Oracle® Communications Unified Inventory Management

API Overview

Release 7.4

E88057-01

December 2017

ORACLE®

Oracle Communications Unified Inventory Management API Overview, Release 7.4

E88057-01

Contents

Preface

This guide explains how to extend Oracle Communications Unified Inventory Management (UIM) through standard Java practices using Oracle Communications Design Studio, which is an Eclipse-based integrated development environment. This guide includes references to both applications, and often directs the reader to see the Design Studio Help and the UIM Help for instructions on how to perform specific tasks.

This guide includes information about the UIM entity managers. This guide also includes the list of Java managers which provide UIM's NFV Orchestration functionality. Similar to extending UIM and using the UIM APIs, the information in this guide applies to extending the NFV Orchestration functionality as well.

This guide should be read after reading *UIM Concepts*, because this guide assumes that the reader has a working knowledge of UIM architecture and concepts. This guide should be read from start to finish because the information presented in a chapter often builds upon information presented in a preceding chapter.

This guide includes examples of typical development code used in given situations. The guidelines and examples may not be applicable in every situation.

Audience

This guide is intended for developers who implement code to extend UIM. The developers should have a good working knowledge of XML and Java development and, in particular, JPA, standard Java practices, and J2EE principles. In working with the NFV Orchestration functionality, this guide assumes you have a working knowledge of NFV concepts.

Related Documentation

For more information, see the following documents in the Oracle Communications Unified Inventory Management documentation set:

- *UIM Installation Guide*: Describes the requirements for installing UIM, installation procedures, and postinstallation tasks.
- *UIM System Administrator's Guide*: Describes administrative tasks such as working with cartridges and cartridge packs, maintaining security, managing the database, configuring Oracle Map Viewer, and troubleshooting.
- *UIM Security Guide*: Provides guidelines and recommendations for setting up UIM in a secure configuration.
- *UIM Concepts*: Provides an overview of important concepts and an introduction to using both UIM and Design Studio.

- *UIM Developer's Guide*: Explains how to customize and extend many aspects of UIM, including the data model, life-cycle management, topology, security, rulesets, user interface, and localization.
- *UIM Web Services Developer's Guide*: Describes the UIM Service Fulfillment Web Service operations and how to use them, and describes how to create custom web services.
- *UIM Information Model Reference*: Describes the UIM information model entities and data attributes, and explains patterns that are common across all entities. This is available on the Oracle Software Delivery Cloud under "Oracle Communications Unified Inventory Management Developer Documentation."
- *Oracle Communications Information Model Reference*: Describes the Oracle Communications information model entities and data attributes, and explains patterns that are common across all entities. The information described in this reference is common across all Oracle Communications products. This is available on the Oracle Software Delivery Cloud under "Oracle Communications Unified Inventory Management Developer Documentation."
- *UIM Cartridge Guide*: Provides information about how you use cartridges and cartridge packs with UIM. Describes the content of the base cartridges.
- *UIM NFV Orchestration Implementation Guide*: Provides information about the NFV Orchestration functional module and includes how to you install, use, and extend this functionality. This guide also provides reference information for the NFV Orchestration RESTful APIs.

For step-by-step instructions for performing tasks, log in to each application to see the following:

- Design Studio Help: Provides step-by-step instructions for tasks you perform in Design Studio.
- UIM Help: Provides step-by-step instructions for tasks you perform in UIM.

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

Overview

This document provides information that you can use when working with the Oracle Communications Unified Inventory Management (UIM) application program interfaces (APIs). This document also provides information that you can use when working with NFV Orchestration Java manager APIs which are also UIM APIs. The UIM APIs can be extended through custom code. The APIs, or extended APIs, can be called from various places, such as from custom rulesets, custom web services, or customized portions of the user interface (UI).

This document provides information on common things you need to do when working with any of the UIM APIs, such as working with transactions, handling errors, and logging messages. This information is described in Chapter 2, "Working with Transactions, Exceptions, and Logging".

The bulk of this document is an overview of numerous UIM APIs, which were specifically selected to describe API usage patterns and best practices for implementing common business scenarios. Code samples are provided to show correct usage of the APIs and expectations of implementing the APIs. This information is described in Chapter 3, "Implementing a Generic Service Fulfillment Scenario" and Chapter 4, "Implementing a Channelized Connectivity Enablement Scenario".

Lastly, this document provides a listing of the following:

- UIM entity manager classes
- NFV Orchestration Java managers
- Code examples for common utility methods

This information is described in Appendix A, "UIM Entity Managers", Appendix B, "NFV Orchestration Java Managers" and Appendix C, "Common Utility Code Examples".

This document does not cover detailed Javadoc information, nor does it cover model and domain information provided in other UIM documentation. This document assumes that you are familiar with UIM functionality, and are planning to extend UIM functionality by implementing a custom solution based on information provided in *UIM Developer's Guide* or *UIM NFV Orchestration Implementation Guide*.

2

Working with Transactions, Exceptions, and Logging

This chapter describes working with transactions, exceptions, and logging. You can use this information when working with all UIM APIs because all APIs must be called from within a transaction, and the calling code must handle exceptions and log any errors.

See the UIM Javadoc for detailed information about API methods, such as the exception thrown by each method.

Working with Transactions

This section describes handling transactions when calling APIs. A standard transaction flow typically includes:

- Starting a transaction
- Calling an API
- Determining if an error occurred
- Performing a commit or rollback of the transaction based on whether an error occurred

Example 2–1 shows a custom method that calls a manager API within a transaction:

Example 2–1 Call to an API from within a Transaction

```
public void sampleCallAPI()
{
    UserEnvironment ue = null;
    UserTransaction ut null;
    try {
        // Step 1: Begin a User Environment and Transaction
        ue = startUserEnvironment(); /* see appendix */
        ut = PersistenceHelper.makePersistenceManager().getTransaction();
        ut.begin();

        // Step 2: Call the API
        PlaceManager mgr = PersistenceHelper.makePlaceManager();
        List<PlaceSpecification> list = mgr.getAllPlaceSpecs();
        // Do something with the list...
    }
    catch (Throwable t) {
        // Step 3: Handle Exception
        try {
```

```
            if (t instanceof ValidationException)
                // Do something with the Exception, such as print it.
                System.out.println("Method call returned validation exception.");
        }
        catch (Exception ignore) {}
    }
    finally {
        // Step 4: Commit or Rollback Transaction
        commitOrRollback(ut); /* see appendix */

        // Step 5: End User Environment
        if (ue != null)
            endUserEnvironment(ue); /* see appendix */
    }
}
```

When managing transactions and calling APIs from within a transaction, consider the following:

- A commit is usually needed between separate groups of API calls that are making updates to the database. The group of APIs is called for an atomic and complete set of operations.
- A rollback is needed when any error occurs.
- Ensure the API call is made within the correct context of **live** or **business interaction**.
- Ensure the User Environment is started before the transaction, and is ended within the **finally** block.

Working with Exceptions

This section describes the exceptions that the UIM APIs can throw. The EntityManager API methods typically throw a ValidationException when a validation error is encountered. However, other exceptions can also be thrown. Table 2–1 describes all of the UIM Exceptions that can be thrown, including the ValidationException.

Table 2–1 Exception Descriptions

Exception	Extends	Description
ValidationException	InventoryException	This exception is widely used and represents all variations of business validation exceptional conditions.
TransientObjectException	ValidationException	This exception is thrown by manager methods if an object is passed into a method in a transient state.
ReadOnlyEntityException	RuntimeException	This exception is thrown when a read-only entity is updated or deleted. A read-only entity can be an entity that is in a queued/planned object state.
InventoryException	Exception	This exception is the Base Inventory Exception and other exceptions extend it.
InvalidBusinessInteraction Exception	RuntimeException	This exception is thrown when the caller attempts to perform an operation against an entity under a BusinessInteraction with an invalid status such as completed or cancelled.

Table 2–1 (Cont.) Exception Descriptions

Exception	Extends	Description
DeletedObjectException	ValidationException	This exception is thrown by manager methods if an object is passed into a method in a deleted state.
BusinessInteraction DisassociationException	ValidationException	This exception is thrown when the manager method is attempting to alter a Business Interaction or Business Interaction Item and the Business Interaction validation determines it is not allowed.
BusinessInteraction CompleteException	ValidationException	This exception is thrown when the manager method is attempting to complete a Business Interaction and the validation determines it is not allowed.

Working with Logging

This section describes logging messages (informational, warning, and debug messages). This section also describes detecting what messages were logged during an API call, which is helpful when trying to determine the success or failure of an API call.

See *UIM System Administrator's Guide* for information on configuring UIM logging, including changing the logging level.

Configuring the Logging Level

The logging level, which is the amount of logging output to the log files from UIM API calls, is determined by the values configured in the *UIM_Home*/**config/loggingconfig.xml** file.

Example 2–2 shows an entry from the **loggingconfig.xml** file. This entry results in any debug messages (through log.debug) existing in the code to be output to the log file when the class exists in the specified package:

Example 2–2 Entry from loggingconfig.xml

```
<Logger name="oracle.communications.inventory.extensibility" additivity="false">
    <level="debug" />
    <AppenderRef ref="stdout"/>
    <AppenderRef ref="rollingFile"/>
</Logger>
```

Working with the Log Interface

The Log interface is located in the package:

oracle.communications.inventory.api.framework.logging

The Log interface provides the ability for an API, or custom code calling an API, to log errors, throw exceptions, and log informational, warning, or debug messages.

Table 2–2 lists the items that can be requested of the Log interface. See the UIM Javadoc for information regarding the specific parameters of each method.

Table 2–2 Log Interface Description

Description	Method to Use	Throws Exception	Checked with Method on FeedbackProvider
Fatal Exception	fatal()	LogFatalException	getFatals()
Validation Exception	validationException()	ValidationException or the exception type provided on method input	getErrors() hasMessages()
Validation Error	validationError()	Currently does not throw a ValidationException	getErrors() hasMessages()
Warning Message	warn()	Not applicable	getWarnings() hasMessages()
Informational Message	info()	Not applicable	getNotes() hasMessages()
Debug Message	debug()	Not applicable	getDebugs()

When calling an API method, additional errors may be thrown. For example, a custom ruleset that calls an API method may throw additional log messages that the developer wants to include in the log file. Example 2–3 shows custom code that adds additional log messages to the log file by calling the Log interface to log an informational message and a debug message:

Example 2–3 Using the Log Interface

```
import oracle.communications.inventory.api.framework.logging.Log;
import oracle.communications.inventory.api.framework.logging.LogFactory;
protected Log log;

public void testLog()
{
    this.log = LogFactory.getLog(this.getClass());
    this.log.validationError("service.findServiceError", service.getId());

    if (this.log.isInfoEnabled())
        this.log.info ("", "This is an informational message");

    if (this.log.isDebugEnabled())
        this.log.debug ("", "This is a debug message.");
}
```

About UIM Log Messages

Messages logged by UIM APIs are defined in several ***.properties** files, per domain. For example, the **service.properties** file defines the messages for the service domain, and the **equipment.properties** file defines the messages for the equipment domain. All message-specific ***.properties** files are located in the *UIM_Home*/**config/resources/logging** directory.

Several of methods on the Log interface define an input parameter of a String key for an error message. These unique keys, along with a corresponding error message String, are defined in the message-specific ***.properties** files. Example 2–4 shows a single message entry from the **servce.properties** file:

Example 2–4 Message Entry from service.properties

```
service.findServiceError.id=110311
service.findServiceError=Error finding service with id {0}.
```

The numbers within the braces are parameter values passed in as arguments to the method call.

Defining Custom Log Messages

You can define custom log messages in the *UIM_Home*/**config/resources/logging /*.properties** files by adding a unique key and corresponding message. The key must be unique across all ***.properties** files in this directory, and across any ***.properties** files contained in any installed cartridges.

Working with the FeedbackProvider Interface

The FeedbackProvider interface is located in the package:

oracle.communications.inventory.api.framework.logging

After calling an API, the code must determine what messages have been logged. The FeedbackProvider interface provides the ability for an API, or custom code calling an API, to interrogate what has occurred. Example 2–5 shows code that checks to see if an error has been logged, and then prints the error:

Example 2–5 Using the FeedbackProvider Interface

```
public void sampleCallAPIWithFeedbackProvider()
{
    UserEnvironment ue = null;
    UserTransaction ut = null;

    try {
        // Step 1: Begin a User Environment and Transaction
        // Step 2: Call the API
        if (!hasErrors()) /* see appendix */
            ut.commit();
        else {
            ut.rollback();
            List<FeedbackMessage> errors =
            ue.getFeedbackProvider().getErrors();
            for ( java.util.Iterator iter = errors.iterator(); iter.hasNext(); )
            {
                FeedbackMessage error = (FeedbackMessage)iter.next();
                System.out.println("Error occurred: " + error.getMessage());
            }
        }
    }
    catch (Throwable t)
    {
        // Step 3: Handle Exception
    }
    finally
    {
        // Step 4: Commit or Rollback Transaction
        // Step 5: End User Environment
    }
}
```

3

Implementing a Generic Service Fulfillment Scenario

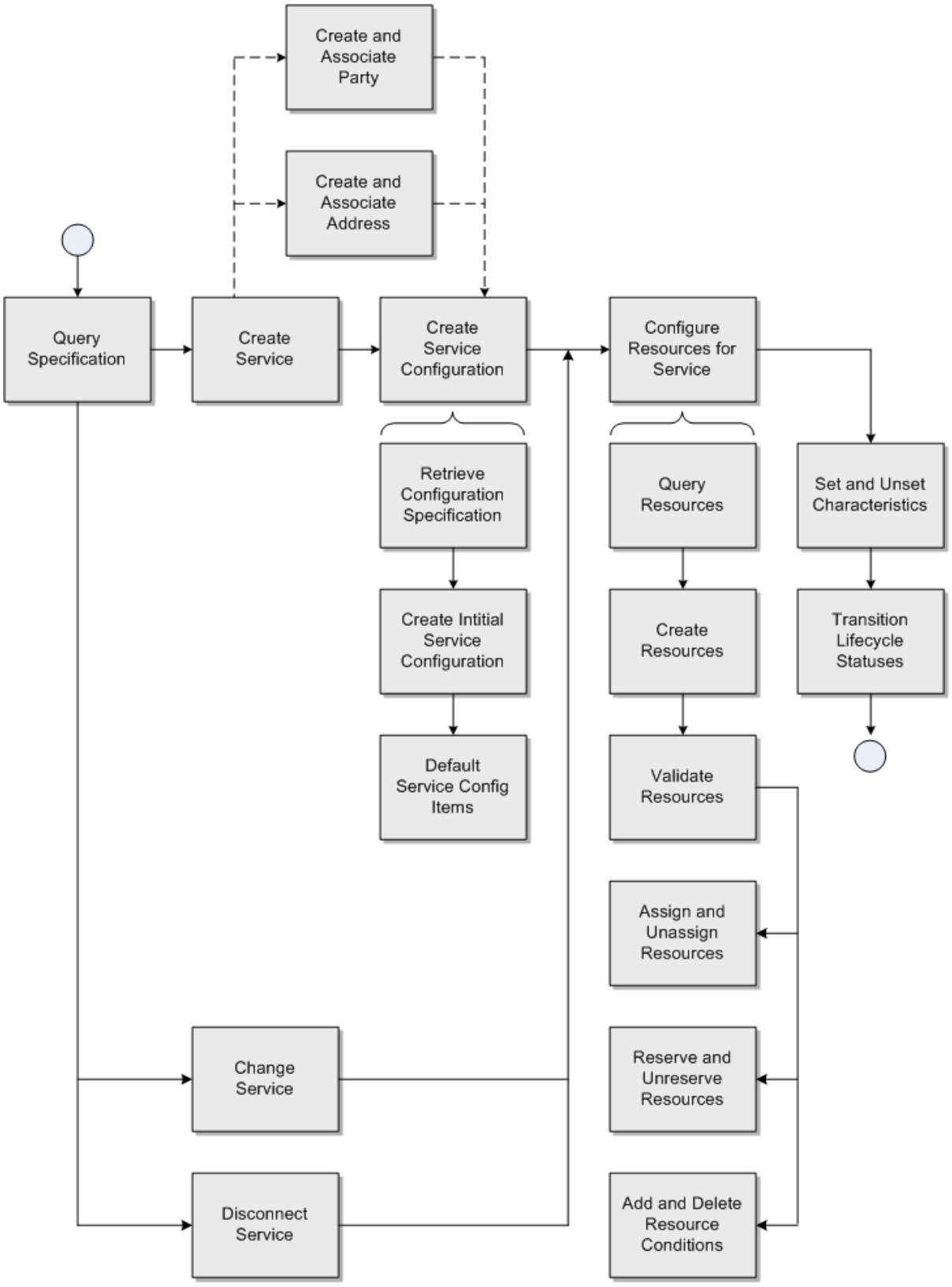
This chapter describes implementing a generic service fulfillment scenario using various Oracle Communications Unified Inventory Management (UIM) application program interfaces (APIs). You can use this information to gain a better understanding of how the UIM APIs can be used to implement any service scenario.

