## Contents

**Preface** .......................................................................................................................................................... xv  
  Audience ....................................................................................................................................................... xv  
  Related Documentation .................................................................................................................. ................ xv  
  Documentation Accessibility ................................................................................................................. ............... xvi

### 1 Overview

**Extending UIM** ........................................................................................................................................ 1-1  
  Creating Cartridges ................................................................................................................................. 1-1  
  Extending the Data Model ........................................................................................................................ 1-2  
  Creating Rulesets ........................................................................................................................................ 1-2  
  Creating Web Services ............................................................................................................................... 1-3  
  Customizing the User Interface ................................................................................................................. 1-3  
  Localizing UIM .......................................................................................................................................... 1-3  
  Optimizing Concurrent Resource Allocation in UIM ............................................................................ 1-3  
  Cooperating with UIM ............................................................................................................................... 1-3  
  Requirements for Extending UIM ............................................................................................................ 1-4

**Tools for Extending UIM** ...................................................................................................................... 1-4  
  Design Studio ........................................................................................................................................... 1-4  
  Additional Tools ........................................................................................................................................... 1-4

**Documentation for Extending UIM** ...................................................................................................... 1-4  
  Information Model Documentation ........................................................................................................ 1-4  
  API Documentation .................................................................................................................................. 1-4  
  Javadoc Documentation ........................................................................................................................... 1-5

**Guidelines for Extending UIM** ........................................................................................................... 1-5  
  Backwards Compatibility ......................................................................................................................... 1-5  
  Detecting Code Changes Between Releases ......................................................................................... 1-6

### 2 Using Design Studio to Extend UIM

**Installing Design Studio** ...................................................................................................................... 2-1  
  Configuring Design Studio ....................................................................................................................... 2-1  
  Setting the Compiler Compliance Level ................................................................................................. 2-2  
  Setting the JRE System Library ............................................................................................................. 2-2  
  Setting System Variables ....................................................................................................................... 2-2  
  Importing the Model Projects ................................................................................................................ 2-3
4 Extending the Data Model

About the UIM Data Model .............................................................................................................. 4-1
About Entities ..................................................................................................................................... 4-2
About Entity Capabilities .................................................................................................................. 4-2
About Entity Relationships ................................................................................................................ 4-2
About Entity Managers .................................................................................................................... 4-2
About Entity ID Sequencing ............................................................................................................. 4-3

About the Metadata Files .................................................................................................................. 4-3
Understanding Metadata File Content ............................................................................................. 4-5
Understanding Entity Definitions ..................................................................................................... 4-6
*-entities.xml Files ............................................................................................................................. 4-6
More on Entity Definitions ............................................................................................................... 4-7
Understanding Entity Attribute Definitions ..................................................................................... 4-8
*-types.xsd Files ................................................................................................................................. 4-8
Understanding Enumeration Definitions .......................................................................................... 4-8
*-enum-entities.xml Files .................................................................................................................. 4-8
*-enum-types.xsd Files ...................................................................................................................... 4-9
Understanding Native Sequence Definitions .................................................................................. 4-9
ocim-entityidsequenceextension-entities.xml File ........................................................................ 4-9
Understanding the Tags that Govern Definitions ............................................................................ 4-10

Extending the Data Model Through the Metadata Files ................................................................ 4-12
Defining New Entities ....................................................................................................................... 4-13
Creating New Entity Managers .......................................................................................................... 4-13
Defining New Entity Attributes ........................................................................................................ 4-13
Defining New Enumerations ............................................................................................................. 4-14
Defining New Native Sequences ....................................................................................................... 4-15
Extending Existing Entities .............................................................................................................. 4-15
Understanding the Extension Tag ..................................................................................................... 4-16
Extending Existing Entities ............................................................................................................... 4-17
Extending Existing Entity Attributes ............................................................................................... 4-17
Extending Existing Enumerations ..................................................................................................... 4-18
Extending Existing Native Sequences .............................................................................................. 4-18

Applying Metadata Static Extensions ............................................................................................ 4-19
About the build.xml File ................................................................................................................... 4-20
Generating, Compiling, and Packaging the Entity Source Files ..................................................... 4-21

More on Entity Definitions .............................................................................................................. 4-24
Understanding Entity Capability Definitions .................................................................................... 4-25
Understanding Entity Relationship and Collection Definitions .................................................... 4-25
Uni-Directional, One-to-One Relationship ....................................................................................... 4-26
Uni-Directional, One-to-Many Relationship .................................................................................... 4-26
5 Extending Life Cycles

About Business Interactions ........................................................................................................... 5-1
Understanding Metadata File Content ........................................................................................... 5-2
Understanding Entity Definitions ................................................................................................. 5-2
Understanding Enumeration Definitions ......................................................................................... 5-3
About Life-Cycle States .................................................................................................................. 5-3
Understanding Business State Enumerations .............................................................................. 5-3
Understanding Transition Definitions ............................................................................................. 5-4
Understanding How Transitions Are Triggered .............................................................................. 5-5
About Transition Groups .................................................................................................................. 5-7
Extending Life Cycles through the Metadata Files ........................................................................ 5-7
Extending Entity Definitions ............................................................................................................ 5-8
Defining an Entity as Life-Cycle Managed ...................................................................................... 5-8
Defining an Entity as Business-Interaction Enabled ...................................................................... 5-8
Defining an Entity as Life-Cycle Managed and Business-Interaction Enabled .............................. 5-9
Extending Enumeration Definitions ................................................................................................ 5-9
Extending Transition Definitions ..................................................................................................... 5-9
Defining New Transitions ................................................................................................................ 5-9
Extending Existing Transitions ....................................................................................................... 5-9
Updating Properties Files ............................................................................................................... 5-10
Updating Security ............................................................................................................................ 5-10
More on Transition Definitions ........................................................................................................ 5-11
About Life Cycle Management Interfaces ...................................................................................... 5-13
LifeCycleManaged ............................................................................................................................ 5-13
TransitionManager .......................................................................................................................... 5-14
Transition Definition Search ............................................................................................................ 5-14

6 Extending the Topology

About Topology Entities and Topology-Managed Entities ............................................................... 6-1
Topology Entities ............................................................................................................................. 6-2
Topology-Managed Entities ............................................................................................................. 6-2
About Topology Mapping .................................................................................................................. 6-3
TopologyEdge ................................................................................................................................... 6-3
TopologyNode .................................................................................................................................. 6-3
Extending the Topology .................................................................................................................... 6-3
Defining an Entity as Topology-Managed ......................................................................................... 6-4
Extending the BusinessObjectType.java File ................................................................................ 6-4
Extending the Mapping ..................................................................................................................... 6-5
Configuring the topologyProcess.properties file ........................................................................... 6-5
7 Extending Security

Securing APIs.................................................................................................................. 7-1
  Securing APIs through the SecurityValidation Aspect .................................................. 7-1
    Creating the Global Extension Point ........................................................................... 7-2
    Creating the Ruleset Global Extension Point ................................................................. 7-2
  Securing APIs through Rulesets and Extension Points .................................................. 7-3

Securing Entity Data........................................................................................................... 7-3
  About Entity Access Control ............................................................................................ 7-3
  Securing Entity Data through Rulesets and Extension Points......................................... 7-3
    Setting Permissions in a Custom Ruleset ...................................................................... 7-4
    Setting Partitions in a Custom Ruleset .......................................................................... 7-5
    Enforcing Security in a Custom Ruleset ......................................................................... 7-6

Creating Custom Rulesets and Extension Points ............................................................... 7-6
  Creating Custom Rulesets .............................................................................................. 7-6
    Securing APIs Example ............................................................................................... 7-6
    Securing Entity Data through Permissions Example .................................................... 7-9
    Securing Entity Data through Partitions Example ......................................................... 7-9
    Securing Entity Data for a Range of Entities Example .................................................. 7-10
    Enforcing Security Example ........................................................................................ 7-11
  Creating Extension Points .............................................................................................. 7-11
  Creating the Ruleset Extension Point .............................................................................. 7-12

8 Extending UIM through Rulesets

About AspectJ .................................................................................................................... 8-1

About JBoss Rules ............................................................................................................. 8-1
  JBoss Rule Structure ....................................................................................................... 8-1
  Rule Attributes ................................................................................................................ 8-2
  Rulesets ........................................................................................................................... 8-2
9 Integrating UIM through Web Services

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Services Overview</td>
<td>9-1</td>
</tr>
<tr>
<td>UIM Web Service Module Overview</td>
<td>9-3</td>
</tr>
<tr>
<td>About Message Queues</td>
<td>9-3</td>
</tr>
<tr>
<td>About Transaction Handling</td>
<td>9-4</td>
</tr>
<tr>
<td>About Exception Stacktraces</td>
<td>9-4</td>
</tr>
<tr>
<td>UIM Web Service Standards and Specifications</td>
<td>9-4</td>
</tr>
<tr>
<td>About the UIM Service Fulfillment Web Service</td>
<td></td>
</tr>
<tr>
<td>CaptureInteraction</td>
<td>9-5</td>
</tr>
<tr>
<td>Validating Input Data</td>
<td>9-6</td>
</tr>
<tr>
<td>Associating Business Interactions</td>
<td>9-7</td>
</tr>
<tr>
<td>About the CaptureInteraction Flow</td>
<td>9-7</td>
</tr>
<tr>
<td>Understanding the Request Content</td>
<td>9-8</td>
</tr>
<tr>
<td>About the CaptureInteraction Response and Faults</td>
<td>9-14</td>
</tr>
<tr>
<td>ProcessInteraction</td>
<td>9-14</td>
</tr>
<tr>
<td>About the ProcessInteraction Flow</td>
<td>9-15</td>
</tr>
<tr>
<td>Customizing ProcessInteraction</td>
<td>9-16</td>
</tr>
<tr>
<td>About the ProcessInteraction Response and Faults</td>
<td>9-20</td>
</tr>
<tr>
<td>GetInteraction</td>
<td>9-20</td>
</tr>
<tr>
<td>About the GetInteraction Flow</td>
<td>9-20</td>
</tr>
<tr>
<td>About the GetInteraction Response and Faults</td>
<td>9-21</td>
</tr>
<tr>
<td>UpdateInteraction</td>
<td>9-21</td>
</tr>
<tr>
<td>About the UpdateInteraction Flow</td>
<td>9-21</td>
</tr>
<tr>
<td>About the UpdateInteraction Response and Faults</td>
<td>9-22</td>
</tr>
<tr>
<td>GetConfiguration</td>
<td>9-22</td>
</tr>
<tr>
<td>About the GetConfiguration Flow</td>
<td>9-23</td>
</tr>
<tr>
<td>About the GetConfiguration Response and Faults</td>
<td>9-24</td>
</tr>
<tr>
<td>UpdateConfiguration</td>
<td>9-24</td>
</tr>
<tr>
<td>About the UpdateConfiguration Response and Faults</td>
<td>9-25</td>
</tr>
<tr>
<td>Understanding the UIM Service Fulfillment Web Service</td>
<td>9-25</td>
</tr>
<tr>
<td>Using the UIM Service Fulfillment Web Service</td>
<td>9-27</td>
</tr>
<tr>
<td>Locating the WSDL and Schema Files</td>
<td>9-27</td>
</tr>
<tr>
<td>About the WSDL File</td>
<td>9-27</td>
</tr>
<tr>
<td>About the Schema Files</td>
<td>9-28</td>
</tr>
<tr>
<td>Deploying the Web Service</td>
<td>9-29</td>
</tr>
<tr>
<td>Verifying the Deployment</td>
<td>9-29</td>
</tr>
<tr>
<td>Testing the Web Service</td>
<td>9-30</td>
</tr>
<tr>
<td>Test Input XML</td>
<td>9-30</td>
</tr>
<tr>
<td>Preconfiguration for Testing</td>
<td>9-30</td>
</tr>
<tr>
<td>Securing the Web Service</td>
<td>9-30</td>
</tr>
</tbody>
</table>
10 Developing Custom Web Services

About the Reference Web Service .......................................................... 10-1
  ReferenceUim.wsd1 ................................................................. 10-1
Schema Files .................................................................................. 10-2
Java Source Files ......................................................................... 10-2
build.xml File ................................................................................ 10-2
  build.xml File Ant Targets ....................................................... 10-2
About the WSDL-First Approach to Developing Custom Web Services .............................................. 10-3
Guidelines and Patterns for Developing Custom Web Services ......................................................... 10-4
  Class Diagrams ........................................................................ 10-4
  WSDL Interface Pattern .......................................................... 10-7
  Operation Name Pattern .......................................................... 10-7
  Operation Signature Pattern ..................................................... 10-8
    Input Parameter ...................................................................... 10-8
    Output Parameter .................................................................... 10-8
    Fault Types ............................................................................ 10-9
  Schemas Pattern ...................................................................... 10-9
    Schema Definition Versus WSDL Definition .......................... 10-9
Developing and Running Custom Web Services ................................................................................. 10-9
  Configuring Your Work Environment ......................................... 10-10
    WebLogic Server .................................................................. 10-10
    UIM ..................................................................................... 10-11
    Design Studio ........................................................................ 10-11
  Importing the Reference Web Service Project ................................................. 10-11
  Configuring the Imported Project .................................................. 10-13
    Configuring the COMPUTERNAM e.properties File .............. 10-13
    Configuring the Project Library List ........................................... 10-14
  Locating the API Method Signature in the Javadoc ......................... 10-15
    Information to Capture .......................................................... 10-16
  Developing the Web Service ......................................................... 10-16
    Creating the WSDL File ........................................................ 10-17
    Creating Schema Files .......................................................... 10-18
    Creating Java Source Files ..................................................... 10-19
  Generating Java Source Based on the WSDL ................................. 10-20
  Creating the WAR File ............................................................. 10-23
    Extracting and Updating the application.xml File .................. 10-24
      Extracting the File .............................................................. 10-25
      Updating the File ............................................................... 10-25
  Importing the WAR File into the EAR File ................................. 10-26
  Deploying the EAR File .......................................................... 10-26
    Verifying the Deployment ....................................................... 10-26
    Specifying a Deployment Plan ................................................. 10-26
Testing Custom Web Services ........................................................................ 10-27
  Generating Test Input XML ....................................................... 10-27
11 Customizing the User Interface

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing JDeveloper</td>
<td>11-1</td>
</tr>
<tr>
<td>Extracting the inventory.ear File into JDeveloper</td>
<td>11-3</td>
</tr>
<tr>
<td>Customizing the User Interface</td>
<td>11-4</td>
</tr>
<tr>
<td>About the UI Files</td>
<td>11-4</td>
</tr>
<tr>
<td>JSFF and XML Files</td>
<td>11-4</td>
</tr>
<tr>
<td>XML Files</td>
<td>11-5</td>
</tr>
<tr>
<td>Java Files</td>
<td>11-6</td>
</tr>
<tr>
<td>XLF Files</td>
<td>11-7</td>
</tr>
<tr>
<td>DCX File</td>
<td>11-8</td>
</tr>
<tr>
<td>Displaying Custom Attributes on a Web Page</td>
<td>11-8</td>
</tr>
<tr>
<td>Adding Custom Input Fields to a Web Page</td>
<td>11-8</td>
</tr>
<tr>
<td>Adding Conditional Components to a Web Page</td>
<td>11-9</td>
</tr>
<tr>
<td>Disabling an Input Field on a Web Page</td>
<td>11-9</td>
</tr>
<tr>
<td>Adding a Custom Action to a Web Page</td>
<td>11-10</td>
</tr>
<tr>
<td>Adding a Custom Search Field</td>
<td>11-10</td>
</tr>
<tr>
<td>Extending the API</td>
<td>11-10</td>
</tr>
<tr>
<td>Extending the UI</td>
<td>11-11</td>
</tr>
<tr>
<td>Deploying User Interface Customizations</td>
<td>11-13</td>
</tr>
<tr>
<td>Customizing Logos</td>
<td>11-13</td>
</tr>
<tr>
<td>Testing User Interface Customizations</td>
<td>11-15</td>
</tr>
</tbody>
</table>

12 Localizing UIM

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting the Language Preference in Internet Explorer</td>
<td>12-1</td>
</tr>
<tr>
<td>Determining the Locale ID</td>
<td>12-2</td>
</tr>
<tr>
<td>Localizing UIM</td>
<td>12-2</td>
</tr>
<tr>
<td>About the UI-Specific Files</td>
<td>12-3</td>
</tr>
<tr>
<td>Localizing the UI-Specific Files</td>
<td>12-3</td>
</tr>
<tr>
<td>Importing the Localization Archive File into Design Studio</td>
<td>12-3</td>
</tr>
<tr>
<td>Locating the UI-Specific Files within the Project</td>
<td>12-3</td>
</tr>
<tr>
<td>Copying and Renaming the UI-Specific Files</td>
<td>12-4</td>
</tr>
<tr>
<td>Editing the UI-Specific Files</td>
<td>12-4</td>
</tr>
<tr>
<td>Deploying the Cartridge Containing the Localized Files</td>
<td>12-6</td>
</tr>
<tr>
<td>Testing the UIM UI Localization</td>
<td>12-6</td>
</tr>
<tr>
<td>Localizing UIM Help</td>
<td>12-6</td>
</tr>
<tr>
<td>About UIM Help</td>
<td>12-6</td>
</tr>
<tr>
<td>About the Oracle Help Configuration File</td>
<td>12-7</td>
</tr>
<tr>
<td>About the UIM Help Files</td>
<td>12-7</td>
</tr>
<tr>
<td>Localizing the UIM Help Files</td>
<td>12-7</td>
</tr>
<tr>
<td>Extracting the Help Files</td>
<td>12-7</td>
</tr>
<tr>
<td>Translating the Help Files</td>
<td>12-8</td>
</tr>
<tr>
<td>Regenerating the Search Index File</td>
<td>12-10</td>
</tr>
<tr>
<td>Creating the Localized Help JAR File</td>
<td>12-11</td>
</tr>
</tbody>
</table>
13 Optimizing Concurrent Resource Allocation

About Concurrent Resource Allocation .............................................................. 13-1
About Row Locking ......................................................................................... 13-2
Understanding How Row Locking Works for Telephone Numbers ................. 13-2
  About the RowLock Pattern ........................................................................ 13-3
  About Releasing Locked Rows .................................................................... 13-3
  About the LockPolicy Object ....................................................................... 13-3
    numberOfResources .................................................................................. 13-3
    expirationTimeStamp ................................................................................ 13-3
    filterExistingLocks .................................................................................. 13-3
    Example LockPolicy Attribute Combinations ........................................ 13-4
Extending UIM Entities to Use Row Locking .................................................. 13-4
  Statically Extending the Data Model ............................................................ 13-4
  Enabling Row Locking ................................................................................ 13-6
  Using Row Locking with Entity Finder APIs .............................................. 13-8
    Understanding How UIM Uses Row Locking .......................................... 13-8
    Writing Custom Code to Use Row Locking .............................................. 13-8
  Using Row Locking Without Entity Finder APIs ........................................ 13-12

14 Understanding the Cooperation Framework

About Cooperation and Data Federation ......................................................... 14-1
About Transaction-Based and Order-Based Cooperations ............................... 14-2
  Transaction-Based ..................................................................................... 14-2
  Order-Based .............................................................................................. 14-2
    Work Order ........................................................................................... 14-3
    Business Interaction Attachment ............................................................. 14-3
About Externally Enabled Entities ................................................................ 14-4
  External Identification ................................................................................. 14-4

A UIM Sample Cartridges

About the Sample Cartridges ......................................................................... A-1
Using the Sample Cartridges .......................................................................... A-1

B Reference Web Service

About the Reference Web Service Operations ................................................. B-2
  CaptureServiceConfigurationInputs ......................................................... B-3
  AutomateServiceConfiguration ................................................................. B-4
  GetServiceConfiguration .......................................................................... B-5
  UpdateServiceConfiguration .................................................................... B-5
    Life Cycle Management ........................................................................... B-6
  ChangeServiceConfigurationState ............................................................... B-6
  CalculateConfigurationDifferences ............................................................. B-6
DeleteParty.................................................................................................................... B-7

Understanding the Reference Web Service Operations Through Examples.................... B-7
New Order/Add Example ........................................................................................................ B-7
New Order/Change Example ............................................................................................... B-8
Amendment Order Example ............................................................................................. B-8
New Order/Disconnect Example ........................................................................................ B-8

Using the Reference Web Service ................................................................................ B-9
Locating the WSDL and Schema Files ............................................................................ B-9
About the WSDL File ........................................................................................................ B-9
About the Schema Files .................................................................................................. B-10
Deploying the Reference Web Service ............................................................................. B-11
Verifying the Deployment ............................................................................................... B-11
Testing the Reference Web Service ................................................................................. B-11
Test Input XML ................................................................................................................ B-11
Generating Test Input XML .......................................................................................... B-11
Preconfiguration for Testing ......................................................................................... B-12
DeleteParty .................................................................................................................... B-12
CaptureServiceConfigurationInputs .............................................................................. B-12
AutomateServiceConfiguration ..................................................................................... B-12
GetServiceConfiguration ............................................................................................. B-12
UpdateServiceConfiguration ......................................................................................... B-12
CalculateConfigurationDifferences .............................................................................. B-12
ChangeServiceConfigurationState ................................................................................. B-12
Securing the Reference Web Service ............................................................................. B-13

C Reference Cooperation Cartridges

About the IP Address Cooperation ....................................................................................... C-1
About the VLAN ID Cooperation ......................................................................................... C-2
About the Connectivity Cooperation ................................................................................... C-2
Using the Reference Cooperation Cartridges ..................................................................... C-2
Creating New or Extending Existing Cooperation Cartridges ........................................... C-2

Cooperative Solution Considerations .............................................................................. C-3
Determining the Cooperative Solution ............................................................................... C-3
Avoiding Cooperation Conflicts ....................................................................................... C-3
Creating New Specifications ............................................................................................ C-4
Managing External Identifiers ........................................................................................ C-4
Creating Externally Enabled Entities in UIM ..................................................................... C-4
Accessing a New External System .................................................................................. C-5
Creating Custom Web Services ....................................................................................... C-6
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 should be read after reading UIM Concepts, because this guide assumes that the reader has a conceptual understanding of UIM. 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.

You should read UIM Concepts before reading this guide.

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 post-installation tasks.
- UIM System Administrator’s Guide: Describes administrative tasks such as working with cartridges and technology 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 Information Model Reference: Describes the UIM information model entities and data attributes, and explains patterns that are common across all entities.
- 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.

- **UIM Cartridge and Technology Pack Guide**: Provides information about how you use cartridges and technology packs with UIM. Describes the content of the base cartridges.

- **UIM technology pack implementation guides**: Describe the content of product technology packs as well as configuration guidelines and implementation considerations.

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


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

**Documentation Accessibility**


**Access to Oracle Support**

This chapter provides an overview of extending Oracle Communications Unified Inventory Management (UIM) that includes the following information:

- Extending UIM
- Tools for Extending UIM
- Documentation for Extending UIM
- Guidelines for Extending UIM

---

**Note:** Throughout this guide, the `UIM_Home` placeholder is used to represent the directory where you installed UIM. For a typical UIM installation, `UIM_Home` is `opt/Oracle/Middleware/user_projects/domains/domain_name/UIM`, where `domain_name` is the domain name you supplied when installing UIM.

---

### Extending UIM

UIM extensions can be categorized as static or dynamic:

- Static extensions are changes made prior to rebuilding the application, which results in the changes becoming a part of the application deployment. For example, extending the data model involves adding content to the existing metadata files, which are contained within the `inventory.ear` file. So, you must rebuild the `inventory.ear` file to include the changed metadata files, and then redeploy the application for the changes to affect.

- Dynamic extensions are made anytime, applied at runtime, and do not require rebuilding the application for the changes to take effect. For example, a cartridge containing specifications can be deployed into UIM, making the specifications available within the application without rebuilding the application.

### Creating Cartridges

Cartridges can contain specifications, characteristics, rulesets, and extended code. You can create cartridges to meet specific business needs in Oracle Communications Design Studio. For example, if your equipment requires specific logic not provided by the EquipmentManager class, you can create your own class, inherited from the EquipmentManager class, and write a new method to address the specific equipment logic. The new method can then be called from within a ruleset.

The extensions defined within a cartridge may be static or dynamic. Cartridges are further explored in Chapter 2, "Using Design Studio to Extend UIM".
Extending the Data Model

You can statically extend the data model by adding new columns to existing tables, or by adding new tables. For example, your business requirements may dictate that you save particular information regarding a telephone number that the existing UIM data model does not save. You can extend the UIM data model to include this piece of information. Your business requirements may dictate that you save information unrelated to any data that the existing UIM data model saves. You can extend the UIM data model to include a new table to retain this information.

This is done through additions to the metadata. The UIM installation provides tools that enable you to automatically regenerate the data model based on the metadata, and to update the application (inventory.ear) to reflect the additions. This topic is further explored in Chapter 4, "Extending the Data Model".

You can dynamically extend the data model through characteristics. For example, you can define a specification for a telephone number and add characteristics that further describe the telephone number. When you create entities in UIM based on a specification that includes characteristics, the characteristics are automatically included in the entities. This topic is further explored in UIM Concepts. For instructions on how to define characteristics in Design Studio, see the Design Studio Help.

Extending Life Cycles, Topology, and Security

An entity is a Java representation of UIM data, and an entity can be defined as life-cycle-managed in the metadata. Life cycle refers to an entity having a start to its life, an end to its life, and a defined state at any given point during its life. Life-cycle transition definitions are part of the UIM metadata, and you can extend these definitions to solve specific business requirements.

An entity can also be defined as topology-managed in the metadata. Topology is a graphical representation of the spatial relationships and connectivity among your inventory entities. Topology-managed entities map to topology entities, which are used in the graphical representation. UIM defines several entities as topology-managed, and you can extend topology by defining additional entities to be topology-managed.

Information on security is provided in UIM System Administrator's Guide. However, this guide (UIM Developer's Guide) provides additional security information specific to securing UIM APIs and UIM entities.

These topics are further explored in:

- Chapter 5, "Extending Life Cycles"
- Chapter 6, "Extending the Topology"
- Chapter 7, "Extending Security"

Additional security information specific to securing Web services is in:

- Chapter 9, "Integrating UIM through Web Services"
- Chapter 10, "Developing Custom Web Services"
- Appendix B, "Reference Web Service"

Creating Rulesets

A ruleset is custom code that extends existing logic at a specified point. You can dynamically extend UIM by creating rulesets to meet specific business needs. For example, if the default telephone number format does not match the telephone
number format used by the country in which you are implementing UIM, you can use a ruleset to reformat the telephone number.

This topic is further explored in Chapter 8, "Extending UIM through Rulesets".

Creating Web Services

Web services are APIs that can be accessed over a network, such as the Internet, and run on a remote system hosting the requested services. UIM provides Web services that are used for service fulfillment and for cartridge management. You can statically extend UIM by creating custom Web services. For example, you can write a Web service that performs a search for a specified entity, such as a pipe, a party, or a telephone number.

This topic is further explored in:
- Chapter 9, "Integrating UIM through Web Services"
- Chapter 10, "Developing Custom Web Services"
- Appendix B, "Reference Web Service"

Customizing the User Interface

You can customize the user interface by adding fields or functionality to existing pages, or by adding new pages. For example, you may want to add a field named Type to the Equipment Maintenance page and populate it with your equipment type. Customizing the user interface statically extends UIM.

This topic is further explored in Chapter 11, "Customizing the User Interface".

Localizing UIM

Localizing UIM is the process of changing the user interface and the online Help from the language in which it was written to another language. This process involves modifying files that contain text that displays in the user interface and the online Help.

This topic is further explored in Chapter 12, "Localizing UIM".

Optimizing Concurrent Resource Allocation in UIM

You can optimize UIM performance by extending entity types that are heavily used in your UIM environment to implement the rowLock pattern.

This topic is further explored in Chapter 13, "Optimizing Concurrent Resource Allocation".

Cooperating with UIM

You can extend UIM to cooperate with other systems through data federation, leasing in data, leasing out data, or sharing data. UIM provides reference cooperation cartridges that you can configure and use, or extend and use as a starting point in creating a custom cooperation.

This topic is further explored in Chapter 14, "Understanding the Cooperation Framework" and Appendix C, "Reference Cooperation Cartridges".
Requirements for Extending UIM
Extending UIM requires the installation of Design Studio, Oracle WebLogic Server, and UIM. Extensions are developed in Design Studio, but you also need access to a UIM development environment into which you can deploy cartridges and run unit tests.

Tools for Extending UIM
Several tools are available for extending UIM and are described in Chapter 2, "Using Design Studio to Extend UIM".

Design Studio
Design Studio is an Eclipse-based integrated development environment. Design Studio is not part of UIM, but it does come with features specific to UIM that enable you to extend UIM. Information on using Design Studio to extend UIM is in Chapter 2, "Using Design Studio to Extend UIM".

Additional Tools
Additional tools such as Ant and a JBoss plug-in are available to you when extending UIM. The UIM installation includes a collection of Apache Ant executable targets that are used to extend the data model. These targets automate entity regeneration, entity recompilation, and repackaging the application EAR file to include the recompiled entities. The JBoss plug-in can be used to edit ruleset syntax within Design Studio. Information on these tools, how to install them, and how to use them is in Chapter 2, "Using Design Studio to Extend UIM".

Documentation for Extending UIM
Additional information needed to extend UIM is described in the following sections. The resources described here are intended to be used together. For example, the Javadoc provides specific information on methods that are available per entity, and method signatures may define specific entity attributes. However, the Javadoc does not get into details regarding the entity itself or any of the attributes it defines; this type of information is covered elsewhere. See "Information Model Documentation" for more information.

Information Model Documentation
Entities are Java representations of UIM data. The entities that comprise UIM are detailed in Oracle Communications Information Model Reference and UIM Information Model Reference. The documents describe each entity, lists the entity attributes, provides examples, and includes information on patterns that are common across all entities. Oracle Communications Information Model Reference and UIM Information Model Reference are located in the UIM Developer Documentation Media Pack on the Oracle Software Delivery Cloud.

API Documentation
Information on UIM APIs is detailed in UIM API Selection Guide. The document provides an overview of UIM APIs that were selected to describe the usage patterns and best practices for service fulfillment scenarios. The document includes code samples that show the correct usage of the APIs and expectations of the implementation.
To access UIM API Selection Guide, search the Knowledge Base on My Oracle Support. After selecting the Oracle Communications Unified Inventory Management product, filter your search using appropriate keywords.

**Javadoc Documentation**

The classes that comprise UIM, and the Platform classes upon which UIM is built, contain Javadoc. The Javadoc that comes with the UIM installation includes both UIM and Platform Javadoc.

To access the Javadoc:

1. Start the application server.
   
   For instructions on how to start the application server, see UIM System Administrator’s Guide.

2. From the application server console, deploy the `UIM_Home/app/inventory.ear` file, which automatically deploys the `UIM_Home/doc/ora_uim_javadoc.war` file.
   
   For instructions on how to deploy a file from the application server console, see UIM System Administrator’s Guide.

3. In your Web browser, do one of the following:
   - If UIM was installed with SSL, enter:
     
     https://server:port/ora_uim_javadoc
   - If UIM was installed without SSL, enter:
     
     http://server:port/ora_uim_javadoc
   
     where server is the specific server on which the application is deployed and port is the port on which the application listens.

**Guidelines for Extending UIM**

You should be aware of backwards compatibility guidelines when extending UIM.

**Backwards Compatibility**

Before you extend UIM, understand the implications of backwards compatibility and the effects on future upgrades.

UIM maintains backwards compatibility for one release for all published external interfaces:

- Manager interfaces and method signatures
- Published extension points
- Web service interfaces

UIM does not maintain backwards compatibility for:

- Metadata and physical data model
- User interface
- Localization
Detecting Code Changes Between Releases

The UIM_Home/doc/ora_uim_delta.war file contains information regarding changes between releases. Oracle recommends that you review the WAR file content when upgrading UIM to determine if any of the upgrades affect your current extensions.

To read about code changes between releases:

1. Start the application server.
   
   For instructions on how to start the application server, see UIM System Administrator’s Guide.

2. From the application server console, deploy the UIM_Home/doc/ora_uim_delta.war file.
   
   For instructions on how to deploy a file from the application server console, see UIM System Administrator’s Guide.

3. In your Web browser, do one of the following:
   
   - If UIM was installed with SSL, enter: https://server:port/ora_uim_delta
   
   - If UIM was installed without SSL, enter: http://server:port/ora_uim_delta

   where server is the specific server on which the application is deployed and port is the port on which the application listens.
Using Design Studio to Extend UIM

This chapter provides information on Oracle Communications Design Studio, an Eclipse-based integration development environment. Design Studio comes with features specific to Oracle Communications Unified Inventory Management (UIM) that enable you to extend UIM.

This chapter contains the following sections:

- Installing Design Studio
- About Design Studio Perspectives
- About Design Studio Views
- About Cartridges and Technology Packs
- About Development Work Within Design Studio
- About the Developer-Facing Inventory Menu Options
- Additional Tools

Installing Design Studio

Design Studio is used to extend Oracle products. Different features are available for the different Oracle products, and each feature provides JAR files that are unique to the product.

For directions on how to install Design Studio, see Design Studio Installation Guide. The instructions describe how to install all available Oracle Communications features with a single installation. Of the features installed, UIM requires:

- Oracle Communications Design Studio Platform
- Oracle Communications Design Studio Domain Modelling
- Oracle Communications Design Studio for Inventory

Configuring Design Studio

To do development work in Design Studio, you must configure the Design Studio environment. This requires:

- Setting the Compiler Compliance Level
- Setting the JRE System Library
- Setting System Variables
- Importing the Model Projects
Configuring the Project Library List

Setting the Compiler Compliance Level
When you install Eclipse, the compiler compliance level is set to 1.6. This must remain 1.6 to reflect the correct version of the JDK.

To set the compiler compliance level in Design Studio:
1. From the menu, select Window, then select Preferences.
   The Preferences window appears.
2. In the navigation panel, expand Java, and click Compiler.
3. Verify that the Compiler compliance level is set to 1.6.
   If it is not, from the Compiler compliance level list, select 1.6.
4. Click Apply, then click OK.

Setting the JRE System Library
When you install Eclipse, the default JRE System Library is already set to the correct version. This information is included should you need to change it or update it.

To set the JRE System Library in Design Studio:
1. From the menu, select Window, then select Preferences.
   The Preferences window appears.
2. In the navigation panel, expand Java, and click Installed JREs.
3. Click Add.
   The Add JRE window appears.
4. From the Installed JRE Types list, select Standard VM and click Next.
5. Click Directory.
   The Browse For Folder window appears.
6. Navigate to the jre directory, located within your Oracle WebLogic Server installation.
   For example, C:/Oracle/Middleware/jdk160_37/jre.
7. Click OK.
   The JRE home, JRE name, and JRE System Libraries fields are set.
8. Click Finish.
   In the Package Explorer view, the JRE System Library now reflects jdk160_37.

Setting System Variables
When you install Eclipse, you must also set system variables to point to the correct version of the JDK.

To set the system variables:
1. From the Windows Start menu, select Control Panel, then select System.
   The System Properties window appears.
2. Click the Advanced tab.
3. Click Environment Variables.
The Environment Variables window appears.

4. Define a new system variable named JAVA_HOME:
   a. In the System Variables section, click New.
      The New System Variables window appears.
   b. In the Variable name field, enter JAVA_HOME.
   c. In the Variable value field, enter the path to the jdk directory that is part of your WebLogic Server installation. For example, C:/Oracle/Middleware/jdk160_37.
   d. Click OK.