About the Generic Service Fulfillment Scenario

The generic service fulfillment scenario is a Service entity with a single Custom Object resource assignment. The example Service entity is simplified, but the API descriptions are applicable and extensible to other types of services with various types of resource assignments.

Figure 3–1 shows the process flow for a generic service fulfillment scenario:

Figure 3–1 Process Flow of Generic Service Fulfillment Scenario



The process flow begins with querying for the service specification, which is used in subsequent steps in the process flow, such as creating the Service and searching for resources.

The process flow continues with creating the service, based upon the retrieved service specification.

Next is creating the service configuration, which involves querying for the service configuration specification, creating the service configuration based upon the retrieved service configuration specification, and any creating default service configuration items.

The process flow continues with the optional steps of creating additional entities, such as Party and Geographic Address (a concrete Geographic Place entity representing a Service Address). These entities are created and associated to the Service with specific inventory roles.

Next in the process flow is configuring the resources for the service (resource management), which involves querying for resources based on specific criteria using core API searches or using custom searches. For example, you can call an API directly to search for a Custom Object by ID, or you can call a custom API to search for a Custom Object by its association to an Inventory Group or association to another Custom Object. You can also create resources for immediate assignment to the service. The main goal of resource management is to retrieve and validate the correct resources for assignment to the service. However, you can also manage the resources with alternate flows, such as creating reservations and conditions. Assignments, reservations, and conditions are the main consumption concepts for a given resource.

In addition to resource assignments, the service and service configuration also have characteristic values. These values are used to setup and configure the service instance.

After the service has been configured through resource and characteristic value assignments, the process flow continues with transitioning the lifecycle status of various entities. APIs are presented to show the transition of the statuses, and how the statuses are managed within the core API functionality.

The process flow shown in Figure 3–1 shows the initial creation of the service, and also shows other scenarios, such as changing the service configuration and disconnecting the service. These additional scenarios are also described.

Now that you have a high-level understanding of the generic service fulfillment process flow, each part of the process flow is further described in the following sections. Each section includes information about the specific UIM APIs used to perform each step and possible alternate flows of each step. Example code is also included for each step.

- Querying for the Specification
- Creating the Service and Service Configuration
- Creating and Associating the Party
- Creating and Associating the Geographic Address with the Service
- Configuring the Resources for the Service Configuration
- Setting Characteristic Values for the Service Configuration Item
- Transitioning the Lifecycle Status

Querying for the Specification

This section describes the UIM API method used to query for the service specification. The retrieved service specification will later be used to create the service.

Table 3–1 and example code provide information about using the API method.

Table 3–1 Querying for the Specification

Topic	Information
Name	SpecManager.findSpecifications
Description	This method retrieves specifications based on input criteria.
Pre-Condition	The service specification already exists.
Internal Logic	The database is queried for specifications meeting the input criteria. Specifications matching the criteria are returned.
Post-Condition	The desired service specification has been retrieved.
Extensions	Not applicable
Tips	If a list of specifications is returned, the list will need to be iterated to select the desired specification to be used to create the service. Set the SpecSearchCriteria.setValidSpecsOnly (true) to instruct the find method to only return active specifications. Set the SpecSearchCriteria.setSpecClass (ServiceSpecification.class) to instruct the find method to only return service specifications. Additional criteria, such as name, may also be set to further constrain the list of service specifications returned by the find method. This method is applicable for retrieving other types of specifications by supplying the correct Specification class as the query parameter. For example, it can be used to retrieve a CustomObject specification to be used later for resource query or creation.

Example 3–1 Querying for the Specification

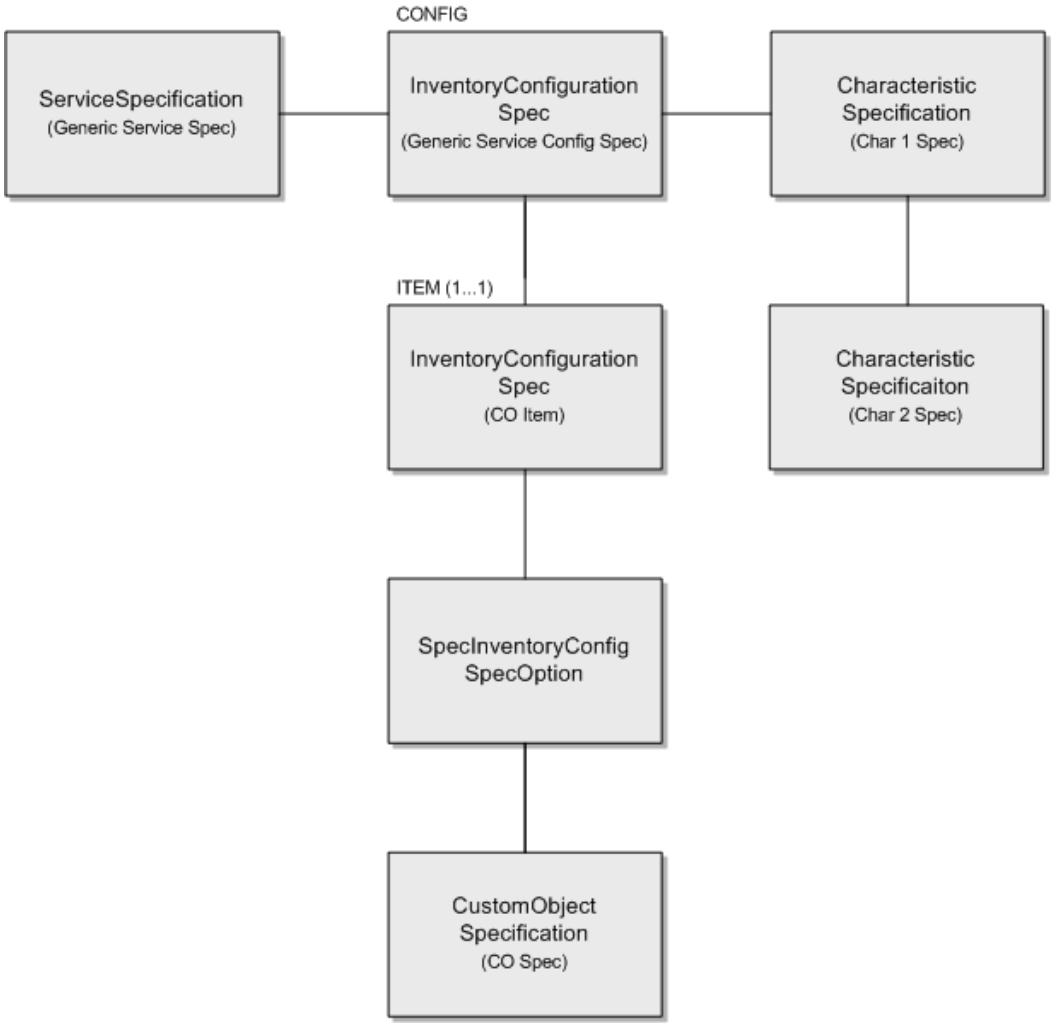
```
Specification spec = null;
SpecManager specMgr = PersistenceHelper.makeSpecManager();

SpecSearchCriteria criteria = specMgr.makeSpecSearchCriteria();
CriteriaItem critSpecName = criteria.makeCriteriaItem();
critSpecName.setValue(specName);
critSpecName.setOperator(CriteriaOperator.EQUALS_IGNORE_CASE);
criteria.setName(critSpecName);
criteria.setSpecClass(ServiceSpecification.class);

List<Specification> specs = specMgr.findSpecifications(criteria);
if (Utils.isEmpty(specs))
{
    /* log error */
}
spec = specs.get(0);
```

Creating the Service and Service Configuration

This section describes the UIM API methods used to create the service and service configuration, and to create default configuration items on the service configuration. The API methods are listed in the order in which they must be called.

Figure 3–2 shows the generic service configuration specification used in the generic service fulfillment scenario:

Figure 3–2 Generic Service Configuration Specification Example



Creating the Service

This section describes the UIM API method used to create the service, based upon the retrieved service specification.

Table 3–2 and example code provide information about using the API method.

Table 3–2 Creating the Service

Topic	Information
Name	ServiceManager.createService
Description	This method creates a service instance built from the input service specification. The service will be populated with the hard facts and characteristics supplied by the caller.
Pre-Condition	A service specification has been selected.
Internal Logic	The service is created using the input service specification.

Table 3–2 (Cont.) Creating the Service

Topic	Information
Post-Condition	The service has been created and is in Pending status.
Extensions	Not applicable
Tips	The Service.startDate and Service.name are required attributes. The Service.characteristics can be populated with the desired characteristics. If the service specification is defined with any required characteristics that do not have default values specified, then those characteristic must be set on the service in order for it to be created successfully.

Example 3–2 Creating the Service

```
ServiceManager smgr = PersistenceHelper.makeServiceManager();

Finder f = PersistenceHelper.makeFinder();
Collection<ServiceSpecification> serviceSpecCollection =
    f.findByName(ServiceSpecification.class,"service_spec");
ServiceSpecification serviceSpec = (ServiceSpecification)
    serviceSpecCollection.iterator().next();

Service serviceModel = smgr.makeService(Service.class);
serviceModel.setName("Service_test22");
serviceModel.setDescription("Service_test22_desc");
serviceModel.setId("Service_test22");
serviceModel.setSpecification(serviceSpec);

Collection<Service> services = new ArrayList<Service>();
services.add(serviceModel);

List<Service> createdServices = smgr.createService(services);
service = createdServices.get(0);
```

Retrieving the Service Configuration Specification

This section describes the UIM API method used to retrieve the service configuration specification. The retrieved service configuration specification will later be used to create the service configuration.

Table 3–3 and example code provide information about using the API method.

Table 3–3 Retrieving the Service Configuration Specification

Topic	Information
Name	ConfigurationManager.getConfigSpecTypeConfig
Description	This method retrieves the configuration specifications related to the input service specification.
Pre-Condition	The service specification is associated to one or more configuration specifications.
Internal Logic	The configuration specifications related to the service specification are retrieved and returned.
Post-Condition	A configuration specification has been selected.
Extensions	Not applicable

Table 3–3 (Cont.) Retrieving the Service Configuration Specification

Topic	Information
Tips	If a list of specifications is returned, the list will need to be iterated to select the desired specification to be used to create the service configuration.

Example 3–3 Retrieving the Service Configuration Specification

```
ConfigurationManager configurationManager =
    PersistenceHelper.makeConfigurationManager();

List< InventoryConfigurationSpec > configSpecs =
    configurationManager.getConfigSpecTypeConfig( serviceSpec, true );

return configSpecs;
```

Creating the Service Configuration

This section describes the UIM API method used to create the service configuration, based upon the retrieved service configuration specification.

Table 3–4 and example code provide information about using the API method:

Table 3–4 Creating the Service Configuration

Topic	Information
Name	BaseConfigurationManager.createConfigurationVersion(Configurable configurable, InventoryConfigurationVersion configuration, InventoryConfigurationSpec configSpec)
Description	This method creates a service configuration version and associates it to the service.
Pre-Condition	The service exists with no service configuration versions.
Internal Logic	Not applicable
Post-Condition	The first configuration version is created and associated to the service. This method will default the configuration items based on the input configSpec.
Extensions	Not applicable
Tips	The service, configuration and configSpec parameters are required.

Example 3–4 Creating the Service Configuration

```
Finder f = PersistenceHelper.makeFinder();
Collection<Service> serviceCollection =
    f.findById(Service.class, servId);
Service serv = serviceCollection.iterator().next();
f.reset();
Collection<InventoryConfigurationSpec> invSpecCollection =
    f.findByName(InventoryConfigurationSpec.class,"Serv_Config");
InventoryConfigurationSpec invSpec =
    invSpecCollection.iterator().next();
BaseConfigurationManager bcd =
    PersistenceHelper.makeConfigurationManager
        (ServiceConfigurationVersion.class);
InventoryConfigurationVersion scv =
    bcd.makeConfigurationVersion(serv);
scv.setDescription(configId);
```

```
scv.setId(configId);
scv.setName(configId);
scv.setEffDate(new Date());
InventoryConfigurationVersion createdConfig =
    bcd.createConfigurationVersion(serv, scv,invSpec);
```

About Alternate Flows

The generic service fulfillment scenario creates a service and initial service configuration. Alternate flows to this scenario may be to change the service, or to disconnect the service.

The alternate flows described in this section are:

- Changing the Service
- Disconnecting the Service

Changing the Service

This section describes the UIM API method used to change an existing service by adding a new service configuration version. The main goal is to create an IN_PROGRESS service configuration version so additional resource or characteristic changes can be executed. For example, after creating an initial service configuration version to assign a custom object to a service, a second service configuration version can be created to unassign the custom object previously allocated.

Table 3–5 and example code provide information about using the API method.

Table 3–5 Changing the Service

Topic	Information
Name	BaseConfigurationManager.createConfigurationVersion(Configurable configurable, InventoryConfigurationVersion configuration)
Description	This method creates new configuration version from the most recently completed previous configuration version.
Pre-Condition	A service with a completed service configuration version must exist.
Internal Logic	Not applicable
Post-Condition	A service configuration version is created with a status of IN_PROGRESS.
Extensions	Not applicable
Tips	The service and configuration parameters are required.

Example 3–5 Changing the Service

```
Finder f = PersistenceHelper.makeFinder();
Collection<Service> serviceCollection = f.findById(Service.class, servId);
Service serv = serviceCollection.iterator().next();
f.reset();
Collection<InventoryConfigurationSpec> invSpecCollection =
    f.findByName(InventoryConfigurationSpec.class,"Serv_Config");
InventoryConfigurationSpec invSpec =
    invSpecCollection.iterator().next();
BaseConfigurationManager bcd =
    PersistenceHelper.makeConfigurationManager(ServiceConfigurationVersion.class);
InventoryConfigurationVersion scv =
    bcd.makeConfigurationVersion(serv);
scv.setDescription(configId);
```

```
scv.setId(configId); scv.setName(configId);
scv.setEffDate(new Date());
InventoryConfigurationVersion createdConfig =
    bcd.createConfigurationVersion(serv, scv);
```

Disconnecting the Service

This section describes the UIM API method used to disconnect a service when the service is no longer needed.

Table 3–6 and example code provide information about using the API method.

Table 3–6 Disconnecting the Service

Topic	Information
Name	ServiceManager.disconnectService
Description	This method will transition the state of a service and invoke necessary business logic for the service and configuration version depending on the type of transition initiated.
Pre-Condition	The service exists and there are no configuration versions in a state other than Completed or Cancelled.
Internal Logic	Not applicable
Post-Condition	The service has a Pending Disconnect status. A new configuration version is created and any resources that are currently assigned, are unassigned. The configuration version has an In Progress status.
Extensions	Not applicable
Tips	The businessAction to be passed as input to the transition method is ServiceAction.DISCONNECT.

Example 3–6 Disconnecting the Service

```
ServiceManager sm = PersistenceHelper.makeServiceManager();
sm.disconnectService(service);
```

Creating and Associating the Party

This section describes the UIM API methods used to create a party, create a party role, and associate the party and party role with the service. The API methods are listed in the order in which they must be called.

> **Note:** The associations of the party and party role with the service are optional, and can be associated before or after the creation of the initial service configuration. Typically, these types of associations do not change for the service, but alternate flows are presented to show how the associations can be changed if necessary.

Creating the Party

This section describes the UIM API method used to create the party.

Table 3–7 and example code provide information about using the API method.

Table 3–7 Creating the Party

Topic	Information
Name	PartyManager.createParties
Description	This method takes a collection of Party entities and persist them into the database. The Party Role and association to the Service is setup by a different API.
Pre-Condition	Party Specification is valid and retrieved from the database. Party has a valid and unique ID.
Internal Logic	Take the collection of transient Party entities and persists them into the database, and return the collection of persisted Party entities. Validate that the Parties are not duplicated by ID and they all have valid PartySpecification.
Post-Condition	Persistent Party entities are returned.
Extensions	This API is defined as an extension point to allow custom validation before or after the Parties are created. For instance, the IDs can be generated based on some custom algorithm.
Tips	Party is a CharacteristicExtensible entity. The characteristic values should be added when the Party instance is created. Use RoleManager APIs to manage the roles played by a given Party, and use AttachmentManager to associate the Party with specific Role to a given Service.

Example 3–7 Creating the Party

```
Finder finder = PersistenceHelper.makeFinder();

PartyManager mgr = PersistenceHelper.makePartyManager();
Party party = mgr.makeParty();
Collection<Party> parties = new ArrayList<Party>();

party.setId(partyId);
party.setName("Party_Name");
party.setDescription("Party_Description");

Collection<PartySpecification> partyspec =
    f.findByName(PartySpecification.class,"Test_Party_Spec");

PartySpecification partySpec =partyspec.iterator().next();
party.setSpecification(partySpec);

parties.add(party);

List<Party> results = mgr.createParties(parties);
Party resulty = results.iterator().next();;
```

Creating the Party Role

This section describes the UIM API method used to create the party role.

Table 3–8 and example code provide information about using the API method.

Table 3–8 Creating the Party Role

Topic	Information
Name	RoleManager.createInventoryRole

Table 3–8 (Cont.) Creating the Party Role

Topic	Information
Description	This method takes a collection of InventoryRole entities and persist them into the database. The roles passed in are the concrete subclass, for instance PartyRole.
Pre-Condition	InvRoleSpecification is valid and retrieved from the database. The Party which has the roles is already created.
Internal Logic	Take the collection of transient InventoryRole entities and persists them into the database, and return the collection of persisted InventoryRole entities. Validate that the roles are not duplicated and they all have valid InvRoleSpecification.
Post-Condition	Persistent concrete subclass (i.e. PartyRole) entities are returned.
Extensions	Not applicable
Tips	Use RoleManager.makePartyRole() API to get a transient instance of the correct concrete subclass of role to create. InvRoleSpecification is required.

Example 3–8 Creating the Party Role

```
Finder finder = PersistenceHelper.makeFinder();
RoleManager roleMgr = PersistenceHelper.makeRoleManager();
PartyRole role = roleMgr.makePartyRole();
/* Utility Method Call - see 3.2.1 Query Spec */
Collection<InvRoleSpecification> invrolespeclist =
    finder.findByName(InvRoleSpecification.class,("Test_Party_Role_Spec");
InvRoleSpecification rolespec =
    (InvRoleSpecification)invrolespeclist.iterator().next();
role.setSpecification(roleSpec);
List<InventoryRole> roles = new ArrayList<InventoryRole>();
roles.add(role);
roleMgr.createInventoryRole(roles);
```

Associating the Party and Party Role with the Service

This section describes the UIM API method used to associate the party and party role with the service. The API method must be called once per association. So, in this scenario, the API is called to associate the party with the service, and then called again to associate the party role with the service.

Table 3–9 and example code provide information about using the API method. The example shows associating the party with the service; it does not show associating the party role with the service, which is accomplished by calling the same API method.

Table 3–9 Associating the Party and Party Role with the Service

Topic	Information
Name	AttachmentManager.createRel
Description	This method creates an involvement (an association) between two entities.
Pre-Condition	Service, Party and PartyRole are already created.
Internal Logic	Creates an involvement entity to represent the relationship from Party to Service with a specific PartyRole. The Party is the parent of this involvement. Validates that the relationship is not duplicated.
Post-Condition	PartyServiceRel is created referencing the entities.

Table 3–9 (Cont.) Associating the Party and Party Role with the Service

Topic	Information
Extensions	Not applicable
Tips	Set the FROM entity to Party and TO entity to Service. Set the FROM entity role to the PartyRole.

Example 3–9 Associating the Party to the Service

```
String roleOid = role.getOid();
AttachmentManager involvementMgr =
    PersistenceHelper.makeAttachmentManager();
Involvement involvement =
    involvementMgr.makeRel(PartyServiceRel.class);
involvement.setToEntity(service);
involvement.setFromEntity(party);
involvement.setFromEntityRoleKey(roleOid);
involvementMgr.createRel(involvement);
PartyServiceRel partyServiceRel = (PartyServiceRel)involvement;
```

About Alternate Flows

The generic service fulfillment scenario creates a party and party role, and associates them with the service. Alternate flows to this scenario may be to disassociate the party and party role from the service, and then delete the party and party role.

The alternate flows described in this section are:

- Disassociating the Party and Party Role from the Service
- Deleting the Party
- Deleting the Party Role

Disassociating the Party and Party Role from the Service

This section describes the UIM API methods used to retrieve a party or service, and then use the retrieved data to disassociate the party from the service. The API methods are listed in the order in which they must be called.

The API methods must be each called once per disassociation. So, in this scenario, an API is called to retrieve the party or service, and another API is called to disassociate the party from the service. This process is repeated to disassociate the party role from the service: An API is called to retrieve the party role or service, and another API is called to disassociate the party role from the service.

Table 3–10 and Table 3–11 provide information about using the API methods.

Table 3–10 Getting the Party and the Service

Topic	Information
Name	Service.getParty() or Party.getService()
Description	These methods are used to retrieve the bidirectional relationship PartyServiceRel between Party and Service. Once retrieved, the correct instance can be deleted.
Pre-Condition	PartyServiceRel is already created.
Internal Logic	Simple relationship attribute on the entities to get list of relationships to iterate through.
Post-Condition	PartyServiceRel is found and passed to next method for deletion.

Table 3–10 (Cont.) Getting the Party and the Service

Topic	Information
Extensions	Not applicable
Tips	Not applicable

Table 3–11 Disassociating the Party from the Service

Topic	Information
Name	AttachmentManager.deleteRel
Description	This method deletes an involvement (an association) between two entities. In this example, an existing relationship between the Party and Service with a specific role is deleted.
Pre-Condition	PartyServiceRel is already created.
Internal Logic	Delete the PartyServiceRel entity.
Post-Condition	PartyServiceRel is deleted.
Extensions	Not applicable
Tips	Delete existing PartyServiceRel and create new ones to change Party to Service relationships.

Deleting the Party

This section describes the UIM API method used to delete a party.

Table 3–12 provides information about using the API method.

Table 3–12 Deleting the Party

Topic	Information
Name	PartyManager.deleteParty
Description	This method deletes an existing Party, and all its existing PartyRoles.
Pre-Condition	Party is already created.
Internal Logic	Delete the Party entity. The Party will not be deleted if it is associated with other entities, such as involvement with a Service.
Post-Condition	Party is deleted.
Extensions	The API is an extension point for adding custom validation logic, such as logging and removing any relationships before deleting.
Tips	Use this method to delete an incorrect or obsolete Party before creating a new Party.

Deleting the Party Role

This section describes the UIM API method used to delete a party role.

Table 3–13 provides information about using the API method.

Table 3–13 Deleting the Party Role

Topic	Information
Name	RoleManager.deleteInventoryRoles
Description	This method deletes an existing InventoryRole on a given entity. In this example, a PartyRole subclass instance is deleted.

Table 3–13 (Cont.) Deleting the Party Role

Topic	Information
Pre-Condition	PartyRole is already created.
Internal Logic	Delete the PartyRole entity.
Post-Condition	PartyRole is deleted.
Extensions	Not applicable
Tips	Use this method to delete an incorrect or obsolete role before creating a new role.

Creating and Associating the Geographic Address with the Service

This section describes the UIM API methods used to create a place, create a place role, and associate the place and place role with the service. (A place is a GeographicPlace entity, which id is a concrete entity representing a geographic address / service address.) The API methods are listed in the order in which they must be called.

Note: The associations of the place and place role with the service are optional, and can be associated before or after the creation of the initial service configuration. Typically, these types of associations do not change for the service, but alternate flows are presented to show how the associations can be changed if necessary.

Creating the Geographic Place

This section describes the UIM API method used to create the geographic place.

Table 3–14 and example code provide information about using the API method.

Table 3–14 Creating the Geographic Place

Topic	Information
Name	PlaceManager.createGeographicPlace
Description	This method takes a collection of Geographic Address entities which represents the Service Address and persist them into the database. The Place Role and association to the Service is setup by a different API. For this example, create a Geographic Address, a concrete subclass of Geographic Place, as an instance of the Service Address.
Pre-Condition	Place Specification is valid and retrieved from the database. Geographic Address has a valid and unique ID.
Internal Logic	Take the collection of transient Geographic Address entities and persists them into the database, and return the collection of persisted Geographic Address entities. Validate that the Geographic Address are not duplicated by ID and they all have valid PlaceSpecification.
Post-Condition	Persistent Geographic Address entities are returned.
Extensions	This API is defined as an extension point to allow custom validation before or after the Geographic Addresses are created. For instance, the IDs can be generated based on some custom algorithm.
Tips	Geographic Address is a CharacteristicExtensible entity. Its characteristic values should be added as the instance is created. Use RoleManager APIs to manage the roles played by a given Geographic Address, and use AttachmentManager to associate the Geographic Address with specific Role to a given Service. (Same as Party.)

Example 3–10 Creating the Geographic Place

```
Finder finder = PersistenceHelper.makeFinder();
PlaceManager placeMgr = PersistenceHelper.makePlaceManager();
GeographicAddress place =
    placeMgr.makeGeographicPlace(GeographicAddress.class);
place.setId("Place_ID");
place.setName("Place_Name");

Collection<PlaceSpecification> placeSpecification = finder.findByName
    (PlaceSpecification.class,(String)paramMap.get("Test_Place_Spec"));

PlaceSpecification pcspec = PlaceSpecification.iterator().next();
place.setSpecification((PlaceSpecification) placeSpec);

List places = new ArrayList<GeographicAddress>();
places.add(place);
places = placeMgr.createGeographicPlace(places);
place = (GeographicAddress) places.iterator().next();
```

Creating the Place Role

This section describes the UIM API method used to create the place role.

Table 3–15 and example code provide information about using the API method.

Table 3–15 Creating the Place Role

Topic	Information
Name	RoleManager.createInventoryRole
Description	This method takes a collection of InventoryRole entities and persist them into the database. The roles passed in are the concrete subclass, for instance PlaceRole.
Pre-Condition	InvRoleSpecification is valid and retrieved from the database. The Geographic Address which has the roles is already created.
Internal Logic	Take the collection of transient InventoryRole entities and persists them into the database, and return the collection of persisted InventoryRole entities. Validate that the roles are not duplicated and they all have valid InvRoleSpecification.
Post-Condition	Persistent concrete subclass (i.e. PlaceRole) entities are returned.
Extensions	Not applicable
Tips	Use RoleManager.makePlaceRole() API to get a transient instance of the correct concrete subclass of role to create. InvRoleSpecification is required.

Example 3–11 Creating the Place Role

```
Finder finder = PersistenceHelper.makeFinder();
RoleManager roleMgr = PersistenceHelper.makeRoleManager();
PlaceRole role = roleMgr.makePlaceRole();

Collection<InvRoleSpecification> invrolespeclist =
    f.findByName(InvRoleSpecification.class, "Test_Place_Role_Spec");

InvRoleSpecification rolespec =
    (InvRoleSpecification)invrolespeclist.iterator().next();
role.setSpecification(roleSpec);
List<InventoryRole> roles = new ArrayList<InventoryRole>();
```

```
roles.add(role);
roleMgr.createInventoryRole( roles);
```

Associating the Geographic Place and Place Role with the Service

This section describes the UIM API method used to associate the geographic place and place role with the service. The API method must be called once per association. So, in this scenario, the API is called to associate the geographic place with the service, and then called again to associate the place role with the service.

Table 3–16 and example code provide information about using the API method. The example shows associating the geographic place with the service; it does not show associating the place role with the service, which is accomplished by calling the same API method.

Table 3–16 Associating the Geographic Place and Place Role with the Service

Topic	Information
Name	AttachmentManager.createRel
Description	This method creates an involvement (an association) between two entities. In this example, a relationship is created between Geographic Address and Service with a specific role created earlier.
Pre-Condition	Service, Geographic Address and PlaceRole are already created.
Internal Logic	Creates an involvement entity to represent the relationship from Geographic Address to Service with a specific PartyRole. The Geographic Address is the parent of this involvement. Validates that the relationship is not duplicated.
Post-Condition	PlaceServiceRel is created referencing the entities.
Extensions	Not applicable
Tips	Set the FROM entity to Geographic Address and TO entity to Service. Set the FROM entity role to the PlaceRole.

Example 3–12 Associating the Geographic Place with the Service

```
String roleOid = role.getOid();

AttachmentManager involvementMgr = PersistenceHelper.makeAttachmentManager();
Involvement involvement = involvementMgr.makeRel(PlaceServiceRel.class);
involvement.setToEntity(service);
involvement.setFromEntity(place);
involvement.setFromEntityRoleKey(roleOid);
involvementMgr.createRel(involvement);

PlaceServiceRel placeServiceRel = (PlaceServiceRel) involvement;
```

About Alternate Flows

The generic service fulfillment scenario creates a geographic place and place role, and associates them with the service. Alternate flows to this scenario may be to disassociate geographic place and place role from the service, and then delete the geographic place and place role.

The alternate flows described in this section are:

- Disassociating the Geographic Place and Place Role from the Service
- Deleting the Geographic Place

- Deleting the Place Role

Disassociating the Geographic Place and Place Role from the Service

This section describes the UIM API methods used to retrieve a place or service, and then use the retrieved data to disassociate the place from the service. The API methods are listed in the order in which they must be called.