5. Update the existing system variable Path:
   a. In the System Variables section, select Path, and click Edit.
      The Edit System Variables window appears.
   b. In the Variable value field, add the path to the bin directory that is part of your WebLogic Server installation. For example, C:/Oracle/Middleware/jdk160_37/bin.
   c. Click OK.
      The Environment Variables window appears.

6. Click OK.
   The System Properties window appears.

7. Click OK.

Importing the Model Projects

The following model projects must be imported into your workspace before modeling any UIM entities in Design Studio. The successful compilation of an Inventory project is dependent upon the model projects; however, the model projects are not compiled in Design Studio, nor are they deployed into UIM. The model projects are located in the UIM_Home/cartridges/base directory.

- ora_uim_mds
- ora_uim_model

For instructions on how to import projects into Design Studio, see the Design Studio Help.

---

Note: The model projects are installed with UIM. As a result, the model projects may change with each new UIM patchset or maintenance release. Contact your System Administrator to get the latest version of the model projects.

---

Configuring the Project Library List

Depending on the contents of your project, you may or may not need to configure the project library list. For example, if you are extending a UIM class, the project library list must be configured to point to the location of the UIM JAR file that contains the UIM class you are extending. In this example, the UIM JAR file is required to compile the project.
Imported projects include a library list of the files needed to compile the project, and the project library list must be configured to point to a location to pick up the cited files.

For instructions on how to configure the project library list, see the Design Studio Help.

---

**Note:** Project library lists include JAR files that are installed with UIM. As a result, these JAR files may change with each new UIM patchset or maintenance release. Contact your System Administrator to get the latest version of these JAR files.

---

**About Design Studio Perspectives**

Perspectives define your Workbench layout and provide different functionality for working with different types of resources. Several perspectives are available within Design Studio. The Java, Studio Design, and Studio Environment perspectives are commonly used when extending UIM.

For instructions on how to open a perspective, see the Design Studio Help.

**About Design Studio Views**

Within a given perspective, views further define your Workbench layout and provide different presentations of resources. Several views are available within Design Studio, and the available views are dependent upon the perspective.

For instructions on how to open a view, see the Design Studio Help.

**Java Perspective Views**

Figure 2–1 shows the views that are available in the Java perspective. When extending UIM, commonly used views are Ant, Navigator, Package Explorer, and Problems.
### Studio Design Perspective Views

Figure 2–2 shows the views that are available in the Studio Design perspective. When extending UIM, commonly used views are Cartridge, Package Explorer, and Problems.

<table>
<thead>
<tr>
<th>View</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blueprint</td>
<td></td>
</tr>
<tr>
<td>Cartridge</td>
<td>Alt+Shift+Q, C</td>
</tr>
<tr>
<td>Dictionary</td>
<td></td>
</tr>
<tr>
<td>Outline</td>
<td>Alt+Shift+Q, D</td>
</tr>
<tr>
<td>Overview</td>
<td></td>
</tr>
<tr>
<td>Package Explorer</td>
<td>Alt+Shift+Q, L</td>
</tr>
<tr>
<td>Problems</td>
<td>Alt+Shift+Q, J</td>
</tr>
<tr>
<td>Properties</td>
<td></td>
</tr>
<tr>
<td>Relation</td>
<td></td>
</tr>
<tr>
<td>Relation Graph</td>
<td></td>
</tr>
<tr>
<td>Solution</td>
<td></td>
</tr>
<tr>
<td>Other…</td>
<td>Alt+Shift+Q, K</td>
</tr>
</tbody>
</table>

### Studio Environment Perspective Views

Figure 2–3 shows the views that are available in the Studio Environment perspective. When deploying cartridges into UIM, commonly used views are Cartridge, Package Explorer, and Problems.

<table>
<thead>
<tr>
<th>View</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blueprint</td>
<td></td>
</tr>
<tr>
<td>Cartridge</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td></td>
</tr>
<tr>
<td>Outline</td>
<td>Alt+Shift+Q, K</td>
</tr>
<tr>
<td>Overview</td>
<td></td>
</tr>
<tr>
<td>Package Explorer</td>
<td>Alt+Shift+Q, P</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
</tr>
<tr>
<td>Properties</td>
<td></td>
</tr>
<tr>
<td>Relation</td>
<td></td>
</tr>
<tr>
<td>Relation Graph</td>
<td></td>
</tr>
<tr>
<td>Solution</td>
<td></td>
</tr>
<tr>
<td>Other…</td>
<td>Alt+Shift+Q, Q</td>
</tr>
</tbody>
</table>
About Cartridges and Technology Packs

A cartridge is a collection of entity specifications, characteristics, rulesets, and extended code defined in Design Studio. Cartridges are built in Design Studio from cartridge projects. When a cartridge project is compiled, the result is a JAR file (the cartridge) that you can deploy into UIM. The name you choose for the cartridge project becomes the name of the cartridge, and everything you create within that project is automatically part of the cartridge.

A technology pack is one or more cartridges that collectively address a particular business need or technology. Oracle offers technology packs that extend UIM for a particular technology, such as Cable TV or GSM 3GPP. Technology packs can also be created by customers and third parties. Technology packs can be deployed as downloaded, or they can be imported into Design Studio and extended prior to deployment.

You can create your own custom cartridges to extend UIM and to organize the extensions. For example, you could create a cartridge that contains all characteristics, another that contains all specifications, and so forth. Or you could create one cartridge per business area, such as telephone numbers or equipment, where each cartridge contains characteristics, specifications, and so forth, that are specific to the business area.

See UIM Cartridge and Technology Pack Guide for additional information.

Working with Cartridges in Design Studio

This section includes a brief overview of how you work with cartridges in Design Studio. For more information see Design Studio Help.

Working With Cartridge Dependencies

A cartridge can be dependent on other cartridges. These dependencies are specified on the Cartridge editor Dependency tab. For example, all Inventory cartridges are dependent upon the ora_uim_model project. So, when you create a new Inventory project, the Dependency tab automatically includes ora_uim_model in the list of project names. Projects that are listed on the Dependency tab indicate that the project is referenceable by the Inventory project at design time.

The projects listed on the Dependency tab do not indicate project compilation dependencies, which simply require that dependent projects be present in the workspace. For example, to compile an Inventory project, both the ora_uim_model and ora_uim_mds projects must be present in your workspace. However, only the ora_uim_model project is listed on the Dependency tab. The ora_uim_mds project is not listed on the Dependency tab because designing the Inventory project content is not dependent upon referencing anything in the ora_uim_mds project; but, the ora_uim_mds project must be present in the workspace to compile runtime artifacts that are used by the UIM UI.
You can define additional cartridge dependencies on the Cartridge editor Dependency tab, and specify the order of compilation by moving the project names up or down within the list. For instructions on how to define cartridge dependencies, see the Design Studio Help.

About Imported Technology Packs

When you import a technology pack into Design Studio, its contents are sealed, meaning that you cannot modify them. Figure 2–4 shows the Studio Design perspective Cartridge Explorer view of the imported projects from the Cable TV Technology Pack. Note that the projects are sealed.

Figure 2–4 imported Technology Packs

Imported projects include a library list of the files needed to compile the project, and the project library list must be configured to point to a location to pick up the cited files. See “Configuring the Project Library List” for more information.

For instructions on how to import a technology pack into Design Studio, see the Design Studio Help.
Viewing Cartridges in Design Studio

There are a number of ways to view the content of cartridge projects in Design Studio.

Figure 2–5 shows a cartridge project called `my_cartridge` as it appears in the Studio Design perspective Cartridge view. The corresponding Inventory Cartridge editor is also shown.

By expanding the cartridge in the Cartridge view, you can see the contents created with each cartridge.

By switching to the Java perspective Package Explorer view and expanding the cartridge, you can see the file types of the contents created with each cartridge. Figure 2–6 shows `my_cartridge` as it appears in the Java perspective Package Explorer view. The resultant JAR file resides in the `cartridgeBin` directory. The corresponding Inventory Cartridge editor is also shown.
Figure 2–6  Package Explorer View of Inventory Cartridge

The Java perspective Package Explorer view shows four files named my_cartridge. They are:

- **my_cartridge**: The Studio Inventory project.
- **my_cartridge.jar**: The JAR file that is deployed into UIM.
- **my_cartridge.xsd** and **my_cartridge_companion.xsdc**: Design Studio core files that are used to store characteristics as data elements within a schema entity (data dictionary).
- **my_cartridge.inventoryCartridge**: The Inventory Cartridge editor, shown on the right side of Figure 2–6.

The **cartridgeBin** and **cartridgeBuild** directories do not exist until you build the project, which also creates the **my_cartridge.jar** file.

**How Content Is Displayed**

The specifications and other content of a cartridge project are grouped based on type in the Cartridge view of the Studio Design perspective. For example, when an Equipment specification is created, it is grouped under **Equipment Specifications**. When a Pipe specification is created, it is grouped under **Pipe Specifications**.

These groupings are purely organizational; they do not represent physical directories. **Figure 2–7** shows cartridge content that includes six entities created from three different specifications.
About Deploying Cartridges and Technology Packs

You can deploy a cartridge into UIM from Design Studio or from the Cartridge Deployer Tool. Design Studio can only deploy a single cartridge; it cannot deploy a cartridge that contains other cartridges.

For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge (including a cartridge that contains other cartridges) using the Cartridge Deployer Tool, see UIM Cartridge and Technology Pack Guide.

About Cartridge Upgrades

Cartridges can be upgraded. For example, the cartridges in a technology pack might be upgraded for a new release. The upgrade process occurs in Design Studio and begins automatically when you open a cartridge that was built in a previous release.

When upgrading a cartridge that is dependent on another cartridge, you must upgrade the dependent cartridge first. During the upgrade process, all dependent cartridges must exist in the workspace to ensure that the upgrade process can convert all cartridges in the correct order.

For instructions on how to upgrade a cartridge, see the Design Studio Help.

About Development Work Within Design Studio

Any custom code that extends UIM can be written in Design Studio by creating a cartridge and adding custom Java code to the cartridge. In the Java perspective Package Explorer view, you can create package structures and Java source files as needed. UIM JAR files are not automatically pulled into the project library list within Design Studio. Therefore, depending on what the custom code references, you may need to update the project library list to include external JAR files. See "Configuring the Project Library List" for more information.
About the Developer-Facing Inventory Menu Options

From the Studio menu, select New, then select Inventory, then select Administration to see the developer-facing options that are available in Design Studio with the installation of the Inventory feature. The options are:

- Sequence Specification
- Extension Point
- Enabled Extension Point
- Ruleset
- Ruleset Extension
- Ruleset Extension - Global
- Inventory Group Specification

The Sequence Specification option is described in the following sections. The remaining developer-facing options, with the exception of Inventory Group Specification, are described in Chapter 8, "Extending UIM through Rulesets". For information on Inventory Group Specification, see UIM Concepts and the Design Studio Help.

Understanding the Sequence Specification

A sequence is a unique, generated number that is used as an identifier. Sequences can be used alone, or concatenated with other attributes to create a larger identifier, such as a connection ID. For example, in the following connection ID, a sequence can be generated to represent the bolded facility designator:

101/T1/PLANTXXAK01/IRVGTXXAK1.

The Sequence specification defines criteria for a sequence. Figure 2–8 shows the Sequence Specification editor Properties tab where the criteria of Minimum Value, Maximum Value, and Increment Value are defined.

For instructions on how to create a Sequence Specification, see the Design Studio Help.
The Sequence specification can be used:

- In custom code to set an identifier
- With the Entity Identifier specification to set the entity identifier

Using the Sequence Specification in Custom Code
UIM provides the SequenceGenerator interface, which is a mechanism for generating sequences. The interface exposes methods that generate three types of sequences:

- Global Sequence
- Context-Based Sequence
- Specification-Based Sequence

Global Sequence
A global sequence is a generated number that starts at 1, is incremented by 1, and is unique. When writing custom code, you can obtain a global sequence by calling the following method on SequenceGenerator:

```java
public long next()
```

This method returns a global sequence (a number that is unique across all calls to the method).

A global sequence does not use the Sequence specification.

Context-Based Sequence
A context-based sequence is a generated number that starts at 1, is incremented by 1, and is unique within a given context. When writing custom code, you can obtain a context-based sequence by calling the following method on SequenceGenerator:

```java
public long next(String context)
```

This method returns a context-based sequence (a number that is unique across all calls to the method that supply the same context).
A context-based sequence does not use the Sequence specification.

**Specification-Based Sequence**
A specification-based sequence is a generated number that starts at 1, is incremented by 1, and is unique within a given context. Additionally, the number is based on criteria that is defined by a Sequence specification (minimum value, maximum value, and increment value). When writing custom code, you can obtain a specification-based sequence by calling the following method on SequenceGenerator:

```
public long next(String sequenceSpecName, String context)
```

This method returns a specification-based sequence (a number that is unique across all calls to the method that supply the same context, and that is based on the sequence criteria as defined by the supplied Sequence specification).

When calling the next method to get a specification-based sequence, your custom code must be hard-coded with the Sequence specification name. Also, the custom cartridge that defines the Sequence Specification must be deployed prior to running the custom code.

---

**Note:** SequenceGenerator operates outside of a transaction. So, if the transaction gets rolled back, any IDs created are not rolled back. (Oracle native sequences work the same way.)

---

**Using the Sequence Specification with the Entity Identification Specification**
The Sequence specification can also be used with the Entity Identification specification to obtain a specification-based sequence to set the entity identifier.

In the metadata, an entity can be defined to have the Entity Identification pattern, as described in Oracle Communications Information Model Reference. When an entity defines this pattern, the entity defines the `id` attribute (entity identifier), which is unique across a specific entity type. For example, EquipmentHolder is defined with the Entity Identification pattern. So, each equipment holder defines the `id` attribute, and the `id` attribute value is unique across all equipment holders.

---

**Note:** The `id` attribute differs from the `entityId` attribute: Only entities that are defined with the `id` attribute can be defined with the Entity Identification pattern. All entities define the `entityId` attribute, which is always unique across the entire database.

---

For entities that define the Entity Identification pattern, the specification editor includes the **Enter Id Manually** check box, as shown in Figure 2–9. When the check box is selected, it indicates that when creating an instance of the specification in UIM, you must manually enter the `id` attribute value through the UIM UI.

In Figure 2–9, the **Enter Id Manually** check box is selected, and the option to select an Entity Identification Specification is disabled.
When the Enter Id Manually check box is deselected, it indicates that when creating an instance of the specification in UIM, the id attribute value is automatically generated. In Figure 2–10, the Enter Id Manually check box is deselected, and the option to select an Entity Identification Specification is now enabled.

When the id attribute value is to be automatically generated (Enter Id Manually is deselected, which is the default), UIM uses the SequenceGenerator interface to obtain a sequence that is used to set the id attribute value. You can optionally format the id attribute value by selecting an Entity Identification specification. When you click Select, a list of all previously defined Entity Identification specifications displays. If no Entity Identification specification is selected, a context-based sequence is generated and used to set the id attribute value. For example, in this scenario the context is the
Equipment Holder entity type, resulting in the sequence being unique across all equipment holders.

**Figure 2–11** shows the Entity Identification Specification Properties tab, where you can define the sequence format. For instructions on how to create an Entity Identification Specification, see the Design Studio Help.

![Figure 2–11 Entity Identification Specification Editor](image)

Based on the Entity Identification specification, the \textit{id} attribute value is generated as:

\[
\text{prefix} + \text{sequence} + \text{suffix}
\]

where:

- prefix is the \textit{Prefix} value specified by the Entity Identification specification. Specifying a prefix is optional.
- sequence is a unique sequence value based on Sequence specification criteria. Specifying a Sequence Specification is required.
- suffix is the \textit{Suffix} value specified by the Entity Identification specification. Specifying a suffix is optional.

**Note:** You can choose to not specify the prefix or suffix. For example, to have your \textit{id} attribute value incremented by 100, define an Entity Identification specification with no prefix or suffix, and specify a Sequence specification that defines an increment value of 100.

### Additional Tools

Third-party tools such as Ant and JBoss are used to extend UIM:

- **Ant** is used to extend the UIM data model, Web services, and user interface.

  Ant is an open source software tool for automating a build process. Ant uses XML to describe a build process and its dependencies. When extending the UIM data...
model, Web services, or user interface, Ant targets are run from within Design Studio. See "Running Ant Targets" for more information.

- **JBoss** is used to extend UIM through rulesets.

  JBoss Rules is an open source rules engine for accessing, changing, and managing business policies. When extending UIM using rulesets, Oracle recommends that you install the JBoss Eclipse plug-in in Design Studio. The plug-in provides JBoss-specific menu options, JBoss online Help, and a JBoss Rule editor. See "Using the JBoss Eclipse Plug-In" for more information.

### Installing Ant

Installing Ant is described in several procedures: Downloading, installing, configuring, and running.

#### Downloading Ant
To download Ant:

1. Go to the following Web site:
   
   http://archive.apache.org/dist/ant/binaries

2. Scroll down and click the `apache-ant-1.7.1-bin.zip` link.
   
   The File Download window appears.

3. Click **Save**.
   
   The Save As window appears.

4. Navigate to a local directory and click **Save**.

#### Installing Ant
To install Ant:

1. In your local hard drive root directory, or in your **Program Files** directory, create a new directory named **ant**.

2. Navigate to the local directory that contains the downloaded `apache-ant-1.7.1-bin.zip` file.

3. Open the `apache-ant-1.7.1-bin.zip` file.

4. Extract the `apache-ant-1.7.1-bin.zip` file contents to the **ant** directory.

#### Configuring Ant
To configure Ant:

1. From the Windows **Start** menu, select **Control Panel**, then select **System**.
   
   The System Properties window appears.

2. Click the **Advanced** tab.

3. Click **Environment Variables**.
   
   The Environment Variables window appears.

4. Define a new system variable named ANT_HOME:
   
   a. In the System Variables section, click **New**.
      
      The New System Variables window appears.
b. In the **Variable name** field, enter ANT_HOME.

c. In the **Variable value** field, enter the path to the extracted `apache-ant-1.7.1` directory. For example, C:/ant/apache-ant-1.7.1 or C:/Program Files/ant/apache-ant-1.7.1.

d. Click **OK**.

5. Update the existing system variable Path:
   a. In the System Variables section, select **Path**, then click **Edit**.
      The Edit System Variables window appears.
   b. In the **Variable value** field, add the path to the **bin** directory of the extracted `apache-ant-1.7.1` directory.
      For example, C:/ant/apache-ant-1.7.1/bin or C:/Program Files/ant/apache-ant-1.7.1/bin.
   c. Click **OK**.
      The Environment Variables window appears.

6. Click **OK**.
   The System Properties window appears.

7. Click **OK**.

**Running Ant Targets**

This procedure is applicable only if the cartridge you imported contains a `build.xml` file that defines Ant targets.

To run an Ant target within Design Studio:

1. Open the Java perspective.
   For instructions on how to open the Java perspective, see the Design Studio Help.
2. From the **Window** menu, select **Show View**, then select **Ant**.
   The Ant window appears.
3. Within the Ant window, right-click and select **Add Buildfiles**.
   The Buildfile Selection window appears.
4. Navigate to and select the file that contains the Ant target you plan to run.
5. Click **OK**.
   The Ant targets defined in the selected file appear in the Ant window, as shown in Figure 2–12.
6. Double-click a target.
   The Ant target runs.

**Installing the JBoss Eclipse Plug-In**

*Note:* UIM 7.2 uses an older version of JBoss.

Installing the JBoss Eclipse plug-in is described in several procedures: Downloading, installing, configuring, and using.

**Downloading the JBoss Eclipse Plug-In**

To download the JBoss Eclipse plug-in:

1. Go to the following Web site:
   
   http://repository.jboss.org/jbossrules/3.0.4/lib/

2. Click the `jbossrules-ide-3.0.4-bin.zip` link.
   The File Download window appears.

3. Click **Save**.
   The Save As window appears.

4. Navigate to a local directory and click **Save**.

**Installing the JBoss Eclipse Plug-In**

To install the JBoss Eclipse plug-in:


2. Navigate to the local directory that contains the downloaded `JBossIDE-Drools-3.0.4.zip` file.

3. Open the `jbossrules-ide-3.0.4-bin.zip` file.

4. Extract the **plugins** directory to the same local directory.

5. Double-click the extracted **plugins** directory to open it.
The `org.drools.ide_3.0.4.jar` file resides within the extracted `plugins` directory.

6. Copy the `org.drools.ide_3.04.jar` file to the `Eclipse_Home/plugins` directory, where `Eclipse_Home` is the location of your Eclipse installation.

You should now see the `org.drools.ide_3.0.4.jar` file in the `Eclipse_Home/plugins` directory, along with the numerous other plug-ins that came with the Eclipse installation.

7. Open Design Studio.

After you install the plug-in, a JBoss Rules icon displays on the Design Studio tool bar. See "Using the JBoss Eclipse Plug-In" for more information.

### Configuring the JBoss Eclipse Plug-In

After you install the JBoss Eclipse plug-in and open Design Studio, you must configure Design Studio to recognize and utilize the plug-in. Configuring the JBoss Eclipse plug-in is described in two procedures: Configuring the preference, and configuring the `.project` and `.classpath` files.

#### Configuring the Preference

To configure the preference in Design Studio:

1. From the menu, select **Window**, then select **Preferences**.

   The Preferences window appears.

2. In the navigation panel, expand **General**, expand **Editors**, and select **File Associations**.

3. In the File types section, ensure that the `.drl` file type is present in the list.

   If the `.drl` file type is not present in the list, add it.

4. In the Associated editors section, select **Rule Editor** and click **Default**.

5. Click **OK** to close the Preferences window.

#### Configuring the `.project` and `.classpath` Files

To configure the `.project` and `.classpath` files in Design Studio:

1. Open the Java perspective, and open the Navigator view.

   **Note:** You must be in the Navigator view to see the `.project` and `.classpath` files; the default Package Explorer view does not show these files.

2. In the Navigator view, open the `.project` file.

3. In the `.project` file editor, click the **Source** tab located at the bottom of the file editor.

4. Within the XML, create a new `<buildCommand>` element by copying and pasting one of the existing `<buildCommand>` elements.
5. Modify the copied `<buildCommand>` element to be:

   ```xml
   <buildCommand>
     <name>org.drools.ide.droolsbuilder</name>
     <arguments>
     </arguments>
   </buildCommand>
   ```

6. Save the `.project` file.

7. In the Navigator view, open the `.classpath` file.

8. In the `.classpath` file editor, click the Source tab located at the bottom of the file editor.

9. Within the XML, create a new `<classpathentry>` element by copying and pasting one of the existing `<classpathentry>` elements.

10. Modify the copied `<classpathentry>` element to be:

    ```xml
    <classpathentry kind="con" path="DROOLS/JBoss Rules"/>
    ```

    **Note:** There is a space between "JBoss" and "Rules".

11. Save the `.classpath` file.

**Using the JBoss Eclipse Plug-In**

The JBoss Eclipse plug-in provides JBoss-specific menu options, JBoss online Help, and a JBoss Rule editor.

**JBoss Menu Options**

After you install the plug-in, a JBoss Rules icon displays on the Design Studio tool bar, which has the following menu options:

- New Rule Project
- New Rule Resource
- New Domain Specific Language
- New Decision Table

See the JBoss online Help for information on using these JBoss-specific menu options. See "JBoss Online Help", below, for information on accessing the JBoss online Help.

**JBoss Online Help**

After you install the plug-in, the JBoss online Help is available in Design Studio.

To access the JBoss online Help in Design Studio:

1. From the menu, select Help, then select Help Contents.
   The Help window appears.

2. In the Content panel, expand Drools, then expand Reference Manual.
   The Reference Manual lists various topic links to select.
**JBoss Rule Editor**

After you install and configure the plug-in, the JBoss Rule editor works in a Drools file opened within Design Studio. The Rule editor catches syntax errors while writing a ruleset, and compiles a ruleset prior to deploying the cartridge that contains the ruleset.
Using the Persistence Framework

This chapter provides information on using the persistence framework, which moves program data (in memory objects) to and from a permanent data store (the database). The persistence framework also manages the database and manages the mapping between the database and the objects.

You use the persistence framework when extending Oracle Communications Unified Inventory Management (UIM). For example, custom rulesets or custom Web services typically have code that reads or updates the database, which is done using the persistence framework. So, UIIM custom code developers need to be familiar with the contents of this chapter.

This chapter contains the following sections:

- About the Persistence Framework Foundation
- Understanding Persistence Framework Concepts
- Persistence Framework Classes and API Methods

About the Persistence Framework Foundation

The persistence framework is built on top of EclipseLink, which implements Java Persistence API (JPA) technology. Functional extensions employ standard Java practices.

This chapter does not replace the EclipseLink or JPA development guides. Both technologies are covered in greater detail at the following Web sites:

**JPA Specifications**
http://wiki.eclipse.org/EclipseLink/Specs

**EclipseLink**
http://wiki.eclipse.org/EclipseLink

**JPA**
http://www.eclipse.org/eclipselink/jpa.php

---

**Note:** Documentation on third-party software products is limited to the information needed to use the UIM persistence framework. If you need additional information on a third-party software application, consult the documentation provided by the product’s manufacturer.
Understanding Persistence Framework Concepts

The persistence framework employs the concepts of eager and lazy fetching and of managed and non-managed entities, as described in the following sections.

Eager and Lazy Fetching

Note: Information on eager and lazy fetching can be found on the Oracle Technology Network Web site at:

http://www.oracle.com/technetwork/articles/javase/index-138213.html

A `fetchType` of `eager` means that a persistence provider loads the attribute of an entity along with the entity, while a `fetchType` of `lazy` is a hint to the provider that the attribute need not be fetched along with the entity. This means that even though you may specify the `fetchType` as `lazy`, the persistence provider may choose to load the attribute eagerly.

By default, all relationships are configured as lazy loading in the metadata, and all basic attributes are configured as eager fetched in the metadata. To configure an attribute as lazy loading in the metadata, set the `<lazy>` attribute to `true`. For example:

```xml
<entity type="cim:DataTypes"
            interface="oracle.communications.platform.entity.DataTypes">
  <attribute name="clobString" lazy="true"/>
</entity>
```

The result is a generated annotation within the DataTypesDAO.java class. For example:

```java
@Basic(fetch=FetchType.LAZY) private java.lang.String clobString;
```

If a field is configured as lazy loading, and you want to eager fetch it when the entity is retrieved from the database, use the Finder.addEagerFetchField() method. For example:

```java
finder = PersistenceHelper.makeFinder();
finder.setResultClass(TelephoneNumber.class);
finder.setJPQLFilter("o.id = :tnId");
finder.setParameter("tnId",'88888888'");
finder.addEagerFetchField(EagerFetch.LEFT_FETCH,"o.specification");
Collection<TelephoneNumber> tns = finder.findMatches();
finder.close();
```

The previous example shows a fetch mode of `LEFT_FETCH`. The persistence framework supports the following fetch modes for an eager fetch:

- **BATCH**: Batch reading may require more than one trip to the database but is usually more efficient than a join fetch, especially join fetches that involve collection relationships. Batch reading configures the query to optimize the retrieval of the related objects, and the related objects for all the resulting objects are read in a single query (instead of multiple queries).
- **FETCH**: This fetch mode uses an inner join.
- **LEFT_FETCH**: This fetch mode uses an outer join.
Managed and Non-Managed Entities

A persistence context is a set of entities such that for any persistent identity there is a unique entity instance. Within a persistence context, entities are managed. An entity manager controls life cycles and accesses data store resources.

When a persistence context ends, previously managed entities become non-managed. A non-managed entity is no longer under the control of the entity manager and no longer has access to data store resources. The major difference between an entity that is managed and an entity that is non-managed is:

- When an entity is managed, the object is connected to the database and changes made to the object are reflected in the database when committed, or flushed in a transaction.
- When an entity is non-managed, the object is not connected to the database, so changes are never applied to the database.

The non-managed object can be stale, which can cause you to receive the OptimisticLockingException when calling the EclipseLink attach() method. In this case, discard the stale non-managed object, retrieve the new object from the data store, and perform any update operation against the new version.

When a transaction already exists and an entity is explicitly retrieved from the database, the entity is managed. There is no need to eager fetch the entity attributes because the attributes and relationships can be lazy-loaded while the transaction is still active.

When no transaction exists, the entity becomes non-managed. EclipseLink supports the lazy loading of relationships of a non-managed entity. EclipseLink also supports the lazy loading of primitive attributes such as String.

---

**Note:** If an entity is serialized and then deserialized, such as sent through a remote EJB interface or Web service, the relationships can not be lazy loaded, and an eager fetch must be used to access the relationships. Alternatively, the application can start a transaction to make the entity managed. When the entity is managed, the attributes and relationships can be accessed directly, and the eager fetch is not required.

---

When an entity is created, it is neither managed nor non-managed. The entity status is Transient because there is no representation of the entity in the database yet. All relational and collection attributes on a transient entity are available (that is, if it is null, then it is really null). When the transient entity is passed to a ManagerX.createX(..) method, the entity is persisted into the database. The entity is persisted by reference. The entity life-cycle status is changed from Transient to Persistent-New. There is a copy of the entity created. When the transaction is committed, the entity becomes non-managed.

Usually, the UI code keeps the non-managed entity in the session so that it can be updated. Sometimes, the UI code does an explicit Find to retrieve and store a non-managed entity in a session. You can store the non-managed entity in a session to avoid table locks on the database. The client code performs a set on the detached entity. When the updated entity is passed into the ManagerX.updateX(..) method, the manager persists the changes by using the EclipseLink attach() method. The current implementation of EclipseLink attach makes a copy of the entity and persists the changes. The new copy represents the managed entity, making the non-managed entity obsolete. For example, the createEntity() and updateEntity() methods each
return a new copy of the entity. The returned copies are managed entities, making the
original non-managed entity obsolete.

\[
A = \text{createEntity}(A)
\]

\[
A = \text{updateEntity}(A)
\]

**Persistence Framework Classes and API Methods**

All of the persistence framework classes covered in this section expose API methods
that you can use when extending UIM. For example, you may want to add additional
validations to existing UIM functionality, or add additional processing.

---

**Note:** For information on the classes described in this section,
including a listing of method names, arguments, and returns, see the
Javadoc. For instructions on how to access the Javadoc, see "Javadoc
Documentation".

---

**PersistenceManager**

Package: oracle.communications.platform.persistence

PersistenceManager generically manages entities defined in the *-entities.xml files.
This manager provides methods to:

- Create, update, and delete an entity or entities
- Check whether an EntityManager is invoked from a Java Transaction API (JTA)
  context
  (JTA specifies standard Java interfaces between a transaction manager and the
  parties involved in a distributed transaction system: The resource manager, the
  application server, and the transactional applications. A JEE application may use
  JTA, but a standalone JSE application does not.)
- Set and get the logging level

This class is a wrapper for the methods defined in the standard
javax.persistence.EntityManager class.

**TypeRegistry**

Package: oracle.communications.platform.persistence

TypeRegistry is a generated class that extends TypeRegistryBase. As part of the entity
code generation process, each entity is added to a class list managed by TypeRegistry.
TypeRegistry provides convenient methods to get a data access object (DAO)
implementation class for each entity. **Table 3–1** contains a list of the methods defined in
the TypeRegistryBase class.

**Table 3–1 TypeRegistryBase APIs**

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classFor(Class)</td>
<td>Gets the concrete class, which implements the given inventory entity class.</td>
</tr>
<tr>
<td>interfaceFor(Class)</td>
<td>Gets the inventory entity interface, based on the concrete class.</td>
</tr>
<tr>
<td>classForDiscriminator(String)</td>
<td>Gets the entity implementation class, based on the discriminator.</td>
</tr>
</tbody>
</table>
Persistence Framework Classes and API Methods

Finder

Package: oracle.communications.platform.persistence

Finder provides methods for querying entities based on simple or complex search criteria. It has convenience methods that set up query parameters and fetch properties. Convenient find methods are provided; however, a complex Java persistence query language (JPQL) query can also be built iteratively.

Finder provides the most frequently used query mechanism. Additional query complexity that can be reused should be incorporated into the entity managers instead. The entity managers should then use Finder for building the queries, or use JPA directly. See "Entity Managers" for more information.

Finder defines methods that enable you to:

- Get an entity based on the entity key
- Refresh an entity or a collection of entities
- Find an entity or entities based on various options, such as name, entity, or ID
- Define a JPQL statement
- Run the defined JPQL statement
- Reset the Finder, which resets all query parameters to null

These methods are further explored in the following sections.

Defining JPQL Statement Methods

The Finder class provides numerous methods that you can use to define a JPQL statement. By using these methods you can:

- Set the result class to query
- Add a join expression
- Set filters, such as a where clause or min/max
- Add an attribute to specify the result set be returned in ascending or descending order
- Set a range to filter the result set
- Add and set parameters
- Declare variables
- Add and set variables
- Add hints, which are JPA-specific
- Add eager fetch fields
- Clear eager fetch fields

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discriminatorForClass(Class)</td>
<td>Gets the discriminator, based on the implementation class.</td>
</tr>
</tbody>
</table>

Table 3–1 (Cont.) TypeRegistryBase APIs
Finder.find() and Finder.findMatches() Methods

The Finder.find(Class<E> candidateType, String filter) method is a convenient method to use because it does not take into consideration any parameters you set on Finder before you call that method. To include parameters, use the Finder.find(Class<E> candidateType, String filter, String [] paramNames, Object [] params) method.

Alternatively, you can build the parameters list using Finder beforehand, then use the findMatches() method. The findMatches() method uses the parameters you set.

Table 3–2 lists some of the commonly used methods defined in the Finder class.

### Table 3–2 Finder APIs

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>Overloaded method that finds an entity or entities based on various arguments, such as entity type, the current filter setting on Finder, a list of the current parameters set on Finder, etc.</td>
</tr>
<tr>
<td>findByName, findById, findByEntity</td>
<td>Various methods that find an entity or entities based on name, ID, or entity type.</td>
</tr>
<tr>
<td>findMin and findMax</td>
<td>Finds the minimum or maximum value based on entity type and the value of min or max, which is used by the method to call Finder.setJPQLFilter().</td>
</tr>
<tr>
<td>findMatches</td>
<td>Overloaded methods that finds an entity or entities based on various arguments, such as an Oracle Text search String, and other arguments that you set.</td>
</tr>
<tr>
<td>findByJPQL</td>
<td>Finds a result set based on a String argument representing a JPQL statement that you define.</td>
</tr>
<tr>
<td>executeUpdateJPQL</td>
<td>Executes an update based on a String argument representing a JPQL statement that you define. This method returns the number of updated entities.</td>
</tr>
<tr>
<td>findByNativeSQL</td>
<td>Finds a result set based on a String argument representing a native SQL statement that you define.</td>
</tr>
<tr>
<td>executeUpdateNativeSQL</td>
<td>Executes an update based on a String argument representing a native SQL statement that you define. This method returns the number of updated entities.</td>
</tr>
<tr>
<td>get</td>
<td>Overloaded method that gets an entity based on entity type and entity key, or based on entity type, entity key, and whether or not the entity is a valid entity.</td>
</tr>
<tr>
<td>refresh</td>
<td>Overloaded method that refreshes the given entity, or the given collection of entities.</td>
</tr>
</tbody>
</table>

PersistenceManager.refresh(), PersistenceManager.attach(), and PersistenceManager.connect() Methods

The basic differences between Finder.refresh(), PersistenceManager.attach(), and PersistenceManager.connect() are:

- Refresh() refreshes the entity content back to the state of the database, and discards any changes made to the entity. If the entity is managed, the refresh API retrieves a copy from the database to refresh the managed entity. If the entity is non-managed, the refresh API makes the entity managed. Any changes previously made to the managed or non-managed entity are discarded. The refresh API returns the reference to the managed entity.
Persistence Framework Classes and API Methods

- Attach() makes the non-managed entity managed, and retains any changes made to the entity. If the entity is already managed, the attach API does nothing in terms of attaching the entity to the database. If the entity is non-managed, the attach API makes the entity managed. Any changes previously made to the managed or non-managed entity are sent to the database by EclipseLink when the transaction is committed or flushed. The attach API returns the reference to the managed entity.