The API methods must be each called once per disassociation. So, in this scenario, an API is called to retrieve the place or service, and another API is called to disassociate the place from the service. This process is repeated to disassociate the place role from the service: An API is called to retrieve the place role or service, and another API is called to disassociate the place role from the service.

Table 3–17 and Table 3–18 provide information about using the API methods.

Table 3–17 Getting the Place and Service

Topic	Information
Name	Service.getPlace() or GeographicPlace.getPlaceservicerels ()
Description	These methods are used to retrieve the bidirectional relationship PlaceServiceRel between Geographic Address and Service. Once retrieved, the correct instance can be deleted.
Pre-Condition	PlaceServiceRel is already created.
Internal Logic	Simple relationship attribute on the entities to get list of relationships to iterate through.
Post-Condition	PlaceServiceRel is found and passed to next method for deletion.
Extensions	Not applicable
Tips	Not applicable

Table 3–18 Disassociating the Place and Place Role from the Service

Topic	Information
Name	AttachmentManager.deleteRel
Description	This method deletes an involvement (an association) between two entities. In this example, an existing relationship between the Geographic Address and Service with a specific role is deleted.
Pre-Condition	PlaceServiceRel is already created.
Internal Logic	Delete the PlaceServiceRel entity.
Post-Condition	PlaceServiceRel is deleted.
Extensions	Not applicable
Tips	Delete existing PlaceServiceRel and create new ones to change Geographic Address to Service relationships.

Deleting the Geographic Place

This section describes the UIM API method used to delete a geographic place.

Table 3–19 provides information about the API method.

Table 3–19 Deleting the Geographic Place

Topic	Information
Name	PlaceManager.deleteGeographicPlace
Description	This method deletes an existing Geographic Address, and all its existing PlaceRoles. In this example, the Service Address as in instance of a Geographic Address is deleted.
Pre-Condition	Geographic Address is already created.
Internal Logic	Delete the Geographic Address entity, and all its existing PlaceRoles. The Geographic Address will not be deleted if it is associated with other entities, such as involvement with a Service.
Post-Condition	Geographic Address is deleted.
Extensions	The API is an extension point for adding custom validation logic, such as logging and removing any relationships before deleting them.
Tips	Use this method to delete an incorrect or obsolete Geographic Address before creating a new Geographic Address.

Deleting the Place Role

This section describes the UIM API method used to delete a place role.

Table 3–20 provides information about the API method.

Table 3–20 Deleting the Place Role

Topic	Information
Name	RoleManager.deleteInventoryRoles
Description	This method deletes an existing InventoryRole on a given entity. In this example, a PlaceRole subclass instance is deleted.
Pre-Condition	PlaceRole is already created.
Internal Logic	Delete the PlaceRole entity.
Post-Condition	PlaceRole is deleted.
Extensions	Not applicable
Tips	Use this method to delete an incorrect or obsolete role before creating a new role.

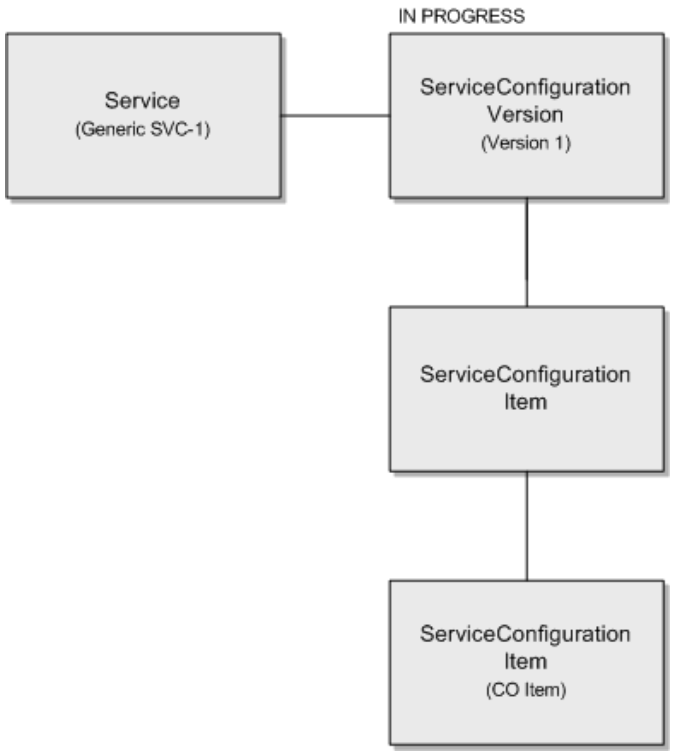
Configuring the Resources for the Service Configuration

This section describes the APIs need to assign a custom object to a service configuration item. The APIs are listed in the order in which they must be called.

Note: If assignment is being done as part of creating the service and service configuration (see "Creating the Service and Service Configuration"), then start at section "Finding the Service Configuration Item" because the service and service configuration are already known.

Figure 3–3 shows how the service and configuration are created by calling the APIs described in Creating the Service and Service Configuration.

Figure 3–3 Generic Service Example



Finding the Service

This section describes the UIM API method used to find the service. The retrieved service will be used to find the service configuration.

Table 3–21 and example code provide information about using the API method.

Table 3–21 Finding the Service

Topic	Information
Name	ServiceManager.findServices
Description	This method retrieves services based on input criteria.
Pre-Condition	The desired service already exists.
Internal Logic	The database is queried for services meeting the input criteria. Services matching the criteria are returned.
Post-Condition	The desired service has been retrieved.
Extensions	Not applicable
Tips	If a list of services is returned, the list will need to be iterated to select the desired service.

Example 3–13 Finding the Service

```
ServiceManager mgr = PersistenceHelper.makeServiceManager();
ServiceSearchCriteria criteria = mgr.makeServiceSearchCriteria();
citem = criteria.makeCriteriaItem();
citem.setValue("Service_Test_22");
citem.setOperator(CriteriaOperator.EQUALS);
criteria.setName(citem);
```

```
List<Service> list = mgr.findServices(criteria);
```

Finding the Current Service Configuration Version

To find the current service configuration version:

1. Find the service. See "Finding the Service".
2. Select the service configuration versions using service.getConfigurations().
3. Process the retrieved service configuration versions, looking for one with a configState of IN_PROGRESS, DESIGNED or ISSUED.

   There will only be one service configuration version in one of these states at a given point in time for a service. If a service configuration version is not found in one of these states, you cannot proceed with resource assignment.

   In the generic service fulfillment scenario, **Version 1** would be selected.

Finding the Service Configuration Item

To find the service configuration item:

1. Find the current service configuration version. See "Finding the Current Service Configuration Version".
2. Select the service configuration items using service.getConfigItems().
3. Process the retrieved service configuration items, looking for one with the configType of ITEM.

   In the generic service fulfillment scenario, **CO Item** would be selected.

> **Note:** In this simplified example, we know there is only one item level configuration item, and we know it is associated to an option for a custom object specification, which is why the following sections find or create a custom object to assign.

Finding the Custom Object to Assign

This section describes the UIM API method used to find the custom object to assign to the retrieved service configuration item. When assigning a custom object to a service configuration item, you can either find an existing custom object, or you can create a new custom object to assign, as described in the following section, "Creating the Custom Object to Assign".

Table 3–22 and example code provide information about using the API method.

Table 3–22 Finding the Custom Object

Topic	Information
Name	CustomObjectManager.findCustomObjects
Description	This method retrieves custom objects based on input criteria.
Pre-Condition	The custom object to be allocated already exists.
Internal Logic	The database is queried for custom objects meeting the input criteria. Custom objects matching the criteria are returned.
Post-Condition	The desired custom object has been retrieved.
Extensions	Not applicable

Table 3–22 (Cont.) Finding the Custom Object

Topic	Information
Tips	Set the CustomObjectSearchCriteria.setAssignmentState(AssignmentState.UNASSIGNED) to instruct the find method to only return available custom objects. In this example, we could choose to set the CustomObjectSearchCriteria.setCustomObjectSpecification (CustomObjectSpecification) to the **CO Spec** instance. If a list of custom objects is returned, the list will need to be iterated to select the desired custom object to be allocated to the service configuration item.

Example 3–14 Finding the Custom Object

```
CustomObjectManager mgr =
    PersistenceHelper.makeCustomObjectManager();
CustomObjectSearchCriteria criteria =
    mgr.makeCustomObjectSearchCriteria();
criteria.setAdminState(InventoryState.INSTALLED);
Finder finder = PersistenceHelper.makeFinder();
finder = PersistenceHelper.makeFinder();

Collection<CustomObjectSpecification> customObjectSpecs =
    finder.findByName(CustomObjectSpecification.class,"Test_Custom_Object_Spec");

criteria.setCustomObjectSpecification(customObjectSpecs.iterator().next());
mgr.findCustomObjects(criteria);

/* another example */
Finder f = PersistenceHelper.makeFinder();
Collection<CustomObject> custObjs = f.findById(CustomObject.class, "CO-1");
```

Creating the Custom Object to Assign

This section describes the UIM API method used to create a custom object to assign to the retrieved service configuration item. When assigning a custom object to a service configuration item, you can either create a new custom object, or you can find an existing custom object to assign, as described in "Finding the Custom Object to Assign".

Table 3–23 and example code provide information about using the API method.

Table 3–23 Creating the Custom Object

Topic	Information
Name	CustomObjectManager.createCustomObjects
Description	This method creates a custom object. The custom object will be populated with the hard facts and characteristics supplied by the caller.
Pre-Condition	Not applicable
Internal Logic	The custom object is created.
Post-Condition	The custom object has been created and is in Installed status.
Extensions	Not applicable
Tips	A custom object can be created with or without a specification.

Example 3–15 Creating the Custom Object

```
CustomObjectManager custMgr =
    PersistenceHelper.makeCustomObjectManager();
Finder f = PersistenceHelper.makeFinder();

Collection<CustomObjectSpecification> specList =
    new ArrayList<CustomObjectSpecification> (
        f.findByName(CustomObjectSpecification.class, "SPEC_CUST_001"));

if (specList != null && !specList.isEmpty())
{
    CustomObjectSpecification custObjSpec =
        specList.iterator().next();

    Collection<CustomObject> custObjects = new ArrayList<CustomObject>();
    CustomObject custObj = custMgr.makeCustomObject();
    custObj.setId("CUST_OBJ_ID");
    custObj.setName("CUST_OBJ_NAME");
    custObj.setDescription("CUST_OBJ_DESC");
    custObj.setSpecification(custObjSpec); /* optional */
    custObjects.add(custObj);

    custMgr.createCustomObjects(custObjects);
}
```

Assigning the Resource to a Configuration Item

This section describes the UIM API method used to assign the resource to a configuration item. In the generic service fulfillment scenario, the resource is the custom object that was either found or created when "Finding the Custom Object to Assign" or "Creating the Custom Object to Assign".

Table 3–24 and example code provide information about using the API method.

Table 3–24 Assigning the Resource to a Configuration Item

Topic	Information
Name	BaseConfigurationManager.assignResource(E item,oracle.communications.inventory.api.entity.common.ConsumableResource resource,java.lang.String reservedFor,java.lang.String reservedForType) In this example, the full signature of the method is included because there are multiple overloaded assignResource methods.
Description	This method assigns the input resource to the input service configuration item. In this example, a custom object is used as the consumable resource for assignment.
Pre-Condition	The configuration item to allocate the custom object to has been selected.
Internal Logic	Not applicable
Post-Condition	The custom object has been allocated to the service configuration item.
Extensions	Not applicable
Tips	The input item is the entity configuration item to assign the resource to (ConsumableResource). In this example, ConsumableResource is set to the CustomObject for **CO-1**. The reservedFor and reservedForType parameters should be populated if the resource to be assigned is reserved, so the reservation can be redeemed.

Example 3–16 Assigning the Resource to a Configuration Item

```
Finder finder = PersistenceHelper.makeFinder();
Collection<CustomObject> custObjs =
    finder.findByName(CustomObject.class, "CO-1");
CustomObject custObj = custObjs.iterator().next();
ServiceManager mgr = PersistenceHelper.makeServiceManager();

ServiceSearchCriteria criteria = mgr.makeServiceSearchCriteria();
CriteriaItem citem = criteria.makeCriteriaItem();
citem.setValue("Service_Test_22");
citem.setOperator(CriteriaOperator.EQUALS);
criteria.setName(citem);

List<Service> list = mgr.findServices(criteria);
Service service = list.get(0);
List<ServiceConfigurationVersion> srvConfigurations =
    service.getConfigurations();
ServiceConfigurationItemAllocationData itemData =
    new ServiceConfigurationItemAllocationData();
int i = srvConfigurations.get(0).getVersionNumber();

//Write logic to get the latest ServiceConfigurationVersion of the Service.
//Process the retrieved service configuration versions,
//looking for one with a configState of IN_PROGRESS, DESIGNED or ISSUED.
ServiceConfigurationVersion latestConfiguration;

//Assign the latest ServiceConfigurationVersion
//to the variable latestConfiguration
List<ServiceConfigurationItem> configItems =
    latestConfiguration.getConfigItems();
for(ServiceConfigurationItem item : configItems)
{
    if((item.getName()!= null && item.getName().equalsIgnoreCase("CO Item")))
    {
        itemData.setResource(custObj);
        itemData.setServiceConfigurationItem(item);
        String reservedFor= null; // "Service-123"
        String reservedForType= null; // "Longterm"
        BaseConfigurationManager bcd =
            PersistenceHelper.makeConfigurationManager
                (ServiceConfigurationVersion.class);
        bcd.assignResource(item, custObj,reservedFor, reservedForType);
        break;
    }
}
```

About Alternate Flows

The generic service fulfillment scenario assigns a custom object resource to a service configuration item. An alternate flow to this scenario may be to unassign the resource from a configuration item.

Additional alternate flows may be to manage consumable resources by creating reservations and conditions. Reservations are created to prevent a given resource to be consumed by another service. The reservation can only be redeemed successfully during resource assignment when the correct token is provided. Also, a reservation can expire if not redeemed within the expiry time period. Conditions are created to add informational or blocking codes to a given resource. A blocking condition prevents a resource from being assigned.

The alternate flows described in this section are:

- Unassigning Resources from a Configuration Item
- Reserving a Custom Object
- Unreserving a Custom Object
- Creating a Blocked Condition for a Custom Object
- Deleting a Blocked Condition for a Custom Object

Unassigning Resources from a Configuration Item

This section describes the UIM API method used to unassign the resource from a configuration item.

Table 3–25 and example code provide information about using the API method.

Table 3–25 Unassigning Resources from a Configuration Item

Topic	Information
Name	BaseConfigurationManager.unallocateInventoryConfigurationItems(java.util.Collection<E> configurationItems)
Description	This method unassigns/deallocates resources that were previously assigned on a configuration item of a service configuration version.
Pre-Condition	A service configuration version exists with a custom object assigned to a configuration item of the version.
Internal Logic	Not applicable
Post-Condition	The custom object/s has been unassigned.
Extensions	Not applicable
Tips	In this example the ConsumableResource to be unassiged is custom object 'CO-1'.

Example 3–17 Unassigning Resources from a Configuration Item

```
BaseConfigurationManager bcd =
    PersistenceHelper.makeConfigurationManager(CustomObject.class);
Finder f = PersistenceHelper.makeFinder();

Collection<CustomObject> custObjs = f.findById(CustomObject.class, "CO-1");
CustomObject custObj = custObjs.iterator().next();
Collection<ServiceConfigurationVersion> scvList =
    f.findByName(ServiceConfigurationVersion.class, "Se_123_2");

ServiceConfigurationVersion scv =
    (ServiceConfigurationVersion)scvList.iterator().next();
BusinessInteractionManager biMgr =
    PersistenceHelper.makeBusinessInteractionManager();
biMgr.switchContext(scv, null);

/* Find Service Configuration Item (SCI) by: */
/*   1) Using Finder query by name, OR */
/*   2) Get Service Configuration and iterate to correct SCI */
//Collection<ServiceConfigurationItem> serviceConfigItems =
//   f.findByName(ServiceConfigurationItem.class, "CO Item");
//ServiceConfigurationItem sci = serviceConfigItems.iterator().next();

ServiceConfigurationItem unSci = null;
```

```
Collection<ServiceConfigurationItem> sciList = scv.getConfigItems();
for (ServiceConfigurationItem sci : sciList)
{
    if (sci.getName().equals("CO Item") &&
    sci.getConfigAction() == ConfigurationItemAction.ASSIGN &&
    sci.getAssignment() != null &&
    sci.getAssignment() instanceof Assignment)
    {
        Assignment assignment = (Assignment) sci.getAssignment();
        if (assignment.getResource().equals(custObj))
        {
            unSci = sci;
            break;
        }
    }
}
if (unSci != null)
{
    Collection<ServiceConfigurationItem> unSciList =
        new ArrayList<ServiceConfigurationItem>();
    unSciList.add(unSci);
    bcd.unallocateInventoryConfigurationItems(unSciList);
}
```

Reserving a Custom Object

This section describes the UIM API methods used to make a reservation and to reserve a custom object using the reservation. To find a custom object to reserve, you must find or create a custom object. See "Finding the Custom Object to Assign" or "Creating the Custom Object to Assign".

Table 3–26, Table 3–27, Table 3–28 and example code provide information about using the API methods.

Table 3–26 Making a Reservation

Topic	Information
Name	ReservationManager.makeReservation(ConsumableResource conRes) In this example, the full signature of the method is included because there are multiple overloaded makeReservation methods.
Description	This method will make an instance of the appropriate Reservation class based on the type of ConsumableResource. For example, if a CustomObject is input, then a CustomObjectReservation will be returned.
Pre-Condition	Not applicable
Internal Logic	This method will determine the appropriate Reservation class to be constructed based on the input ConsumableResource.
Post-Condition	The caller has an instance of the appropriate Reservation class. In this scenario, it will be a CustomObjectReservation.
Extensions	Not applicable
Tips	The CustomObject instance for **CO-1** should be passed as input to the method.

Table 3–27 Reserving a Resource

Topic	Information
Name	ReservationManager.reserveResource(Collection <? extends ConsumableResource> resources, Reservation reservation)
Description	This method will reserve the input resources.
Pre-Condition	The resource exists. In this scenario the resource is Custom Object **CO-1**.
Internal Logic	The input parameters are validated, and if no errors are detected each input resource is reserved. The system will generate a new reservation number. All the input resources will be reserved for this reservation number.
Post-Condition	The resource (Custom Object **CO-1**) is reserved.
Extensions	The RESERVATION_EXPIRATION ruleset can be customized to change the default behavior of setting the expiry date for a resource reservation. By default, a long term reservation will expire after 30 days and a short term reservation will expire after 10 minutes.
Tips	At least one ConsumableResource must be input. For this scenario, it will be the CustomObject instance for **CO-1**. The Reservation passed to the method must have the following attributes set: ■ Reservation.reservedFor (Free form text identifying the reserver.) ■ Resevation.reservedForType (A ReservedForType such as CUSTOMER.) ■ Reservation.reservationType (This would be set to ReservationType.LONGTERM for this scenario.) Optionally, the Reservation.reason can be set. This is free form text. The startDate, endDate, and expiry can also be set, but for this example we will allow them to be defaulted by the system.

You can also add a resource to an existing reservation number by calling the ReservationManager.addResourceToReservation method using this API method:

Table 3–28 Adding a Resource to a Reservation

Topic	Information
Name	ReservationManager.addResourceToReservation(Collection <? extends ConsumableResource> resources, Reservation reservation)
Description	This method will reserve the input resources.
Pre-Condition	The resource exists. In this scenario the resource is Custom Object **CO-1**.
Internal Logic	The input parameters are validated, and if no errors are detected each input resource is reserved. The resources will be reserved with an existing reservation number. The reservedFor and reservedForType values will always be the same for all resource reservations for the same reservation number. Other reservation information, such as reason and expiry, can differ among resource reserved with the same reservation number.
Post-Condition	The resource (Custom Object **CO-1**) is reserved.

Table 3–28 (Cont.) Adding a Resource to a Reservation

Topic	Information
Extensions	The RESERVATION_EXPIRATION ruleset can be customized to change the default behavior of setting the expiry date for a resource reservation. By default, a long term reservation will expire after 30 days and a short term reservation will expire after 10 minutes.
Tips	At least one ConsumableResource must be input. For this scenario, it will be the CustomObject instance for **CO-1**. The Reservation passed to the method must have the following attributes set: ■ Reservation.reservationNumber An existing resource reservation must already exist with this same reservation number. ■ Reservation.reservationType In the generic service fulfillment scenario, this would be set to ReservationType.LONGTERM. If Reservation.reservedForType or Reservation.ReservedFor are populated, they must match the equivalent values for existing resource reservations for the reservationNumber. The startDate, endDate, and expiry can also be set, but for this scenario, these dates are defaulted by the system.

Example 3–18 Reserving a Custom Object

```
ReservationManager resMgr = PersistenceHelper.makeReservationManager();
ConsumableResource cr = (ConsumableResource) custObj;
List<ConsumableResource> crList = new  ArrayList<ConsumableResource>();
crList.add(cr);

Reservation reservation = resMgr.makeReservation(cr);
reservation.setReason("Future reqiurement");
reservation.setReservedFor("Order-333");
reservation.setReservedForType(ReservedForType.ORDER);
reservation.setReservationType(ReservationType.LONGTERM);

resMgr.reserveResource( crList, reservation);

ReservationManager resMgr = PersistenceHelper.makeReservationManager();
ConsumableResource cr = (ConsumableResource) custObj;
List<ConsumableResource> crList = new ArrayList<ConsumableResource>();
crList.add(cr);

Reservation reservation = resMgr.makeReservation(cr);
reservation.setReservationNumber("111111111");
reservation.setReservedFor("Order-333");
reservation.setReservedForType(ReservedForType.ORDER);
reservation.setReservationType(ReservationType.LONGTERM);

resMgr.addResourceToReservation( crList, reservation);
```

Unreserving a Custom Object

This section describes the UIM API methods used to unreserve a custom object. To find the custom object to unreserve, you must find the custom object. See "Finding the Custom Object to Assign".

Table 3–29 and example code provide information about using the API method.

Table 3–29 Unreserving a Custom Object

Topic	Information
Name	ReservationManager.unreserveResource(Collection<? extends ConsumableResource> resources, String redeemer, ReservedForType redeemerType) In this example, the full signature of the method is included because there are multiple overloaded unreserveResource methods.
Description	This method will delete the reservation for the input resources.
Pre-Condition	The resource exists and is reserved.
Internal Logic	The input parameters are validated, and if no errors are detected each input resource is unreserved. The input redeemer and redeemerType must match the persisted reservation information for each of the input resources.
Post-Condition	The resource (custom object **CO-1**) is no longer reserved.
Extensions	Not applicable
Tips	At least one ConsumableResource must be input. For this scenario, it will be the CustomObject instance for **CO-1**. The redeemer and redeemerType are required.

Example 3–19 Unreserving a Custom Object

```
ReservationManager resMgr = InventoryHelper.makeReservationManager();
ConsumableResource cr = (ConsumableResource) custObj;
List<ConsumableResource> crList = new ArrayList<ConsumableResource>();
crList.add(cr);

resMgr.unreserveResource(crList, "Order-333", ReservedForType.ORDER);
```

Creating a Blocked Condition for a Custom Object

This section describes the UIM API methods used to create a blocked condition for a custom object. To find a custom object to create the condition for, you must find or create a custom object. See "Finding the Custom Object to Assign" or "Creating the Custom Object to Assign".

Table 3–30, Table 3–31 and example code provide information about using the API methods.

Table 3–30 Making a Condition

Topic	Information
Name	ConditionManager.makeCondition(ConsumableResource conRes) In this example, the full signature of the method is included because there are multiple overloaded makeCondition methods.
Description	This method will make an instance of the appropriate Condition class based on the type of ConsumableResource. For example, if a CustomObject is input, then a CustomObjectCondition will be returned.
Pre-Condition	Not applicable
Internal Logic	This method will determine the appropriate Condition class to be constructed based on the input ConsumableResource.
Post-Condition	The caller has an instance of the appropriate Condition class. In this scenario, it will be a CustomObjectCondition.

Table 3–30 (Cont.) Making a Condition

Topic	Information
Extensions	Not applicable
Tips	The CustomObject instance for **CO-1** should be passed as input to the method.

Table 3–31 Creating Conditions

Topic	Information
Name	ConditionManager.createConditions
Description	This method will create a condition on each of the input resources.
Pre-Condition	The resource exists. In this scenario the resource is Custom Object **CO-1**.
Internal Logic	The input Condition instances are validated, and if no errors are detected a condition is created for each resource specified in the input Condition collection.
Post-Condition	The resource (custom object **CO-1**) has a blocked condition.
Extensions	Not applicable
Tips	The Condition passed to the method must have the following attributes set: ■ Condition.resource This should be set to the CustomObject instance for **CO-1**. ■ Condition.reason This is free form text describing the reason for the condition. For example, **Under Repair**. ■ Condition.type This should be set to ConditionType.BLOCKED. Optionally, the Condition.validFor can be set with a startDate and endDate value. If startDate is not specified, it is defaulted to the current date. If endDate is not specified, it is defaulted to the java max date value of 18- Jan-2038. Optionally, the Condition.description can be set. This is free form text.

Example 3–20 Creating a Blocked Condition for a Custom Object

```
ConditionManager conMgr = PersistenceHelper.makeConditionManager();
Collection<Condition> inputCons = new ArrayList<Condition>();

Finder f = PersistenceHelper.makeFinder();
Collection<CustomObject> custObjs = f.findById(CustomObject.class, "CO-1");
CustomObject custObj = custObjs.iterator().next();

Condition con = conMgr.makeCondition(custObj);
con.setDescription("Test Failure");
con.setReason("Under Repair");
con.setType(ConditionType.BLOCKED);

Date now = new Date();
Date later = getEndDate(now); /* call to an utility method */
con.setValidFor(new TimePeriod(now, later));
con.setResource(custObj);
con.setMaster(true);
```

```
inputCons.add(con);

Collection <? extends Condition> cons = conMgr.createConditions(inputCons);
```

Deleting a Blocked Condition for a Custom Object

This section describes the UIM API methods used to delete a blocked condition from a custom object. To find the custom object to delete the blocked condition from, you must find the custom object. See "Finding the Custom Object to Assign". To delete the condition from the custom object, you must first find the condition to be deleted using the API method described here.

Table 3–32, Table 3–33, Table 3–34 and example code provide information about using the API methods.

Table 3–32 Making a Condition Search Criteria

Topic	Information
Name	ConditionManager.makeConditionSearchCriteria
Description	This method will make an instance of ConditionSearchCriteria.
Pre-Condition	Not applicable
Internal Logic	Not applicable
Post-Condition	The caller has an instance of ConditionSearchCriteria.
Extensions	Not applicable
Tips	Not applicable

Table 3–33 Finding Conditions

Topic	Information
Name	ConditionManager.findConditions
Description	This method retrieves conditions based on input criteria.
Pre-Condition	The custom object to find conditions for has been selected. The desired condition exists.
Internal Logic	The database is queried for conditions meeting the input criteria. Conditions matching the criteria are returned.
Post-Condition	The desired condition has been retrieved.
Extensions	Not applicable
Tips	In this scenario, the following CriteriaItems could be populated on the ConditionSearchCritiera: ▪ resource The CustomObject instance for **CO-1**. ▪ type ConditionType.BLOCKED If a list of conditions is returned, the list will need to be iterated to select the desired condition to be deleted.

Table 3–34 Deleting Conditions

Topic	Information
Name	ConditionManager.deleteConditions

Table 3–34 (Cont.) Deleting Conditions

Topic	Information
Description	This method will delete conditions on resources.
Pre-Condition	The condition to be deleted has been selected.
Internal Logic	The input Condition instances are validated, and if no errors are detected the conditions are deleted.
Post-Condition	The resource (Custom Object **CO-1**) no longer has the blocked condition.
Extensions	Not applicable
Tips	Not applicable

Example 3–21 Deleting a Blocked Condition from a Custom Object

```
Finder f = PersistenceHelper.makeFinder();
Collection<CustomObject> custObjs = f.findById(CustomObject.class, "CO-1");
CustomObject custObj = custObjs.iterator().next();

ConditionManager conMgr = PersistenceHelper.makeConditionManager();
ConditionSearchCriteria criteria = conMgr.makeConditionSearchCriteria();

CriteriaItem res = criteria.makeCriteriaItem();
res.setValue(custObj);
res.setOperator(CriteriaOperator.EQUALS);
criteria.setResource(res);

CriteriaItem type = criteria.makeCriteriaItem();
type.setValue(ConditionType.BLOCKED);
type.setOperator(CriteriaOperator.EQUALS_IGNORE_CASE);
criteria.setType(type);

Collection <CustomObjectCondition> cons = conMgr.findConditions(criteria);
CustomObjectCondition con = cons.iterator().next();

conMgr.deleteConditions(cons);
```

Setting Characteristic Values for the Service Configuration Item

The following APIs are used to set characteristic values on a service configuration item. The set of allowable characteristic values for a given service configuration item are defined by the service configuration specification used to create the service configuration.

The following shows a configuration item hierarchy that has two characteristic values associated with the Customer Equipment (CE) Router ITEM:

ITEM - Site

- ITEM - Customer Equipment Router
  - Specification - Logical Device
  - Characteristic - Customer
  - Instructions - Characteristics
  - Additional Information

The Configuration ITEMs are used to create the Service Configuration Item instances. Characteristics will be related to the Service Configuration Item. Since Service Configuration Item is a Characteristic Extensible entity, we can use the CharacteristicManager.init API to initialize the set of characteristic values on the entity. In the example above, the two Characteristics under the Customer Equipment Router ITEM would create two instances on the ServiceConfigurationItemCharacteristic, and if there is default values defined, it is also copied.

Table 3–35 and example code provide information about using the API method.

Table 3–35 Setting Characteristic Values for the Service Configuration Item

Topic	Information
Name	CharacteristicManager.init(CharacteristicExtensible<CharValue> characteristicExtensible, Specification spec)
Description	This method initializes the CharacteristicExtensible entity. In this case, the ServiceConfigurationItem). It sets the default value for each characteristic which has one.
Pre-Condition	A service configuration item exists and the InventoryConfigurationSpec is known.
Internal Logic	The InventoryConfigurationSpec is used to get the CharacteristicSpecUsage, from the CharacteristicSpecUsage to get the CharacteristicSpecification, so that the default spec value can be retrieved and set to the CharValue. And the Charvalue will be set to the Service configuration item.
Post-Condition	ServiceConfigurationItem has the default characteristics set.
Extensions	Not applicable
Tips	Not applicable

Note: When creating a Service Configuration Item, call CharacteristicManager.init (CharacteristicExtensible<CharValue> characteristicExtensible, Specification spec) method to initiate the default characteristics value.