- Connect() makes the non-managed entity managed, and discards any changes made to the entity. If the entity is already managed, the connect API does nothing in terms of connecting the entity to the database. If the entity is non-managed, the connect API makes the entity managed. Any changes previously made to the managed or non-managed entity are discarded. The connect API returns the reference to the managed entity.

Refresh() does a get from the database. refresh() takes a detached entity, connects it to the database, but does not merge the entity attribute into the database. Refresh() re-retrieves the entity even when it is already attached.

Attach() takes a detached entity and merges its data into the database. The operation fails if the detached entity is stale. When attach attaches the detached entity to the database, it also merges the entity attribute values into the database. Attach() ignores the entity if it is already attached.

If you do not intend to merge the entity attributes of an entity in the database, do not use attach(). If you do, you may be updating an attribute in the database. Also, the last modified fields for the entity are updated, and the entity version is updated.

**Note:** Using attach() may cause an OptimisticLockVerificationException because it tries to merge values in the database. If the detached entity is a stale entity (some other code thread has modified the same entity and has incremented the entity version), using attach again causes this exception.

InventoryFinder

Package: oracle.communications.inventory.api.framework.persistence

InventoryFinder extends Finder and provides a few additional methods, as described in the following table.

<table>
<thead>
<tr>
<th>Table 3-3 InventoryFinder APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API</strong></td>
</tr>
<tr>
<td>find(String queryExpression, Object... parms);</td>
</tr>
<tr>
<td>findTotalCounts();</td>
</tr>
</tbody>
</table>

PersistenceHelper

Package: oracle.communications.platform.persistence

PersistenceHelper is a generated class that provides factory methods to get an instance of an entity manager, the TypeRegistry, a Finder, an InventoryFinder, or the PersistenceManager.
Persistent

Package: oracle.communications.platform.persistence

All persistent entities implement the Persistent API. It provides convenience methods for determining the state of an entity. These methods are all read-only; so, the methods can run whether or not there is an active transaction. The following methods are defined in the Persistent API and are available on all entities.

```java
public Class getEntityType();
public String getOid();
public long getEntityId();
public String getEntityClass();
public int getEntityVersion();
public boolean isEntityIdValid();
public Identifier makeIdentifier();
public void makeTransient();
public boolean isPopulated(String fieldName);
public void unpopulate(String fieldName);
public boolean isTransient();
public boolean isPersistent();
public boolean isTransactional();
public boolean isNew();
public boolean isDirty();
public boolean isDeleted();
public boolean isDetached();
public <E extends Persistent> E connect();
public <E extends Persistent> E refresh();
public <E extends Persistent> E attach();
public String getEntityDescription();
```

Entity Managers

Entity managers are not part of the persistence framework: they are additional managers that use the persistence framework to support the overall application logic. An entity manager manages the database tables for a specific functional area. For example, EquipmentManager manages the Equipment table, but it also manages EquipmentHolder, PhysicalPort, PhysicalConnector, PhysicalDevice, and so forth.

Defining Entity Managers

Entity managers are defined in the metadata by the `<manager>` element and `<interface>` attribute. Example 3–1 is an excerpt from the `uim-equipment-entities.xml` file:

```xml
<manager
  interface="oracle.communications.inventory.api.equipment.EquipmentManager"
  class="oracle.communications.inventory.api.equipment.impl.EquipmentManagerImpl"/>
```

Every entity manager defined in the metadata has a corresponding entity manager and implementation of the manager. So, based on Example 3–1, the following classes exist:

- EquipmentManager
- EquipmentManagerImpl

Entity managers are not generated classes; however, the factory methods in PersistenceHelper that allow for the instantiation of the managers are generated. These factory methods are generated based on the metadata definition.
The relationship of entity to entity manager is not one-to-one. For example, in `ocim-equipment-entities.xml` file, there are a number of entities defined, and each entity defines its own entity interface (which differs from a manager interface). An entity interface defines the getter and setter methods for data defined for the entity. Example 3–2 is an excerpt from the `ocim-equipment-entities.xml` file that shows two entity definitions. The definitions include the interface that is defined for an entity (not for a manager).

**Example 3–2  ocim-equipment-entities.xml**

```xml
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true">
  . . .
<entity type="ocim:EquipmentHolder"
interface="oracle.communications.inventory.api.entity.EquipmentHolder"
accessControlled="true">
```

**Entity Manager Implementation Inheritance Structure**

The PersistenceManagerBean class is the common base class for all entity manager implementations, and all entity manager implementations extend BaseInvManager. TransitionManagerBean is another layer of inheritance. The inheritance structure of all entity manager implementations is shown below. The following sections discuss each of these classes.

```plaintext
PersistenceManagerBean  |
| TransitionManagerImpl  |
| BaseInvManager         |
| EntityNameManagerImpl  |
```

**Note:** In some cases, there are additional layers between BaseInvManager and EntityNameManagerImpl, but these four layers of inheritance are always present. An example that has additional layers is LogicalDeviceManagerImpl.

Specifying the `<managedBy>` attribute for an entity in the metadata allows the entity manager to override the default behavior of the following methods:

- TransitionManager.transition(LifeCycleManaged, Object)
- PersistenceManagerBean.completeCreate(Persistent)
- PersistenceManagerBean.completeUpdate(Persistent)
- PersistenceManagerBean.completeDelete(Persistent)

**PersistenceManagerBean**

Package: oracle.communications.platform.persistence.impl
PersistenceManagerBean is the common base class for all entity managers. It provides convenient create, read, update, and delete (CRUD) methods for managing entity persistence. It also provides methods to attach an object to the persistence engine, and methods to test for object equality. Developing entity managers requires the use of the PersistenceManagerBean class. It defines all the persistence-related methods used by entity managers, it hides the JPA standard PersistenceManager, and it wraps the persistence logic required.

**TransitionManagerImpl**

Package: oracle.communications.inventory.api.common.impl

TransitionManagerImpl transitions an entity’s business and object states, which is only applicable for entities defined as life-cycle managed in the metadata. This layer of inheritance is always in place, but it is used only by life-cycle managed entities. See Chapter 5, "Extending Life Cycles" for more information.

**BaseInvManager**

Package: oracle.communications.inventory.api.common

BaseInvManager extends PersistenceManagerBean and provides application-specific logic to the PersistenceManagerBean methods. All entity manager classes must extend this class.
Extending the Data Model

This chapter provides information on how to extend the Oracle Communications Unified Inventory Management (UIM) data model through additions to the metadata. The information describes statically extending the UIM data model, which can result in backwards compatibility issues. See "Backwards Compatibility" for the implications regarding this type of extension. Another option is to dynamically extend UIM through characteristics. For information about characteristics, see UIM Concepts.

This chapter contains the following sections:

- About the UIM Data Model
- About the Metadata Files
- Understanding Metadata File Content
- Extending the Data Model Through the Metadata Files
- Applying Metadata Static Extensions
- More on Entity Definitions

About the UIM Data Model

The UIM data model extends the Oracle Communications Information Model (Information Model). The Information Model is shared by several Oracle Communications products, including UIM. The data model for each product is defined by a collection of XML and XSD files called metadata. Some metadata files are defined by the Information Model, and some metadata files are defined by the product. Regardless of their origin, all metadata files are part of the product installation.

Metadata files define:

- Tables and columns that comprise the UIM database
- Entities and attributes that correspond to the tables and columns
- Enumerated data
- Life-cycle state transition data
- Native sequences
- Tags that govern the definition of an entity, entity manager, enumeration, and native sequence

The metadata files, in conjunction with UIM-provided Ant targets, are used to regenerate the database tables and the corresponding entity Java source files. Another UIM-provided Ant target compiles the entity source files into entity class files and rebuilds the inventory.ear file to include the entity class files.
About the UIM Data Model

Note: The generated entities and compiled source files reside in the inventory.ear file upon installation. The Ant targets for regeneration and compilation are needed only if you statically extend the data model.

About Entities

Entities are Java representations of UIM data and are used to persist data in the database. For example, in the UIM database, the TelephoneNumber table defines several columns of data including ID, name, and description, each of which are defined with a data type of String. For each table, there is a corresponding Java entity class, such as TelephoneNumber.class, that is compiled from a Java source file, such as TelephoneNumber.java. Each source file defines data attributes with the same names and same data types as the data columns defined for the corresponding table. Each row in the TelephoneNumber table is persisted by an instance of TelephoneNumber.class. The TelephoneNumber table name, column names, and data types correspond directly to TelephoneNumber.java attributes and data types because both are generated from the same entity definition in the metadata.

About Entity Capabilities

A capability is a design pattern that is applied to an entity, such as enabling an entity to be life-cycle managed. For example, an entity that is life-cycle managed progresses through a succession of states during the course of its life. For life-cycle managed entities, UIM tracks two states: administrative state and object state. To support this capability, an entity must define the adminState and objectState attributes. Rather than define these attributes for every entity that supports this pattern, the capability is declared as part of the entity definition. As a result of this declaration, the adminState and objectState attributes are generated for the entity. So, a capability that is declared in an entity definition can result in the generation of attributes, as well as the generation of any related entities that support the capability, neither of which are explicitly defined in metadata.

When extending the data model, you can extend existing entities to declare capabilities, or you can create new entities that declare capabilities. See "Understanding Entity Capability Definitions" for more information.

About Entity Relationships

Entity relationships describe how an entity relates to other entities. Entity relationships can be defined as one-to-one, one-to-many, many-to-one, or many-to-many. An entity definition can specify a relationship to an explicitly-defined entity, or to a capability-generated entity.

When extending the data model, you can extend existing entities to define additional relationships, or you can create new entities that define relationships to other entities. See "Understanding Entity Relationship and Collection Definitions" for more information.

About Entity Managers

Entity managers are Java classes that manage a specified set of database tables for a specific functional area. For example, the EquipmentManager class manages the Equipment table, but it also manages other tables in the equipment functional area such as EquipmentHolder, PhysicalPort, PhysicalConnector, and PhysicalDevice.
Entity manager class files are part of UIM and work with the Persistence Framework in managing the UIM database. The metadata defines entity manager interfaces, citing existing entity manager classes in the definition.

When extending the data model, you can extend existing entities to be managed by a specific entity manager, or you can create new entities which requires the creation of new entity managers. See “Understanding Entity Manager Definitions” for more information.

About Entity ID Sequencing

The Oracle database provides a mechanism for obtaining a generated unique number known as a sequence. This mechanism is called an Oracle native sequence. Each UIM entity defines the `entityId` attribute, which the persistence framework uses to uniquely identify an object. In previous releases, the `entityId` attribute value was set using just one Oracle native sequence, resulting in the value being a unique number across the entire database. However, this scenario does not provide for optimal processing performance.

To improve processing performance, UIM now defines several additional Oracle native sequences. Each native sequence is given a sequence generator name that is based on a functional area, such as ConnectivitySeqGen, EquipmentSeqGen, and TelephoneNumberSeqGen. The native sequences and corresponding sequence generator names are defined in the metadata, and an entity definition may specify a sequence generator, indicating the native sequence that the entity is to use when setting the `entityId` attribute value for the entity. For example, the Pipe, PipeTerminationPoint, and PipeRel entity definitions specify the ConnectivitySeqGen sequence generator. In this scenario, the `entityId` values for the entities that use a specific native sequence are unique; `entityId` values are not unique across the entire database.

To keep this information centrally located, all native sequences and their corresponding sequence generator names are defined in the same type of file. Additionally, all entity definitions that specify a sequence generator are extended to specify it in the same type of file. Entity definitions that do not specify a sequence generator name use the default native sequence provided by the database.

Depending on your implementation of UIM, you may determine that you have heavily-used entities that, upon installation, use the default database native sequence. You can extend the data model by extending your heavily-used entity definitions to specify one of the UIM-defined sequence generators. You can also define your own native sequences and corresponding sequence generator names in the metadata, and extend your heavily-used entity definitions to specify one of your new sequence generators. See “Understanding Native Sequence Definitions” for more information.

---

**Note:** The remainder of this chapter refers only to entities and attributes, rather than to tables and columns and corresponding entities and attributes, all of which are generated from entity definitions in the metadata.

---

About the Metadata Files

The metadata files are contained in the `UIM_Home/cartridges/tools/ora_uim_entity_sdk_cartproj.zip` file. Within the ZIP file, the metadata files are located in the `src/ora_uim_poms.jar` file unless otherwise noted.

The metadata files include:
**ocim-*.***
File names that start with `ocim-*.*` indicate that the file is part of the Information Model. These files are common to several Oracle Communications products, including UIM.

**uim-*.***
File names that start with `uim-*.*` indicate that the file defines UIM-specific entities, entity attributes, entity managers, enumerations, and transitions.

**-*entities.xml**
File names that end with `-entities.xml` indicate that the file defines entities for a specific area, such as service, equipment, or connectivity. Entities are defined through XML tags that are governed by the `package.xsd` file and the `*-plugin.xsd` files, which are described below. Any tags used in an entity definition are a subset of the tags defined in the `package.xsd` file and the `*-plugin.xsd` files.

In addition to defining entities, `*-entities.xml` files also define entity managers, enumerations, and native sequences. The `*-entities.xml` file content is further explored in "Understanding Entity Definitions".

**-*types.xsd**
File names that end with `-types.xsd` indicate that the file defines entity attributes (name and data type), or inherits entity attributes from a specified entity. For example, the Equipment entity defines several attributes including `id`, `name`, and `description`, all of which are defined as String. In another example, the EquipmentRole entity does not define any attributes; rather, it inherits all of InventoryRole entity attributes. The `*-types.xsd` file content is further explored in "Understanding Entity Attribute Definitions".

**-*enum-*.***
File names that contain `-enum-*.*` indicate that the file defines either enumeration types or enumeration values. The `*-enum-*.*` file content is further explored in "Understanding Enumeration Definitions".

**-*entityidsequenceextension-entities.xml**
File names that end with `-entityidsequenceextension-entities.xml` indicate that the file defines native sequences and corresponding sequence generator names, and that it extends both explicitly-defined and capability-generated entities by specifying a sequence generator for the entity to use. The `-entityidsequenceextension-entities.xml` file content is further explored in "Understanding Native Sequence Definitions".

**-*transitions.xml**
File names that end with `-transitions.xml` indicate that the file defines life-cycle state transitions. The `*-transitions.xml` file content is further explored in Chapter 5, "Extending Life Cycles".

**-*plugin.xsd**
File names that end with `-plugin.xsd` indicate that the file defines XML tags that govern definitions in the `*-entities.xml` files. The `*-plugin.xsd` files include:

- `uim-plugin.xsd`
- `core-plugin.xsd`
- `capability-plugin.xsd` files, where `capability` represents a specific capability such as capacity, characteristic, or consumable
The file content of the *-plugin.xsd files is similar to the package.xsd file content. See "package.xsd" for more information.

The uim-plugin.xsd resides in the src/ora_uim_poms.jar file. The core-plugin.xsd file resides in the src/platformFiles/poms/core_poms_lib.jar file.

The capability-plugin.xsd files reside in the correspondingly named src/platformFiles/poms/capability_poms_lib.jar files, where capability represents a specific capability such as capacity, characteristic, or consumable.

*-libs.xml
The capacity-caps-libs.xml and capacity-model-libs.xml files are internal files that support the modularity of the capability-plugin.xsd files.

The *-libs.xml files reside in the correspondingly named src/platformFiles/poms/capability_poms_lib.jar files, where capability represents a specific capability such as capacity, characteristic, or consumable.

package.xsd
The package.xsd file, and the *-plugin.xsd files, defines XML tags that govern definitions in the *-entities.xml files. For example, <entity>, <implements>, and <relationship> are XML tags used to define an entity, to specify an interface that the entity implements, and to define the entity’s relationship to other entities.

The XML tags defined in these files are enforced by the build that generates the database and entities. If an XML tag is added to an *-entities.xml file that is not defined in the package.xsd file or in a *-plugin.xsd file, the build fails with an error citing the invalid XML tag.

There is only package.xsd file and copy of the file resides in each of the src/platformFiles/poms/capability_poms_lib.jar files, where capability represents a specific capability such as capacity, characteristic, or consumable.

---

**Note:** Platform is the base code upon which all Oracle Communications products are built. Platform provides common code used by all Oracle Communications products, including the code that generates and builds the each product’s database and entities using the metadata files.

---

XMLSchema.xsd

The XMLSchema.xsd file is industry-standard specific.

---

Understanding Metadata File Content

The metadata file content defines:

- Entities
- Entity attributes
- Enumerations
- Native sequences
XML tags that govern the definition of an entity, entity manager, enumeration, and native sequence

Understanding Entity Definitions

Entity definitions result in the creation of database tables and corresponding entity source files, which are compiled into entity class files. Entity classes are used to persist data in the database, and each entity class instance mirrors a unique database record in a table.

*-entities.xml Files

Entities are defined by XML elements and attributes that identify various properties of the entity. An entity definition can reside in an \texttt{ocim-*-entities.xml} file, in a \texttt{uim-*-entities.xml} file, or in both files. When an entity definition resides in both files, the UIM portion of the definition extends from the Information Model portion of the definition. For the UIM data model, most entities are defined by both files. There are a handful of entities that are UIM-specific, in which case the entity definition resides only in the \texttt{uim-*-entities.xml} file.

Example of an Entity Defined by Both Files

In the UIM data model, most entity definitions reside in an \texttt{ocim-*-entities.xml} file, with the entity definition extended in a \texttt{uim-*-entities.xml} file, as shown in the following examples. Example 4–1 is an excerpt from the \texttt{ocim-number-entities.xml} file that shows the TelephoneNumber entity definition. The entity definition includes any interfaces the entity implements, any capabilities for which the entity is enabled, and any relationships that the entity has to other entities.

Example 4–1  Entity Definition

```xml
<entity type="ocim:TelephoneNumber"
    interface="oracle.communications.inventory.api.entity.TelephoneNumber"
    accessControlled="true" entityIdsSequenceGenerator="TelephoneNumberSeqGen">
    <implements interface="oracle.communications.inventory.api.entity.common.NetworkAddress"/>
    <attribute name="id" index="true"/>
    <attribute name="name" index="true"/>
    <!-- **************** Capabilities ******************-->
    <lifecycle stateType="ocim:InventoryState"/>
    <consumable prefix="TN" attribute="telephoneNumber"
        assignmentStateType="ocim:AssignmentState">
        <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
    </consumable>
    <referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
    <characteristic spec="ocim:CharacteristicSpecification">
        <characteristicName name="ocim:TNCharacteristic"
            interface="oracle.communications.inventory.api.entity.TNCharacteristic"
            table="TN_CHAR"/>
    </characteristic>
    <businessInteractionEnabled history="true" visibilityState="SHOW"/>
    <groupEnabled/>
    <!-- **************** Relationships ******************-->
    <!-- One-Sided Many-to-One TelephoneNumber to TelephoneNumberSpec -->
    <relationship name="specification">
        <otherSide type="ocim:TelephoneNumberSpecification"/>
    </relationship>
</entity>
```
Example 4–2 is an excerpt from the uim-number-entities.xml file. The example shows the TelephoneNumber entity definition that extends the TelephoneNumber entity definition from the ocim-number-entities.xml file. In this example, the entity declares an entity manager through the managedBy tag because the entity is business-interaction enabled, as defined in the ocim-number-entities.xml file. Any methods that an entity needs to implement are also defined in the UIM portion of the entity definition.

Example 4–2 Extended Entity Definition

```xml
<entity type="ocim:TelephoneNumber"
    managedBy="oracle.communications.inventory.api.number.TelephoneNumberManager">
    <method name="getDisplayInfo">
        <signature><![CDATA[String getDisplayInfo()]]></signature>
        <body><![CDATA[
            return getName();
        ]]]>
        </body>
        <javadoc>Return an identifiable info String for this resource.</javadoc>
    </method>
</entity>
```

Example of a UIM-Specific Entity Definition

Very few entities are defined by a uim-*-entities.xml file only. One such example is the uim-rule-entities.xml file. Rules are UIM-specific functionality, so the Information Model does not define any rule entities.

Example 4–3 an is an excerpt from the uim-rule-entities.xml file. The file defines four entities that deal with rules. The excerpt shows the ExtensionPoint entity definition, which includes any interfaces the entity implements, any capabilities for which the entity is enabled, and any relationships that the entity has to other entities.

Example 4–3 Entity Definition

```xml
<entity type="ocim:ExtensionPoint"
    interface="oracle.communications.inventory.api.entity.ExtensionPoint">
    <!-- Two-Sided One-to-Many ExtensionPoint to EnabledExtensionPoint -->
    <relationship name="enabledExtensionPoints">
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:EnabledExtensionPoint"/>
    </relationship>
    <!-- Two-Sided One-to-Many ExtensionPoint to ExtensionPointRuleSet -->
    <relationship name="extensionPointRuleSets">
        <thisSide inverse="true" collection="java.util.HashSet"/>
        <otherSide dependent="true" type="ocim:ExtensionPointRuleSet"/>
    </relationship>
</entity>
```

More on Entity Definitions

Entity definitions include capability and relationship declarations. See "Understanding Entity Capability Definitions" and "Understanding Entity Relationship and Collection Definitions" for more information.

The *-entities.xml files also define entity managers. See "Understanding Entity Manager Definitions" for more information.
Understanding Entity Attribute Definitions

Each entity defines a set of attributes in which to store data. Entity attribute definitions result in the creation of database table columns and corresponding entity attributes in source files, which are compiled into entity class files. Entity classes are used to persist data in the database, and each entity class instance mirrors a unique database record in a table.

*-types.xsd Files

Entity attributes are defined by XML elements and attributes that define an attribute’s name and data type. Entity attribute definitions reside in *-types.xsd files. Each *-entities.xml file has a corresponding *-types.xsd file. For example, ocim-number-entities.xml and ocim-number-types.xsd, or uim-rule-entities.xml and uim-rule-types.xsd. For each set of corresponding files:

- The *-entities.xml files define entities and entity managers
- The *-types.xsd files define entity attributes (name and data type)

Example 4–4 is an excerpt from the ocim-number-types.xsd file that defines the entity attributes (name and data type) for the TelephoneNumber entity. The attribute names defined in the excerpt are id, name, and description, and all of the attribute data types are defined as string.

Example 4–4 Entity Attributes Definition

```xml
<xs:complexType name="TelephoneNumber">
  <xs:sequence>
    <xs:element name="id" type="xs:string">
    </xs:element>
    <xs:element name="name" type="xs:string">
    </xs:element>
    <xs:element name="description" type="xs:string">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Understanding Enumeration Definitions

Enumeration definitions result in attributes that are defined with an enumeration type being limited to storing only previously defined data values. Enumerations are used to regulate data upon which code is based. For example, code can be written to handle a finite number of scenarios based on a finite number of defined enumeration values.

The following *-enum-* files are in the metadata:

- ocim-enum-entities.xml
- uim-enum-entities.xml
- ocim-enum-types.xsd
- uim-enum-types.xsd

*-enum-entities.xml Files

Enumeration types are defined in the ocim-enum-entities.xml and uim-enum-entities.xml files. Example 4–5 is an excerpt from the uim-enum-entities.xml file that shows the definition of two enumeration types: BusinessInteractionState and BusinessInteractionAction.
**Example 4–5  Enumeration Type Definition**

```xml
<enum type="ocim:BusinessInteractionState"
     enumType="oracle.communications.inventory.api.entity.BusinessInteractionState"
     adminState="true"/>
<enum type="ocim:BusinessInteractionAction"
     enumType="oracle.communications.inventory.api.entity.BusinessInteractionAction"/>
```

**-*-enum-types.xsd Files**

Enumeration values are defined in the `ocim-enum-types.xsd` and `uim-enum-types.xsd` files. **Example 4–6** is an excerpt from the `ocim-enum-types.xsd` file that shows the definition of two sets of enumeration values. The `ocim-enum-entities.xml` file and the `ocim-enum-types.xsd` file both show excerpts of `BusinessInteractionState` and `BusinessInteractionAction`. The `ocim-enum-entities.xml` file defines the enumeration type, and the `ocim-enum-types.xsd` file defines the enumeration values that are valid for each enumeration type.

**Example 4–6  Enumeration Value Definition**

```xml
<xs:simpleType name="BusinessInteractionState">
  <xs:restriction base="xs:string">
    <xs:enumeration value="CANCELLED"/>
    <xs:enumeration value="COMPLETED"/>
    <xs:enumeration value="CREATED"/>
    <xs:enumeration value="IN_PROGRESS"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="BusinessInteractionAction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="CANCEL"/>
    <xs:enumeration value="COMPLETE"/>
    <xs:enumeration value="PROCESS"/>
    <xs:enumeration value="TRANSFER"/>
  </xs:restriction>
</xs:simpleType>
```

**Understanding Native Sequence Definitions**

Several native sequences and their corresponding sequence generator names are defined in the metadata. Native sequence definitions result in the creation of the native sequence in the database. Several entity definitions are extended to specify a sequence generator, which results in the corresponding native sequence being used to set the `entityId` attribute value for the entity. The native sequence definitions, their corresponding sequence generator names, and the entity definitions that are extended to specify a sequence generator, reside in the `ocim-entityidsequenceextension-entities.xml` file.

---

**Note:** Entity definitions that do not specify a native sequence use the default native sequence provided by Platform.

---

**ocim-entityidsequenceextension-entities.xml File**

**Example 4–7** is an excerpt from the `ocim-entityidsequenceextension-entities.xml` file that shows the definition of three native sequences:

- ENTITYID_CONNECTIVITY_SEQ
- ENTITYID_EQUIPMENT_SEQ
- ENTITYID_TN_SEQ

The corresponding sequence generator names given to these native sequences are:
- ConnectivitySeqGen
- EquipmentSeqGen
- TelephoneNumberSeqGen

**Example 4–7 Native Sequence Definition**

```xml
<entityIdSequenceGenerator name="ConnectivitySeqGen" sequence="ENTITYID_CONNECTIVITY_SEQ" />
<entityIdSequenceGenerator name="EquipmentSeqGen" sequence="ENTITYID_EQUIPMENT_SEQ" />
<entityIdSequenceGenerator name="TelephoneNumberSeqGen" sequence="ENTITYID_TN_SEQ" />
```

Example 4–8 is an excerpt from the `ocim-entityidsequenceextension-entities.xml` file that shows the extended definition of three entities to include a sequence generator. In this example, the Pipe, PipeTerminationPoint, and PipeRel entities all specify the ConnectivitySeqGen sequence generator, which results in these entities using the corresponding ENTITYID_CONNECTIVITY_SEQ native sequence to set their respective `entityId` attribute values.

**Example 4–8 Extended Entity Definition**

```xml
<entity type="ocim:Pipe" entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
<entity type="ocim:PipeTerminationPoint" entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
<entity type="ocim:PipeRel" entityIdSequenceGenerator="ConnectivitySeqGen" extension="true"/>
```

### Understanding the Tags that Govern Definitions

The `package.xsd` and `-plugin.xsd` files define the tags that govern definitions of entities, entity managers, enumerations, and native sequences in `-entities.xml` files.

This section introduces the `package.xsd` and `-plugin.xsd` file content to help you better understand the content when you are viewing it. This section also explains the entity definition’s use of the tags to help you better understand how the tags correlate to an entity definition.

**Note:** This section does not explain the functionality of the governing tags; it explains where the governing tags are defined and how they are used within the `-entities.xml` files.

For information on the functionality that the tags provide, see:
- The documentation for each tag within the `package.xsd` and `-plugin.xsd` files
- *Oracle Communications Information Model Reference*

The `package.xsd` defines the following complexType elements:
- entity
- manager
The `<entity>` element defines several elements, and some of the elements define attributes. For example, the `<entity>` element defines the `<import>`, `<implements>`, `<attribute>`, `<relationship>`, and `<method>` elements, as well as several other elements. The `<relationship>` element defines the `join`, `thisSide`, and `otherSide` attributes. The `<entity>` element also defines several attributes directly (as opposed to the attributes being defined for an element). For example, the `<entity>` element defines the `interface` and `managedBy` attributes. Within the file, each complexType, element, and attribute is described.

In a similar fashion, the `*-plugin.xsd` files also define tags that are used in the `*-entities.xml` files. For example, the `capability-*-plugin.xsd` files (where `capability` represents a specific capability such as capacity, characteristic, and consumable) only define tags that are used in an entity definition to declare a particular capability for an entity.

Example 4–9 shows the `TelephoneNumber` entity definition. The example is numbered so that the information describing the example can be referenced.

**Example 4–9  Entity Definition**

```
01  <entity type="ocim:TelephoneNumber"
02      interface="oracle.communications.inventory.api.entity.TelephoneNumber"
03      accessControlled="true" >
04      <imports interface=
05      "oracle.communications.inventory.api.entity.common.NetworkAddress"/>
06      <attribute name="id" index="true"/>
07      <attribute name="name" index="true"/>
08      <!-- **************** Capabilities ******************-->
09      <lifecycle stateType="ocim:InventoryState"/>
10      <consumable prefix="TN" attribute="telephoneNumber"
11          assignmentStateType="ocim:AssignmentState">
12          <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
13      </consumable>
14      <referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
15      <characteristic spec="ocim:CharacteristicSpecification">
16          <characteristicName name="ocim:TNCharacteristic"
17              interface= "oracle.communications.inventory.api.entity.TNCharacteristic"
18              table="TN_CHAR"/>
19      </characteristic>
20      <businessInteractionEnabled history="true" visibilityState="SHOW"/>
21      <groupEnabled/>
22      <!-- **************** Relationships ******************-->
23      <!-- One-Sided Many-to-One TelephoneNumber to TelephoneNumberSpec -->
24      <relationship name="specification">
25          <otherSide type="ocim:TelephoneNumberSpecification"/>
26      </relationship>
27      </entity>
```

Lines 01 through 07 define the entity with various tags. For example:

- Line 01 uses `<entity>` and `type`; `<entity>` is defined as a complexType in `package.xsd`, and `type` is defined as an attribute of the `<entity>` element in `package.xsd`.
- Line 02 uses `interface`, which is defined as an attribute of the `<entity>` element in `package.xsd`. 
Extending the Data Model Through the Metadata Files

- Line 03 uses `accessControlled`, which is defined in the `core-plugin.xsd` file.
- Lines 04 and 05 use `<implements>` and `interface`; `<implements>` is defined as an element of the `<entity>` element in `package.xsd`, and `interface` is defined as an attribute of the `<implements>` element in `package.xsd`.
- Lines 06 and 07 use `<attribute>` and `name`; `<attribute>` is defined as a complexType in `package.xsd`, and `name` is defined as an attribute of the `<attribute>` element in `package.xsd`.

Lines 08 through 22 continue the entity definition by defining the entity’s capabilities. For example:

- Line 09 uses `<lifecycle>` and `stateType`, both of which are defined in `uim-plugin.xsd`.
- Lines 10 through 13 use several tags that are defined in `consumable-plugin.xsd`.
- Line 14 uses `<referenceEnabled>`, which is defined in `uim-plugin.xsd`.
- Lines 15 through 20 use several tags that are defined in `characteristic-plugin.xsd`.
- Line 21 uses `<businessInteractionEnabled>` and `visibilityState`, both of which are defined in `uim-plugin.xsd`.
- Line 22 uses `<groupEnabled>`, which is defined in `groupenabled-plugin.xsd`.

Lines 23 through 27 continue the entity definition by defining the entity’s relationships to other entities. For example, `<relationship>`, `name`, `<otherSide>`, and `type` are all defined in `package.xsd`.

## Extending the Data Model Through the Metadata Files

You extend the data model by creating new metadata files.

---

**Caution:** Do not make modifications to the existing metadata files. See "Backwards Compatibility" for the issues involved with modifying the existing metadata files.

---

You use Oracle Communications Design Studio to create new metadata files by importing the `ora_uim_entity_sdk_cartproj.zip` file, and creating new XML or XSD files within the imported project. Any new metadata files you create must reside in the `ora_uim_entity_sdk` project, within the `src` directory.

When you define new entities and attributes, or extend existing entities and attributes, the changes are picked up by the Ant target that generates the database and the corresponding entity Java source files. For example, if you add the new entity `myNewEntity`, `myNewEntity` is generated as a new table in the database, and `MyNewEntity` is generated as an entity Java source file. If you add `myNewAttribute` to an existing entity, `myNewAttribute` is generated as a new column on the existing table in the database, and `myNewAttribute` is generated as an attribute within the generated entity Java source file.

The following sections describe extending the data model through the creation of new metadata files.
**Defining New Entities**

When defining new entities, look at existing `-entities.xml` files for examples of how to define various entity properties. The XML tags you use to define a new entity are governed by the `package.xsd` and `-plugin.xsd` files. Be sure to include any referenced schemas in the package statement.

To define a new entity:

   
   The file name must end with `-entities.xml`. For example, `myNewFile-entities.xml`.

2. Open an existing `-entities.xml` file.

3. Copy and paste an entity definition from the existing file to your new file.

4. Modify the copied entity definition as needed:
   a. Change the name of the entity to reflect the name of your new entity.
   b. Remove or update the tags to reflect the definition of your new entity.

5. Write an entity manager that defines the interfaces to manage the new entity. See "Creating New Entity Managers".

6. Write an entity manager implementation that inherits from `BaseInvManager` and defines the methods to manage the new entity. See "Creating New Entity Managers".

7. Include the entity manager interface definition in your new `-entities.xml` file.

**Creating New Entity Managers**

When creating new entity managers, the entity managers:

- Should provide coarse-grained methods that may involve other entity managers
- Assume that the caller is managing transaction boundaries
- Must be developed so that they are stateless and thread safe so they can be exposed to Web service calls
- Should avoid creating duplicate records using `makePersistent()`. To avoid this, call the `connect()` method before you set entities to the transient state. See "PersistenceManager.refresh(), PersistenceManager.attach(), and PersistenceManager.connect() Methods" for more information.

See Chapter 3, "Using the Persistence Framework" for more information on entity managers.

**Defining New Entity Attributes**

When defining new entity attributes, look at the existing `-types.xsd` files for examples of how to define various attributes. The XML tags you use to define new entity attributes are governed by the `XMLSchema.xsd` file. Be sure to include any referenced schemas in the package statement.

XSD is an industry standard. For information about writing XSD, see the W3C Web site:


To define attributes for a new entity, or to add new attributes to an existing entity:

1. Create a new XSD file.
The file name must end with -types.xsd. For example, myNewFile-types.xsd.

2. Open an existing *-types.xsd file.

3. Copy and paste an entity attribute definition from the existing file to your new file.

4. If defining attributes for a new entity, modify the copied entity attributes definition as needed:
   a. Change the entity name to reflect the new entity name.
   b. Change the attribute names to reflect the new attribute names.
   c. Change the attribute types to reflect the new attribute types.

5. If adding new attributes to an existing entity, modify the copied entity attributes definition as needed:
   a. Change the entity name to reflect the entity name that defines the attributes to which you are adding new attributes.
   b. Change the attribute names to reflect the new attribute names you are adding.
   c. Change the attribute types to reflect the new attribute types you are adding.

Defining New Enumerations

When defining new enumeration types, look at the existing *-enum-entities.xml files for examples of how to define them; when defining new enumeration values, look at the existing *-enum-types.xsd files for examples of how to define them. The XML tags you use to define new enumerations are governed by the XMLSchema.xsd file. Be sure to include any referenced schemas in the package statement.

Enumerations are an industry standard. For information about writing enumerations, see the W3C Web Services Enumeration Web site:

http://www.w3.org/Submission/WS-Enumeration/

You can place all new enumeration types in one new file, and all new sets of enumeration values in another new file.

To define a new enumeration type:

   The file name must contain -enum- and end with -entities.xml. For example, myNew-enum-entities.xml.

2. Open an existing *-enum-entities.xml file.

3. Copy and paste an enumeration type definition from the existing file to your new file.

4. Modify the copied enumeration type definition as needed.

To define enumeration values for a new enumeration type, or to add new enumeration values to an existing enumeration type:

1. Create a new XSD file.
   The file name must contain -enum- and end with -types.xsd. For example, myNewFile-enum-types.xsd.

2. Open an existing *-enum-types.xsd file.

3. Copy and paste a set of enumeration values from the existing file to your new file.
4. If defining new enumeration values for a new enumeration type, modify the copied enumeration values as needed:
   a. Change the enumeration type to reflect your new enumeration type.
   b. Change the data type to reflect the data type of your new enumeration values.
   c. Change the enumeration values to reflect your new enumeration values.

5. If adding new enumeration values to an existing enumeration type, modify the copied enumeration values as needed:
   a. Change the enumeration type to reflect the name of the existing enumeration type to which you are adding the new enumeration values.
   b. Change the data type to reflect the same data type as defined by the enumeration type to which you are adding the new enumeration values.
   c. Change the enumeration values to reflect the new enumeration values you are adding to the enumeration type.

**Defining New Native Sequences**

When defining new native sequences, look at the existing `ocim-entityidsequenceextension-entities.xml` file for examples of how to define a native sequence and corresponding sequence generator name. Be sure to include any referenced schemas in the package statement.

To define a new native sequence, and specify an entity to use it:

   The file name must end with `-entities.xml`. The file name should also contain a meaningful reference so you can readily recognize the file content, such as `-seqext-`. For example, `myNewSeqExts-entities.xml`.

2. Open the existing `ocim-entityidsequenceextension-entities.xml` file.

3. Copy and paste a native sequence definition from the existing file to your new file.

4. Modify the copied native sequence definition as needed:
   a. Change the name of the native sequence to reflect the name of your new native sequence.
   b. Change the corresponding sequence generator name to reflect a functional name for your new native sequence.

5. Copy and paste an extended entity definition from the existing file to your new file.

6. Modify the copied extended entity definition as needed:
   a. Change the entity name.
   b. Change the specified sequence generator to your new sequence generator.

**Extending Existing Entities**

You can extend an existing entity using the XML tags defined in the `package.xsd` and `-plugin.xsd` files that enable an entity’s use of framework functionality. For example, you can extend an entity to be life-cycle enabled, capacity enabled, business-interaction enabled, place enabled, group enabled, and so forth. Each of these are functional areas of UIM that become available to an entity through the entity’s original definition, or through an extension and custom code.
There are two kinds of existing entities: Explicitly-defined entities and capability-generated entities. For example, the Equipment entity is explicitly defined in the metadata, and includes the declaration of the consumable capability. As a result of this declaration, the EquipmentConsumer entity is generated, even though the EquipmentConsumer entity is not explicitly defined in the metadata.

Explicitly-defined entities and capability-generated entities are extended the same way, with one slight difference: When extending a capability-generated entity, you must include the `extension` tag.

### Understanding the Extension Tag

The `package.xsd` file defines tags that are used to govern entity definitions. One such tag is the `extension` tag. Since entities are defined in `-entities.xml` files, the `extension` tag may only be used within an `-entities.xml` file.

You can use the `extension` tag to extend:

- Capability-generated entities
- Generated attributes of an explicitly-defined entity
- Generated relationships of an explicitly-defined entity

For example, the following is an excerpt from the `ocim-entityidsequenceextension-entities.xml` file that extends the capability-generated EquipmentConsumer entity to use the ConsumerSeqGen native sequence:

```xml
<entity type="ocim:EquipmentConsumer" entityIdSequenceGenerator="ConsumerSeqGen" extension="true"/>
```

A similar extension in the same file for the explicitly-defined Equipment entity does not require the `extension` tag because the entity is not capability-generated:

```xml
<entity type="ocim:Equipment" entityIdSequenceGenerator="EquipmentSeqGen"/>
```

In another example, the TelephoneNumber entity is explicitly defined in the metadata, and by default has the Trackable Pattern as described in Oracle Communications Information Model Reference. The Trackable Pattern generates the `createdDate`, `createdUser`, `lastModifiedDate`, and `lastModifiedUser` attributes. You can extend any of these generated attributes through the use of the `extension` tag. For example, the following adds an index to the `createdUser` generated attribute:

```xml
<entity type="ocim:TelephoneNumber" extension="true">
    <attribute name="createdUser" index="true"/>
</entity>
```

The TelephoneNumber entity explicitly defines several attributes, such as `id`, `name`, and `description`. So, a similar extension for an explicitly-defined attribute of TelephoneNumber does not require the `extension` tag because the attribute is not generated:

```xml
<entity type="ocim:TelephoneNumber">
    <attribute name="id" index="true"/>
</entity>
```
Extending the Data Model Through the Metadata Files

Extending Existing Entities

You can extend an existing entity through a new *-entities.xml file that defines the same entity name and includes additional properties for the entity. When regenerating the entities:

- For explicitly-defined entities, the properties defined for the entity in both files are merged together, resulting in the entity possessing the original properties and the extended properties.
- For capability-generated entities, the properties of the generated entities and any extended definitions are merged together, resulting in the entity possessing the original generated properties and the extended properties.

To extend an existing entity:

   The file name must end with -entities.xml. For example, myNewFile-entities.xml.
2. Open an existing *-entities.xml file.
3. Copy and paste an entity definition from the existing file to your new file.
4. Modify the copied entity definition as needed:
   a. The entity name must be the same name as the entity you are extending.
   b. Add any tags to reflect the extension.
   c. If you are extending a capability-generated entity, include the extension tag.

Extending Existing Entity Attributes

Extending an existing entity attribute is done in an *-entities.xml file, not in the *-types.xsd file.
You can extend an existing attribute using the XML tags defined for attributes in the `package.xsd` file. For example, you can extend an attribute definition to have an index, to be encrypted, or to have a maximum length that the database stores for an attribute. See "Understanding the Tags that Govern Definitions" for more information.

There are two kinds of existing attributes: Explicitly-defined attributes and capability-generated attributes. For example, the Equipment entity and its attributes are explicitly defined in the metadata. The Equipment entity definition includes the declaration of the life cycle management capability. As a result of this declaration, the `adminState` and `objectState` attributes are generated for the Equipment entity, even though they are not explicitly defined in the metadata.

Explicitly-defined attributes and capability-generated attributes are extended the same way, with one slight difference: When extending a capability-generated attribute, you must include the `extension` tag. See "Understanding the Extension Tag" for more information.

You can extend an existing attribute through a new `*-entities.xml` file that defines the same entity name, and includes the attribute extension. When regenerating the entities:

- For explicitly-defined attributes, the properties defined for the attribute in both files are merged together, resulting in the attribute possessing the original properties and the extended properties.
- For capability-generated attributes, the properties of the generated attributes and any extended definitions are merged together, resulting in the attribute possessing the original generated properties and the extended properties.

To extend an existing attribute:

   The file name must end with `*-entities.xml`. For example, `myNewFile-entities.xml`.
2. Open an existing `*-entities.xml` file.
3. Copy and paste an entity definition from the existing file to your new file.
4. Modify the copied entity definition as needed:
   a. The entity name must be the same name as the entity that defines or generates the attribute you are extending.
   b. Update the tags to reflect the extension.
   c. If you are extending a capability-generated entity, include the `extension` tag.

**Extending Existing Enumerations**

You cannot extend existing enumerations. To clarify, you can add new enumeration values to an existing set of enumeration values. See "Defining New Enumerations" for more information.

**Extending Existing Native Sequences**

You can extend the use of any UIM-defined native sequences to include additional entities.

To extend the use of UIM-defined native sequences:

The file name must end with -entities.xml. The file name should also contain a meaningful reference so you can readily recognize the file content, such as -seqext-. For example, myNewSeqExts-entities.xml.

2. Open the existing ocim-entityidsequenceextension-entities.xml file.
3. Copy and paste an extended entity definition from the existing file to your new file.
4. Modify the copied extended entity definition as needed:
   a. Change the entity name.
   b. Change the specified sequence generator to a UIM-defined sequence generator that you want the entity to use.
   c. If the entity is a capability-generated entity, include the extension tag.

You can also modify existing UIM-defined native sequences.

---

**Note:** Oracle recommends that modifications to existing sequences be made before using UIM (before any sequence numbers have been generated).

---

To modify existing UIM-defined native sequences:

   The file name must end with -entities.xml. The file name should also contain a meaningful reference so you can readily recognize the file content, such as -seqext-. For example, myNewSeqExts-entities.xml.
2. Open the existing ocim-entityidsequenceextension-entities.xml file.
3. Copy and paste an extended entity definition from the existing file to your new file.
4. In the copied file, modify the sequence definition as needed.
5. In the copied file, set the entityIdSequenceGeneratorPriority attribute to a value higher than the entityIdSequenceGeneratorPriority value in the original file so that the modified sequence overrides the original sequence.
6. If modifying the existing sequence after using UIM (after sequence numbers have been generated):
   a. Determine which entities use the modified sequence.
   b. For each entity using the sequence, determine the current maximum entityId value.
   c. For each entity using the sequence, extend the entity definition to set the initialValue attribute to a value 50 to 100 higher than the current maximum entityId value.

See "Extending Existing Entities" and "Extending Existing Entity Attributes" for information on how to do this.

---

**Applying Metadata Static Extensions**

Statically extending the data model involves manually extending the metadata files. To apply the metadata static extensions:
- Generate the entity source files from the metadata files
- Compile the generated entity source files
- Package the compiled entity source files in the `inventory-adapter.ear`, `inventory.ear`, and `custom.ear` files.
- Deploy the `inventory_adapter.ear`, `inventory.ear`, and `custom.ear` files

You generate, compile, and package the entity source files using Ant targets that are provided in the `ora_uim_entity_sdk_cartproj.zip/src/build.xml` file.

**About the build.xml File**

The `build.xml` file defines several Ant targets that you can run to manage the database. An Ant target is a set of executable tasks that can be run using Ant. See "Running Ant Targets" for more information.

Table 4–1 describes the Ant targets defined in the `build.xml` file.

<table>
<thead>
<tr>
<th>Ant Target</th>
<th>Description</th>
</tr>
</thead>
</table>
| entities    | From the metadata, this target generates entity source files and compiles them into entity Java classes. As a result, this target creates:  
  - The `api/build` directory, where all of the generated source files and compiled classes reside. This directory is placed in the `ora_uim_entity_sdk/src` directory.  
  - The `uim-entities.jar` file, which contains all of the entity classes. This file is placed in the `ora_uim_entity_sdk/src/generated/inventory/entities/APP-INF/lib` directory. |
| clean       | This target deletes the `api/build` and `generated` directories created by the `entities` Ant target.  
  Oracle recommends that you always run the `clean` target prior to running the `entities` target. |
| create.tables | This target creates new database tables based on any new entities defined in the metadata; this target does not alter existing tables that were created as part of the UIM installation.  
  To connect to the database in which you are creating tables, set the database credentials in step 8 of the procedure to generate, compile, and package the entity source files. See "Generating, Compiling, and Packaging the Entity Source Files" for more information. |
| alter.tables | This target creates new database tables based on any new entities defined in the metadata. This target also alters existing database tables based on any new attributes defined for existing entities in the metadata.  
  To connect to the database in which you are creating or altering tables, set database credentials in step 8 of the procedure to generate, compile, and package the entity source files. See "Generating, Compiling, and Packaging the Entity Source Files" for more information. |
Generating, Compiling, and Packaging the Entity Source Files

To generate, compile, and package the entity source files for deployment:

1. Configure your Design Studio environment. Define the following system variables:
   - JAVA_HOME
     See "Configuring Design Studio" for more information.
   - ANT_HOME
     See "Installing Ant" for more information.

2. In Design Studio, open the Studio Design perspective.
   For instructions on how to open perspectives in Design Studio, see the Design Studio Help.

   For instructions on how to import a project into Design Studio using archive files, see the Design Studio Help.

4. Change to the Java perspective.
   For instructions on how to change perspectives in Design Studio, see the Design Studio Help.

5. Copy and rename the ora_uim_entity_sdk/etc/COMPUTERNAME.properties file to HOSTNAME.properties, where HOSTNAME is the name of the computer where Design Studio is installed.
   You can determine the computer name by running the following DOS command:
   ```
   echo %COMPUTERNAME%
   ```

6. In the HOSTNAME.properties file, set the following properties to reflect your configuration:
   - UIM_HOME
   - DB_HOME
   - PROJECT_HOME

7. From a command line, navigate to your workspace ora_uim_entity_sdk/src directory, and run the following Ant command:
   ```
   ant -f build_extract_poms_zip.xml
   ```

---

### Table 4–1 (Cont.) build.xml Ant Targets

<table>
<thead>
<tr>
<th>Ant Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create.DDL</td>
<td>This target creates the createDDL.jdbc and alterDDL.jdbc scripts based on any new entities defined in the metadata, or any new attributes defined for existing entities in the metadata. These files are placed in the ora_uim_entity_sdk/src/generated/entities/scripts directory. Run these scripts to create new database tables or alter existing database tables. The create.DDL target is used in place of the create.tables and alter.tables targets.</td>
</tr>
<tr>
<td>update.earwithEntities</td>
<td>This target updates the inventory.ear, inventory-adapter.ear, and custom.ear files with the generated uim-entities.jar file.</td>
</tr>
</tbody>
</table>
Applying Metadata Static Extensions

This command extracts the POMS SDK into the `ora_uim_entity_sdk/src/platformFiles/extract` directory.

8. In the `ora_uim_entity_sdk/src/platformFiles/extract/objectmgmt/poms/config/poms.properties` file, set the `ConnectionUserName`, `ConnectionPassword`, and `ConnectionURL` database credentials to reflect your database:

```java
#Set datastore connection information for offline utilities.
poms.ConnectionDriverName = oracle.jdbc.OracleDriver
poms.ConnectionUserName = uimuser
poms.ConnectionPassword = welcome@123
```

9. Configure the project library list.

   For instructions on how configure the project library list, see the Design Studio Help.

   Figure 4-1 shows the imported project library list, which includes the JAR files needed to compile the project.

![Figure 4-1 Project Library List Before Configuring](image)

**Note:** The `uim-entities.jar` file gets created and placed in the specified directory by a later step in this procedure.

The project library list of JAR files does not indicate the location of the files, so you must configure the project library list to point to the location of the JAR files. To do this, you need to add new variables named `POMS_LIB` and `UIM_LIB` that point the specified directory, as listed in Table 4-2.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Directory Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>POMS_LIB</td>
<td><code>Oracle_Home/POMSClient/lib</code></td>
</tr>
</tbody>
</table>

The `uim-entities.jar` file gets created and placed in the specified directory by a later step in this procedure.
Table 4–2 (Cont.) Location of JAR Files

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Directory Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIM_LIB</td>
<td>UIM_Home/lib</td>
</tr>
</tbody>
</table>

Figure 4–2 shows the project library list after the variables are added. Notice that the library list now includes the location of the JAR files, not just the JAR file names.

10. Add any new metadata files to the ora_uim_src_entity/src/api directory. See "Extending the Data Model Through the Metadata Files" for more information.

11. Modify the ora_uim_src_entity/src/api/custom-model-lib.xml file to include any new metadata files in the build. For example, if you have created new files named my-entities.xml and my-entities.xsd, you need to add these file names in the custom-model-lib.xml as follows:

Example 4–10  custom-model-lib.xml

```xml
<?xml version='1.0' encoding='UTF-8'?>
<libsl>
  <modellib>
    <id>
      <name>http://xmlns.oracle.com/communications/persistence/UimModel</name>
      <version>1.0</version>
    </id>
    <art>my-entities.xml</art>
    <art>my-types.xsd</art>
  </modellib>
</libsl>
```

12. Add any custom Java code that supports new entities to the ora_uim_src_entity/src/api/src-man directory. For example, defining new entities in the
metadata requires creating new entity managers. See “Creating New Entity Managers” for more information.

13. From a command line, navigate to your workspace ora_uim_src_entity/src directory, and run the following Ant command:

```
ant entities
```

This command creates the uim-entities.jar file and places it in the ora_uim_src_entity/src/generated/inventory/APP_INF/lib directory. The uim-entities.jar file contains the entity Java classes for all UIM entities and all custom entities. See "About the build.xml File" for more information.

14. Make a backup copy of the following files, located in the UIM_Home/app directory:

- inventory.ear
- inventory-adapter.ear
- custom.ear

15. From a command line, navigate to your workspace ora_uim_src_entity/src directory and run the following Ant command:

```
ant update.earwithEntities
```

This command updates the inventory.ear, inventory-adapter.ear, and custom.ear files with the uim-entities.jar file created in step 13. See "About the build.xml File" for more information.

16. From a command line, navigate to your workspace ora_uim_src_entity/src directory and run one of the following set of Ant commands:

```
ant create.tables
ant alter.tables
```

or

```
ant create.DDL
```

These commands update the database either directly (create.tables, alter.tables), or indirectly through a script (create.DDL). See "About the build.xml File" for more information.

17. Deploy the inventory-adapter.ear file.

18. If you have custom Java code in the ora_uim_entity_sdk project, deploy the resultant ora_uim_entity_sdk.jar file (cartridge) from the Studio environment into UIM.

Deploying the cartridge adds your custom Java code to the uim_custom_lib.ear file.

19. Deploy the updated inventory.ear file. At the time of deploying this EAR file, you must name this application oracle.communications.inventory.

20. Deploy the custom.ear file if you added custom code.

**More on Entity Definitions**

This section further describes entity definitions, focusing on:

- Understanding Entity Capability Definitions
Understanding Entity Capability Definitions

A capability is a design pattern that is applied to an entity, such as enabling an entity to be life-cycle managed. A capability is declared in the metadata using tags, and results in the generation of attributes and related entities that are not explicitly defined in the *-entities.xml or *-types.xsd files.

For example, an entity that is life-cycle-managed progresses through a succession of states during the course of its life. For life-cycle-managed entities, UIM tracks two states: administrative state and object state. To support this capability, an entity must define the adminState and objectState attributes. Rather than declare these attributes in the *-types.xsd of every entity that supports this pattern, the capability is declared for the entity in the *-entities.xml file using the <lifeCycle> element stateType tag. As a result of this tag, the adminState and objectState attributes are generated on the entity, and the corresponding columns are generated on the database table.

The package.xsd and *-plugin.xsd files defines the tags that are available to declare a capability. Some capability definitions are modularized, such as the capacity, characteristic, consumable, and group-enabled capabilities, as defined in the following *-plugin.xsd files:

- capacity-plugin.xsd
- characteristic-plugin.xsd
- consumable-plugin.xsd
- groupenabled-plugin.xsd

Other *-plugin.xsd files include:

- core-plugin.xsd
- uim-plugin.xsd

The design patterns that are declared as capabilities are documented in Oracle Communications Information Model Reference.

Understanding Entity Relationship and Collection Definitions

---

**Note:** For information on all possible elements and attributes that can be used to define entity relationships and collections, see the package.xsd file.

---

The *-entities.xml files define entities and their relationships to other entities. Relationships between entities can be categorized into two types: Uni-directional and bi-directional. A uni-directional relationship only allows one-way traversal from one entity to another; a bi-directional relationship allows traversal both ways. The relationships can also be separated into three cardinalities: One-to-one, one-to-many, and many-to-many. Types and cardinalities of relationships result in any given entity relationship falling into one of six different combinations. Examples of these six different combinations are described in the following sections.

The direction of the relationships is not the determining factor of how primary and foreign keys are defined in the physical model. It affects only how the logical object
model is defined. However, the relationship cardinality and ownership dictates the primary-foreign keys and join table definitions.

Relational and collection-type attributes can also be dependent. The entity or collection of entities referenced by a dependent attribute is deleted when the owning entity is deleted.

**Uni-Directional, One-to-One Relationship**

In Example 4–11, TopologyProfileEdge has a reference to TopologyEdge, but TopologyEdge does not have a reference to TopologyProfileEdge.

**Example 4–11  Uni-Directional, One-to-One Relationship**

```xml
<entity type="ocim:TopologyProfileEdge"
interface="oracle.communications.inventory.api.entity.TopologyProfileEdge"
accessControlled='true' entityIdSequenceGenerator="TopologySeqGen">
  . . .
  <!-- One-Sided One-to-One TopologyProfileEdge to TopologyEdge -->
  <relationship name="topologyEdge">
    <otherSide type="ocim:TopologyEdge"/>
    <javadoc>
      The TopologyEdge that contains the TopologyProfileEdge.
    </javadoc>
  </relationship>
</entity>
```

**Uni-Directional, One-to-Many Relationship**

In Example 4–12, CharactersticSpecification has a reference to a collection of CharacteristicSpecValue entities, but a CharacteristicSpecValue does not have a reference back to the CharacteristicSpecification. The relationship can be omitted only on the many side. The CharacteristicSpecification ENTITYID foreign key is still realized physically as a column in the CharacteristicSpecValue table. However, a CharacteristicSpecValue entity does not have a Java attribute generated that allows the traversal back to the CharacteristicSpecification.

**Example 4–12  Uni-Directional, One-to-Many Relationship**

```xml
<entity type="ocim:CharacteristicSpecification"
interface="oracle.communications.inventory.api.entity.CharacteristicSpecification"
timeBound="true">
  <implements interface="java.lang.Cloneable">
  . . .
  <!-- One-Sided One-to-Many CharacteristicSpecification to CharacteristicSpecValue -->
  <relationship name="values">
    <thisSide collection="java.util.HashSet"/>
    <otherSide dependent='true' type="ocim:CharacteristicSpecValue"/>
  </relationship>
</entity>
```

The collection data type is defined as a java.util.ArrayList. EclipseLink suggests that java.util.HashSet be used whenever possible to achieve better performance on their smart proxies logic because list-type collections such as ArrayList require sequential ordering for the elements for indexed access and allow for duplicate values of elements. Therefore, if the usage pattern of the collection attributes does not involve direct indexed access to a specific element, and the elements are unique within the collection, set-type collections should be used instead.
Uni-Directional, Many-to-Many Relationship
There is no example in UIM of a uni-directional, many-to-many relationship; however, it is a valid relationship. Using a scenario of entity1 and entity2, entity1 is applicable to multiple entity2s, and each entity2 has access to multiple entity1s. However, only entity1 has a collection of entity2s. A join table is required for the many-to-many relationship.

In this relationship, the logical object model does not provide immediate insight that the relationship is many-to-many. From the entity1 point of view, it is one-entity1-to-many-entity2s. The logical object model does not show the many-to-many cardinality because there is no relationship back to entity1. However, the physical model exhibits the many-to-many relationship through the use of the join table.

Bi-Directional, One-to-One Relationship
In Example 4–13, Equipment has a reference to EquipmentEquipmentRel, and EquipmentEquipmentRel has a reference back to its sole Equipment. The relationship is owned by the Equipment and the EquipmentEquipmentRel is dependent on the Equipment.

The relationship name is used for generating the attribute name in the entity.

Example 4–13  Bi-Directional, One-to-One Relationship
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
  ...
  <!-- Two-Sided One-to-One Equipment to EquipmentEquipmentRel (B) -->
  <relationship name="parentEquipment">
    <thisSide inverse="true"/>
    <otherSide type="ocim:EquipmentEquipmentRel"
      attribute="childEquipment"/>
    <javadoc>
      The holding parent equipment.
    </javadoc>
  </relationship>
</entity>
  ...
<entity type="ocim:EquipmentEquipmentRel"
interface="oracle.communications.inventory.api.entity.EquipmentEquipmentRel"
table="Eq_EqRel" accessControlled="true"
entityIdSequenceGenerator="EquipmentSeqGen">
  ...
  <!-- Two-Sided One-to-One EquipmentEquipmentRel to Equipment -->
  <relationship name="childEquipment">
    <otherSide type="ocim:Equipment" attribute="parentEquipment"/>
  </relationship>
</entity>

Bi-Directional, One-to-Many Relationship
In Example 4–14, Equipment has a reference to a collection of EquipmentHolderEquipmentRel entities, and EquipmentHolderEquipmentRel has a reference back to Equipment.

The inverse relationship is always on the one side of the relationship because the foreign key is on the many side. The relationship is owned by the Equipment. The collection of EquipmentHolderEquipmentRel entities is defined as dependent. As a
dependent collection, the entities in the collection are deleted automatically when the
owner entity is deleted. A dependent property is also applicable to simple
no-collection type attributes.

**Example 4–14 Bi-Directional, One-to-Many Relationship**

```xml
<entity type="ocim:Equipment"
  interface="oracle.communications.inventory.api.entity.Equipment"
  accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
  . . .
  <!-- Two-Sided One-to-Many Equipment to EquipmentEquipmentHolderRel -->
  <relationship name="parentEquipmentHolders">
    <thisSide inverse="true" collection="java.util.HashSet"/>
    <otherSide dependent="true" type="ocim:EquipmentHolderEquipmentRel"
      attribute="equipment"/>
    <javadoc>
      Set of parent equipment holders the equipment is held by.
    </javadoc>
  </relationship>
</entity>

. . .

<entity type="ocim:EquipmentHolderEquipmentRel"
  interface="oracle.communications.inventory.api.entity.EquipmentHolderEquipmentRel"
  table="EqHolder_EqRel" accessControlled="true"
  entityIdSequenceGenerator="EquipmentSeqGen">
  . . .
  <!-- Two-Sided Many-to-One EquipmentHolderEquipmentRel to Equipment -->
  <relationship name="equipment">
    <otherSide type="ocim:Equipment" attribute="parentEquipmentHolders"/>
    <javadoc>
      The child equipment.
    </javadoc>
  </relationship>
</entity>
```

**Bi-Directional, Many-to-Many Relationship**

In **Example 4–15**, Equipment can have multiple DeviceInterface entities, and each
DeviceInterface entity can be applicable to many Equipment entities. A value of
inverse="true" means that the other side of a two-way relationship owns the foreign
key. In a many-to-many relationship, the inverse is arbitrary because there should be a
join table created. In this case, the inverse="true" defines the other side as the owner of
the relationship. Collection types include ArrayList and HashMap. See the EclipseLink
documentation for all supported collection types.

**Example 4–15 Bi-Directional, Many-To-Many Relationship**

```xml
<entity type="ocim:Equipment"
  interface="oracle.communications.inventory.api.entity.Equipment"
  accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
  . . .
  <!-- Two-Sided Many-to-Many Equipment to DeviceInterface-->
  <relationship name="supportedDeviceInterfaces">
    <join table="equipment_deviceinterface"/>
    <thisSide inverse="true" collection="java.util.HashSet"/>
    <otherSide type="ocim:DeviceInterface"/>
    <javadoc>
      The list of mapped device interfaces supported by the equipment.
    </javadoc>
  </relationship>
</entity>
```
Relationship Definition Affected on Generated Entities

Each relationship definition adds an attribute to the generated entity for which it is defined. For a uni-directional relationship, an attribute is generated for the owning entity. For a bi-directional relationship, an attribute is generated for the owning entity and for the dependent entity.

Example 4–16 is an excerpt from the uim-rule-entities.xml file that defines the extensionPoint, ruleSetEntity, and specification relationships for the ExtensionPointRuleSet entity.

Example 4–16 Entity Definition

```
<entity type="ocim:ExtensionPointRuleSet"
interface="oracle.communications.inventory.api.entity.ExtensionPointRuleSet">
  <identifier>
    <attribute>extensionPoint</attribute>
    <attribute>ruleSetEntity</attribute>
    <attribute>specification</attribute>
  </identifier>
  <!-- Two-Sided Many-to-One ExtensionPointRuleSet to ExtensionPoint -->
  <relationship name="extensionPoint">
    <otherSide type="ocim:ExtensionPoint" attribute="extensionPointRuleSets"/>
  </relationship>
  <!-- Two-Sided Many-to-One ExtensionPointRuleSet to RuleSetEntity -->
  <relationship name="ruleSetEntity">
    <otherSide type="ocim:RuleSetEntity" attribute="extensionPointRuleSets"/>
  </relationship>
  <!-- Two-Sided Many-to-One ExtensionPointRuleSet to Specification -->
  <relationship name="specification">
    <otherSide type="ocim:Specification" attribute="extensionPointRuleSets"/>
  </relationship>
</entity>
```

Example 4–17 is an excerpt from the uim-rule-types.xsd file that defines the type attribute and the sequence attribute for the ExtensionPointRuleSet entity.

Example 4–17 Entity Attributes Definition

```
<xs:complexType name="ExtensionPointRuleSet">
  <!-- ... -->
</xs:complexType>
```
Example 4–18 is a code excerpt from the ExtensionPointRuleSet generated source file that defines the attributes for ExtensionPointRuleSet entity. The type attribute and the sequence attribute are generated based on the attributes defined in the uim-rule-types.xsd file for the ExtensionPointRuleSet entity. The fields extensionPoint, ruleSetEntity, and specification are generated based on the relationships defined in the uim-rule-entities.xml file for the ExtensionPointRuleSet entity.

Example 4–18  Generated Source File

```java
/*
 * ExtensionPointRuleSet.java
 * [CODE-GENERATED]
 */
package oracle.communications.inventory.api.entity;
/**
 * Associates extension points and rule sets.
 */
public interface ExtensionPointRuleSet
extends java.io.Serializable,
oracle.communications.platform.persistence.Persistent,
oracle.communications.inventory.api.Trackable
{
public static final
oracle.communications.platform.persistence.impl.EntityField _type
= new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.class, "type");

public static final
oracle.communications.platform.persistence.impl.EntityField _sequence
= new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.class, "sequence");

public static final
oracle.communications.platform.persistence.impl.EntityField _extensionPoint
= new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.class, "extensionPoint");

public static final
oracle.communications.platform.persistence.impl.EntityField _ruleSetEntity
= new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.class, "ruleSetEntity");

public static final
oracle.communications.platform.persistence.impl.EntityField _specification
= new
oracle.communications.platform.persistence.impl.EntityField(ExtensionPointRuleSet.class, "specification");
```
class, 'specification');

Taking the example one step further, you can look at the generated source code for the ExtensionPoint, RuleSetEntity, and Specification entities. These entities are defined as the other side of the bi-directional relationships in Example 4–16, "Entity Definition". All three generated source files define the `extensionPointRuleSets` attribute, as defined by the `otherSide` attribute for each.

Understanding Entity Manager Definitions

The persistence framework manages the database and the mapping between the database and the entity classes. An entity manager manages the database tables for a specific functional area. For example, EquipmentManager manages the Equipment table, but it also manages EquipmentHolder, PhysicalPort, PhysicalConnector, PhysicalDevice, and so forth.

Defining Entity Managers

Entity managers are UIM classes. As a result, entity managers are defined in `uim-*-entities.xml` files, and not in `ocim-*-entities.xml` files. Entity managers are defined using the `<manager>` element and `interface` attribute that are defined in the `package.xsd` file. Example 4–19 is an excerpt from the `uim-equipment-entities.xml` file that shows the EquipmentManager entity manager definition.

Example 4–19  Entity Manager Definition

```xml
<manager
    interface="oracle.communications.inventory.api.equipment.EquipmentManager"
    class="oracle.communications.inventory.api.equipment.impl.EquipmentManagerImpl"/>
```

Upon installation of UIM, every entity manager that is defined in the metadata has a corresponding entity manager and implementation of the manager. For example, based on the excerpt shown in Example 4–19, the following classes exist:

- EquipmentManager
- EquipmentManagerImpl

If you extend the data model by creating a new entity, you must also create a new entity manager, and implementation of the manager, to manage the entity data. See "Creating New Entity Managers" for more information.

The relationship of entity to entity manager is not one-to-one. For example, `ocim-equipment-entities.xml` defines several entities, each of which defines its own entity interface. An entity interface differs from a manager interface; an entity interface defines the getter and setter methods for entity attributes, while a manager interface defines methods for the entity, such as the createEquipment(), getEquipment(), or updateEquipment() methods. Example 4–20 is an excerpt from the `ocim-equipment-entities.xml` file that shows a portion of the Equipment and EquipmentHolder entity definitions.

Example 4–20  Entity Definition

```xml
<entity type="ocim:Equipment"
    interface="oracle.communications.inventory.api.entity.Equipment"
    accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">

<entity type="ocim:EquipmentHolder"
    interface="oracle.communications.inventory.api.entity.EquipmentHolder"
    accessControlled="true" entityIdSequenceGenerator="EquipmentSeqGen">
```
Both of these entities are extended in the uim-equipment-entities.xml file, as shown in Example 4–21.

**Example 4–21 ManagedBy Declarations**

```xml
<entity type="ocim:Equipment"
    managedBy="oracle.communications.inventory.api.equipment.EquipmentManager"/>

<entity type="ocim:EquipmentHolder"
    managedBy="oracle.communications.inventory.api.equipment.EquipmentManager"/>
```

The `managedBy` tag is only present on entities that are business-interaction enabled. For business-interaction enabled entities, the `managedBy` tag specifies which entity manager manages the entity.
This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) entity life cycles. An entity life cycle refers to an entity having a start to its life, an end to its life, and a defined state at any given point during its life. Life-cycle state transition definitions are part of the UIM metadata, and these definitions can be extended to solve specific business requirements.

An entity can be defined as life-cycle managed in the metadata. Life-cycle managed entities transition through various states throughout the life cycle. The states are determined by the transition definition specified for the entity in the metadata.

The information presented in this chapter describes statically extending UIM, which can result in backwards compatibility issues. See "Backwards Compatibility" for the implications regarding this type of extension.

This chapter contains the following sections:

- About Business Interactions
- Understanding Metadata File Content
- Extending Life Cycles through the Metadata Files
- About Life Cycle Management Interfaces

Note: Before you begin reading about extending life cycles, you should have an understanding of the following concepts described in UIM Concepts:

- Business Interactions
- Life Cycles

About Business Interactions

Business interactions represent business transactions or events that affect products, services, and resources in inventory. They include service requests, sales orders, and network planning projects. Business interactions are modeled in inventory to facilitate change in the inventory, provide traceability, and enable transaction cancellations and changes. They can involve current business transactions, such as service orders, or future planned events, such as grooming projects.

In the UIM user interface (UI), you can switch between business interactions and current inventory by choosing Current on the menu bar. The Current menu has the following options:

- Current: Switches from a business interaction to current inventory.
Recent BIs: Lists the five most recently accessed business interactions.

Search: Opens the Business Interaction Search page. Accessing a business interaction from the Search page switches the current business interaction to the selected business interaction, and also adds the selected business interaction as an option on the Current menu.

Business interactions tie in with transition definitions because the business states through which an entity transitions depend on whether the entity is within the context of a business interaction or current inventory. Each transition definition can define different <from> and <to> business states for business interaction versus current inventory. See Example 5–4, "Create Transition".

Understanding Metadata File Content

Extending an entity to be life-cycle managed, and extending life-cycle state transitions, is done through the metadata files and involves the definitions of:

- Entities
- Enumerations
- Transitions

Understanding Entity Definitions

This section builds upon the information presented in "Understanding Entity Definitions" in Chapter 4, Extending the Data Model.

An entity can be defined as life-cycle managed and business-interaction enabled in the metadata. A business-interaction enabled entity is, by inheritance, automatically a life-cycle managed entity. Conversely, an entity can be defined as life-cycle managed in the metadata without being a business-interaction enabled entity. The elements and attributes used to define an entity as life-cycle managed and business-interaction enabled are defined in the uim-plugin.xsd file. For example, the <lifecycle> element is used to define an entity as life-cycle managed, and the <businessInteractionEnabled> element is used to define an entity as business-interaction enabled.

Example 5–1 is an excerpt from the ocim-number-entities.xml file that shows the TelephoneNumber entity definition. The definition includes the declaration of the life-cycle managed and business-interaction enabled capabilities, which are bolded in the example. The <lifecycle> element defines the stateType attribute, which defines a value of InventoryState. InventoryState is an enumeration and is described in "Understanding Enumeration Definitions".

Example 5–1 Entity Definition

```
<entity type="ocim:TelephoneNumber"
interface="oracle.communications.inventory.api.entity.TelephoneNumber" accessControlled="true">

<!-- **************************** Capabilities ****************************-->
<lifecycle stateType="ocim:InventoryState"/>
<consumable prefix="TN" attribute="telephoneNumber"
assignmentStateType="ocim:AssignmentState">
  <consumer name="ocim:Service" ConfigurationItemEnabled="true"/>
</consumable>
<referenceEnabled prefix="TelephoneNumber" attribute="telephoneNumber"/>
<characteristic spec="ocim:CharacteristicSpecification">
  <characteristicName name="ocim:TNCharacteristic"
  interface="oracle.communications.inventory.api.entity.TNCharacteristic">
```
Understanding Metadata File Content

Extending Life Cycles

Understanding Enumeration Definitions

This section builds upon the information presented in "Understanding Enumeration Definitions" in Chapter 4, *Extending the Data Model*.

About Life-Cycle States

Life-cycle managed entities transition through various states throughout the life cycle. These life-cycle states are defined as enumerations. There are two types of life-cycle states that an entity transitions through: Business states and object states.

- A **business state** represents the current state as a result of a business action such as validate, approve, issue, complete, or cancel.
- An **object state** represents the current state as a result of an object activity such as create, update, or delete.

Business state enumerations are defined in the *-enum-entities.xml* and *-enum-types.xsd* metadata files. Numerous business state enumerations are defined in the metadata upon installation of UIM, and you can extend the business state enumerations to solve business requirements.

Object state enumerations are defined in a Java class and cannot be extended. The object state enumerations are:

- PLANNED
- QUEUED
- ACTIVE
- INACTIVE
- CANCELLED
- DELETED

Understanding Business State Enumerations

**Example 5–2** is an excerpt from the *ocim-enum-entities.xml* file, which defines the InventoryState enumeration type.

**Example 5–2   Enumeration Type Definition**

```xml
<enum type="ocim:InventoryState"
    enumType="oracle.communications.inventory.api.entity.InventoryState"
    adminState="true"/>
```

**Example 5–1, "Entity Definition"** defined the TelephoneNumber entity to be life-cycle managed, and the definition included the stateType attribute value of **InventoryState**, which is an enumeration.

**Example 5–3** is an excerpt from the *ocim-enum-types.xsd* file, which defines the enumeration values for the InventoryState enumeration type. The enumeration type and enumeration values indicate that the TelephoneNumber entity may transition through up to eight business states during its life cycle.
Understanding Metadata File Content

**Example 5–3  Enumeration Values Definition**

```xml
<xs:simpleType name="InventoryState">
  <xs:annotation>
    <xs:documentation>Inventory Status</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="PLANNED"/>
    <xs:enumeration value="PENDING_INSTALL"/>
    <xs:enumeration value="INSTALLED"/>
    <xs:enumeration value="PENDING_UNAVAILABLE"/>
    <xs:enumeration value="UNAVAILABLE"/>
    <xs:enumeration value="PENDING_REMOVE"/>
    <xs:enumeration value="END_OF_LIFE"/>
    <xs:enumeration value="PENDING_AVAILABLE"/>
  </xs:restriction>
</xs:simpleType>
```

Understanding Transition Definitions

*Note:* Transition definitions for current inventory are defined within the `<live>` element; `<live>` displays as **current** in the UIM UI.

A transition defines the intermediate step from one business state to another business state, or from one object state to another object state. For example, within the context of a business interaction, the create transition moves an entity from inception to the initial PENDING_INSTALL business state, and the createComplete transition moves an entity from the PENDING_INSTALL business state to the INSTALLED business state. Similarly, the create transition moves an entity from inception to the initial QUEUED object state, and the createComplete transition moves an entity from the QUEUED object state to the ACTIVE object state.

Transition definitions are defined in files that start with **uim**- and end with `-transitions.xml`. For example, **uim-default-transitions.xml**. The transition definition files are located in the `UIM_Home/cartridges/tools/ora_uim_entity_sdk.zip/src/uim_poms_lib.jar` file.

You can extend business state enumerations, but you cannot extend object state enumerations. For transitions, you can extend both business state and object state transitions.

**Example 5–4** is an excerpt from the **uim-default-transitions.xml** file, which defines the **Create** transition for the business state:

- From inception to PENDING_INSTALL within the context of a business interaction
- From inception to INSTALLED within the context of current inventory

The example also defines the **Create** transition for the object state:

- From inception to QUEUED within the context of a business interaction
- From inception to ACTIVE within the context of current inventory

**Example 5–4  Create Transition**

```xml
<transition name="Create" priority="0" default="true">
  <objectActivity value='CREATE'/>
  <businessState type='ocim:InventoryState'>
```

---

5-4  UIM Developer's Guide
<attribute name="adminState" isCharacteristic="false"/>
<businessInteraction>
  <from/>
  <to>PENDING_INSTALL</to>
</businessInteraction>
<live>
  <from/>
  <to>INSTALLED</to>
</live>
</businessState>
<objectState>
  <businessInteraction>
    <from/>
    <to>QUEUED</to>
  </businessInteraction>
  <live>
    <from/>
    <to>ACTIVE</to>
  </live>
</objectState>
</transition>

### Understanding How Transitions Are Triggered

Transitions can be triggered automatically from within custom code or manually from within the UIM user interface.

For information on the life cycle management interfaces that are available when writing custom code to automatically transition an entity’s life-cycle state, Custom code can be called from:

- Customized user interface
- Rulesets
- Web services

You can manually transition an entity’s life-cycle state from the Actions menu on the Summary page of any entity that is defined as life-cycle managed. The Actions menu options reflect the applicable transitions defined for the entity, based on the entity’s current state.

**Note:** Manually transitioning through an entity’s life cycle by selecting the options on the Actions menu implies that the correct life-cycle state is dependent on user interaction to initiate the transition.

Example 5–5 is an excerpt from the uim-default-transitions.xml file, which defines the Activate and Deactivate transitions. The Activate and Deactivate transitions are shown in Figure 5–1, "Summary Page Actions Menu".

The example defines the Activate transition for the business state:

- From UNAVAILABLE to PENDING_AVAILBLE within the context of a business interaction
- From UNAVAILABLE to INSTALLED within the context of current inventory

The example defines the Activate transition for the object state:

- Nothing is defined within the context of a business interaction
From INACTIVE to ACTIVE within the context of current inventory

The example defines the **Deactivate** transition for the business state:

- From INSTALLED to PENDING_UNAVAILABLE within the context of a business interaction
- From INSTALLED to UNAVAILABLE within the context of current inventory

The example defines the **Deactivate** transition for the object state:

- Nothing is defined within the context of a business interaction
- From ACTIVE TO INACTIVE within the context of current inventory

**Example 5–5 Activate and Deactivate Transitions**

```xml
<transition name="Activate" priority="0" default="true">
  <businessAction type="ocim:ResourceAction" value="ACTIVATE"/>
  <businessState type="ocim:InventoryState">
    <attribute name="adminState" isCharacteristic="false"/>
    <businessInteraction>
      <from>UNAVAILABLE</from>
      <to>PENDING_AVAILABLE</to>
    </businessInteraction>
    <live>
      <from>UNAVAILABLE</from>
      <to>INSTALLED</to>
    </live>
  </businessState>
  <objectState>
    <live>
      <from>INACTIVE</from>
      <to>ACTIVE</to>
    </live>
  </objectState>
  ...
</transition>

<transition name="Deactivate" priority="0" default="true">
  <businessAction type="ocim:ResourceAction" value="DEACTIVATE"/>
  <businessState type="ocim:InventoryState">
    <attribute name="adminState" isCharacteristic="false"/>
    <businessInteraction>
      <from>INSTALLED</from>
      <to>PENDING_UNAVAILABLE</to>
    </businessInteraction>
    <live>
      <from>INSTALLED</from>
      <to>UNAVAILABLE</to>
    </live>
  </businessState>
  <objectState>
    <live>
      <from>ACTIVE</from>
      <to>INACTIVE</to>
    </live>
  </objectState>
  ...
</transition>
```

Figure 5–1 shows the Telephone Number Summary page Actions menu, which reflects the applicable transitions defined for the TelephoneNumber entity based on its current
state. The telephone number, shown in the context of current inventory, has an inventory status of **Installed**, so **Deactivate** is the only available transition option. If you select **Deactivate**, the inventory status changes to **Unavailable**, and **Activate** becomes the only available transition option. When in the context of a business interaction, the value of the **Inventory Status** field reflects the states defined for `<businessInteraction>` based on the entity’s current state. When in the context of current inventory, the value of the **Inventory Status** field reflects the status defined for current (live) based on the entity’s current state.

**Figure 5–1  Summary Page Actions Menu**

![Summary Page Actions Menu](image)

**About Transition Groups**

A transition group provides the ability to associate a group of transition definitions with a specification. A transition group requires a name, which is used to associate it with a specification. A transition group can be associated with multiple specifications. By default, transition definitions defined within a transition group are templates. Templates are transition definitions that are not active/searchable until the group in which they are defined is associated with a specification. Within a `uim-*-transitions.xml` file, the `<transitionGroup>` element can define the `templateOnly` optional attribute, which defaults to **true**. If set to **true**, the transition definitions in the group are active/searchable, even though they are not yet associated with a specification.

Example 5–6, "Transitions Definition" shows an example of a transition group.

**Extending Life Cycles through the Metadata Files**

You extend life cycles by creating new metadata files.

---

**Caution:** Do not make modifications to the existing metadata files. See "Backwards Compatibility" for the issues involved with modifying the existing metadata files.
The metadata files are contained in the `UIM_Home/cartridges/tools/ora_uim_entity_sdk.zip/src/uim_poms_lib.jar` file.

You can use Oracle Communications Design Studio to create new metadata files. For example, you can import the `ora_uim_entity_sdk.zip` file and create any new XML or XSD files within the imported project.

This section builds upon information presented in Chapter 4, "Extending the Data Model". Any new metadata files you create must reside in `ora_uim_entity_sdk` project to be picked up by the generator. See "Applying Metadata Static Extensions" for more information.

### Extending Entity Definitions

You can extend an entity definition to be business-interaction enabled, life-cycle managed, or both.

#### Defining an Entity as Life-Cycle Managed

The presence of the `<lifecycle>` element in the entity definition defines an entity as life-cycle managed. `stateType` is a required attribute of the `<lifecycle>` element. The value of `stateType` is an enumeration type that is defined in the `ocim-enum-entities.xml` file. This file defines several enumeration types that can be specified for `stateType`. Each enumeration type defines a set of enumeration values that represent the states of a specific life cycle. `initialState` is an optional attribute of the `<lifecycle>` element. `initialState` defines an enumeration that represents the default initial life-cycle state.

To define a new entity as life-cycle managed, add the `<lifecycle>` element to the entity definition in the new `*-entities.xml` file. See "Defining New Entities" for more information.

To define an existing entity as life-cycle managed, add the `<lifecycle>` element to the existing entity by extending the entity definition in the new `*-entities.xml` file. See "Extending Existing Entities" for more information.

Any new entity files you create must end with `-entities.xml` and reside in the `ora_uim_entity_sdk/src/api` directory, to be picked up by the entity generator.

#### Defining an Entity as Business-Interaction Enabled

The presence of the `<businessInteractionEnabled>` element in the entity definition defines an entity as business-interaction enabled. `history` is an optional attribute of the `<businessInteractionEnabled>` element. The `history` attribute is a boolean: if it is set to `true`, the version object is kept in the data store; if it is set to `false`, the version object is deleted. (Versioning is not covered in this guide. For information about versioning, see UIM Concepts.) `visibilityState` is also an optional attribute of the `<businessInteractionEnabled>` element. `visibilityState` defines an enumeration that is the default initial display level for the business-interaction enabled entity.

To define a new entity as business-interaction enabled, add the `<businessInteractionEnabled>` element to the new entity definition in the new `*-entities.xml` file. See "Defining New Entities" for more information.

To define an existing entity as business-interaction enabled, add the `<businessInteractionEnabled>` element to the existing entity by extending the entity definition in the new `*-entities.xml` file. See "Extending Existing Entities" for more information.
Any new entity files you create must end with `-entities.xml` and reside in the `ora_uim_entity_sdk/src/api` directory to be picked up by the entity generator.

**Note:** If an entity is inherited from a business-interaction enabled entity, the entity can not be defined as business-interaction enabled.

### Defining an Entity as Life-Cycle Managed and Business-Interaction Enabled

A business-interaction enabled entity is, by inheritance, automatically a life-cycle managed entity. The presence of the `<businessInteractionEnabled>` element in the entity definition defines an entity as business-interaction enabled and as life-cycle managed. However, the presence of the `<lifecycle>` element in the entity definition is still required to specify `stateType`. Example 5–1, "Entity Definition" showed both the `<lifecycle>` and `<businessInteractionEnabled>` elements in the entity definition.

### Extending Enumeration Definitions

You can create new enumeration files to address business requirements. New files you create must end with `-entities.xml` or `-types.xsd`. New files that follow this naming convention, and that reside the `ora_uim_entity_sdk/src/api` directory, are picked up by the entity generator.

See "Defining New Enumerations" and "Extending Existing Enumerations" for more information.

### Extending Transition Definitions

When extending transitions by either creating new transitions or extending existing transitions, look at the existing `uim-*-transitions.xml` files for examples. Any new entity files you create that end with `-transitions.xml` and that reside in the `ora_uim_entity_sdk/src/api` directory are picked up by the entity generator.

The following transition procedures state how to create a new file, but you do not need to create a new transitions file for each new transition. For example, you can optionally define all new transitions and extending existing transitions in the same file.

#### Defining New Transitions

To add a new transition to a new `*-transitions.xml` file:

   
   The file name must end with `-transitions.xml`. For example, `myNewFile-transitions.xml`.

2. Open an existing `uim-*-transitions.xml` file.

3. Copy and paste a transition definition from the existing file to your new file.

4. Modify the copied transition definition as needed:
   
   a. Change the name of the transition to reflect the name of your new transition.
   
   b. Remove or update the tags to reflect the definition of your new transition.

#### Extending Existing Transitions

To extend an existing transition in a new `*-transitions.xml` file:

The file name must end with -transitions.xml. For example, myNewFile-transitions.xml.

2. Open the existing uim*-transitions.xml file that you plan to extend.

3. Copy and paste the transition definition from the existing file to your new file.

4. Modify the copied transition definition as needed:
   a. Add additional <businessState> elements as needed.
   b. Do not change the transition name.
   c. If the copied transition does not define the priority attribute, add it and set the value to 1 (the default is 0). If the copied transition already defines the priority attribute, increase the value. The priority attribute value is used when the transition name is not unique. The higher the value, the higher the priority.

**Updating Properties Files**

If you extend life cycles, you need to update some properties files that are used to display life-cycle statuses. The following properties files are located in the UIM_Home/config/resources/logging directory:

- **status.properties**
  This file defines statuses that are referenced by the UI. If life cycles are extended by introducing new statuses through the metadata transition files, and the statuses are referenced by the UI, the status.properties file must be updated to reflect the new statuses.

- **enum.properties**
  This file defines enumerations that are referenced by the UI. If life cycles are extended by introducing new enumerations through the metadata enumeration files, and the enumerations are referenced by the UI, the enum.properties file must be updated to reflect the new enumerations.

**Updating Security**

If you extend life cycles, you need to update security for any new actions to display in the UIM UI.

To update security:

1. Log in to the Enterprise Manager Console.

2. In the navigation panel, expand Application Deployments and navigate to the UIM deployment (oracle.communications.inventory).

3. Right-click on oracle.communications.inventory and select Security, then select Application Policies.
   The Application Policies page appears.

4. Expand Search, enter search criteria, and search for existing policies.
   The search results display.

5. Select the policy to be modified and click Edit.
   The Edit Application Grant page appears.

6. Click Add to add a new permission.
   The Add Permission page appears.
7. In the Customize section, enter the following details:
   - **Permission Class**: oracle.security.jps.ResourcePermission
     **Resource Name**: resourceType=PAGE_ACTION, resourceName=CustomObject.MY_TRANSITION
   - **Permission Actions**: view
8. Click OK to add the permission.
9. Click OK to grant the application.

More on Transition Definitions
The following information is provided to help you define *-transitions.xml files. Each transition file can define multiple transition definitions, and each transition definition can define multiple states. Example 5–6, “Transitions Definition” includes all the possible elements and attributes described below.
   - `<transition>` can be defined multiple times within the same file.
     - **name** is required and should be unique. If duplicate transition names are found, the one with the higher **priority** attribute value is used.
     - **entityType** is optional. If it is not specified, the transition definition is available for all entity types.
     - **priority** is optional, and has a default value of 0. The higher the value, the higher the priority. The value is used when name is not unique. If the same name and same priority are specified, an error occurs.
   - `<specification>` is optional. If it is not specified, the transition definition is available for entities with any specification.
   - `<businessAction>` and `<objectActivity>` are optional, but one of them must be specified. These values are used by the lookup process to determine the transition definition.
   - `<businessState>` can be defined multiple times within a transition. This defines the business states that the entity transitions through during its life cycle.
     - `<businessState>` can be set on an entity’s attribute or a custom attribute.
     - **type** must be a valid enumeration.
     - **isCharacteristic** indicates whether `<businessState>` is an attribute or a custom attribute.
     - **name** is either the attribute name or the custom attribute name.
     - If **isCharacteristic** is set to true, you can specify the **characteristicSpecName** attribute. If this attribute is not set, the system uses the **name** attribute value as the characteristic-specific name.
     - `<businessState>` can optionally define zero, one, or many `<businessInteraction>` blocks, or zero or one `<live>` block, or both.
     - If `<businessInteraction>` is defined, its `<from>` state is used to match the entity’s current business state if the transition happens within the context of a business interaction. If `<from>` is not specified, it is considered a wild card and can be matched with any entity’s current state.
     - If `<live>` is defined, its `<from>` state is used to match the entity’s current business state if the transition happens within the context of current inventory.
– If the transition happens within the context of a business interaction and
  <businessInteraction> is not defined, the search for a match continues.
  Similarly, if the transition happens within the context of current inventory and
  <live> is not defined, the search for a match continues.

■ The <businessInteraction> block and <live> block can define multiple <from>
states. This allows matching multiple <from> states without defining them
separately in each <businessState> block. If <from> is not specified, it is
considered a wild card and can be matched with any entity’s current state.

■ There can be only one <to> state defined in the <businessInteraction> and <live>
blocks. The value is used to set the entity’s business state. If <to> is not specified,
the entity’s current state is not changed.

■ Only one <objectState> block can be defined for the transition definition.

■ <objectState> can define zero, one, or many <businessInteraction> blocks, and
  zero or one <live> block, and each can define multiple <from> states.

■ The <dependants> block defines the methods to retrieve the dependent entities
and how to transition them. Multiple <dependants> blocks can be defined in a
transition definition.
  – attribute is the attribute name of the parent entity and is used to hold the
    dependent entities by the parent entity.
  – isCollection is a boolean that indicates whether attribute holds a collection
    (true) or a single dependent entity (false).
  – If the dependent entity is accessed indirectly through the weak reference of the
    attribute, then weakReference is the name of the access method to resolve the
    weak reference. For example, the BusinessInteraction entity has an items
    attribute that holds a collection of BusinessInteractionItem entities, but the
    BusinessInteractionItem entity has a weak reference that refers to the real
dependent entity (TelephoneNumber). The toEntity attribute is specified to
    resolve the TelephoneNumber entity from the BusinessInteractionItem entity.
  – After the dependent entities have been resolved, the system is ready to
    transition the dependent entities with the parent’s business action and object
    activity. However, if useDependentObjectActivity is true, the system uses the
    dependent entity’s object activity and parent entity’s business action to
    transition the dependents.
  – If the parent’s business action is not valid for transitioning the dependents,
you have the option to specify one or more <transitionName> elements in the
  <dependants> block. TransitionNames retrieves the transitions in sequence,
  then uses the transition’s business action and object activity to look up the
  matching transition for each dependent. If there is no matching transition by
  using transitionNames, the action described in the previous bullet is
  performed.

Example 5–6 Transitions Definition

<transitionGroup name="defaultBusinessInteractionGroup" templateOnly="false">
  <transition name="BusinessInteractionCreate"
    entityType="ocim:BusinessInteraction" assignable="true" priority="0">
    <objectActivity value="CREATE"/>
    <businessState type="ocim:BusinessInteractionState">
      <attribute name="adminState" isCharacteristic="false"/>
      <businessInteraction>
        <from/>
    </businessInteraction>
  </transition>
</transitionGroup>
About Life Cycle Management Interfaces

The following sections describe life cycle management interfaces. For information on the methods defined by any of these interfaces, see the Javadoc. For instructions on how to access the Javadoc, see "Javadoc Documentation".

LifeCycleManaged

An entity that is defined as life-cycle managed in the metadata automatically implements the LifeCycleManaged interface. It is not necessary to include the tag:

<implements interface="oracle.communications.inventory.api.LifeCycleManaged"/>

The LifeCycleManaged interface:

- Defines a business state for the entity

  A business state represents the current state as a result of a business action such as validate, approve, issue, complete, or cancel.
- Defines an object state for the entity
  An object state represents the current state as a result of an object activity such as create, update, or delete.

**TransitionManager**

An entity that is defined as life-cycle managed in the metadata automatically implements the LifeCycleManaged interface. This enables you to call methods on the oracle.communications.inventory.api.common.TransitionManager interface, which takes in a LifeCycleManaged entity as an input parameter.

The TransitionManager interface:

- Defines methods that take in a business action and appropriately transition the business and object states
- Automatically updates the business state and object state of any life-cycle managed dependent entities when the parent life-cycle managed entity business state or object state is updated
- Provides the ability to associate or disassociate a specification with a transition group

**Transition Definition Search**

The transition() method provides the ability to transition through the defined business states and object states. To do this, it must first determine the transition definitions for business state and object state that apply to the entity. This is accomplished through a search that takes place within the transition() method.

The transition() method input parameters are the life-cycle managed entity, business action, and object activity. The life-cycle managed entity parameter contains entity type and specification, which are used in the transition definition search. If no match is found, a less relevant search is performed until a transition definition is found. The following lists the search criteria in the most-significant to least-significant order. The least-relevant transition definition returned would be the default transition definition.

1. Business action, object activity, entity type, specification
2. Business action, object activity, entity type
3. Business action, object activity

At this point, the search has returned one or more transition definitions that matched the criteria. This list of transition definitions is now interrogated to find one that defines a <from> business state that matches the entity’s current business state. Whether the entity is within the context of a business interaction determines which <from> business state is interrogated: Business interaction or current (live).
This chapter provides information on extending the topology in Oracle Communications Unified Inventory Management (UIM). The topology is a graphical representation of the spatial relationships and connectivity among your inventory entities.

The topology uses a specific set of entities and a specific algorithm to determine the path between any two entities. This algorithm is called the path analysis. You can extend the topology to include additional entities in the topology, and you can modify the path analysis to suit your business needs.

The information presented in this chapter describes statically extending UIM, which can result in backwards compatibility issues. See "Backwards Compatibility" for the implications regarding this type of extension.

This chapter contains the following sections:

- About Topology Entities and Topology-Managed Entities
- About Topology Mapping
- Extending the Topology
- About Path Analysis
- Configuring and Customizing Path Analysis
- About Topology Interfaces
- About the topologyProcess.properties File

**Note:** Before you begin reading about extending topology, it is important that you have an understanding of the following subjects described in *UIM Concepts*:

- Connectivity
- Topology

**About Topology Entities and Topology-Managed Entities**

Topology entities are defined in the metadata and are used to display the topology. Topology-managed entities are also defined in the metadata and are indirectly used to display the topology. UIM maps topology-managed entities to one of two topology entities, and, as a result of the mapping, topology-managed entities indirectly display in the topology.
About Topology Entities and Topology-Managed Entities

Topology Entities

The metadata defines the following topology entities:

- TopologyEdge
- TopologyNode

TopologyNode entities represent locations, network nodes, or devices, and TopologyEdge entities represent pipes or network edges.

The metadata defines the topology entities in the topology-entities.xml file. Example 6–1 is an excerpt from this file that shows the definition of the TopologyNode entity.

Example 6–1  topology-entities.xml

```xml
<entity type="ocim:TopologyNode"
  interface="oracle.communications.inventory.api.entity.TopologyNode"
  accessControlled='true' entityIdSequenceGenerator="TopologySeqGen">
  <implements interface="java.lang.Cloneable"/>
  <implements interface="oracle.communications.inventory.api.entity.common.TopologyObject"/>
  <attribute name="isTopLevelNode" index='true'/>
  <attribute name="geometry" spatial='true'/>
  <relationship name="businessObject">
    <thisSide inverse='true'/>
    <otherSide dependent='true' type="ocim:TopNodeAssociation"
      attribute="topologyNode"/>
  </relationship>
</entity>
```

The TopologyEdge entity is also defined in the topology-entities.xml file in the same manner.

---

Note: There are actually several topology entities defined in the topology-entities.xml file that support topology. However, within the context of extending topology, this chapter focuses solely on the TopologyEdge and TopologyNode entities.

---

Topology-Managed Entities

The metadata defines the following entities as topology-managed:

- Equipment
- GeographicPlace
- LogicalDevice
- Network
- NetworkEdge
- NetworkNode
- PhysicalDevice
- Pipe

The metadata defines these entities as topology-managed throughout the various *-entities.xml files. Example 6–2 is an excerpt from the equipment-entities.xml file. The example shows the entity definition for PhysicalDevice, which includes the
implementation of the TopologyObject interface. Implementing the TopologyObject interface in the entity definition is what defines an entity as topology-managed.

Example 6–2 Topology-Managed Entity Definition

```xml
<entity type="ocim:PhysicalDevice"
interface="oracle.communications.inventory.api.entity.PhysicalDevice" accessControlled="true"
entityIdSequenceGenerator="PhyDeviceSeqGen">
  <implements interface="oracle.communications.inventory.api.entity.common.PhysicalResource"/>
  <implements interface="java.lang.Cloneable"/>
  <implements interface="oracle.communications.inventory.api.entity.common.TopologyObject"/>
  <implements interface="oracle.communications.inventory.api.entity.common.PhysicalMappingObject"/>
  <implements interface="oracle.communications.inventory.api.entity.common.NetworkNodeEnabled"/>
  . . .
</entity>
```

About Topology Mapping

Entities defined as topology-managed in the metadata are mapped to either TopologyEdge or TopologyNode by the UIM-provided TopologyMapperImpl class.

TopologyEdge

The following topology-managed entities are mapped to TopologyEdge:

- NetworkEdge
- Pipe

TopologyNode

The following topology-managed entities are mapped to TopologyNode:

- Equipment
- GeographicPlace
- LogicalDevice
- Network
- NetworkNode
- PhysicalDevice

Note: The GeographicPlace entity is defined as topology-managed in the UIM metadata, and the UIM mapping logic indirectly maps this entity to TopologyNode. The mapping logic actually checks for GeographicLocation and GeographicSite, not GeographicPlace. GeographicPlace is a parent to GeographicLocation and GeographicSite. A place becomes a topology object when it is associated to a resource such as Logical Device or Physical Device.

Extending the Topology

To extend the topology:

1. Determine entities that you plan to define as topology-managed. (This step is performed by the business analyst, who relays the information to the developer.)
2. Determine the mapping of each topology-managed entity to TopologyEdge or TopologyNode. (This step is performed by the business analyst, who relays the information to the developer.)

3. Define identified entities as topology-managed in the metadata by creating new ext-*-entities.xml files. See "Defining an Entity as Topology-Managed" for more information.

4. Regenerate the entities to pick up the new ext-*-entities.xml files. See "Applying Metadata Static Extensions" for more information.

5. Extend the mapping logic to include the mapping of any additional entities defined as topology-managed in the metadata. See "Extending the Mapping" for more information.

**Defining an Entity as Topology-Managed**

An entity can be defined as topology-managed through a new file in the metadata.

---

**Caution:** Do not modify existing metadata files. See "Backwards Compatibility" for the issues involved with making additions to the existing metadata files.

---

To define a new entity as topology-managed, add the <implements> element to the entity definition in the new *-entities.xml file to implement the TopologyObject interface. See "Defining New Entities" for more information.

To define an existing entity as topology-managed, add the <implements> element to the entity by extending the entity definition in the new *-entities.xml file to implement the TopologyObject interface. See "Extending Existing Entities".

**Extending the BusinessObjectType.java File**

If you define an entity as topology-managed in the metadata, you must also extend the BusinessObjectType class by modifying it to include an enumerated value for that entity. This provides the ability to keep a weak reference between the topology entity and the business object.

For example, the BusinessObjectType class defines the BusinessObjectType enumeration, and you must assign an enumerated value to any entities you define as topology-managed:

```java
/**
 * This class defines the business IDs for mapping Business objects to TopologyEdges and TopologyNodes in the topology model.
 * Every different business entity must have a unique ID.
 * Once a value has been set it cannot be changed.
 */
public enum BusinessObjectType {
    LogicalDeviceDao(1), GeographicPlaceDao(2), PipeDao(3),
    PhysicalDeviceDao(4), NetworkDao(5), NetworkNodeDao(6),
    NetworkEdgeDao(7), EquipmentDao(8), PhysicalConnectorDao(9),
    PhysicalPortDao(10), EquipmentHolderDao(11), CustomObjectDao(12),
    ServiceDao(13), GeographicSiteDao(14), ServiceConfigurationVersionDao(15),
    TopologyOnly(9999);
```

*Caution:* Do not modify existing metadata files. See "Backwards Compatibility" for the issues involved with making additions to the existing metadata files.
Extending the Mapping

Entities defined as topology-managed in the metadata must be mapped to TopologyEdge or TopologyNode by extending the TopologyMapperImpl class. This class is located in the oracle.communications.inventory.api.topology package.

Configuring the topologyProcess.properties file

If you extend the mapping, you must also configure the topologyProcess.properties file to point to your new mapper class. For example, the file includes the following upon installation, and you must configure it to point to your new mapper class instead:

```
# mapperClass - The Class Object that maps the business model to Topology
mapperClass=oracle.communications.api.topology.mapper.impl.TopologyMapperImpl
```

About Path Analysis

Path analysis is an automated process in UIM that helps you locate and assign pipes for enablement. You specify a starting point (the source), an ending point (the target), and a variety of optional criteria. Path analysis evaluates possible paths based on the criteria you provide and returns paths from which you can select. See UIM Concepts for more information.

Path analysis uses the topology to find paths.

Configuring and Customizing Path Analysis

Path analysis evaluates connections based on topology-managed entity data. Only entities in the topology are included in path analysis. You can configure and customize path analysis, as described in the following sections.

Configuring the Path Analysis Mode

Path analysis can use two different algorithms to determine paths:

- The Complex algorithm (the default) considers all possible paths between end points, which means evaluating a large number of permutations. You can use filtering to limit the amount of data to be processed. This mode of path analysis is suitable for complex networks with many possible connections.

- The Simple Linear algorithm works by iteratively analyzing paths working from the end points toward a common node. This mode of analysis is suited to relatively simple scenarios where paths are inherently linear and include 10 or fewer hops, such as POTS. The Simple Linear algorithm has less impact on system performance than the Complex algorithm.

You can use the topologyProcess.properties file to configure path analysis. For example, the properties file includes the following upon installation:

```
# Path Analysis Properties
simpleLinearMode=false
simpleLinearModeMaxCycles=5
continueProcessingIndicator=true
```

- The `simpleLinearMode` parameter is used to denote the path analysis mode. The default value is `false`, indicating that Complex mode is the default path analysis mode.
You can extend path analysis so that Simple Linear mode is used when analyzing paths for particular pipe specifications, even when Complex mode is used for the application in general. See "Customizing Path Analysis" for more information.

- The `simpleLinearModeMaxCycles` parameter denotes the number of connected neighbors that a Simple Linear path analysis finds before determining that a path cannot be found. The default value is 5. You can increase the value if path analysis fails to find paths.

- The `continueProcessingIndicator` parameter denotes whether UIM will try to find a path with the Complex mode if no path can be found by using Simple Linear mode. The default value is `true`, indicating that if no path is found using Simple Linear mode, path analysis continues by attempting to find a path using Complex mode. Setting the value to `false` indicates that if no path is found using Simple Linear mode, path analysis stops.

**Customizing Path Analysis**

You can use rulesets to customize path analysis. By associating rulesets to individual Pipe specifications, you can tailor path analysis to meet various business scenarios.

A sample ruleset is provided with UIM to serve as a starting place for three types of customization:

- Adding additional filter criteria to the analysis. See "Adding Filtering Criteria" for more information.

- Setting Simple Linear mode for path analysis involving a particular Pipe specification. See "Setting the Analysis Mode" for more information.

- Specifying that only pipes based on particular specifications be included in a path analysis. See "Limiting the Analysis by Pipe Specification" for more information.

The `PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA` sample ruleset is included in the `UIM_Home/cartridges/sample/ora_uim_pathanalysis_sample` cartridge.

You can customize path analysis by appending code to the body of the rule. The sample rule includes examples of each of the three types of customizations mentioned in this section.

The ruleset is applicable to Pipe specifications and must be associated with the `PathAnalysisManager_findPaths` base extension point and the `oracle.communications.inventory.api.entity.PipeSpecification` enabled extension point. The placement of the ruleset extension point must be BEFORE.

**Adding Filtering Criteria**

You can add filtering criteria to a path analysis. Filtering criteria restrict the amount of data that UIM considers when locating paths, reducing the amount of processing required.

---

**Note:** Before changing the value of this parameter, you need to be certain that the Simple Linear mode is appropriate for your needs. Path analysis will not find some kinds of paths in this mode.
Configuring and Customizing Path Analysis

Extending the Topology

For example, you can limit the analysis to consider only nodes or edges that include particular characters in their names or only pipes in a particular status. Including the following code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset limits the path analysis to pipes in the Installed state.

```
filterStr.append("businessObject.referenceId == vPipe.ext:getColumn('ENTITYID')
    ");
filterStr.append(" && vPipe.adminState == pStatus ");
params.add('pStatus');
values.add(InventoryState.INSTALLED);
criteria.setAppendQuery (params, values, filterStr.toString());
```

Setting the Analysis Mode

You can configure path analysis to use Simple Linear mode when enabling pipes based on a particular specification. Including the following code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset sets the mode to Simple Linear when the rule runs. It also sets values for the SimpleLinearModeMaxCycles and ContinueProcessingIndicator parameters.