Example 3–22 Setting Characteristic Values for the Service Configuration Item

```
CharacteristicManager characteristicManager =
    PersistenceHelper.makeCharacteristicManager();

// Initialize the characteristics to the item
characteristicManager.init((CharacteristicExtensible)childConfigItem,
    inventoryConfigurationSpec);

// Get the characteristics from service config item
HashSet<CharValue> characteristics = serviceConfigItem.getCharacteristics();

// Loop through the HashSet of characteristics and set the value as defined
for (CharValue charValue : characteristics)
{
    charValue.setValue("myValue");
    charValue.setLabel("myLabel");
}
```

About Alternate Flows

The generic service fulfillment scenario sets characteristic values for the service configuration item. An alternate flow to this scenario may be to unset characteristic values from the service configuration item.

The alternate flow described in this section is "Unsetting Characteristic Values for the Service Configuration Item".

Unsetting Characteristic Values for the Service Configuration Item

The following API is to unset characteristic values on a service configuration.

The following example code provides information about using the API method.

> **Note:** From ServiceConfigurationItem, get the characteristics and then delete the ServiceConfigurationItemCharacteristics to remove the characteristic values. If only one particular characteristic needs to be deleted for the ServiceConfigurationItem, then a name match should be compared before deleting the ServiceConfigurationItemCharacteristic.

Example 3–23 Unsetting Characteristic Values for the Service Configuration

```
HashSet<ServiceConfigurationItemCharacteristic> characteristics =
    serviceConfigItem.getCharacteristics();

Iterator<ServiceConfigurationItemCharacteristic> itr =
    characteristics.iterator();

while (itr.hasNext())
{
    ServiceConfigurationItemCharacteristic characteristic = itr.next();
    if characteristic.getName().equals("myName")
        itr.remove();
}
```

Transitioning the Lifecycle Status

The transition APIs are used for transitioning the lifecycle status of a given entity which implements the LifeCycleManaged interface. The state transition rules are defined in the ***-transitions.xml** files.

Table 3–36 and example code provide information about using the API method.

Table 3–36 Transitioning the Lifecycle Status

Topic	Information
Name	TransitionManager.transition
Description	Transitions a LifeCycleManaged entity by finding the matching transition definition which has the business action defined and the object activity defined the same as the input parameters, and which **from** business state matches the entity's business state.
Pre-Condition	TransitionManager.isValidTransition has successfully validated that the specified business action can trigger the transition of either the business state or the object state.

Table 3–36 (Cont.) Transitioning the Lifecycle Status

Topic	Information
Internal Logic	Finds a matching transition definition. For a version object it matches on business action and object activity only. Other objects are matched from most specific to least specific in the following order: **1.** Match businessAction, objectActivity, entity type, and the specification. **2.** Match businessAction, objectActivity, entity type. **3.** Match businessAction, objectActivity. Switches to a Business Interaction context if applicable and updates the business or object state of the object and its dependents based on the transition definition.
Post-Condition	The object state or business state is updated.
Extensions	BusinessInteractionSpec_TransitionManager_ validateBusinessStateTransitions BusinessInteractionSpec_TransitionManager_ validateObjectStateTransitions
Tips	See *UIM Developer's Guide* for more information.

Example 3–24 Transitioning the Lifecycle Status

```
TransitionManager transitionManager =
    PersistenceHelper.makeTransitionManager(service);

boolean success = false;
success = transitionManager.transition(service, ServiceAction.COMPLETE);
```

4

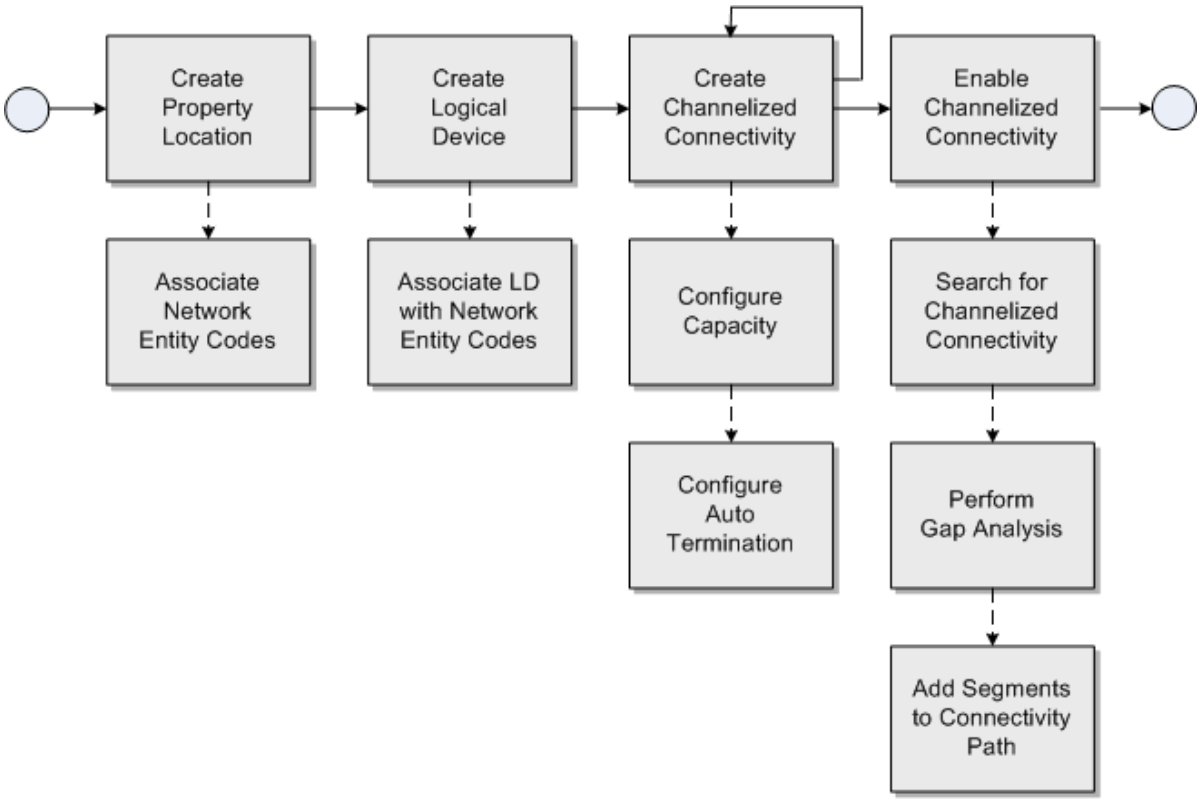
Implementing a Channelized Connectivity Enablement Scenario

This chapter describes implementing a channelized connectivity enablement scenario using various Oracle Communications Unified Inventory Management (UIM) application program interfaces (APIs). You can use this information to gain a better understanding of how the UIM APIs can be used to implement any channelized connectivity enablement scenario.

About the Channelized Connectivity Enablement Scenario

Figure 4–1 shows the process flow for a channelized connectivity enablement scenario:

Figure 4–1 Process Flow for a Generic Channelized Connectivity Scenario



This process flow begins with creating a property location and associating network entity codes with the property location. The network entity codes are used in subsequent steps in the process flow, such as associating them with logical devices.

The process flow continues with creating logical devices with device interfaces that can terminate on the bearer channelized connectivity, and associating logical devices with the network entity codes previously created. This involves creating logical device search criteria to find the required logical device specification.

Next is creating channelized connectivity, which represents bearer channelized connectivity between two network entity codes that define attributes of technology, rate code, and channelized connectivity function.

The process flow continues by configuring the capacity for the channelized connectivity to channelize it, and by optionally terminating them on the device interfaces of logical devices previously created. This is called auto termination of device interfaces because it also terminates the sub-device interfaces down the hierarchy to the channels when the channelized connectivity is terminated automatically. This represents the bearer channelized connectivity that will be used in enablement in subsequent steps of the process flow.

The process flow continues with creating channelized connectivity to represent the rider between two network entity codes that define attributes of technology, rate code, and channelized connectivity function. For a channelized connectivity entity to be enabled by a channel, its rate code must match or be compatible with the rate code of the channel.

Next is enabling channelized connectivity, which can be manually done by searching for and adding the bearer channelized connectivity's channel. This involves creating channelized connectivity search criteria to search for the bearer channelized connectivity and selecting the appropriate channel. Enablement can also be done by adding bearer channelized connectivity through gap analysis to the rider that involves creating path analysis criteria to search for the bearer channelized connectivity between a source/intermediate/target property locations or logical devices.

Now that you have a high-level understanding of the channelized connectivity enablement scenario process flow, each part of the process flow is further described in the following sections. Each section includes information about the specific UIM APIs used to perform each step. Example code is also included for each step.

- Creating a Property Location and Associating Network Entity Codes
- Creating a Logical Device and Associating LD Interfaces with Network Entity Codes
- Creating Channelized Connectivity
- Enabling Channelized Connectivity

Creating a Property Location and Associating Network Entity Codes

This section describes the UIM API methods used to create a property location and to associate network entity codes with the property location.

Table 4–1, Table 4–2, and example code provide information about using the API methods to create a property location and to associate network entity codes to the property location.

Table 4–1 Creating a Property Location

Topic	Information
Name	LocationManager.createPropertyLocation (Collection<PropertyLocation> locations)
Description	Creates the Property Location instances with the given inputs. User has to specify one mandatory Primary address as input with which a property Location has to be created. Every property location also has a property address associated with it.
Pre-Condition	The **locations** parameter needs to be prepared with necessary attributes
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	■ The same method is also used to create Network Location when the Network Location code is populated in the input. As part of creation of Network location, the same method also enables users to create Network entity codes corresponding to the Network Location. ■ The Location Identifier which is a concatenated Address format is used to uniquely identify the Property Location. ■ If horizontal/vertical coordinates are given as inputs, the latitude/longitude coordinates are automatically populated for the created Property Location and vice versa.

Table 4–2 Associating Network Entity Codes with a Property Location

Topic	Information
Name	LocationManager.associateNetworkEntityCodeToNetworkLocation (List<NetworkEntityCode> entitycodes, PropertyLocation location)
Description	This method is called during the association or creation of the network entity code in the context of property location.
Pre-Condition	The **location** parameter already exists.
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	■ Check if the network entity code is unique. ■ Check for the length of the network entity code.

Example 4–1 Creating a Property Location and Associating Network Entity Codes with the Property Location

```
Finder finder = PersistenceHelper.makeFinder();
PropertyLocation propertyLocation = locationManager.makePropertyLocation();
PropertyAddress propertyAddress = locationManager.makePropertyAddress();
LocationManager locationManager = PersistenceHelper.makeLocationManager();

//Set all necessary attributes needed for Property Address and Property Location
propertyAddress.setStreetAddress((String)paramMap.get("streetAddress"));
propertyAddress.setCity((String)paramMap.get("city"));
propertyAddress.setState((String)paramMap.get("state"));
propertyAddress.setCountry((String)paramMap.get("country"));
```

```
propertyAddress.setIsValidated(Boolean.valueOf
    ((String)paramMap.get("isValidated")));
propertyAddress.setIsNonValidatedAddressAccepted(true);
propertyAddress.setIsPrimaryAddress(true);
Set<PropertyAddress> addressSet = new HashSet<PropertyAddress>(1);
addressSet.add(propertyAddress);
propertyLocation.setPropertyAddresses(addressSet);
propertyLocation.setNetworkLocationCode("PLANO");
propertyLocation.setLatitude("34");
propertyLocation.setLongitude("54");

Collection<PropertyLocation> list = new ArrayList<PropertyLocation>(1);
list.add(propertyLocation);
List<PropertyLocation> propLocobjects =
    locationManager.createPropertyLocation(list);
networkLocation = propLocobjects.get(0);
List<NetworkEntityCode> networkEntityCodes = new ArrayList<NetworkEntityCode>();
NetworkEntityCode nec = locationManager.makeNetworkEntityCode();
nec.setName(necStr);
networkEntityCodes.add(nec);
if (!Utils.isEmpty(networkEntityCodes))
{
    locationManager.associateNetworkEntityCodeToNetworkLocation
        (networkEntityCodes,networkLocation);
}
```

Creating a Logical Device and Associating LD Interfaces with Network Entity Codes

This section describes the UIM API methods used to create a logical device with default logical device interfaces, and to associate the logical device interfaces with the previously created network entity codes.

Table 4–3 and example code provide information about using the API method to create a logical device with default logical device interfaces.

Table 4–3 Creating a Logical Device

Topic	Information
Name	LogicalDeviceManager.createLogicalDevice (Collection<LogicalDevice> logicalDevices)
Description	Creates logical device entities and their provided device interfaces and sub-device interfaces based on the specification.
Pre-Condition	Logical device specification with device interfaces is defined and exists already.
Internal Logic	Device interfaces can also provide other device interfaces. The number of device interfaces to be created will be determined by the minimum value defined in the specification relationships. The input logical device entities should be sparsely populated with the specification, hard attributes and characteristics. The provided device interfaces will be derived based on the specification. Characteristics will be defaulted based on the specification. The id of the device interfaces will be generated. If required characteristics exist for a provided device interface that are not defaulted, then the logical device will still be created.
Post-Condition	Not applicable

Table 4–3 (Cont.) Creating a Logical Device

Topic	Information
Extensions	Not applicable
Tips	Not applicable

Example 4–2 Creating a Logical Device with Default Logical Device Interfaces

```
Finder finder = PersistenceHelper.makeFinder();
LogicalDeviceManager ldMgr = PersistenceHelper.makeLogicalDeviceManager();

Collection<Specification> specs =
    finder.findByName(Specification.class,"ldSpecName");

LogicalDeviceSpecification ldSpec =
    (LogicalDeviceSpecification)specs.iterator().next();

LogicalDevice ld = ldMgr.makeLogicalDevice();
ld.setName("ldName");
ld.setId("ldId");
ld.setSpecification(ldSpec);
List<LogicalDevice> ldList = new ArrayList<LogicalDevice>();
ldList.add(ld);
ldMgr.createLogicalDevice(ldList);
```

The following table and example code provide information about using the API method to associate a logical device with a network entity code.

Table 4–4 Associating a Logical Device with a Network Entity Code

Topic	Information
Name	LogicalDeviceManager.updateLogicalDevice (Collection<LogicalDevice> logicalDevices)
Description	This method is intended to update the hard attributes and characteristics of a logical device.
Pre-Condition	Logical device exists already. The location of a logical device can only be changed if it does not have any active consumers or interconnections on the logical device or any of its device interfaces.
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–3 Associating a Logical Device with a Network Entity Code

```
Finder finder = PersistenceHelper.makeFinder();
LogicalDeviceManager ldMgr = PersistenceHelper.makeLogicalDeviceManager();
LocationManager locationManager = PersistenceHelper.makeLocationManager();

// find an existing logical device
LogicalDevice ld = finder.findById(LogicalDevice.class, "ldId").iterator().next();

// find an existing property location that has network entity code
PropertyLocation pls =
(PropertyLocation)locationManager.findNetworkEntityLocation("PLANO");
```

```
ld.setPropertyLocation(pls);

NetworkEntityCodeSearchCriteria criteria =
    locationManager.makeNetworkEntityCodeSearchCriteria();
criteria.setPropertyLocation(pls);

//find network entity code matching "001"
List<NetworkEntityCode> networkEntityCodes =
    locationManager.findNetworkEntityCodes(criteria);
NetworkEntityCode networkEntCd = null;

if (!Utils.isEmpty(networkEntityCodes))
{
    String networkEntityCod= "001";
    for (NetworkEntityCode nec : networkEntityCodes)
    {
        if ((pls.getNetworkLocationCode() + "." + networkEntityCode).equals
        nec.getNetworkLocationEntityCode()))
        {
            networkEntCd = nec;
        }
    }
}
ld.setNetworkEntityCode(networkEntCd);
networkEntCd.setLogicalDevice(ld);
List<LogicalDevice> ldList = new ArrayList<LogicalDevice>();
ldList.add(ld);
ldMgr.updateLogicalDevice(ldList);
```

Creating Channelized Connectivity

This section describes the UIM API methods used to:

- Create Channelized Connectivity
- Configure Capacity on the Channelized Connectivity
- Configure Auto Termination on the Channelized Connectivity

Create Channelized Connectivity

Table 4–5 and example code provide information about using the API method to create channelized connectivity. (You use the same API method to create the bearer channelized connectivity and the rider channelized connectivity.)