```
criteria.setSimpleLinearMode(true);
criteria.setSimpleLinearModeMaxCycles(10);
criteria.setContinueProcessingIndicator(true);
```

Limiting the Analysis by Pipe Specification

You can limit the pipe analysis so that it considers only transport pipes based on a particular specification. For example, you can filter out trunk and ISDN lines that are not valid connections for POTS. Similarly, if there are cables between a switch and an MDF that are not used for POTS, you can exclude them from the pipe analysis.

```
Note:  You can also limit path analysis to particular Pipe specifications by including a specification in the Transport configuration item of a Pipe configuration.
```

For example, including the follow code in the PATHANALYSIS_FINDPATHS_SETCUSTOMCRITERIA ruleset limits the path analysis to pipes based on the Sample Terminated Pipe specification:

```
SpecManager sm = InventoryHelper.makeSpecManager();
SpecSearchCriteria specCriteria = sm.makeSpecSearchCriteria();
CriteriaItem critSpecName = specCriteria.makeCriteriaItem();
critSpecName.setValue("SampleTerminatedPipe");
critSpecName.setOperator(CriteriaOperator.EQUALS_IGNORE_CASE);
specCriteria.setName(critSpecName);
List<Specification> specs = sm.findSpecifications(specCriteria);
ArrayList includeSpecs = new ArrayList();
for (Specification pipespec : specs){
    includeSpecs.add(new Long(pipespec.getEntityId()));
}
criteria.setIncludeSpecifications(includeSpecs);
```

Note: Because the additional criteria are defined using standard JPAQL syntax, knowledge of JPAQL is required to implement this feature.
About Topology Interfaces

You can use the topology interfaces when writing rulesets or Web services to meet business requirements that involve extending the topology or customizing path analysis.

The following sections describe the available topology interfaces. For information on the methods defined by any of these interfaces, see the Javadoc. For instructions on how to access the Javadoc, see "Javadoc Documentation".

TopologyObject is the only topology interface described in this section that is available to all entities. Defining an entity to implement this interface makes the entity topology-managed. Topology-managed entities must be mapped to TopologyEdge or TopologyNode.

The remaining interfaces described in this section are available to TopologyEdge and TopologyNode entities. Example 6–3 is an excerpt from the uim-common-entities.xml file showing the common manager interfaces defined for the entities, including TopologyEdge and TopologyNode.

Example 6–3  uim-common-entities.xml Manager Interfaces

```xml
<manager interface="oracle.communications.inventory.api.framework.policy.SearchPolicy"
  class="oracle.communications.inventory.api.framework.policy.impl.SearchPolicyImpl"/>
<manager interface="oracle.communications.inventory.api.common.TransitionManager"
  class="oracle.communications.inventory.api.common.impl.TransitionManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.AttachmentManager"
  class="oracle.communications.inventory.api.common.impl.AttachmentManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.SequenceGenerator"
  class="oracle.communications.inventory.api.common.impl.SequenceGeneratorImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ConsumerManager"
  class="oracle.communications.inventory.api.consumer.impl.ConsumerManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.AssignmentManager"
  class="oracle.communications.inventory.api.consumer.impl.AssignmentManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.ConfigurationInputManager"
  class="oracle.communications.inventory.api.common.impl.ConfigurationInputManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ConditionManager"
  class="oracle.communications.inventory.api.consumer.impl.ConditionManagerImpl"/>
<manager interface="oracle.communications.inventory.api.consumer.ReservationManager"
  class="oracle.communications.inventory.api.consumer.impl.ReservationManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.FederationManager"
  class="oracle.communications.inventory.api.common.impl.FederationManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.EntityIdGenerator"
  class="oracle.communications.inventory.api.common.impl.EntityIdGeneratorImpl"/>
<manager interface="oracle.communications.inventory.api.admin.SecurityManager"
  class="oracle.communications.inventory.api.admin.impl.SecurityManagerImpl"/>
<manager interface="oracle.communications.inventory.api.topology.TopologyManager"
  class="oracle.communications.inventory.api.topology.impl.TopologyManagerImpl"/>
<manager interface="oracle.communications.inventory.api.topology.mapper.TopologyProfileMapper"
  class="oracle.communications.inventory.api.topology.mapper.impl.TopologyProfileMapperImpl"/>
<manager interface="oracle.communications.inventory.api.topology.mapper.PathAnalysisMapper"
  class="oracle.communications.inventory.api.topology.mapper.impl.PathAnalysisMapperImpl"/>
```
class="oracle.communications.inventory.api.role.impl.RoleManagerImpl"/>
<manager interface="oracle.communications.inventory.api.common.RowLockManager"
class="oracle.communications.inventory.api.common.impl.RowLockManagerImpl"/>
<manager interface="oracle.communications.inventory.api.framework.policy.LockPolicy"
class="oracle.communications.inventory.api.framework.policy.impl.LockPolicyImpl"/>

**TopologyObject**

Package: oracle.communications.api.inventory.entity.common

This interface defines getter methods for the object’s IDs: ID, ENTITYID, and OID. There are no setter methods because these IDs are generated for the object, not set for the object.

**TopologyManager**

Package: oracle.communications.inventory.api.topology

This interface defines methods for finding and maintaining TopologyEdge and TopologyNode entity objects.

**TopologyMapper**

Package: oracle.communications.inventory.api.topology.mapper

This interface defines the business rules for mapping topology-managed entity objects to a TopologyEdge entity object or a TopologyNode entity object.

**PathAnalysisManager**

Package: oracle.communications.inventory.api.topology

This interface defines methods for finding paths (edges and nodes) through the topology network based on specified criteria.

**PathAnalysisMapper**

Package: oracle.communications.inventory.api.topology.mapper

This interface defines the business rules for mapping business object path analysis criteria to values used in the topology model. This object provides a mapping layer between the business model and the topology model for cases where the data in the topology model must be converted from a value in the business model.

**TopologyProfileMapper**

Package: oracle.communications.inventory.api.topology.mapper

This interface defines mapping for service topology. While topology is extended through the metadata, service topology is extended through characteristics, specifications, extension points, and rulesets, all of which can be defined in Oracle Communications Design Studio. UIM provides a service topology sample cartridge that is a working example of how you could extend service topology. See Appendix A, "UIM Sample Cartridges" for more information on the service topology sample cartridge.

**TopologyEdgeSearchCriteria**

Package: oracle.communications.inventory.api.topology
This interface defines the available search criteria for the TopologyEdge entity object and is an input parameter to topology manager and topology mapper interface methods.

**TopologyNodeSearchCriteria**

Package: oracle.communications.inventory.api.topology

This interface defines the available search criteria for the TopologyNode entity object and is an input parameter to topology manager and topology mapper interface methods.

**About the topologyProcess.properties File**

Topology logic references the **UIM_Home/config/resources/event/topologyProcess.properties** file for specifying the mapper class and for configuring path analysis. You can also use this file to:

- Turn off topology updates. If you turn off topology updates, you can rebuild the topology if you need to use a topology-related feature. See *UIM System Administrator’s Guide* for more information.

  For example, the file includes the following upon installation:
  
  ```
  # disableTopology - turns Topology Refresh On or Off
  disableTopology=false
  ```

  - Opt whether to update the topology synchronously or asynchronously with business model updates. See *UIM System Administrator’s Guide* for more information.

    For example, the file includes the following upon installation:

    ```
    # processSynchronous - Topology is refreshed as part of the transaction (true)
    # or asynchronoulsy in a seperate transaction (false)
    processSynchronous=true
    ```
This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) security to include APIs and entity data.

Security for other parts of UIM is handled by external systems, such as the Oracle WebLogic Server Administration Console and Oracle Enterprise Manager. See *UIM System Administrator’s Guide* for more information.

This chapter contains the following sections:

- Securing APIs
- Securing Entity Data

---

**Note:** For information on securing Web services, see:

- Chapter 9, "Integrating UIM through Web Services"
- Chapter 10, "Developing Custom Web Services"
- Appendix B, "Reference Web Service"

---

**Securing APIs**

By default, UIM APIs are not secured. To secure an API, you must extend UIM security to include the APIs. This can be done by:

- Securing APIs through the SecurityValidation Aspect
- Securing APIs through Rulesets and Extension Points

**Securing APIs through the SecurityValidation Aspect**

You can secure access to an API by adding the API method to the UIM-provided security extension point (securityExtensionPoint) definition, which is defined within the SecurityValidation aspect in the *aop.xml* file. See Chapter 8, "Extending UIM through Rulesets" for more information about aspects and the *aop.xml* file.

At the framework level, security is automatically enforced at the security extension point for any methods that the extension point defines. For example, if no API methods are defined for the security extension point within the SecurityValidation aspect, then no APIs are secured. If 20 API methods are defined for the security extension point within the SecurityValidation aspect, then those 20 API methods are validated/secured.

Example 7–1 shows API security definitions that are provided as a comment in the *aop.xml* file. If uncommented, these definitions would secure the createConditions,
updateConditions, and deleteConditions APIs using the SecurityValidation aspect through the specified extension point (securityExtensionPoint). The result of this entry in the aop.xml file is that security validations are run prior to every call to the createConditions, updateConditions, and deleteConditions APIs.

You can use this example as a starting point by modifying it and uncommenting it in the aop.xml file to secure any API.

**Example 7–1 SecurityValidation Aspect**

```xml
<concrete-aspect
   name="oracle.communications.extensibility.extension.SecurityValidation"
   extends=" oracle.communications.extensibility.extension.SecurityValidationExtension">
   <pointcut name="securityExtensionPoint" expression="call(public *
       oracle.communications.inventory.api.consumer.ConditionManager.
       createConditions(java.util.Collection))
   call(public *
       oracle.communications.inventory.api.consumer.ConditionManager.
       updateConditions(java.util.Collection))
   call(public *
       oracle.communications.inventory.api.consumer.ConditionManager.
       deleteConditions(java.util.Collection))"/>
</concrete-aspect>
```

**Creating the Global Extension Point**

Global extension points are created in Oracle Communications Design Studio. For information on global extension points, see Chapter 8, "Extending UIM through Rulesets". For instructions on how to create a global extension point, see the Design Studio Help.

When using this approach to secure APIs, you must also create one global extension point that defines the handleSecurityViolation API, which enables the rulesets to generate errors. The handleSecurityViolation API is located in the oracle.communications.inventory.api.admin.SecurityManager package. **Example 7–2** shows the API method signature to use when defining the global extension point for the handleSecurityViolation API.

**Example 7–2 Custom Global Extension Point Signature**

```java
public void oracle.communications.inventory.api.admin.SecurityManager.
   handleSecurityViolation([])
```

**Creating the Ruleset Global Extension Point**

Ruleset global extension points are created in Design Studio. For information on ruleset global extension points, see Chapter 8, "Extending UIM through Rulesets". For instructions on how to create a ruleset global extension point, see the Design Studio Help.

After you have created the ruleset and global extension point in Design Studio, you must also create the corresponding ruleset global extension point in Design Studio. A ruleset global extension point associates a ruleset with a global extension point, so the global extension point knows which ruleset to run.
Securing APIs through Rulesets and Extension Points

You can also secure access to an API by creating custom rulesets that run at specified extension points. The custom rulesets set permissions for an API, enforces any permissions that are set for an API, and logs error messages whenever a security violation is detected.

Setting and enforcing API permissions through rulesets is done in the same manner as setting and enforcing entity data permissions. See "Securing Entity Data through Rulesets and Extension Points" for more information.

Securing Entity Data

By default, UIM entity data is not secured. To secure entity data, you must extend UIM security to control data access to individual entities. This is done by creating custom rulesets that run at specified extension points. The custom rulesets set permissions or partitions for an entity, enforces any permissions or partitions that are set for an entity, and logs error messages whenever a security violation is detected.

About Entity Access Control

To configure access control for an entity, the entity must be declared as access-controlled in the metadata. For example, the following is an excerpt from the metadata that shows the Equipment entity definition, which is declared as access-controlled:

```
<entity type="ocim:Equipment"
interface="oracle.communications.inventory.api.entity.Equipment"
accessControlled="true">
```

Most, but not all, entities are declared as access-controlled. If you want to configure access control for an entity that is not declared as access-controlled in the metadata, you must first extend the data model to declare the entity as access-controlled. See Chapter 4, "Extending the Data Model" for more information.

Access-controlled entities define additional attributes that contain security-specific data. For example, access-controlled entities define the owner, permissions, and partition attributes. Access-controlled entities also extend the AccessControlled class, so each entity class has access to the setOwner(), setPermissions(), and setPartition() methods defined in the AccessControlled parent class. The value of these attributes can be set by custom rulesets that call these methods.

---

**Note:** When controlling access to a range of entities, the ruleset custom code must iterate through the range and call the method for each entity in the range. See "Securing Entity Data for a Range of Entities Example" for more information.

---

Securing Entity Data through Rulesets and Extension Points

You can secure entity data through rulesets and extension points by:

- Setting Permissions in a Custom Ruleset
- Setting Partitions in a Custom Ruleset
- Enforcing Security in a Custom Ruleset
Setting Permissions in a Custom Ruleset

Note: This section also applies to securing APIs through permissions.

To control data access to an entity through permissions, set the permissions attribute for the entity through custom code that calls the setPermissions() method, which is defined as:

```java
public void setPermissions(String acl);
```

This method is defined in the oracle.communications.inventory.api.AccessControlled class, which is the parent class of all entities that are declared as access-controlled in the metadata. In the custom ruleset, you can call this method on the parent class (AccessControlled) or on the child class (EntityName, such as TelephoneNumber, Equipment, and so forth).

See "Creating Custom Rulesets and Extension Points" for examples of setPermissions() method calls.

Understanding ACL

The permissions are defined as an access control list (ACL). The ACL is a Java string that specifies who is allowed to access an object and what operations they can perform on an object.

An ACL consists of one or more entry statements separated by semicolons. Each statement includes the type of permission (allow or deny), the permission (r for read or w for write), and a principal or role to whom the permission is granted. (A principal is a user or group. It is easier to manage permissions at the level of roles, however.)

The syntax is as follows:

```java
allow|deny r|w = principal|roles[role1,role2,role3...];
```

where principal is the name of any user or group and role is the name of any role.

Example 7–3 shows the ACL syntax in Extended Backus-Naur Format (EBNF).

Example 7–3  ACL Syntax

```
 acl:= acl_entry (‘;’,‘acl_entry)*
 acl_entry:=(‘allow’|‘deny’)(permission? target_list
 permission:= (‘r’|‘w’)’=’
 target_list:= target (‘,’target)*
 target:= principal|’roles’ [[‘role_list’]
 role_list:= role(‘,’role)*
```

Note the following about the ACL:

- The ACL is evaluated left to right until a security decision of allow or deny is enforced.
- If no permission is stated, allow is implied.
  - Allowing write access implies allowing read access.
- Denying read access also implies denying write access.
Any user having the **uimuser** role is permitted full access to an entity, regardless of the permissions set for the entity. This role exists by default and is defined as a superuser.

Table 7–1 lists examples of permissions and how they work together.

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow roles[billing_admin]; deny all</td>
<td>Anyone assigned to the billing_admin role can read or write the entity, but no one else.</td>
</tr>
<tr>
<td>Allow all</td>
<td>Everyone can read or write the entity. The same can be achieved by simply not defining permissions for the entity.</td>
</tr>
<tr>
<td>Allow r=all,w=roles[location_admin]</td>
<td>Everyone can read the entity; anyone having the location_admin role can write the entity.</td>
</tr>
<tr>
<td>Deny all</td>
<td>No one may can the entity except superusers.</td>
</tr>
<tr>
<td>Deny w=all</td>
<td>No one can write the entity except superusers, but everyone may read the entity.</td>
</tr>
<tr>
<td>Deny roles[OrderEntryUser,GeoMapAdmin User]</td>
<td>Anyone having either the OrderEntryUser or the GeoMapAdminUser role is denied access. Everyone else has full access.</td>
</tr>
</tbody>
</table>

**Setting Partitions in a Custom Ruleset**

To control data access to an entity through partitions, set the **partition** attribute for the entity through custom code that calls the setPartition() method, which is defined as:

```java
public void setPartition(String partition);
```

This method is defined in the oracle.communications.inventory.api.AccessControlled class, which is the parent class of all entities that are declared as access-controlled in the metadata. In the custom ruleset, you can call this method on the parent class (AccessControlled) or on the child class (EntityName, such as TelephoneNumber, Equipment, and so forth).

See "Creating Custom Rulesets and Extension Points" for examples of setPartition() method calls.

**Configuring Partitions**

To control data access to an entity through partitions, some additional configuration is required:

1. In the WebLogic Server Administration Console, you must define a user group within a security realm. The group you define represents a data partition in UIM. For instructions on how to do this, see **UIM System Administrator’s Guide**.

   **Caution:** The group name must begin with **ora_uim_partition#** to be recognized by UIM. For example, if you define a group name of **ora_uim_partition#myPartition**, then the custom ruleset would set the partition to **/myPartition**.

2. In the **UIM_Home/config/system-config.properties** file, set the **uim.security.filter.enabled** property to **true**, as shown here:

   ```
   uim.security.filter.enabled=true
   ```
Creating Custom Rulesets and Extension Points

Enforcing Security in a Custom Ruleset

---

**Note:** This section also applies to enforcing security permissions set for APIs.

---

API access that is controlled through set permissions, and entity data access that is controlled through set permissions and partitions is enforced through custom code that calls the checkPermissions() method, which is defined as:

```java
public void checkPermissions(String perm, AccessControlled instance);
```

This method is defined in the `oracle.communications.inventory.api.framework.security.UserEnvironment` class. The checkPermissions() method calls the hasAccessToPartition() method, so the checkPermissions verifies access for both permissions and partitions.

If a security violation is detected, the application throws a `java.security.AccessControlException`. The custom code catches and logs the AccessControlException by calling the error() method, which is defined as:

```java
public void error(String s, Throwable t);
```

This method is defined in the `oracle.communications.inventory.api.framework.logging.Log` class.

See "Creating Custom Rulesets and Extension Points" for examples of error() method calls.

Creating Custom Rulesets and Extension Points

When using custom rulesets to secure an API or entity data, you must also create an extension point or global extension point to run the ruleset. The following sections provide additional information and examples for creating the ruleset and extension point. If creating a global extension point, see "Creating the Global Extension Point" for more information.

Creating Custom Rulesets

Rulesets are created in Oracle Communications Design Studio. For information on rulesets, see Chapter 8, "Extending UIM through Rulesets". For instructions on how to create a ruleset, see the Design Studio Help.

---

**Note:** In the following custom ruleset examples, all import statements are omitted.

Securing APIs Example

Example 7–4 shows a custom ruleset that secures access to the createConditions, updateConditions, and deleteConditions APIs by setting permissions. The ruleset defines four rules:

- Default Condition Validation Rule
  This rule always runs and calls the validate() method, which simply logs the method name and logs the user that is calling the method.

- Create Condition Validation Rule
This rule runs only when the ruleset is called from an extension point that defines the createConditions API. This rule calls the setConditionsOwner() method, which sets permissions.

■ Update Condition Validation Rule
This rule runs only when the ruleset is called from an extension point that defines the updateConditions API. This rule calls the validateConditionsOwner() method, which enforces security and logs an error if a security violation is detected.

■ Delete Condition Validation Rule
This rule runs only when the ruleset is called from an extension point that defines the deleteConditions API. This rule also calls the validateConditionsOwner() method, which enforces security and logs an error if a security violation is detected.

**Example 7–4 Custom Ruleset**

```java
package oracle.communications.rules.
.
.
.global Log log;

function void validate(ExtensionPointRuleContext context, Log logger,
UserEnvironment env) {
    logger.info('', new String[]{"********"});
    logger.info('', new String[]{"method: ", context.getMethodName()});
    logger.info('', new String[]{"user: ", env.getUserName()});
    logger.info('', new String[]{"********"});
}

function void setConditionsOwner(ExtensionPointRuleContext context, Log logger,
UserEnvironment env) {
    logger.info('', new String[]{"********"});
    logger.info('', new String[]{"setConditionsOwner"});
    Collection conditions = (Collection) context.getArguments()[0];
    if (conditions != null && !conditions.isEmpty()) {
        String owner = env.getUserName();
        for (Iterator itr = conditions.iterator(); itr.hasNext();) {
            Condition cond = (Condition) itr.next();
            if (cond instanceof AccessControlled) {
                ((AccessControlled)cond).setOwner( owner );
                ((AccessControlled)cond).setPermissions("deny contractEmployees");
            }
        }
    }
    logger.info('', new String[]{"********"});
}

function void validateConditionsOwner(ExtensionPointRuleContext context, Log logger,
UserEnvironment env) {
    logger.info('', new String[]{"********"});
    logger.info('', new String[]{"validateConditionsOwner"});
    Collection conditions = (Collection) context.getArguments()[0];
    String methodName = context.getMethodName();
    String targetName = context.getDeclaringTargetType().getSimpleName();
    String policyName = targetName + "." + methodName;
    
    for (Iterator itr = conditions.iterator(); itr.hasNext();) {
        Condition cond = (Condition) itr.next();
        if (cond instanceof AccessControlled) {
            ((AccessControlled)cond).setPermissions("deny contractEmployees");
        }
    }
    logger.info('', new String[]{"********"});
}
```
logger.info('', new String[]"policyName: ", policyName));

if (conditions != null && !conditions.isEmpty()) {
    for (Iterator itr = conditions.iterator(); itr.hasNext();) {
        Condition cond = (Condition) itr.next();
        if (cond instanceof AccessControlled) {
            try {
                env.checkPermissions( policyName, (AccessControlled) cond );
            }
            catch (java.security.AccessControlException ace) {
                logger.error('', new String[] ace.getMessage());
                logger.error('', new String[] "My error message for: "+
                cond.toString());
            }
        }
    }
    logger.info('', new String[]"********");
}

rule "Default Condition Validation Rule"
salience 10
when
    context: ExtensionPointRuleContext()
then
    UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
    RuleDebug.breakPoint(context);
    RuleDebug.breakPoint(env);
    validate(context, log, env);
end

rule "Create Condition Validation Rule"
salience 1
when
    context: ExtensionPointRuleContext(methodName == "createConditions")
then
    UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
    setConditionsOwner(context, log, env);
end

rule "Update Condition Validation Rule"
salience 1
when
    context: ExtensionPointRuleContext(methodName == "updateConditions")
then
    UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
    validateConditionsOwner(context, log, env);
end

rule "Delete Condition Validation Rule"
salience 1
when
    context: ExtensionPointRuleContext(methodName == "deleteConditions")
then
    UserEnvironment env = UserEnvironmentFactory.getUserEnvironment();
    validateConditionsOwner(context, log, env);
end
Securing Entity Data through Permissions Example

Example 7–5 shows a custom ruleset that secures access to party entities by setting permissions. The ruleset name implies that it is intended to run when a party is created.

Example 7–5  Custom Ruleset

```java
package oracle.communications.inventory.rules

global Log log;

rule "Create Party with Permissions"
  salience 2
  when
    partyList : Collection()
  then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((partyList != null) && !(partyList.isEmpty())) {
      for (Object partyO : partyList) {
        Party party = (Party)partyO;
        party.setOwner("inv");
        party.setPermissions("allow inv; deny all");
      }
    }
end
```

Retrieving Permissions Information Example

Example 7–6 shows a custom ruleset that retrieves user and role information so you can view the permissions that are set for a user through roles.

Example 7–6  Custom Ruleset

```java
package oracle.communications.inventory.rules

rule "Get Permissions Info"
  salience 2
  when
    true
  then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    String user = env.getUser();
    String userName = env.getUserName();
    Collection roles = env.getRoles();
end
```

Securing Entity Data through Partitions Example

Example 7–7 shows a custom ruleset that secures access to logical device entities by setting a partition. The ruleset name implies that it is intended to run when a logical device is created.

Example 7–7  Custom Ruleset

```java
package oracle.communications.inventory.rules
```
Creating Custom Rulesets and Extension Points

... 

Securing Entity Data for a Range of Entities Example

When securing entity data for a range of entities, the ruleset custom code must iterate through the range and call the access control method for each entity in the range. To do this, you must configure your custom ruleset to run After the API call.

Example 7–8 shows a custom ruleset that secures access to a range of logical devices by iterating through the range of logical devices, and setting a partition for each logical device in the range. The ruleset name implies that it is intended to run when a range of logical devices are created.

Note: Example 7–8 shows the use of the setPartition() method to secure entity data for a range, but the same concept applies when using the setOwner() or setPermissions() methods to secure entity data for a range.

Example 7–8 Custom Ruleset

```java
package oracle.communications.inventory.rules;

// ...

global Log log;

rule "Create LogicalDevice with Partitions"
salience 2
when
    ldList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((ldList != null) && !(ldList.isEmpty())) {
        for (Object ld : ldList) {
            ((LogicalDevice)ld).setPartition("/US_PARTITION/NY_PARTITION");
        }
    }
end

// ...
```

```java
package oracle.communications.inventory.rules;

// ...

global Log log;

rule "Create Range of LogicalDevices with Partitions"
salience 2
when
    ldList : Collection()
then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((ldList != null) && !(ldList.isEmpty())) {
        for (Object obj : ldList) {
            LogicalDevice ld = (LogicalDevice)obj;
            ld.setPartition("/US_PARTITION/NY_PARTITION");
        }
    }
end
```

Note: Example 7–8 shows the use of the setPartition() method to secure entity data for a range, but the same concept applies when using the setOwner() or setPermissions() methods to secure entity data for a range.
Creating Custom Rulesets and Extension Points

Enforcing Security Example

Example 7–9 shows a custom ruleset that enforces security access to a party. The ruleset name implies that it is intended to run when a party is updated.

Example 7–9  Custom Ruleset

```java
package oracle.communications.inventory.rules

global Log log;

rule "Secure Update Party"
  salience 2
  when
    partyList : Collection()
  then
    UserEnvironment environment = UserEnvironmentFactory.getUserEnvironment();
    if ((partyList != null) && !(partyList.isEmpty()))
    {
      for (Object partyO : partyList )
      {
        Party party = (Party)partyO;
        try
        {
          environment.checkPermissions
            (WritePermission.getInstance().toString(), party);
        }
        catch(Throwable t){
          log.error("", t);
        }
      }
    }
end
```

Creating Extension Points

---

**Note:** Check the ora_uim_baseextpts cartridge to determine if any extension points you may need are already defined. Depending on what you are securing, you may or may not need to create new extension points.

---

Extension points are created in Design Studio. For information on extension points, see Chapter 8, “Extending UIM through Rulesets”. For instructions on how to create an extension point, see the Design Studio Help.

When securing APIs, you must create one extension point per API to secure, where each extension point defines the specific API method to secure. In the same vein, when securing entity data, you must create one extension point per entity to secure, where each extension point defines the specific entity method to secure. The same ruleset can be called from multiple extension points. Example 7–10 shows the API method signatures to use when defining the extension point for each API secured by the custom ruleset shown in Example 7–4.

Example 7–10  Custom Extension Point Signatures

```java
public void
oracle.communications.inventory.api.consumer.ConditionManager.createConditions
(java.util.Collection)
```
public void
oracle.communications.inventory.api.consumer.ConditionManager.updateConditions
(java.util.Collection)

public void
oracle.communications.inventory.api.consumer.ConditionManager.deleteConditions
(java.util.Collection)

Creating the Ruleset Extension Point

Ruleset extension points are created in Design Studio. For information on ruleset
extension points, see Chapter 8, "Extending UIM through Rulesets". For instructions
on how to create a ruleset extension point, see the Design Studio Help.

After you have created the ruleset and extension point in Design Studio, you must also
create the corresponding ruleset extension point in Design Studio. A ruleset extension
point associates a ruleset with an extension point, so the extension point knows which
ruleset to run.
This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) using rulesets. A ruleset is custom code that extends existing UIM code at a specified point. Rulesets are built upon the existing open source projects of AspectJ and JBoss Rules.

This chapter contains the following sections:

- About AspectJ
- About JBoss Rules
- About UIM Rulesets
- About the UIM Extensibility Framework
- Extending UIM through Rulesets
- About Base Rulesets
- Troubleshooting

**About AspectJ**

AspectJ is an Eclipse open source project that enables aspect-oriented programming (AOP). AOP provides the ability to insert code at various points across a code base. For example, when the UIM application is started, the AspectJ engine inserts (or weaves) custom extension points into the UIM code stream.

The AspectJ engine is called the Weaver.

**About JBoss Rules**

JBoss Rules is an open source rules engine for accessing, changing, and managing business policies. JBoss Rules provides the ability for a business analyst or auditor to view business rules because they are completely decoupled from the code. This allows for greater flexibility in changing, adding, or removing rules as business needs change.

The JBoss Rules engine is called Drools.

**JBoss Rule Structure**

A JBoss Rule is a two-part structure that defines a left-hand side (LHS) and a right-hand side (RHS). The LHS defines a condition, and the RHS defines an action. When the condition evaluates to true, the action occurs. The action is the custom code. Example 8–1 shows the structure of a JBoss rule.
Example 8–1  JBoss Rule

```
rule "RuleName"
  salience 0
  when
    LHS
  then
    RHS
  end
```

**Note:** The RHS does not have to define a condition. If no condition is defined, the default is true, and the action occurs.

### Rule Attributes

A rule can optionally define attributes, one of which is salience. The salience attribute defines the priority of when a rule runs, which is necessary for cases when multiple rules are fired at the same time. salience is defined as an Integer value; the higher the number, the higher the priority. The rule with the higher priority runs first. If not specified, the salience value defaults to zero.

See the JBoss Rules online Help for information on other available attributes. The JBoss Rules online Help is installed with JBoss Eclipse plug-in. See "Using the JBoss Eclipse Plug-In" for more information.

### Rulesets

A ruleset is a file that contains one or more JBoss rules. A ruleset may also contain:

- A package statement
- Import statements
- Global and local variables
- Functions (very much like Java methods)

The content of a ruleset file is very much like the content of a Java source file, but a ruleset is not a .java file, it is a .drl file. The .drl extension is pronounced “drools”, as in a ruleset is a Drools file.

Example 8–2 shows a ruleset that contains a package statement, import statements, a global variable, a function, and the rule.

Example 8–2  Ruleset

```
package oracle.communications.inventory.rules

import oracle.communications.inventory.api.framework.logging.Log;
import java.util.*;
import java.lang.*;
import oracle.communications.inventory.api.entity.TelephoneNumber;
import oracle.communications.inventory.api.entity.TelephoneNumberSpecification;
import oracle.communications.inventory.extensibility.extension.util.
  ExtensionPointRuleContext;

global Log log;

function String getEditMask(TelephoneNumberSpecification tnSpec) {
  // Set the default edit mask.
  String editMask = "##########";
  if ( tnSpec == null )
```
About UIM Rulesets

UIM rulesets provide the ability to run custom code that extends UIM code at a specified point. The specified point is called an extension point.

UIM rulesets:

- Use JBoss Rules and are enabled by AspectJ
- Functionally extend UIM through custom code
- Dynamically extend UIM because they can be added, changed, or deleted without rebuilding the application or restarting the application server

Within a ruleset, functions must be defined prior to the rule, so the rule can recognize the function when compiling. However, when a ruleset runs, execution begins at the rule name; it does not begin at any functions that the ruleset may define prior to the rule.

Rule Action

The Rule action is the custom code, which may reside:

- In the LHS of the rule
- In a function within the ruleset that is called from the LHS
- In a separate Java class that is called from the LHS

If the custom code is short and simple, placing it in the LHS of the rule, or within a function in the ruleset, is fine. If the custom code is even slightly complex, Oracle recommends that you place it in a separate Java class. The advantage of placing custom code in a separate Java class is the use of the Java editor. For example, the Java editor catches syntax errors, creates import statements, and provides a list of method names when you type a class name.

---

Note: The ruleset always runs, but this does not mean that the custom code always runs. Whether or not the custom code runs depends on the outcome of the rule condition:

- If the condition evaluates to true, the custom code runs
- If the condition evaluates to false, the custom code does not run

---

return editMask;

// Set the edit mask based on specification name
if(tnSpec.getName().equals("TNspec NPA-NXX"))
   editMask = "###-###-####";
return editMask;
}

rule 'Get TN Edit Mask'
   salience 0
when
   telephoneNumberSpecification : TelephoneNumberSpecification()
   context : ExtensionPointRuleContext()
then
   String editMask = getEditMask(telephoneNumberSpecification);
   context.setReturnValue(editMask);
end
About UIM Rulesets

- Are provided in the ora_uim_baserulesets cartridge (base rulesets)
- Are created in Oracle Communications Design Studio within an Inventory project (custom rulesets)
- Are deployed into UIM as part of a cartridge

To understand how rulesets work, you must understand the following terms:

- **Ruleset**
- **Extension Point**
- **Ruleset Extension Point**
- **Enabled Extension Point**

**Ruleset**

A ruleset defines a JBoss Rule, or multiple JBoss Rules. A ruleset can define custom code, or call a separate Java class that defines custom code. UIM provides several base rulesets in the ora_uim_baserulesets cartridge, and also provides a framework that enables you to create custom rulesets.

Custom rulesets are created in Design Studio in the Ruleset editor, and in the corresponding Drools file editor, which you access from the Ruleset editor. See "Creating Rulesets" for more information.

**Extension Point**

An extension point defines a UIM API method signature to establish a specific point in the code at which to call a ruleset. UIM provides several base extension points in the ora_uim_baseextpts cartridge, and also provides a framework that enables you to create custom extension points.

Custom extension points are created in Design Studio in the Extension Point editor, and in a corresponding custom aop.xml file. See "Creating Extension Points" for more information.

An extension point is defined as specification-based or global, based on the Design Studio Extension Point editor Global check box. If the Global check box is not selected, the extension point is specification-based. If the Global check box is selected, the extension point is global.

Specification-based extension points pertain to a particular specification, and global extension points do not.

**Specification-Based Extension Points**

The signature argument for specification-based extension points must define a specific UIM entity object, such as TelephoneNumber, Equipment, or Pipe; or define a generic object, such as a java.util.Collection that can contain specific UIM entity objects. For example, the TelephoneNumberManager_createTelephoneNumbers base extension point defines the following signature:

```java
public abstract interface java.util.List
oracle.communications.inventory.api.number.TelephoneNumberManager.
createTelephoneNumbers(java.lang.String, java.lang.String,
oracle.communications.inventory.api.entity.TelephoneNumber)
```

The signature defines TelephoneNumber as the method argument. This indicates the extension point is intended to be used with the Telephone Number specification.
Global Extension Points
The signature argument for global extension points is not restricted; it may define any
type of argument, including no argument at all. For example, the
TimeoutEventListener_timerExpired base global extension point defines the following
signature:
public void
oracle.communications.inventory.api.common.TimeoutEventListener.timerExpired()

Extension Point Type
Extension points also define a type, which dictates how the extension point is weaved
into the UIM code stream. There are two extension point types:
- Execution
- Call

To understand extension point types, you must first understand ruleset extension
points. See "Ruleset Extension Point" and "Understanding Extension Point Type and
Ruleset Placement".

Ruleset Extension Point
A ruleset extension point configures a ruleset to run at an extension point, and
configures the placement of the ruleset with respect to the method signature defined
by the extension point. UIM provides no ruleset extension points, but does provide a
framework that enables you to create ruleset extension points.

Through a ruleset extension point, you can configure:
- A base ruleset to run at a base extension point
- A base ruleset to run at a custom extension point
- A custom ruleset to run at a base extension point
- A custom ruleset to run at a custom extension point

Through a ruleset extension point, you can also configure the placement of a ruleset
(with respect to the method defined by the extension point) to run:
- Before the method
- After the method
- Instead of the method

Ruleset extension points are created in Design Studio in the Ruleset Extension Point
and Ruleset Extension Point Global editors. See "Creating Ruleset Extension Points"
and "Creating Ruleset Extension Points Global" for more information.

Understanding Extension Point Type and Ruleset Placement
An extension point defines a type of execution or call, and a ruleset extension point
defines a placement of before, after, or instead. Together, this information dictates how
the extension point is weaved into the UIM code stream, as explained using the
following figures.

Figure 8–1 represents the UIM code stream. Within the UIM code stream, method A is
shown, as well as calls to UIM method A from various places within the UIM code
stream. The dots within method A represent executable lines of code.
Figure 8–1  UIM Code Stream

Call to method A

Method A

Call to method A

Figure 8–2, Figure 8–3, and Figure 8–4 show an extension point type of execution, which dictates the extension point is weaved within the method defined by the extension point. For this type, the extension point is weaved in only one place: within the method itself.

Figure 8–2 represents a ruleset configured to run before the method. For this type of configuration, the extension point is weaved into the method, immediately prior to the first line of the method’s executable code. The result is the ruleset custom code runs before the method runs.

Figure 8–2 Type Execution, with Placement Before

Call to method A

Method A

extension point

Call to method A

Figure 8–3 represents a ruleset configured to run after the method. For this type of configuration, the extension point is weaved into the method, immediately following the last line of the method’s executable code. The result is the ruleset custom code runs after the method runs.

Figure 8–3 Type Execution, with Placement After

Call to method A

Method A

Call to method A

Call to method A

extension point

Figure 8–4 represents a ruleset configured to run instead of the method. For this type of configuration, the extension point is weaved into the method, and the method’s executable code does not run. The result is the ruleset custom code runs instead of the method.

Figure 8–4 represents a ruleset configured to run instead of the method. For this type of configuration, the extension point is weaved into the method, and the method’s executable code does not run. The result is the ruleset custom code runs instead of the method.
About UIM Rulesets

Figure 8–4  Type Execution, with Placement Instead

Call to method A

extension point

Call to method A

Method A

// . . .
// . . .

type point

Figure 8–5, Figure 8–6, and Figure 8–7 show an extension point type of call, which dictates the extension point is weaved at the call to the method defined by the extension point. For this type, the extension point may be weaved in multiple places: at each place from where the method is called.

Figure 8–5 represents a ruleset configured to run before the method. For this type of configuration, the extension point is weaved into the UIM code stream, immediately prior to the method call. The result is the ruleset custom code runs before the method runs.

Figure 8–5  Type Call, with Placement Before

extension point

Call to method A

extension point

Call to method A

Method A

. . .

Figure 8–6 represents a ruleset configured to run after the method. For this type of configuration, the extension point is weaved into the UIM code stream, immediately after the method call. The result is the ruleset custom code runs after the method runs.

Figure 8–6  Type Call, with Placement After

Call to method A

extension point

Method A

. . .

Call to method A

extension point

Figure 8–7 represents a ruleset configured to run instead of the method. For this type of configuration, the extension point is weaved into the UIM code stream, and the method is not called. The result is the ruleset custom code runs instead of the method.
Runtime performance is not affected by extension point type; however, server startup performance is affected because that is when custom extension points are weaved, and there is more to weave for type call. For this reason, Oracle recommends that extension points be defined as type execution.

Based on this recommendation, you cannot specify extension point type in the Design Studio Extension Point editor; all extension points default to type execution. However, there are cases when you may need to use type call. For example, if your custom code needs to know the calling class for processing reasons, or needs to know if the call originated from a Web service for processing reasons. In such cases, you must define type call, and there is a way to do this. See “Creating Extension Points” for more information.

The benefit of specifying type call is the ruleset can retrieve the caller from the ExtensionPointRuleContext. The drawback of specifying type call is the extension point does not fire (custom code does not run) if the extension point is called by a method defined in the same class or subclass.

For example, Figure 8–8 shows ClassA, which defines methods x() and y(), and y() calls x(). ClassB extends ClassA and defines method z(), and z() also calls x(). The aop.xml file defines an extension point for method x() of type call. The extension point fires when method x() is called from anywhere outside ClassA or ClassB; but the extension point does not fire when method x() is called from y() or z(), because x() is called from within the same class or subclass.
Enabled Extension Point

Note: Enabled extension points are used only with specification-based extension points; they are not used with global extension points.

An enabled extension point enables a specification-based extension point for a particular specification. UIM provides several base enabled extension points in the ora_uim_baseextpts cartridge, and also provides a framework that enables you to create enabled extension points.

Enabling a specification-based extension point for a particular specification is accomplished by associating an entity specification Java class, to a specification-based extension point. To understand an enabled extension point, you must first understand that, for specification-based extension points, you must configure the specification for a ruleset extension point. This configuration is done on the Rules tab of any specification editor, where you simply select a ruleset extension point from a list. The list is populated for the specification, based on extension points that are enabled for the specification. If no extension points are enabled for the specification, no ruleset extension points are available for selection on the Rules tab of the specification.

For example, if 10 extension points are defined, along with 10 ruleset extension points, and no enabled extension points are defined; the Equipment Specification editor Rules tab lists no ruleset extension points from which to choose. However, if 10 extension points are defined, along with 3 ruleset extension points, and 3 of these extension points are enabled for the EquipmentSpecification Java class through enabled extension points; the Equipment Specification editor Rules tab lists 3 ruleset extension points from which to choose.


About the UIM Extensibility Framework

The extensibility framework supports the functionality that rulesets and extension points provide. The following sections describe various parts of the extensibility framework that are critical to understanding how rulesets work.
Supplying Classes

The following sections describe classes that support ruleset functionality.

**RulesExecutor**

Package: oracle.communications.inventory.extensibility.rules

This class defines the methods listed below, which support the execution of rulesets. While the extensibility framework provides the ability to automatically run rulesets at extension points, you can also write custom code that directly runs a ruleset by calling the execute() method on this class. See the Javadoc for information on this class. For instructions on how to access the Javadoc, see "Javadoc Documentation".

- load()
- execute()
- unload()

**ExtensionPointContext and ExtensionPointRuleContext**

Package: oracle.communications.extensibility.extension.util

ExtensionPointRuleContext extends ExtensionPointContext.

For any given extension point, ExtensionPointRuleContext is constructed and made available to the ruleset as an argument. The extensibility framework adds the ExtensionPointContext as an argument, following any arguments defined by the extension point signature.

For extension points of type call, the context contains the calling class. This is provided so custom code can process differently based on the caller. For example, the custom code may need to perform a different process if called from the UI, versus being called from a Web service. In this scenario, the custom code can use the context’s getCaller method to make the determination. For extension points of type execution, the context does not contain the calling class. So, the getCaller method should not be used for extension points of type execution because the return is always null.

Regardless of type, the context contains the target class and method arguments. Method arguments are placed into the argument collection in left-to-right parameter order. Integral types are placed in the corresponding wrapper object. For example, int arguments are passed by reference using an Integer.

**ExtensionPointRuleContext.returnValue**

Data is returned from an extension point ruleset in the `ReturnValue` attribute defined in the ExtensionPointRuleContext class. For example, you use the ExtensionPointRuleContext.setReturnValue(Object) method to set the `ReturnValue` attribute. The placement of the ruleset affects the use of the `ReturnValue` attribute as follows:

- **Before**
  The ruleset runs before the intercepted method. If the ruleset populates the `ReturnValue` attribute, there is no effect because the intercepted method removes any `ReturnValue` set by the ruleset.

- **After**
  The ruleset runs after the intercepted method. Data in ExtensionPointRuleContext is available to the ruleset to manipulate. The ruleset can change the `ReturnValue` attribute either by setting a new return object in the context or by changing
attribute values of the return object already in the context. For this scenario, the return value type must match the value type that is normally returned by the intercepted method or an exception is thrown.

- Instead

The ruleset runs instead of the intercepted method. The ruleset completely controls what is returned to the caller by setting the \texttt{returnValue} attribute. For this scenario, the return value type must match what is normally returned by the intercepted method or an exception is thrown.

For an example of the use of ExtensionPointRuleContext, view the TELEPHONE\_NUMBER\_FORMATTING ruleset that is provided in the ora\_uim\_baserulesets cartridge. See "Viewing Base Rulesets" for more information.

\textbf{aop.xm}l File

The \texttt{UIM\_Home/config/extensibility/META-INF/aop.xml} file is provided as example to follow when creating custom extension points, which is a two-part process: Creating the extension point in the Design Studio Extension Point editor, and creating a custom \texttt{aop.xml} file. Both are deployed into UIM as part of a cartridge.

The custom \texttt{aop.xml} file must reside in the cartridge’s \texttt{model/aspects} directory, and must be named \texttt{aop.xml}. This file is used at UIM application startup to weave custom extension points into the UIM code stream.

Example 8–3 is an excerpt from the provided \texttt{aop.xml} file, and shows all of the XML elements that the file defines, as well as several of the example extension points that the file defines. (Many of the extension point definitions were removed for readability.)

\textbf{Example 8–3 \texttt{aop.xml} File}

```xml
<aspectj>
  <!--
  <aspects>
  <concrete-aspect name="oracle.communications.inventory.extensibility.extension.SpecAsTarget"
      extends="oracle.communications.inventory.extensibility.extension.SpecTargetExtension">
    <pointcut name="ruleExtensionPoint" expression="execution(public * oracle.communications.inventory.api.impl.entity.SpecificationDAO.getName(..))"/>
  </concrete-aspect>
  <concrete-aspect name="oracle.communications.inventory.extensibility.extension.SpecBasedAsArgument"
      extends="oracle.communications.inventory.extensibility.extension.SpecBasedArgumentExtension">
    <pointcut name="ruleExtensionPoint" expression="call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.createTelephoneNumbers(String, String, oracle.communications.inventory.api.entity.TelephoneNumber, java.util.Set, java.util.List))
        || call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.deleteTelephoneNumbers(oracle.communications.inventory.api.entity.TelephoneNumber...))
        || call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.updateTelephoneNumbers(java.util.List, java.util.Set, java.util.List))"/>
  </concrete-aspect>
  </aspects>
  <!--
  <aspect name="oracle.communications.inventory.extensibility.extension.SpecAsTarget"
      extends="oracle.communications.inventory.extensibility.extension.SpecTargetExtension">
    <pointcut name="ruleExtensionPoint" expression="execution(public * oracle.communications.inventory.api.impl.entity.SpecificationDAO.getName(..))"/>
  </aspect>
  <aspect name="oracle.communications.inventory.extensibility.extension.SpecBasedAsArgument"
      extends="oracle.communications.inventory.extensibility.extension.SpecBasedArgumentExtension">
    <pointcut name="ruleExtensionPoint" expression="call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.createTelephoneNumbers(String, String, oracle.communications.inventory.api.entity.TelephoneNumber, java.util.Set, java.util.List))
        || call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.deleteTelephoneNumbers(oracle.communications.inventory.api.entity.TelephoneNumber...))
        || call(public * oracle.communications.inventory.api.number.TelephoneNumberManager.updateTelephoneNumbers(java.util.List, java.util.Set, java.util.List))"/>
  </aspect>
  </aspectj>
```

"oracle.communications.inventory.extensibility.extension.SpecAsArgument"
   extends="oracle.communications.inventory.extensibility.extension.SpecArgumentExtension">
   <pointcut name="ruleExtensionPoint" expression="
      call(public * oracle.communications.inventory.api.consumer.ReservationManager.extendReservation(oracle.communications.inventory.api.entity.ServiceSpecification, java.util.List, java.lang.String, oracle.communications.inventory.api.entity.ReservedForType))
      || call(public * oracle.communications.inventory.api.consumer.ReservationManager.reserveResource(oracle.communications.inventory.api.entity.ServiceSpecification, java.util.Collection, oracle.communications.inventory.api.entity.common.Reservation))
      || call(public * oracle.communications.inventory.api.service.ServiceManager.createService(oracle.communications.inventory.api.entity.Service, oracle.communications.inventory.api.entity.ServiceSpecification))="/\n   </concrete-aspect>

<concrete-aspect name="oracle.communications.inventory.extensibility.extension.SecurityValidation"
   extends="oracle.communications.inventory.extensibility.extension.SecurityValidationExtension">
   <pointcut name="securityExtensionPoint" expression="
      call(public * oracle.communications.inventory.api.group.InventoryGroupManager.createInventoryGroup(oracle.communications.inventory.api.entity.InventoryGroup))
      || call(public * oracle.communications.inventory.api.group.InventoryGroupManager.deleteInventoryGroup(oracle.communications.inventory.api.entity.InventoryGroup))
      || call(public * oracle.communications.inventory.api.group.InventoryGroupManager.updateInventoryGroup(oracle.communications.inventory.api.entity.InventoryGroup))="/\n   </concrete-aspect>
</aspects>

<weaver>
   <include within= ""/>
The provided `aop.xml` file defines the following:

- **aspects**
  
  This element defines the concrete extensions through the `<concrete-aspect>` element. The implemented aspects are:

  - **SpecBasedAsArgument**
    This `<concrete-aspect>` element defines extension points for method signatures that define specification-based arguments. An example of a specification-based argument is an instance of an entity, such as Equipment. For example, the `Equipment.createPhysicalPorts(Equipment equip, List physPorts)` method defines an argument of Equipment.

  - **SpecAsArgument**
    This `<concrete-aspect>` element defines extension points for method signatures that define specification arguments. An example of a specification argument is a specification itself, such as `TelephoneNumberSpecification`. For example, the `SpecManager.getEditMask(TelephoneNumberSpecification)` method defines an argument of `TelephoneNumberSpecification`.

  - **SpecBasedAsTarget**
    This `<concrete-aspect>` element defines extension points for method signatures defined on a specification-based object. An example of a specification-based object is an instance of an entity, such as `Equipment`. For example, when the `Equipment.createPhysicalPorts(Equipment equip, List physPorts)` method is called, `Equipment` is the target of the invocation. Oracle recommends that you do not use this type of aspect.

  - **SpecAsTarget**
    This `<concrete-aspect>` element defines extension points for method signatures defined on a specification object. An example of a specification object is a specification itself, such as `EquipmentSpecification`. For example, when the `EquipmentSpecification.setModelNumber(String modelNbr)` method is called, `EquipmentSpecification` is the target of the invocation. Oracle recommends that you do not use this type of aspect.

  - **GlobalRule**
    This `<concrete-aspect>` element defines global extension points for method signatures that define arguments that are neither a specification nor specification-based. For example, `ReservationManager.expireReservation(boolean)` is a method defined as a global extension point.

  - **SecurityValidation**
    This `<concrete-aspect>` element defines extension points for APIs that require authorization to access. Extension points defined in this element are neither specification-based nor global because they are not part of the extensibility
Extending UIM through Rulesets

The following sections describe:

- Creating Rulesets
- Creating Ruleset Extension Points
- Creating Ruleset Extension Points Global
Extending UIM through Rulesets

- Creating Enabled Extension Points
- Configuring a Specification for a Ruleset Extension Point
- Validating and Compiling Rulesets
- Deploying Rulesets
- Running Rulesets
- Steps for Extending UIM through Rulesets

It is assumed that:

- You are in Design Studio with the Inventory features installed.
- You understand what a cartridge is, how to create one in Design Studio, and how to deploy one into UIM.
- You understand about Design Studio perspectives and views, and how to switch between them.

Creating Rulesets

For instructions on how to create a ruleset in Design Studio, see the Design Studio Help. When creating a ruleset, use the following information.

Name Field
When entering the name of the ruleset in the Name field, the text editor forces you to enter all capitals. Oracle recommends that you use underscores for readability, such as MY_RULE_SET.

DRL File
A ruleset resides in a DRL file, which you access from the Ruleset editor. When writing the ruleset, Oracle recommends that you copy the DRL file from a base ruleset to the DRL file from your custom ruleset, and modify it as needed. The base rulesets provide great examples of rulesets. See "Viewing Base Rulesets" for more information.

When writing a ruleset, Oracle recommends that you install the JBoss Eclipse plug-in in Design Studio. The plug-in provides JBoss-specific menu options, JBoss online Help, and a JBoss Rule editor that catches syntax errors when writing the ruleset, and compiles a ruleset prior to deploying the cartridge that contains the ruleset. See "Installing the JBoss Eclipse Plug-In" for information on downloading, installing, configuring, and using the JBoss Eclipse plug-in.

All DRL files are saved in the inventory project’s model directory. You can view this directory by switching to the Java perspective Package Explorer view, and expanding the inventory project.

Creating Extension Points

**Note:** Before creating a custom extension point, check the ora_uim_baseextpts cartridge to see if a base extension point already exists that defines the UIM API method you need to use.

Before you create an extension point, you must first determine the UIM API method signature that you want the extension point to define. For example, you may want to create an extension point that deals with disassociating a telephone number from an
inventory group. To make this determination, access the Javadoc and search for the manager class using a search of *.Manager.class to return a list of all manager classes such as TelephoneNumberManager, EquipmentManager, or PipeManager. After locating the appropriate manager class, search the list of methods for the most likely method. For example, the TelephoneNumberManager class defines the disassociateTN() method.

For instructions on how to access the Javadoc, see "Javadoc Documentation".

For instructions on how to create an extension point in Design Studio, see the Design Studio Help. When creating an extension point, use the following information.

**Name Field**

When entering a name, Oracle recommends that you follow the same naming convention used in the ora_uim_baseextpts cartridge, which is ClassName_methodName. For example TelephoneNumberManager_disassociateTN.

**Point Name Field**

When entering a point name, Oracle recommends that you follow the same naming convention used in the ora_uim_baseextpts cartridge, which is ClassName.methodName. For example, TelephoneNumberManager.disassociateTN.

**Signature Field**

Defining the Signature field correctly is critical for the ruleset to run. An incomplete signature, incorrect spellings, or spacing errors result in the ruleset not executing. Oracle recommends that you copy the signature from the UIM_Home/lib/uim-core-interfaces.txt file, and paste it into the Signature field. The uim-core-interfaces.txt file provides a generated listing of all API method signatures.

Copy the text from public abstract interface through the end of the line.

*Example 8–4* shows a typical signature definition:

```java
public abstract interface java.lang.String
oracle.communications.inventory.api.businessinteraction.BusinessInteractionManager.getEntityAction(
oracle.communications.inventory.api.entity.common.RootEntity,
oracle.communications.inventory.api.entity.BusinessInteraction, java.lang.String)
```

A signature requires the following:

- **Visibility modifier (public, private, protected)**
  
  The visibility modifier must be defined as public abstract interface, as shown in Example 8–4. The existence of the Javadoc for the method indicates that the method is a public interface. AspectJ requires that all methods defined for an extension point be declared as abstract, even if the method in the Java code is not defined as abstract.

- **Return**
  
  This part of the signature defines the return values. In Example 8–4, getEntityAction() returns a java.lang.String.

- **Fully qualified method call, which includes:**
  
  - Package
In Example 8–4, the package that contains the BusinessInteractionManager class is oracle.communications.inventory.api.businessinteraction.

- Class
  In Example 8–4, the class is BusinessInteractionManager.

- Method
  In Example 8–4, the method is getEntityAction().

- Arguments
  In Example 8–4, the arguments are the fully qualified objects of RootEntity, BusinessInteraction, and String.

Putting all parts together results in the signature being defined as shown in Example 8–4.

When entering the signature:
- Characters that define the signature are case sensitive.
- No extra spaces can exist within the signature.
- If the signature defines multiple arguments, the arguments are separated by a comma followed by a space.
- Signatures that define arrays must use the notation of “[]” to represent the array of objects. The notation of “...” to represent an array of objects is not recognized by the AspectJ framework. See "aop.xml File".
- Signatures that define an array of objects as a parameter must contain the keyword transient. The AspectJ framework requires this keyword to retrieve the extension point, as shown in Example 8–5.

Example 8–5  Signature with Transient and Array

```
public abstract transient interface oracle.communications.inventory.api.entity.BusinessInteraction
oracle.communications.inventory.api.businessinteraction.BusinessInteractionManager.transferItems(or
acle.communications.inventory.api.entity.BusinessInteraction,
oracle.communications.inventory.api.entity.BusinessInteraction,
oracle.communications.inventory.api.entity. BusinessInteractionItem[])
```

Caution: Do not copy the signature from the Javadoc and paste it into the Signature field: Copying from an HTML file results in spacing errors. If copied from the Javadoc, it may appear to look correct, but not be correct. Avoiding these spacing errors is critical because it is difficult to determine why an extension point does not fire because you do not get an error; the extension point simply does not fire.

Creating the Custom aop.xml File

Creating a custom extension point is a two-part process: Creating the extension point in the Design Studio Extension Point editor, and creating a custom aop.xml file. Both are deployed into UIM as part of a cartridge.

To create a custom aop.xml file:

1. In Design Studio, create an Inventory project.
2. Switch to the Java perspective.
3. Expand the inventory project, and expand the **model** directory. (The **model** directory gets created when the inventory project is created.)

4. Create a new directory named **aspects** within the **model** directory.

5. Create a new file named **aop.xml** in the **model/aspects** directory.

6. Model the contents of the custom **aop.xml** file after the **UIM_Home/config/extendibility/META-INF/aop.xml** file. See "**aop.xml File**" for more information on the **aop.xml** file.

7. Determine the aspect of the extension point, which is based on the method signature defined by the extension point. See "**aop.xml File**" for more information on aspects.

8. Define the extension point within the determined aspect:
   
a. Define the type of call or execution. See "**aop.xml File**" for more information on type.
   
b. In the **aop.xml** file, the return value is always an asterisk (*). See "**aop.xml File**" for more information.
   
c. Define the package, class, method, and arguments.

9. Delete any **<concrete-aspect>** elements that do not define any extension points.

10. Ensure that the **<weaver>** element is not commented out.

11. Update the **<weaver>** include elements to reflect the correct package or packages that are applicable to your custom extension point.

12. Ensure the weaver is turned on in the **setUIMEnv.cmd** file. See "**aop.xml File**" for more information.

---

**Caution:** The concrete-aspect name must be unique across all extension points installed on your application server or the following error appears on server startup: "Error Attempt to concretize but chosen aspect name already defined: name in aop.xml warning register definition failed."

To fix this error, change the concrete-aspect name. Do not change the extends portion of the concrete-aspect. Example 8–6, "Concrete-Aspect Name" shows the before and after reflecting this change.

---

**Example 8–6  Concrete-Aspect Name**

// Before
<concrete-aspect name="oracle.communications.extendibility.extension.SpecBasedAsArgument"
extends="oracle.communications.extendibility.extension.SpecBasedArgumentExtension">
<pointcut name="ruleExtensionPoint" expression="call (public abstract interface java.util.List
oracle.communications.inventory.api.connectivity.PipeManager.updatePipes(java.util.Collection))"/>

// After
<concrete-aspect name="oracle.communications.extendibility.extension.SpecBasedAsArgumentXyz"
extends="oracle.communications.extendibility.extension.SpecBasedArgumentExtension">
<pointcut name="ruleExtensionPoint" expression="call (public abstract interface java.util.List
oracle.communications.inventory.api.connectivity.PipeManager.updatePipes(java.util.Collection))"/>
Creating Ruleset Extension Points

The ruleset extension point configures a ruleset to run at a specification-based extension point, and configures the placement of the ruleset.

For instructions on how to define a ruleset extension point, see the Design Studio Help.

Creating Ruleset Extension Points Global

The ruleset extension point global configures a ruleset to run at a global extension point, and configures the placement of the ruleset.

For instructions on how to define a ruleset extension point global, see the Design Studio Help.

Creating Enabled Extension Points

---

**Note:** Enabled extension points are used only with specification-based extension points; they are not used with global extension points.

---

Create an enabled extension point for every specification-based extension point you create. For instructions on how to create an enabled extension point, see the Design Studio Help. When creating an enabled extension point, use the following information.

**Name Field**

When entering a name, Oracle recommends that you follow the same naming convention used in the ora_uim_baseextpts cartridge, which is `SpecificationName_ClassName_methodName`. For example, `usTelephoneNumber_TelephoneNumberManager_disassociateTN`.

**Specification Class Name Field**

You must select a value for **Class Specification Name**, which is preloaded with the available entity specification Java class names. The selection list displays the fully qualified Java class names.

Nearly all of the Studio Inventory entities are recognizable by their Java class name. For example, the Telephone Number Specification entity displays in the list as `oracle.communications.inventory.api.entity.TelephoneNumberSpecification`, and the Equipment Specification entity displays in the list as `oracle.communications.inventory.api.entity.EquipmentSpecification`.

The only exceptions to this recognizable naming convention are the configuration specifications:

- Logical Device Configuration Specification
- Network Configuration Specification
- Pipe Configuration Specification
- Place Configuration Specification
- Service Configuration Specification

To enable an extension point for a configuration specification, you must select the `oracle.communications.inventory.api.entity.InventoryConfigurationSpec` class, which
extends the **Configuration Version Instance Type** field. See "**Configuration Version Instance Type Field**" for more information.

### Configuration Version Instance Type Field

The **Configuration Version Instance Type** field is only enabled when you select `oracle.communications.inventory.api.entity.InventoryConfigurationSpec` for the **Specification Class Name** field.

When enabled, you may optionally select a value for **Configuration Version Instance Type**, which is preloaded with the available entity configuration specification Java class names. The selection list displays the fully qualified Java class names of:

- `oracle.communications.platform.entity.impl.LogicalDeviceConfigurationVersionDAO`
- `oracle.communications.platform.entity.impl.NetworkConfigurationVersionDAO`
- `oracle.communications.platform.entity.impl.PipeConfigurationVersionDAO`
- `oracle.communications.platform.entity.impl.PlaceConfigurationVersionDAO`
- `oracle.communications.platform.entity.impl.ServiceConfigurationVersionDAO`

If you select a value for **Configuration Version Instance Type**, the extension point is enabled for the selected entity configuration specification. If you do not select a value for **Configuration Version Instance Type**, the extension point is enabled for all of the entity configuration specifications.

### Configuring a Specification for a Ruleset Extension Point

**Note:** This section applies only to a ruleset extension point; it does not apply to ruleset extension point global.

To run a ruleset that is configured to run at a specification-based extension point, you must also configure the specification for the ruleset extension point. This configuration is done in Design Studio, on the Rules tab of any Specification editor. For example, Figure 8–9 shows that, when you click **Select** to select a ruleset extension point, only the ruleset extension points that are enabled for the Equipment Specification display.

For instructions on how to configure a specification for a ruleset extension point, see the Design Studio Help.
Validating and Compiling Rulesets

Rulesets are validated at the following times:

- As you write a ruleset, the JBoss Rule editor validates syntax to prevent compilation errors. The JBoss Rule editor is not part of UIM; it is part of the JBoss Eclipse plug-in. See "Installing the JBoss Eclipse Plug-In".
- When you build the project, validations are performed to ensure that required values are supplied, and that the specification configured for a ruleset extension point complies with the definitions in the enabled extension points.

Rulesets are compiled at the following times:

- When the application server is started and there are uncompiled rulesets (such as after an upgrade), rulesets are compiled and the serialized compilation is stored in the database. If compilation errors are encountered, the startup fails and the errors are cited.
- When a cartridge that contains rulesets is deployed, the rulesets are compiled and the serialized compilation is stored in the database. If the compilation errors are encountered, the deployment fails and the errors are cited.

Deploying Rulesets

Deploying a cartridge that contains rulesets and extension point is no different than deploying any other cartridge. See "About Deploying Cartridges and Technology Packs" for more information.

Running Rulesets

Rulesets can be run manually or automatically.

Manually Running Rulesets

Rulesets can be run manually from within UIM by clicking the Execute Rule link located in the Administration group of the Tasks panel on the main window. Manually running rulesets is commonly used to manage UIM data. For example, you can use the
SYSTEM_EXPORT base ruleset to export data from one environment, and the
SYSTEM_IMPORT base ruleset to load the exported data into another environment.
See "About Base Rulesets" for more information.

For instructions on manually running a ruleset from within UIM, see the UIM Help.

Automatically Running Rulesets
More commonly, rulesets run automatically after they are deployed into UIM: When
an event occurs that runs an existing UIM method that was defined as an extension
point, the extensibility framework calls the RulesExecutor.execute() method, which
runs the ruleset associated with the extension point. If the ruleset condition evaluates
to true, the ruleset custom code runs.

Steps for Extending UIM through Rulesets
Now that you have an understanding of rulesets and extension points, the following
high-level steps are provided as a summary:

1. Determine the functionality that you plan to extend and how you plan to extend it.
2. Create an Inventory project in which you can write custom Java code to extend the
   functionality.
3. Write a ruleset that calls the custom Java code.
4. Determine if you need to create a custom extension point by checking the
   UIM-provided extension points contained in the UIM_Home/cartridges/base/
   ora_uim_baseextpts cartridge.
5. If needed, create a custom extension point. This is a two-part process: Custom
   extension points are define in the Design Studio Extension Point editor, and in a
   custom aop.xml file.
6. If you create a custom extension point that is specification-based, create an enabled
   extension point.
7. Create a ruleset extension point to configure the ruleset to run at the extension
   point.
8. For specification-based extension points, configure the specification for the ruleset
   extension point.
9. Build the project to create the cartridge.
10. Deploy the cartridge into UIM.
11. If you created any custom extension points, make sure the weaver is turned on,
    and restart the UIM application server.
12. Run the ruleset.

About Base Rulesets
UIM provides numerous base rulesets that are called by UIM code, and that provide
examples for you to view when creating custom rulesets. These rulesets are contained
in the UIM_Home/cartridges/base/ora_uim_baserulesets cartridge.

Viewing Base Rulesets
Base rulesets can be viewed in Design Studio or in UIM.
Viewing Base Rulesets in Design Studio
To view base rulesets in Design Studio, import the ora_uim_baserulesets cartridge into Design Studio. After you import the base cartridge, you can view the rulesets in the Studio Design perspective Cartridge Explorer view.

For instructions on how to import a cartridge into Design Studio, see the Design Studio Help.

Viewing Base Rulesets in UIM
To view base rulesets in UIM, deploy the ora_uim_baserulesets cartridge into UIM. After you deploy the base cartridge, you can view the rulesets in UIM by clicking the Rulesets link in the Tasks panel of the UIM Home page.

For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge into UIM using the Cartridge Deployer Tool, see UIM Cartridge and Technology Pack Guide.

List of UIM Rulesets
The following is a list of the base rulesets provided in the ora_uim_baserulesets cartridge. Each base ruleset includes a description of the ruleset, the execution point that initiates the ruleset, and the input and output parameters defined for the ruleset. For the base rulesets in boldface, additional information is provided in the following sections.

- Address Range Validation
- Clean Up Extension Points
- Create Address Characteristic Map
- Find Address Range
- Import Enabled Extension Points
- Import Extension Point Ruleset
- Import Extension Points
- Import Inventory
- Import Security Extensions
- Place Formatted Identifier
- Recall Disconnected TN
- Reservation Check Redeemer
- Reservation Expiration
- System Export
- System Import
- Telephone Number Disconnect
- Telephone Number Formatting
- Telephone Number Grading
- TN Selection
- Trail Pipe Topology Edge
- Validate Address For Range
Import Inventory

The Import Inventory base ruleset does the following, using an input text file that provides a telephone number ID, a logical device account ID, and an equipment ID:

- Creates an instance of a telephone number if it does not exist
- Creates an instance of a logical device account if it does not exist
- Creates an instance of equipment if it does not exist
- Validates and creates a custom involvement between the telephone number and logical device account
- Validates and creates a custom involvement between the logical device account and equipment

The formatted content of the input text file is:

```
TN,LDA,Equipment
<TelephoneNumberId>,<LogicalDeviceAccountId>,<EquipmentId>
<TelephoneNumberId2>,<LogicalDeviceAccountId2>,<EquipmentId2>
<TelephoneNumberId3>,<LogicalDeviceAccountId3>,<EquipmentId3>
```

The content of the input text file is comma delimited. The first line lists the attribute names that are hard-coded in the ruleset. These attribute names must be spelled and capitalized as shown to be recognized by the ruleset. The remaining lines list the corresponding values of the attributes. Only one row of values is required. Listing multiple rows of values, as shown, is optional.

To use this ruleset, perform the following steps within Design Studio. For instructions on how to perform each step, see the Design Studio Help and the UIM Help.

1. Create the following specifications:
   - Telephone Number Specification
   - Logical Device Account Specification
   - Equipment Specification

2. Deploy the cartridge containing the specifications created in Step 1.

3. Import the ora_uim_baserulesets cartridge.

4. Modify the Import Inventory ruleset to reflect the specification names that you created in Step 1.

5. Deploy the extended ora_uim_baserulesets cartridge that now contains the modified Import Inventory ruleset.

6. Create the input text file and save it.

   The file name, and the location in which it is saved, is arbitrary. When you run the ruleset in Step 7, you will need to browse to this location and select the file, so make note of both.

7. Within UIM, run the Import Inventory ruleset, using the input text file that you created in Step 6.

The result of these actions is a custom involvement created between the supplied telephone number and logical device account, and a custom involvement created between the logical device account and equipment.
Place Formatted Identifier

The Place Formatted Identifier ruleset defines the display format of the Formatted Identifier field for places that are associated with a specific Place specification. The field is comprised of the characteristics defined for the Place specification with which the place is associated. This ruleset runs in UIM whenever the Formatted Identifier field displays for a place. Figure 8–10 shows the Place Summary page for an address where the ruleset concatenates the State, City, Zip Code, Country, Address Line 1, and Address Line 2 characteristics to set the Formatted Identifier field.

Figure 8–10  Formatted Identifier Field

You can edit this rule in Design Studio. To modify the ruleset, open it within Design Studio and modify the boldface code in Example 8–7. The default code randomly appends the characteristics defined for the specification. By modifying the code, you can specify the characteristics to include in the display, and you can specify the order in which the characteristics appear in the display.

Example 8–7  Place Formatted Identifier Rule

```
rule "Locations"
  salience 0
when
  place : GeographicPlace()
  eval(place instanceof GeographicLocation)
then
  // execute for location
  StringBuilder formattedIdentifier = new StringBuilder();
  List pc = new ArrayList(place.getCharacteristics());
  if (pc != null) {
    for(int i=0;i<pc.size();i++) {
      String value = ((PlaceCharacteristic)pc.get(i)).getValue();
      if (value != null) formattedIdentifier.append(value).append(" ");
    }
  }
  place.setFormattedIdentifier(formattedIdentifier.toString().trim());
end
rule "Addresses"
  salience 0
when
  place : GeographicPlace()
  eval(place instanceof GeographicAddress)
then
  // execute for address
  StringBuilder formattedIdentifier = new StringBuilder();
```

List pc = new ArrayList(place.getCharacteristics());
if (pc != null) {
    for(int i=0;i<pc.size();i++) {
        String value = ((PlaceCharacteristic)pc.get(i)).getValue();
        if (value != null) formattedIdentifier.append(value).append(" ");
    }
    place.setFormattedIdentifier(formattedIdentifier.toString().trim());
}

Reservation Check Redeemer
You can use the Reservation Check Redeemer ruleset to enable reservation redemption validations. By default, reservation redemption validations are disabled in UIM.

To enable reservation redemption validations, create a ruleset extension point global to configure the Reservation Check Redeemer base ruleset to run at the ReservationManager.checkRedeemer global base extension point. By default, the ReservationManager.checkRedeemer method returns false, and the Reservation Check Redeemer base ruleset returns true. By configuring the base ruleset to run instead of the method, reservation redemption validations are enabled.

If you have enabled reservation validations, and want to disable them, modify the ruleset to return false.

System Export and System Import
The System Export and System Import rulesets are used to manage UIM data. For example, you may wish to create a new UIM test environment and load the test environment with data from another UIM environment, or you may wish to export UIM data to send the data upstream in the order fulfillment process.

Caution:  The System Export ruleset exports data from the UIM database. In this release, Metadata Services (MDS) was introduced to store additional data that is used by the UIM UI in the presentation of specifications and characteristics. The export ruleset does not export this additional data from the MDS.

If you are using the System Export ruleset without System Import, for example to send data upstream, this limitation is not an issue.

If you are using System Export to export data from environment A, along with System Import to import data into environment B, you can do the following to work around the issue:

1. Run the System Export ruleset in environment A.
2. Run the System Import ruleset in environment B.
3. Deploy the cartridge or cartridges that define the specifications and characteristics that were previously deployed into environment A (resulting in the additional specification and characteristic data being stored in the MDS) into environment B.

The System Export ruleset exports database entities into an XML formatted output file. An export configuration is the input file, and the export ruleset name is the data file. The output is an XML file containing all exported entities that match the specified criteria in the export configuration.

The System Import ruleset imports previously exported XML-formatted data into the system. Before you import data, you should check the data in the XML file for entity
IDs that clash with those already in use. You can modify the IDs in the file if necessary. The way UIM handles duplicate IDs during import depends on the value of the **DuplicateAction** parameter in the configuration file.

Input configuration data for these rulesets includes various processing instructions. In the case of System Export, supply this information to the ruleset through the file input. The same configuration is included in the export bundle that is created as part of the export operation. **Example 8–8** shows the file content for an export operation.

**Example 8–8  Export Operation**