Table 4–5 Creating Channelized Connectivity

Topic	Information
Name	ConnectivityManager.createConnectivity(N connectivity, String aNetworkLocationEntityCode, String zNetworkLocationEntityCode, int quantity, boolean contiguousSerialAllocation)
Description	This method will create channelized connectivity. Valid A Location and Z Location must be set on the channelized connectivity instance.
Pre-Condition	Two property locations to represent A and Z side of the channelized connectivity already exists. ora_uim_basetechnologies is already installed.
Internal Logic	Not applicable

Table 4–5 (Cont.) Creating Channelized Connectivity

Topic	Information
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–4 Creating Channelized Connectivity

```
String rateCode = "STM1;
String function = "SM01";
String aLocation = "DALLAS";
String zLocation = "PLANO";
String aEntityCode = "DALLAS.001";
String zEntityCode = "PLANO.001";

int qtyInt = 1;
boolean isContiguos = "true";

TDMConnectivityManager manager =
    (TDMConnectivityManager)PersistenceHelper.makeConnectivityManager
        (TDMConnectivity.class);

Finder finder = PersistenceHelper.makeFinder();

NetworkConnectivity c = manager.makeTDMFacility();
NetworkConnectivity nc = (NetworkConnectivity)c;

String technology =
    finder.findByName(Technology.class, "SDH").iterator().next();
nc.setTechnology(technology);
finder.reset();

String rateCode =
    finder.findByName(RateCode.class, "STM1").iterator().next();
nc.setRateCode(rateCode);
finder.reset();

String function =
    finder.findByName(ConnectivityFunction.class,"SM01").iterator().next();

nc.setConnectivityFunction(function);
String aLocationCode = aLocation;
if(!Utils.isEmpty(aEntityCode)){
    aLocationCode = aLocation+"."+aEntityCode;}

String zLocationCode = zLocation;
if(!Utils.isEmpty(zEntityCode)){
    zLocationCode = zLocation+"."+zEntityCode;}

int tempQty = qtyInt;
while(tempQty >0)
{
    if(tempQty > 99){
        qtyInt = 99;}
    else{
        qtyInt = tempQty;}

    Collection<TDMConnectivity> createdConnectivities =
```

```
        manager.createConnectivity(c, aLocationCode, zLocationCode,
            qtyInt, isContiguos);
}
```

Configure Capacity on the Channelized Connectivity

Table 4–6 and example code provide information about using the API method to configure capacity on the channelized connectivity.

Table 4–6 Configuring Capacity on the Channelized Connectivity

Topic	Information
Name	SignalTerminationPointManager.applyCapacityConfiguration (MultiplexedFacility connectivity, List<RateCode> orderedRateCodes, String signalAddress)
Description	This method configures a connectivity to the required rate code level and also creates channels at those levels.
Pre-Condition	Not applicable
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Also call TDMConnectivityManager.createAndAutoTerminateChannels(M multiplexedFacility, boolean doValidation) to ensure terminations are also adjusted accordingly.

Example 4–5 Configuring Capacity on the Channelized Connectivity

```
Finder finder = PersistenceHelper.makeFinder();

String connectivityIdentifier = "ALLNTXC01 / FRSCTXC01 / STM1 / SM01 / 1";
String sourceRateCode = "OM80";
String destinitionRateCode = "OM32";

RateCode sourceRC =
    finder.findByName(RateCode.class, sourceRateCode).iterator().next();

RateCode destinitionRC =
    finder.findByName(RateCode.class, destinitionRateCode).iterator().next();

TDMConnectivityManager mgr =
    (TDMConnectivityManager)PersistenceHelper.makeConnectivityManager
        (TDMFacility.class);

TDMConnectivitySearchCriteria criteria = mgr.makeTDMSearchCriteria();
CriteriaItem item = criteria.makeCriteriaItem();
item.setName("connectivityIdentifier");
item.setValue("connectivityIdentifier);
item.setOperator(CriteriaOperator.EQUALS);
criteria.setConnectivityIdentifier(item);
TDMFacility tdm = mgr.findTDMConnectivities(criteria).iterator().next();

SignalTerminationPointManager stpMgr =
    PersistenceHelper.makeSignalTerminationPointManager();

List<RateCode> orderedRateCodes = new ArrayList<RateCode>();
if (sourceRC != null){
```

```
    orderedRateCodes.add(sourceRC);}
if (destinitionRC != null){
    orderedRateCodes.add(destinitionRC);}

stpMgr.applyCapacityConfiguration(tdm, orderedRateCodes, "");
mgr.createAndAutoTerminateChannels(tdm, true);
```

Configure Auto Termination on the Channelized Connectivity

Table 4–7 and example code provide information about using the API method to configure auto-termination on the channelized connectivity.

Table 4–7 Auto-terminating the Channelized Connectivity

Topic	Information
Name	ConnectivityManager.assignDeviceInterface(E connectivity, DeviceInterface di, ConnectivityEndpoint endpoint)
Description	This method terminates the channelized connectivity with the device interface at the given end point. Also auto-terminates the channels on the sub-device interfaces.
Pre-Condition	Ensure the capacity is configured at the required level on the channelized connectivity and the sub-device interfaces are created beforehand until that level.
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–6 Auto-Terminating the Channelized Connectivity

```
Finder finder = PersistenceHelper.makeFinder();
String tdmName = "DS3_TDM_Tail";
String diId = "DS3-1-1";
ConnectivityEndpoint endPoint = ConnectivityEndpoint.A_ENDPOINT;

DeviceInterface di =
    finder.findById(DeviceInterface.class, diId).iterator().next();
finder.reset();

TDMFacility tdm =
    finder.findByName(TDMFacility.class, tdmName).iterator().next();

TDMConnectivityManager manager = (TDMConnectivityManager)
    PersistenceHelper.makeConnectivityManager(TDMConnectivity.class);

tdm = (TDMFacility) manager.assignDeviceInterface(tdm, di, endPoint);
```

Enabling Channelized Connectivity

This section describes the UIM API methods used to enable channelized connectivity by:

- Manually Enabling Channelized Connectivity
- Performing Gap Analysis
- Adding Segments To Connectivity Path Based on the Gap Analysis Results

Manually Enabling Channelized Connectivity

Table 4–8 and example code provide information about using the API method to manually enable channelized connectivity by manually searching for the channelized connectivity and adding segments to the connectivity path.

Table 4–8 Manually Enabling Channelized Connectivity

Topic	Information
Name	ConnectivityManager.addSegmentsToConnectivityPath(E connectivityTrail, PipeConfigurationItem connectivityPath, PipeConfigurationItem gapItem, List<Pipe> bearerList) throws ValidationException
Description	The **connectivityTrail** parameter is the channelized connectivity that will be enabled. The **connectivityPath** parameter is the PipeConfigurationItem of the path. The **gapItem** parameter is the PipeConfigurationItem of the gap that will be resolved. The **bearerList** parameter contains other connectivities to be added for enablement. See *Oracle Communications Information Model Reference* for information on PipeConfigurationItem.
Pre-Condition	Not applicable
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–7 Manually Enabling Channelized Connectivity by Searching for the Connectivity and Adding Segments to the Connectivity Path

```
String trailName = "EDINBURGH.002 / LONDON.001 / VC12 / VC12 / 1";

//We want to add connectivities to first path
int pathIndex = "0";

//Assuming there are other connectivities already added to this path
int gapIndex = "0";

PersistenceHelper.makeBusinessInteractionManager().switchContext
    ((String)null, null);

Finder finder = PersistenceHelper.makeFinder();

Connectivity connectivityTrail =
    finder.findByName(Connectivity.class, trailName).iterator().next();

List<String> bearers = new ArrayList<String>();
bearers.add("EDINBURGH.001 / EDINBURGH.002 / STM4 / SM04 / 139 / 1-1-1-2");
bearers.add("EDINBURGH.001 / MACHESTER.001 / STM4 / SM04 / 139 / 1-1-1-2");
bearers.add("LONDON.001 / MACHESTER.001 / STM4 / SM04 / 139 / 1-1-1-2");

List<Pipe> bearerList = new ArrayList<Pipe>(bearers.size());
for (String bearerName : bearers)
{
```

```
    finder.reset();
    Pipe connectivity = finder.findByName
        (TDMFacility.class, bearerName).iterator().next();
    bearerList.add(connectivity);
}

PipeConfigurationVersion designVersion =
    ConnectivityUtils.getInProgressDesignVersion((Pipe)connectivityTrail);

List<PipeConfigurationItem> allPaths =
    PipeHelper.getAllTransportItems(designVersion);

PipeConfigurationItem connectivityPath = allPaths.get(pathIndex);

PipeConfigurationItem gapItem =
    connectivityPath.getChildConfigItems().get(gapIndex);

ConnectivityManager manager = PersistenceHelper.makeConnectivityManager();
manager.addSegmentsToConnectivityPath
    (connectivityTrail, connectivityPath, gapItem, bearerList);
```

Performing Gap Analysis

Table 4–9 and example code provide information about using the API method to perform gap analysis.

Table 4–9 Performing Gap Analysis

Topic	Information
Name	List<PathResultSet> findPaths(PipeSpecification enabledPipe, PathAnalysisCriteria criteria) throws ValidationException
Description	The **enabledPipe** parameter is the channelized connectivity to be enabled. The **criteria** parameter is used in performing gap analysis.
Pre-Condition	Ensure the channelize connectivities that you are expecting the results are already created, terminated, and their capacity is configured.
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–8 Performing Gap Analysis

```
String sourceLocationCode = "EDINBURGH.002";
String intermediateLocationCode = "MACHESTER.001";
String targetLocationCode = "LONDON.001";
String rateCodeName = "VC12";

LocationManager locationManager =
    PersistenceHelper.makeLocationManager();

TopologyObject sourceNode =
    (TopologyObject)locationManager.findNetworkEntityLocation(sourceLocationCode);

TopologyObject targetNode =
    (TopologyObject)locationManager.findNetworkEntityLocation(targetLocationCode);
```

```
TopologyObject intermediateNode = null;
if(!Utils.isEmpty(intermediateLocationCode)){
    intermediateNode =
        (TopologyObject)locationManager.findNetworkEntityLocation
            (intermediateLocationCode);
}
if(sourceNode == null || targetNode == null ||
(!Utils.isEmpty(intermediateLocationCode) && intermediateNode == null)){
    throw new IllegalArgumentException("Invalid source/intermediate/target");
}
RateCode rateCode = null;
CapacityManager capacityManager = PersistenceHelper.makeCapacityManager();
RateCodeSearchCriteria rateCodeSC = capacityManager.makeRateCodeSearchCriteria();

CriteriaItem rateCodeNameItem = rateCodeSC.makeCriteriaItem();
rateCodeNameItem.setName(rateCodeName);
rateCodeNameItem.setOperator(CriteriaOperator.EQUALS);
rateCodeNameItem.setValue(rateCodeName);
rateCodeSC.setName(rateCodeNameItem);

List<RateCode> rateCodes = capacityManager.findRateCode(rateCodeSC);
if (!Utils.isEmpty(rateCodes)) {
    rateCode = rateCodes.get(0);
}
if(rateCode == null){
    throw new IllegalArgumentException("Invalid rateCode");
}
PathAnalysisCriteria criteria = new PathAnalysisCriteria();
criteria.setSourceNode(sourceNode);
criteria.setIntermediateNode(intermediateNode);
criteria.setTargetNode(targetNode);
criteria.setRateCode(rateCode);
criteria.setGapAnalysis(true);

PathAnalysisManager pathAnalysisManager =
    PersistenceHelper.makePathAnalysisManager();

List<PathResultSet> paths = pathAnalysisManager.findPaths(criteria);
```

Adding Segments To Connectivity Path Based on the Gap Analysis Results

Table 4–10 and example code provide information about using the API method to add segments to the connectivity path based on the gap analysis results.

Table 4–10 Adding Segments to Connectivity Path Based on Gap Analysis Results

Topic	Information
Name	ConnectivityManager.addSegmentsToConnectivityPath (E connectivityTrail, PipeConfigurationItem connectivityPath, PipeConfigurationItem gapItem, PathResultSet path) throws ValidationException;

Table 4–10 (Cont.) Adding Segments to Connectivity Path Based on Gap Analysis

Topic	Information
Description	The **connectivityTrail** parameter is the channelized connectivity that will be enabled. The **connectivityPath** parameter is the PipeConfigurationItem representing the path to which the segments have to be added. The **gapItem** parameter is the PipeConfigurationItem of the gap that will be resolved. The **path** parameter is the results returned from gap analysis. (You can pass the results retrieved in the previous example. For example, paths.get(0)). See *Oracle Communications Information Model Reference* for information on PipeConfigurationItem.
Pre-Condition	Not applicable
Internal Logic	Not applicable
Post-Condition	Not applicable
Extensions	Not applicable
Tips	Not applicable

Example 4–9 Adding Segments to Connectivity Path Based on Gap Analysis Results

```
String trailName = "EDINBURGH.002 / LONDON.001 / VC12 / VC12 / 1";

//We want to add connectivities to first path
int pathIndex = "0";

//Assuming there are other connectivities already added to this path
int gapIndex = "0";

PersistenceHelper.makeBusinessInteractionManager().switchContext
    ((String)null, null);

Finder finder = PersistenceHelper.makeFinder();

Connectivity connectivityTrail =
    finder.findByName(Connectivity.class, trailName).iterator().next();

PipeConfigurationVersion designVersion =
    ConnectivityUtils.getInProgressDesignVersion((Pipe)connectivityTrail);

List<PipeConfigurationItem> allPaths =
    PipeHelper.getAllTransportItems(designVersion);

PipeConfigurationItem connectivityPath = allPaths.get(pathIndex);

PipeConfigurationItem gapItem =
    connectivityPath.getChildConfigItems().get(gapIndex);

ConnectivityManager manager = PersistenceHelper.makeConnectivityManager();

/*Here paths are the path returned by gap analysis.
Assuming the first one is the list is selected*/
manager.addSegmentsToConnectivityPath
    (connectivityTrail, connectivityPath, gapItem, paths.get(0));
```

A

UIM Entity Managers

This appendix provides a listing of Oracle Communications Unified Inventory Management (UIM) entity manager class names, the package in which they reside, the entities they manage, and a brief description.

These Java manager classes are found in the **uim_managers.jar** which is located in the UIM Software Development Kit (SDK). See *UIM Developer's Guide* for more information on the UIM SDK.

Note: The package references in Table A–1 assume the package prefix of **oracle.communications.inventory.api**.

Table A–1 List of UIM Entity Managers

Manager Name	Package	Managed Entities	Description
ActivityManager	project.activity	Activity ActivityItem Project	Defines the methods for managing Activity entities within a Project along with their ActivityItem entities.
AddressRangeManager	place	GeographicAddress	Defines a GeographicAddress being used as a range.
AssignmentManager	consumer	Assignment	Extends ConsumerManager, managing Assignment logic. Assignment such as PipeAssignment, EquipmentAssignment.
AttachmentManager	common	Involvement	Administers Attachments and Involvements, for example preconfiguring TelephoneNumber with LogicalDeviceAccount.
BaseInvManager	common	<Base Class>	Provides application-specific behavior to methods in the JdoBean. The JdoBean doesn't know about entities that are specific to the inventory application.
BOMManager	bom	Activity Inventory	Defines the methods to support retrieving Bill of Materials information as well as populating additional information on an activity or resource.

Table A–1 (Cont.) List of UIM Entity Managers

Manager Name	Package	Managed Entities	Description
BusinessInteractionManager	businessinteraction	BusinessInteraction	Defines methods for managing Business Interactions.
CapacityManager	capacity	Capacity	Defines the methods for managing capacity such as PipeCapacityProvided, PipeCapacityRequired, PipeCapacityConsumption.
CharacteristicManager	characteristic	Characteristics	Defines the methods for managing Characteristics such as CharacteristicSpecUsage, CharacteristicSpecValue, CharacteristicSpecValueUsage.
ConditionManager	consumer	Condition	Extends InventoryManager, managing Condition logic. Condition such as PipeCondition, EquipmentCondition.
ConfigurationManager	configuration	Configuration	Administers a configuration and its subtypes such as ServiceConfiguration, PlaceConfiguration.
ConnectivityManager	connectivity	Connectivity Pipe DeviceInterface InterConnection CrossConnect	Defines the methods for managing the creation, updates, deletions, and retrieving of connectivity data. This manager references a large number of different entities so the primary entities are listed here as the managed entities.
ConsumerManager	consumer	Assignment Condition Reservation	Validates resource availability.
CustomNetworkAddressManager	custom	CustomNetworkAddress	Defines the methods for managing CustomNetworkAddress objects.
CustomObjectManager	custom	CustomObject	Defines the methods for managing CustomObject objects.
EquipmentManager	equipment	Equipment EquipmentHolder PhysicalPort PhysicalConnector PhysicalDevice	Defines the methods for managing equipment and provided equipment holders, physical ports and physical connectors of the equipment. This interface also defines the methods for maintaining and finding physical devices and provided physical ports and physical connectors of the physical devices.
FlowIdentifierManager	networkaddress	FlowIdentifier InventoryGroup	Defines the methods for managing flow identifiers and relating them to inventory groups.
InventoryBaseManager	inventory	InventoryConfigurationItem	Gets and validates inventory configuration item for configuration.
InventoryGroupManager	group	InventoryGroup InvGroupRef	Defines the methods for managing inventory groups and related entities.

Table A–1 (Cont.) List of UIM Entity Managers

Manager Name	Package	Managed Entities	Description
IPAddressManager	ip	IPAddress NetworkAddressDomain	Defines the methods for managing IP Addresses.
IPNetworkManager	ip	IPSubnet IPAddress NetworkAddressDomain	Defines the methods for creating, deleting, finding, and updating IP network objects.
LocationManager	location	PropertyLocation PropertyAddress NetworkEntityCode	Defines the methods for managing behaviors of property locations.
LogicalDeviceManager	logicaldevice	LogicalDevice DeviceInterface FlowInterface	Defines the methods for managing LogicalDevice, Device Interface, and Flow Interface objects.
LogicalDeviceAccountManager	logicaldevice.account	LogicalDeviceAccount	Defines the methods for managing LogicalDeviceAccount objects.
LogicalPhysicalResourceBase	resource		Contains shared methods and variables for managing logical and physical resources.
MediaManager	media	Media	Defines the methods for managing Media objects. Most of the methods for creating, updating, and deleting Media objects are deprecated because the functionality was replaced in Design Studio.
MediaResourceManager	mediaresource	MediaStream MediaResourceLogicalDeviceRel	Defines the methods for managing MediaStream objects and its relationships to LogicalDevice objects. MediaStream is also a MediaResource which is an abstract entity for various types of media.
MultiplexedConnectivityManager	connectivity	MultiplexedConnectivity MultiplexedChannel MultiplexedFacility	Defines the methods for managing MultiplexedConnectivity objects as well as creating and retrieving channels for a facility. This interface also creates and removes terminations for a facility.
NetworkAddressBlockManager	networkaddress	NetworkAddressBlock	Defines methods for managing NetworkAddressBlock objects.
NetworkAddressDomainManager	networkaddress	NetworkAddressDomain NetworkAddressType	Defines methods for managing NetworkAddressDomain objects.
NetworkManager	network	Network NetworkNode NetworkEdge	Defines methods for managing Network, NetworkNode, and NetworkEdge objects.
NetworkReconfigurationActivityManager	project.activity	Network NetworkNode NetworkEdge	Defines methods for managing Network, NetworkNode, and NetworkEdge objects, and their relationships to Activities.

Table A–1 (Cont.) List of UIM Entity Managers

Manager Name	Package	Managed Entities	Description
NetworkReconfiguration Manager	network	Network NetworkNode NetworkEdge	Defines methods for managing Network, NetworkNode, and NetworkEdge objects for Network Configuration scenarios.
PacketConnectivityManager	connectivity	NetworkConnectivity	Defines the methods for creating Packet Network Connectivity objects.
PartyManager	party	Party	Defines the methods for managing Party objects.
PathAnalysisManager	topology	TopologyEdge ToplogyNode	Defines the methods for finding paths of interconnected TopologyEdge and TopologyNode objects.
PipeConfigurationManager	connectivity	PipeConfigurationVersion PipeConfigurationItem Pipe PipeTerminationPoint	Defines the methods for managing Pipe Configurations and their related entities.
PipeManager	connectivity	Pipe PipeTerminationPoint	Defines the methods for managing Pipe and PipeTerminationPoint objects.
PlaceConfigurationManager	place	PlaceConfiguration	Defines the methods for managing PlaceConfiguration objects.
PlaceManager	place	GeographicPlace GeographicAddress GeographicLocation GeographicSite	Defines the methods for maintaining GeographicPlace objects and their concrete subclasses.
ProductManager	product	Product	Defines the methods for managing Product objects.
ProjectManager	project	Project	Defines the methods for managing Project objects.
ReservationManager	consumer	Reservation	Extends ConsumerManager, managing Reservation logic. Reservation such as PipeReservation, EquipmentReservation.
RoleManager	role	Role	Defines methods for managing Role objects.
SecurityManager	admin	User Role Partition SecurityPolicy	Defines the methods for managing User, Role, Partition, and SecurityPolicy objects.
ServiceConfigurationManager	service	ServiceConfigurationVersion ServiceConfigurationItem	This manager is used to configure a service using configuration versions and items.

Table A–1 (Cont.) List of UIM Entity Managers

Manager Name	Package	Managed Entities	Description
ServiceConnectivityManager	connectivity	ServiceConnectivity ServiceNetwork ServiceConfigurationVersion	This manager is used to create service connectivity objects with and without a ServiceConfigurationVersion.
ServiceManager	service	Service	Defines the methods for managing Service objects.
SignalTerminationPointManager	signalterminationpoint	SignalTerminationPoint TrailTerminationPoint ConnectionTerminationPoint	Defines methods for managing Signal Structure and SignalTerminationPoint.
SpecManager	specification	Specification	Administers a specification and its subtypes such as PipeSpecification, EquipmentSpecification.
TagManager	tag	Tag	Defines the methods for managing Tag objects.
TDMConnectivityManager	connectivity	TDMChannel TDMFacility	Defines the methods for managing TDMChannel and TDMFacility objects.
TelephoneNumberManager	number	TelephoneNumber	Defines the methods for managing TelephoneNumber objects.
TopologyManager	topology	TopologyEdge TopologyNode	Defines the methods for managing TopologyEdge and ToplogyNode objects.
TransitionManager	common		Transitions an entity's business and object states by finding the matching transition definitions with business action, object activity, entity type, and specification. If the definition's **from** state matches the entity's state, then the entity's state is set to the definition's **to** state.
VirtualNetworkManager	network	Network NetworkNode NetworkEdge FlowInterface FlowIdentifier	Defines the methods for managing Virtual Networks, Service Networks, and Packet Virtual Network objects.
WorkflowManager	businessinteraction	EngineeringWorkOrder Checklist Activity	Defines the methods for managing Engineering Work Orders and Activities. This manager also updates Activity properties like duration and their checklists and, also transitioning an Activity's status.

B

NFV Orchestration Java Managers

This appendix provides a listing of Oracle Communications Unified Inventory Management (UIM) NFV Orchestration Java manager names, the package in which they reside, and a brief description.

These Java manager classes are found in the **nso_managers.jar** which is located in the UIM Software Development Kit (SDK). See *UIM Developer's Guide* for more information on the UIM SDK.

Table B–1 contains the list of Java managers in alphabetical order by manager name.

Note: The package references in Table B–1 assume the package prefix of **oracle.communications.inventory.nso**.

Table B–1 List of NFV Orchestration Java Managers

Manager Name	Package	Description
DescriptorManager	api.descriptor	Defines numerous find methods for retrieving the descriptors and specifications for Network Services, VNFs, PNFs and orchestration requests.
EMSManager	api.ems	Defines the methods for finding, creating, updating and deleting EMSs, which perform the typical management functionality for one or several VNFs.
NetworkServiceDesignManager	api.c2a	Defines the methods for creating, disconnecting and changing the configuration version for a Network Service.
NetworkServiceManager	api.ns	Defines various methods to instantiate, activate, terminate and update Network Service entities. This manager also includes several find methods for Network Services and methods for Design and Assign of various Network Service entities.
NFVIManager	nfvi	Defines the methods for managing the NFV infrastructure. This manager includes methods to create, get and delete objects such as flavors, ports, networks and virtual routers for the VIM. By default, NFV Orchestration supports integration with OpenStack, but you can implement this interface to provide integration to a custom VIM, for instance supporting VMware vCloud.
NSONotificationManager	api.ns	Defines the methods to process a notification. This manager provides the mechanism to extend and provide your own custom required notifications.

Table B–1 (Cont.) List of NFV Orchestration Java Managers

Manager Name	Package	Description
NSOResponseManager	api.ns	Defines the methods to aid in sending a response to a topic in the WebLogic server. By default, NFV Orchestration includes a response manager that publishes the status of the VNF and Network Service life-cycle operations to a topic. You can also implement this interface to provide a custom response manager.
PNFManager	api.pnf	Defines the methods to find, create, update, delete, and manage PNFs.
PNFServiceDesignManager	api.c2a	Defines the methods to process the actions performed during a PNF addition to a Network Service or termination from a Network Service.
ResourceOrchestrationManager	api.ro	Defines the methods used to choose a data center based on the requirement to provision a Network Service. An instance can be obtained from the NSOHelper class.
SBSytemManager	api.sb	Defines the south-bound system manager providing methods to manage the VNF, such as reboot, replace, upgrade, scale and instantiate. You can implement this interface to integrate NFV Orchestration with a third-party VNF manager or Oracle's VNF Manager.
SDNController	nfvi	Defines the methods to create, update, and delete network forwarding paths (NFPs) for VNF forwarding graphs (VNFFGs). By default, NFV Orchestration supports integration with OpenStack Neutron Networking-SFC (Service Function Chaining) using Open vSwitch (OVS) driver, but you can also implement a custom SDN controller.
VNFCapabilityServiceManager	api.vnf.capability	Defines the methods to configure a VNF service. This also contains a designAndAssign() method, as well as the issueConfigurationVersion() method.
VNFConfigManager	nfvi	Defines the methods to return the configuration files of a VNF and generates configuration content for VNF configuration. You can implement this interface to extend the VNF manager functionality and its configuration files.
VNFConnectionManager	nfvi	Defined the methods to connect and configure a VNF. You can implement this interface to extend the VNF manager functionality for these methods.
VNFLifeCycleManager	nfvi	Defines methods to manage the life cycle of a VNF, such as instantiate, reboot and terminate. You can implement this interface to extend the VNF manager functionality for these methods. By default, NFV Orchestration manages the VNF life-cycle operations by using OpenStack Compute services (referred to as Nova), but you can also implement and use a custom VNF life-cycle manager.
VNFMonitoringManager	nfvi	Defines the methods to manage the monitoring of a VNF, such as create, get and update alarms. By default, NFV Orchestration supports integration with OpenStack Ceilometer, but you can also implement and use a custom monitoring engine.
VNFServiceDesignManager	api.c2a	Defines the methods for creating, disconnecting and changing the configuration version for a VNF.
VNFServiceManager	api.vnf	Defines various methods to instantiate, activate, terminate and update VNFs. This manager also includes several find methods for VNFs.

See *UIM NFV Orchestration Implementation Guide* for more information on extending the Java managers.

C

Common Utility Code Examples

This appendix provides example code of common utilities that are often used when working with the Oracle Communications Unified Inventory Management (UIM) application program interfaces (APIs).

Example C–1 Common Utility Code

```
public boolean hasErrors()
{
    boolean hasErrors = false;
    UserEnvironment userEnvironment = UserEnvironmentFactory.getUserEnvironment();
    if (userEnvironment != null)
    {
        FeedbackProvider feedbackProvider = userEnvironment.getFeedbackProvider();
        hasErrors = feedbackProvider.hasMessages(FeedbackLevel.ERROR);
    }
    return hasErrors;
}


public FeedbackProvider getFeedbackProvider()
{
    FeedbackProvider feedbackProvider = null;
    UserEnvironment userEnvironment = getUserEnvironment();
    if (userEnvironment != null)
    {
        feedbackProvider = userEnvironment.getFeedbackProvider();
    }
    return feedbackProvider;
}


protected static void commitOrRollback(UserTransaction ut)throws Exception
{
    FeedbackProvider feedbackProvider =
        getUserEnvironment().getFeedbackProvider();
    if (feedbackProvider.hasMessages(FeedbackLevel.ERROR))
    {
        if (ut != null && ut.getStatus() == Status.STATUS_ACTIVE)
            ut.rollback();
    }
    else
    {
        if (ut != null && ut.getStatus() == Status.STATUS_ACTIVE)
            ut.commit();
    }
```

```
}


protected static UserEnvironment startUserEnvironment()throws Exception
{
    UserEnvironment userEnvironment = null;
    try {
        UserEnvironment = getUserEnvironment();
        if (userEnvironment != null)
        {
            //Reset the User Context in User Environment.
            userEnvironment.reset();
            //Begin the UserEnvironment before it is first used.
            userEnvironment.begin();
            //Reset the Feedback Provider in User Environment.
            userEnvironment.getFeedbackProvider().reset();
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
    return userEnvironment;
}


protected static void endUserEnvironment(UserEnvironment userEnvironment)
{
    if (userEnvironment == null)
        return;

    userEnvironment.getFeedbackProvider().reset();
    userEnvironment.end();
}


protected static UserEnvironment getUserEnvironment() throws Exception
{
    UserEnvironment userEnvironment = null;
    try {
        //Utils is oracle.communications.platform.util.Utils
        InitialContext initialContext = Utils.getInitialContext();
        String jndiContextName = "inv";
        String userEnvironmentName = "UserEnvironment";

        userEnvironment = (UserEnvironment)initialContext.lookup
                          (jndiContextName + "/" + userEnvironmentName);

        initialContext.close();
    }
    catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
    return userEnvironment;
}
```