```
Select T from TelephoneNumberSpecification T where T.name = 'Sample'
Select E from EquipmentSpecification E where E.name = 'Sample'
duplicateAction=Update
relationshipsToInclude=Meta
```

The first two lines in this example are the JPA Queries for the export operation. Multiple queries can be specified within a configuration file. The instructions section must appear directly after any queries and can contain the following options. If any instructions are not specified, the default is assumed.

- **DuplicateAction**: action to take when duplicate data is encountered in the target data store. Options are:
  - **Update**: (default) any duplicate record is updated with the newly imported data values.
  - **Ignore**: do not process, skip the duplicate record.
  - **Error**: when a duplicate record exists in the target data store, report it as such and end the transaction without taking any action.

- **RelationshipsToInclude**: describes how to process objects that are related to the selected entity. Options are:
  - **Meta**: (default) only include relationships that are metadata type relationships, such as specification relationships.
  - **Data**: only include relationships that are business data in nature, such as characteristics.
  - **All**: include all relationships including metadata and normal business data associations.
  - **None**: do not include any relationships and only process the integral type attributes of the selected entity.

**Telephone Number Formatting**

The Telephone Number Formatting ruleset defines the telephone number length and display format of telephone numbers. The ruleset defines a default editMask that is applied to all telephone numbers, unless otherwise specified. The ruleset can also define specific editMasks to apply to specific Telephone Number specifications. This ruleset runs in UIM during telephone number readiness, and applies the default edit mask of 10 digits (############).

To modify the base ruleset in Design Studio, scroll to the FUNCTIONS section of the code. You can modify the default editMask, or the specification name and corresponding editMask, or both. You can also replicate the code and define multiple editMasks that are specific to a particular specification. In **Example 8–9**, the boldface code is the code you need to modify.
Example 8–9  Telephone Number Formatting Rule

```java
// FUNCTIONS

function String getEditMask(TelephoneNumberSpecification tnSpec)
{
    // The character # is reserved and represents a required digit.
    // The default mask is eight required digits.
    String editMask = "##########";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if ( tnSpec.getName().equals("TNspec NPA-NXX") )
        editMask = "###-###-####";

    return editMask;
}

// RULES

rule "Get TN Edit Mask"
    salience 0
    when
        telephoneNumberSpecification : TelephoneNumberSpecification()
        context : ExtensionPointRuleContext()
    then
        String editMask = getEditMask(telephoneNumberSpecification);
        context.setReturnValue(editMask);
end
```

Example 8–10 shows the modified portion of the Telephone Formatting rule in boldface. In this example, the modified rule redefines the default EditMask length format from 10 digits to 11 digits. The example also redefines the telephone number display format for telephone numbers created from the NANPA telephone number specification to display as +# (xxx) xxx-xxxx. For example, +1 (972) 555-8495.

Example 8–10  Telephone Number Formatting Rule

```java
function String getEditMask(TelephoneNumberSpecification tnSpec)
{
    // The character # is reserved and represents a required digit.
    // The default mask is eleven required digits.
    String editMask = "############";
    if ( tnSpec == null )
        return editMask;

    // Define the edit mask based on the spec name
    if ( tnSpec.getName().equals("NANPA") )
        editMask = " +# (###) ###-####";

    return editMask;
}
```

Create a ruleset extension point to configure the ruleset to run at the SpecManager.getEditMask base extension point, and deploy the cartridge.

From this point forward, all telephone numbers created from the NANPA telephone number specification and the TELEPHONE_NUMBER_FORMATTING ruleset are based on the new default length and display format. Any telephone numbers created from the NANPA telephone number specification prior to this ruleset being deployed...
do not reflect this new default length and display format, nor will they; the telephone number formatting is not applied retroactively.

You can also modify the telephone number default edit mask in the `number.properties` file. See *UIM System Administrator’s Guide* for more information.

## Troubleshooting

This section provides information on troubleshooting potential problems you may encounter when working with custom rulesets and extension points.

### Troubleshooting Custom Rulesets

When troubleshooting custom rulesets, check the following:

- Does the Ruleset compile?
  - Use the JBoss Eclipse plug-in Rule editor.
  - Check the import statements.
  - Check the project library list.
- Does the cartridge build and deploy successfully?
  - Check the UIM application server log.
- Does the ruleset RHS condition ever evaluate to true?
  - Debug to find out.
- Are the ruleset argument values and return values correct?
  - Debug to find out.

For information on turning on debugging, see *UIM System Administrator’s Guide*.

### Troubleshooting Custom Extension Points

When troubleshooting custom extension points, check the following:

- Is the extension point defined in both Design Studio, and in a custom `aop.xml` file?
- Is the signature defined correctly in both places?
  - Check spacing.
  - Check spelling of package and class names.
- Regarding the weaver section in the custom `aop.xml` file:
  - Did you include it?
  - Are the package names correct?
- Did the cartridge build and deploy successfully?
  - Check the UIM application server log.
- Is the weaver turned on?
  - Check the `UIM_Home/Domain_Home/bin/setUIMEnv.cmd` file.
- After deploying a cartridge containing custom extension points, did you restart the UIM application server?
– Check the UIM application server log to see if the custom extension point was successfully weaved into the UIM code stream.

Troubleshooting Configuring a Ruleset to run at an Extension Point

When using base rulesets and base extension points, remember that base ruleset extension points are not provided. It is up to you to configure rulesets to run at extension points. This is true for base rulesets and base extension points, custom rulesets and custom extension points, or a combination of base and custom.

- Is the ruleset configured to run? That is, did you create a ruleset extension point?
- Did you select the correct ruleset?
- Did you select the correct extension point?
- Did you select the correct placement of the rule to run before, after, or instead of the method?

It is important to be mindful of the rule placement. For example, if you are expecting your ruleset custom code to perform a process based on something the extension point method does, and you configure the ruleset to run before or instead of the method, you will not get the results you are expecting.

Troubleshooting Using Timing Events

If you set up rulesets based on timing events, be sure the UIM Home/config/timers.properties file has the timing event you are using turned on. For example, if you configure a ruleset to run based on the timing of telephone number aging, and the timing event for this is not turned on, your ruleset will never run. For more information on the timers.properties file, see UIM System Administrator’s Guide.
This chapter provides information on integrating Oracle Communications Unified Inventory Management (UIM) with external systems through Web services. UIM provides the following Web services:

- The UIM Service Fulfillment Web service defines operations that enable you to create and modify business interactions, through which you can create and modify services, service configurations, and service configuration items. Information on this Web service is provided in this chapter.

- The UIM Reference Web service defines operations that enable you to create and modify services, service configurations, and service configuration items, but does not involve the use of business interactions. This Web service is deprecated: Oracle recommends that you use the UIM Service Fulfillment Web service. If you have used the UIM Reference Web service in a previous release, see Appendix B, "Reference Web Service". This Web service also serves an example to follow when developing custom Web services.

- The UIM Cartridge Management Web service defines various operations that support deploying and undeploying cartridges. The Cartridge Deployer Tool and Oracle Communications Design Studio use this Web service to manage cartridges. While this Web service is published, its target users are internal developers. As a result, information on this Web service is not provided in this document. It is mentioned here only because the UIM Cartridge Management Web service is deployed as an installation step and displays on the Oracle WebLogic Server Administration Console.

- UIM also provides a way for you to develop, build, and deploy custom Web services. See Chapter 10, "Developing Custom Web Services" for more information.

This chapter contains the following sections:

- Web Services Overview
- UIM Web Service Module Overview
- UIM Web Service Standards and Specifications
- About the UIM Service Fulfillment Web Service
- Understanding the UIM Service Fulfillment Web Service
- Using the UIM Service Fulfillment Web Service

**Web Services Overview**

Web services support interoperable machine-to-machine interaction over a network. Web services are APIs that can be accessed over a network, and run on a remote
system hosting the requested services. Web service operations are described by the Web Service Definition Language (WSDL).

Figure 9–1 shows the different paths traveled by a call originating from the UIM UI client, and a call originating from outside UIM that is then processed by the UIM Web service client.

**Figure 9–1 Web Services**

The path of the Web service includes:

- **Web service client**
  This represents the Web service user (client, Web service client, or customer). Web service operations are called by sending SOAP messages over http or https, or by posting SOAP messages on a UIM-defined JMS message queue. See "About Message Queues" for more information.

- **Web service module**
  This represents all the sub-modules required for implementing a Web service, including the Web service, the Web service framework, WSDL interfaces, and WSDL implementations. The Web service module is deployed as a WAR file. Details of this module are shown in Figure 9–2, "Web Service Module".

- **UIM business logic**
  This represents all the sub-modules required for attaining business functionality. This includes the Java API, the Java API framework, business logic, and persistence framework. Details of the UIM business logic are not within the scope of this document.
UIM Web Service Module Overview

Figure 9–2 shows the Web service module in more detail.

Figure 9–2  Web Service Module

The Web service module includes:
- Platform-provided Web Services Framework
  This represents the Web service framework provided by Java EE platforms, such as Oracle WebLogic Server.
- Web service operations
  This represents the Java Web service implementation class. This is the entry point to a UIM Web service. The Web service operations are Java representations of the WSDL.
- Adapters
  The Web service operations layer calls the adapters, which direct the calls and collect data from the appropriate UIM API managers. Transaction handling is performed in the adapters.
- Mappers
  Mapper classes convert data representations by providing operations that convert data from Entity representation to ValueType representation, and from ValueType representation to Entity representation. Mapper classes are typically called by the adapter code.
- Helpers
  Helper classes assist the working of the adapters.

The Web service operations, adapter, mapper, and helper classes are further explored in Chapter 10, "Developing Custom Web Services".

About Message Queues

The UIM installation provides the following message queues to use when calling Web services:
- inventoryWSQueue
- inventoryWSQueueAlternate

Two message queues are provided for efficient processing of Web service calls. For example, you may have all Web service operation calls except ProcessInteraction use
inventoryWSQueue, and have ProcessInteraction use inventoryWSQueueAlternate because the ProcessInteraction operation takes longer to run than the other operations.

### About Transaction Handling

The adapter layer performs transaction handling. Transactions are started only if the thread is not already within a transaction.

### About Exception Stacktraces

Exception stacktraces are available in the WebLogic server logs. Exception stacktraces are not available in the UIM Web service responses.

### UIM Web Service Standards and Specifications

Table 9–1 lists the UIM Web service standards and specifications.

<table>
<thead>
<tr>
<th>Standard and Specification</th>
<th>Version Release</th>
<th>Description</th>
<th>Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAX-RPC</td>
<td>1.1</td>
<td>XML &lt;-&gt; Java binding specification</td>
<td>Compliant.</td>
</tr>
<tr>
<td>JSR-181</td>
<td></td>
<td>Java Web service annotations</td>
<td>Deprecated. Uses basic annotations for interoperability.</td>
</tr>
<tr>
<td>SOAP</td>
<td>1.1</td>
<td>Simple Object Access Protocol (Also referred to as Service Orientated Architecture Protocol.)</td>
<td>Compliant. Uses XML/SOAP/HTTP and XML/SOAP/JMS.</td>
</tr>
<tr>
<td>Transport Protocols</td>
<td>HTTP 1.0, HTTPS 1.0 (HTTP 1.1), JMS 1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSDL</td>
<td>1.1</td>
<td>Web Service Definition Language</td>
<td>Compliant.</td>
</tr>
<tr>
<td>XML</td>
<td>1.1</td>
<td></td>
<td>Compliant. Uses XML/SOAP/HTTP and XML/SOAP/JMS.</td>
</tr>
</tbody>
</table>
About the UIM Service Fulfillment Web Service

Note: You should have an understanding of the following subjects described in *UIM Concepts*:

- Planning (business interactions and business interaction items)
- Services (services, service configurations, and service configuration items)
- Connectivity (pipes, pipe configurations, and pipe configuration items)
- Life cycle management
- Service fulfillment

The UIM Service Fulfillment Web service is packaged in the *inventory.ear* file, within the *UIMServiceFulfillment.war* file. When the installer deploys the *inventory.ear* file, the UIM Service Fulfillment Web service automatically deploys and is ready to use.

At a high level, the UIM Service Fulfillment Web service enables an external system to create new business interactions and change existing business interactions in UIM. Through business interactions, an external system can manage services and connectivity, including the relationships between services and the relationships between pipes, as well as the resources associated with them.

At a more granular level, the UIM Web Service Fulfillment Web service supports operations that enable you to:

- Create business interactions and change existing business interactions
- Process business interactions to create or change services or connectivity in current inventory, and, using a custom ruleset that calls custom code, create or change configuration items and allocate resources for services or connectivity in current inventory
- Transition business interactions and associated business entities through their respective life cycles
- Retrieve business interactions
- Retrieve configurations
- Update configurations

Only business interactions that support services and connectivity can be added through the Web service. Specifically, these include services, service configurations, pipes, and pipe configurations. However, after the business interaction is created in UIM, you can use the UI to add business interaction items of any type.

Note: The remainder of this chapter references only services, service configurations, and service configuration items (and not pipes, pipes configurations, and pipe configuration items). Any information regarding services, service configurations, and service configuration items applies to pipes, pipe configurations, and pipe configuration items.

The UIM Service Fulfillment Web service defines the following operations:
About the UIM Service Fulfillment Web Service

- CaptureInteraction
- ProcessInteraction
- GetInteraction
- UpdateInteraction
- GetConfiguration
- UpdateConfiguration

Information on the WSDL file and schema files that support these operations is presented in "Using the UIM Service Fulfillment Web Service". The information includes the location of the files, and what the files define. If you are not already familiar with WSDL and schema files, it may be beneficial to read this section first, and to look at the files before reading about the details of each operation. The following sections on each operation assumes that you are already familiar with WSDL and schema file content.

**CaptureInteraction**

The CaptureInteraction operation enables external systems to send an order request to UIM to add, change, or disconnect a service. The CaptureInteraction request defines one order per request. The order can define multiple line items, and multiple child orders. Each child order is defined by the same structure as the order on the request. That is, each child order can define multiple line items and multiple child orders, and so forth.

The order information from the request is stored in UIM as a business interaction. Business interactions are used for planning inventory resources, prior to making the inventory resources available in current inventory.

The CaptureInteraction request must specify a business interaction action of CREATE or CHANGE. The business interaction action is defined as an enumeration in the BusinessInteraction.xsd schema file. The enumeration defines several actions, but CREATE and CHANGE are the only valid actions for CaptureInteraction.

When the CaptureInteraction request specifies the CREATE action, it creates a business interaction to contain the order information sent in the request, and creates an attachment that contains the entire <interaction> element of the XML from the CaptureInteraction request. CaptureInteraction also associates the attachment to the business interaction.

When the CaptureInteraction request specifies the CREATE action, you can specify an external ID for the business interaction. The external ID should be unique within UIM, but UIM does not enforce uniqueness; it is up to the calling system to enforce uniqueness. When an external ID is specified, UIM captures it and stores it with all of the other request data. A subsequent request can then specify a CHANGE action and supply the external ID to identify the business interaction to be changed.

When the CaptureInteraction request specifies the CHANGE action, the request must provide either the external ID or the business interaction ID to indicate the business interaction to change. If the request provides an external ID, CaptureInteraction assumes the external ID was supplied when the business interaction was created. CaptureInteraction then retrieves the business interaction and updates it with the order information sent in the request. CaptureInteraction also retrieves the business interaction attachment and updates the XML.

You can view the XML that is contained in the attachment from within the UIM UI. Also, the ProcessInteraction operation retrieves the business interaction attachment.
and uses the XML to process the business interaction into current inventory. This is further explained in “ProcessInteraction”.

**Validating Input Data**

You can validate the request input data using a custom ruleset. Data validations specific to your implementation can reside in the ruleset, or reside in custom code that the ruleset calls. Either way, you can configure the ruleset to run using the provided `BusinessInteractionManager_createBusinessInteractionAttachment` extension point, located in the `ora_uim_baseextpts` cartridge. The extension point defines the `createBIAttachment()` method, so configure your ruleset to run before the method. By using this extension point, your custom validations run before the attachment is created. If the validation fails, the session rolls back and the business interaction that was created is not committed. See Chapter 8, “Extending UIM through Rulesets” for more information.

**Associating Business Interactions**

In UIM, business interactions can be associated with one another. The `CaptureInteraction` request defines an element that enables you to associate one or more child business interactions to the business interaction you are creating or changing. Furthermore, you can associate one or more child business interactions to each child business interaction, which would be the grandchild business interactions to the business interaction you are creating or changing, and so forth.

**About the CaptureInteraction Flow**

Figure 9–3 shows what occurs when the `CaptureInteraction` operation is called. A business interaction is represented as BI in the figure.

---

![Figure 9–3 CaptureInteraction](image-url)
In Figure 9–3, the **Validate XML Input** box represents the custom ruleset that you can configure to run before the creation of the attachment.

The CaptureInteraction operation also wraps the BusinessInteraction.captureInteraction() API method. The API method defines two arguments: the parent business interaction, and the XML. When the CaptureInteraction operation calls the API method, the parent business interaction argument is always null. Depending on the XML business interaction action of CREATE or CHANGE, the API method creates or changes the business interaction, and creates or changes the attachment. When the XML business interaction action is CREATE, the attachment is associated to the business interaction; when the XML business interaction action is CHANGE, the association already exists.

If the business interaction defines a child business interaction, the API method is called from within itself. In this scenario, the parent business interaction argument is no longer null. As a result, after the business interaction is created, and the attachment created and associated, the business interaction is associated to the parent business interaction that was specified by the argument. For example, a request defines one new business interaction that has one child business interaction. The CaptureInteraction operation calls the API method with a parent business interaction argument of null. Business interaction A is created. The attachment is created and associated to business interaction A. Because the parent business interaction argument is null, the **Associate BI to Parent BI** box does nothing. Next, the first (in this example, the only) child business interaction is processed and calls the API method with a parent business interaction argument (business interaction A). Business interaction B is created. The attachment is created and associated to business interaction B. Because the parent business interaction argument is not null, business interaction B is associated to the parent business interaction argument that was supplied (business interaction A).

**Understanding the Request Content**

Figure 9–4 shows the high-level content of the CaptureInteractionRequest. Each request defines a single interaction, which specifies the data used to create the business interaction. The interaction defines a header and a body. The body defines a sequence of items: each item defines a service, and each service defines a service configuration. (The service and service configuration can also be a pipe and pipe configuration. This is further explored in "About the Service or Connectivity"). The body also defines a sequence of interactions, which specifies the data used to create any child business interactions.
Example 9–1 is a condensed version of the CaptureInteraction request that highlights the main content to better understand the CaptureInteraction operation. The example is numbered so that information describing the example can be referenced.

Example 9–1 omits the following:

- Namespaces, and assumes that they are properly defined
- Elements such as notes, start and end dates, effective dates, and descriptions
- Structures that detail an external ID, specification, configuration, and configurationItem
- Structures and elements within party and place, which are designated with "..."

**Note:** CaptureInteraction, ProcessInteraction, GetInteraction, and UpdateInteraction all use the same structure for the request and for the response. The only difference is the actual request/response name (line 01 and line 71). When reading about the remaining operations in the following sections, refer back to this example for a better understanding.

**Example 9–1 Condensed CaptureInteraction Request**

01 <captureInteractionRequest>
<invbi:interaction>
  <invbi:header>
    <invbi:specification/>
    <invbi:action/>
    <invbi:id/>
    <invbi:name/>
    <invbi:externalIdentity/>
    <invbi:state/>
  </invbi:header>
  <invbi:body>
    <invbi:item>
      <invbi:externalIdentity>
        <invbi:action/>
        <invbi:service>
          <invsvc:specification/>
          <invsvc:id/>
          <invsvc:action/>
          <invsvc:name/>
          <invsvc:externalIdentity/>
          <invsvc:state/>
          <invsvc:place>
            ...<invplace:service>
            ...<invsvc:party>
            ...<invparty:service>
            ...<invsvc:configuration/>
          </invsvc:party>
          <invsvc:configuration/>
        </invsvc:place>
        <invsvc:party>
          ...<invparty:service>
          ...<invsvc:place>
          ...<invplace:service>
          ...<invsvc:configuration/>
        </invsvc:place>
        <invsvc:configuration/>
      </invbi:service>
      <invbi:parameter>
        <invbi:name/>
        <invbi:value/>
      </invbi:parameter>
    </invbi:item>
  </invbi:body>
</invbi:interaction>
Throughout Example 9-1, the <specification> element that is shown is actually a structure that defines the following elements:

**Example 9-2 Specification Structure**

```xml
<invbi:specification>
  <invent:entityNote/>
  <invspec:name/>
  <invspec:entityClass/>
  <invspec:description/>
  <invspec:startDate/>
  <invspec:endDate/>
</invbi:specification>
```

Within the specification structure, the <name> element is the name of a specification. This <name> element is not be confused with the <name> element that is specified for the business interaction (line 07) or for the service (line 19). For example, a request that specifies the CREATE interaction action must supply the Business Interaction specification name (within the specification structure on line 04), and the name of the business interaction being created by the request (line 07). Similarly, a request that specifies the ADD service action must supply the Service specification name (within the specification structure on line 16), and the name of the service being created by the request (line 19).

Within the specification structure, the <entityClass> element is defined as an enumeration in the **Specification.xsd** schema file. The enumeration values reflect UIM entity specification types, such as BusinessInteraction, Service, Equipment, and so forth. The Service Fulfillment Web service does not use the <entityClass> element, so the request does not need to specify it.

**About the Business Interaction**

The CaptureInteraction request captures one interaction per request (lines 02 through 68). For each interaction, the request captures one or more items (lines 12 through 56), and one or more child interactions (lines 57 through 66).

When calling CaptureInteraction, the request must specify an action (line 05) of CREATE or CHANGE. The interaction <action> element is defined as an enumeration in the **BusinessInteraction.xsd** schema file.

If the action is CREATE, the request must provide an arbitrary name for the business interaction (line 07) being created, as well as the Business Interaction specification name (within line 04) upon which the business interaction is being based. (The specification name will typically be **Service Order**, which is the Business Interaction specification provided in the ora_uim_basespecifications cartridge.) The request can optionally provide an external ID for the business interaction. You do not need to provide the specification entityClass enumeration value of **BusinessInteraction**; this is
assumed based on the placement of the specification structure within the <interaction> element.

If the action is CHANGE, the request must provide the external ID (within line 08) or the business interaction ID (line 06) to indicate the business interaction to change, as well as the actual changes.

**About the Business Interaction Item**

The CaptureInteraction request captures one or more items per interaction. Example 9–1 shows just one item (lines 12 through 56). To include multiple items, replicate the item and place it between lines 56 and 57.

Each item defines an action (line 14), which must be **ADD** regardless of the request.

---

**Note:** The action must be ADD. It cannot be another action, and it cannot be left blank. If the action is not ADD, the operation errors.

---

**About the Service or Connectivity**

Each item defines a service or connectivity. Example 9–1 shows a service (lines 15 through 51). However, the <service> element (line 15) is actually defined as a choice in the BusinessInteraction.xsd schema file, with the choices being service, connectivity, and entity. When the choice is service, a service and service configuration are captured; when the choice is connectivity, a pipe and pipe configuration are captured. The choice of entity is not used for the request; it is used only for the response. (The request and response use the same structure.) When the request choice is either service or connectivity, you must supply a valid specification name (within line 16) from which to create an instance of the specification in UIM. For example, if the choice is service, supply a valid Service specification; if the choice is connectivity, supply a valid Pipe specification.

---

**Note:** The preceding paragraph required the mention of pipes and pipe configurations due to the element that defines a choice. The following paragraphs return to citing only services and service configurations, but pipes and pipe configurations continue to apply.

---

Each service also defines an action (line 18). The service action is not an enumeration, as are the interaction action and item action. Rather, the service action is a custom action that is recognized by, and acted upon by, custom code. The service action is further explored in "ProcessInteraction".

The service may also specify a place (lines 22 through 35) or a party (lines 36 through 49) to associate to the service. The details of the service or connection are captured within the <parameter> element (lines 52 through 55). These parameters are used to drive custom code and are further explored in "ProcessInteraction". Party and place information for the service can also be defined within parameters, instead of within the place and party structures (lines 22 through 35 and lines 36 through 49). The end result is the same: a party or a place is associated to the service. The difference is in the processing: If the party or place is defined for the service within the party and place structures, the Web service processes them. If the party and place are defined for the service within the <parameter> element, custom code is responsible for processing them.

The request and response use the same structure. Most of the elements are used only by the response, so there are numerous elements that are not used by the request. For
example, a service and configuration for the place (lines 24 through 34), a service and configuration for the party (lines 38 through 48), and the configuration for the service itself (line 50).

About the Associated Business Interaction
The CaptureInteraction request captures one or more child interactions per interaction. Example 9–1 shows just one child interaction (lines 57 through 66). To include multiple interactions, replicate the child interaction (lines 57 through 66) and place it between lines 66 through 67.

About the ExecuteProcess Element
The <executeProcess> element (line 69) is defined after the interaction and applies to the interaction. This element is defined as a boolean and is used only by the CaptureInteraction operation. When the value of <executeProcess> is true, CaptureInteraction executes and, upon completion, ProcessInteraction executes. This eliminates the need to place two separate Web service calls; one for CaptureInteraction and one for ProcessInteraction. When the value of <executeProcess> is false, just CaptureInteraction executes. The default value is false.

About the ResponseLevel Element
For the CaptureInteraction, ProcessInteraction, UpdateInteraction, and GetInteraction requests, the <responseLevel> element (line 70) is defined as an enumeration in the InteractionMessages.xsd schema file. Depending on the enumeration value specified in the request, the level of information returned by the response can vary:

- INTERACTION
  Returns just the interaction information.

- INTERACTION_ITEM
  Returns the interaction and item information.

- INTERACTION_ITEM_ENTITY
  Returns the interaction, item, and entity information.

- INTERACTION_ITEM_ENTITY_CONFIGURATION (default option)
  Returns the interaction, item, entity, and configuration information.

- INTERACTION_ITEM_ENTITY_CONFIGURATION_EXPANDED
  Returns the interaction, item, entity, configuration, and any child configurations.

For the GetConfiguration and UpdateConfiguration requests, the <responseLevel> element is defined as an enumeration in the ConfigurationMessages.xsd schema file, which defines a different set of enumeration values for the <responseLevel> element (because the <interaction> and <item> elements are not applicable to the GetConfiguration and UpdateConfiguration request structures). Depending on the enumeration value specified in the request, the level of information returned by the response can vary:

- ENTITY_CONFIGURATION (default option)
  Returns the entity and configuration information.

- ENTITY_CONFIGURATION_EXPANDED
  Returns the entity, configuration, and any child configurations.
About the CaptureInteraction Response and Faults

The CaptureInteraction response returns a varying level of information based on the <responseLevel> value the request specifies. See "About the ResponseLevel Element" for more information.

The CaptureInteraction response always includes the business interaction ID and the current business interaction state. If a new business interaction is created, the business interaction ID generated by UIM is returned. If an existing business interaction is changed, the business interaction ID sent with the request is returned. The valid business interaction states are CREATED, IN_PROGRESS, COMPLETED, or CANCELLED, as defined by an enumeration in the BusinessInteraction.xsd schema file.

The CaptureInteraction response returns an error message for the following scenarios:

- The request specifies a business interaction action of CREATE with a business interaction ID that already exists.
- The request specifies a business interaction action of CHANGE with a business interaction ID that does not exist. (The specified business interaction can be specified directly, or specified indirectly through the external ID.)
- An optional extension point is used to validate the input, and the associated ruleset logs an error. For example, the XML input payload does not validate.
- The purpose of the request is to add a child business interaction to an existing business interaction, and the request specifies the CREATE business interaction action for both the parent and child business interactions. In this scenario, the child action needs to be CREATE and the parent action needs to be CHANGE because you are creating the child, but modifying the existing business interaction to associate the child. (A request can specify the CREATE business interaction action for both the parent and child business interactions, as long as neither specified business interaction ID exists.)

ProcessInteraction

The ProcessInteraction operation performs the work that is necessary to move a planned service into current inventory. The planned service is represented by the interaction in the XML, which is stored in UIM as a business interaction attachment, having been placed there by CaptureInteraction.

ProcessInteraction retrieves the business interaction and attachment and, based on the items defined for the interaction in the XML, processes each item. Each item creates or updates a service, including any default service configuration items defined by the specified Service specification. ProcessInteraction also calls the BaseConfigurationManager.automateConfiguration() method per service configuration. This is the same method that the UIM UI calls to automatically configure a service.

Custom code that creates or updates service configuration items is called through a custom ruleset that you configure to run using a provided extension point that defines the same API method that ProcessInteraction wraps (the automateConfiguration() method). This topic is further explored in "Customizing ProcessInteraction".

When calling ProcessInteraction, the request must specify the external ID or the business interaction ID to indicate the business interaction to process.

The request can specify whether to process the entire business interaction, or just specific business interaction items. If the request specifies the external ID or business interaction ID only, the entire business interaction is processed; if the request specifies
the external ID or business interaction ID and specific business interaction items, only the specified business interaction items are processed.

**About the ProcessInteraction Flow**

Figure 9–5 shows what occurs when the ProcessInteraction operation is called. A business interaction is represented as BI in the figure.

**Figure 9–5 ProcessInteraction**

Figure 9–5 shows the ProcessInteraction flow for a request that specifies the CREATE business interaction action. For a request that specifies the CHANGE business interaction action, the Create Service, Create Service Configuration, and Associate Service Configuration to BI boxes would reflect Find or Change Service, Find Service Configuration, and Find Service Configuration Associations. This is because ProcessInteraction handles changes to a service, but not to the service configuration or service configuration items, which is handled by custom code. For a CHANGE action, the change may be applicable to the interaction, or to the items, or both. If the change is applicable to the interaction, the service and service configuration need to change. If the change is applicable to the items, the service and service configuration need to be found to get to the service configuration items that need to change.

In Figure 9–5, the light grey boxes represent the work performed by ProcessInteraction, prior to calling the custom ruleset. ProcessInteraction handles the processing of the business interaction. The dark grey boxes represent the work that is performed by the custom code. The custom code handles the processing of the business interaction items.

The processing of the business interaction items involves customizations that are necessary to meet the business requirements of providing the specific type of service. Customizations may involve custom specifications, characteristics, rulesets, ruleset extension points, and Java code, all of which are written in Design Studio within a
Inventory project. The Inventory project is then built, resulting in a cartridge that you can deploy into UIM.

Regarding the **Associate Service Configuration to BI** box: A service configuration is indirectly associated to a business interaction through the business interaction items. This association is shown by the dotted line in Figure 9–6. To associate the service configuration to the business interaction, ProcessInteraction:

- Creates business interaction items based on the items for the interaction in the request
- Associates the business interaction items to the service configuration

**Figure 9–6  Association of Service Configuration to BI**

---

**Customizing ProcessInteraction**

You customize the process of business interaction items using a custom ruleset that calls custom code. You can configure the custom ruleset to run using the `BaseConfigurationManager_automateConfiguration.rstp` extension point, located in the provided ora_uim_baseextpts cartridge. The extension point defines the same API method that ProcessInteraction wraps. Configure your ruleset to run instead of this method. See Chapter 8, "Extending UIM through Rulesets" for more information.

Customizations are based on the request’s service action (Example 9–1, line 18) and parameters (Example 9–1, lines 52 through 55). You must establish a finite list of service actions and parameters that can be specified in the request, which can then be recognized by, and processed by, the custom code, which in turn must call API methods to realize the service in UIM. To support the customizations, you also need to create custom specifications, characteristics, rulesets, and ruleset extension points as needed.

Customizations must also include a mapping of any custom service actions to UIM-defined entity actions. The following entity actions are recognized by the Service Fulfillment Web service code, and are case-sensitive:

- create
- change
- delete
- disconnect
■ suspend
■ suspendWithConfiguration
■ resume
■ resumeWithConfiguration
■ no_action

All of these entity actions, with the exception of suspendWithConfiguration and resumeWithConfiguration, are UIM-defined entity actions. The Web service recognizes these two additional entity actions so that it can perform additional functionality. For example, the suspend action suspends a service but does not touch the service configuration. The suspendWithConfiguration action suspends a service and creates a new service configuration version. Similarly, the resume action resumes a suspended service but does not touch the service configuration. The resumeWithConfiguration action resumes a suspended a service and creates a new service configuration version. (For either action, if an existing service configuration version does not exist, an error is thrown because the service configuration must already exist if you are suspending or resuming it.)

To run your custom code that maps your custom service actions to UIM-defined entity actions, use the BusinessInteractionManager_getEntityAction.rstp extension point, located in the ora_uim_baseextpts cartridge. To do this, create a ruleset that calls your custom code, and create a ruleset extension point that uses this extension point to call your custom ruleset instead of ProcessInteraction calling the BusinessInteractionManager.getEntityAction() method.

After you have completed your customizations, build the project and deploy the resultant cartridge into UIM.

To customize ProcessInteraction:

1. Determine finite list of service actions and parameters to process.
2. In Design Studio, create an Inventory project.
3. Within the project, create a custom ruleset that is the entry point to your custom code.
4. Within the project, create a custom ruleset extension point that associates your custom ruleset with the provided extension point and configure it to run instead of BaseConfigurationManager.automateConfiguration.
5. Within the project, create a custom ruleset that maps your finite list of service actions to the entity actions that are defined in the Service Fulfillment Web service code.
6. Within the project, create a custom ruleset extension point that associates your custom ruleset with the provided extension point and configure it to run instead of BusinessInteractionManager.getEntityAction.
7. Within the project, write custom code that evaluates the mapped entity actions and custom parameters from the request that are specific to the service action mapped to the entity action, and that calls the required API methods to create the service in UIM.
8. Within the project, create any custom specifications or characteristics that are needed to support the custom code that calls the API methods.
9. Within the project, create any custom rulesets and ruleset extension points that are needed to extend any of the API methods that your custom code calls.
10. Build the project, which creates the custom cartridge.
11. Deploy the custom cartridge into UIM.
12. Ensure that the UIM Service Fulfillment Web service is deployed.
13. Send Web service operation requests, such as CaptureInteraction, ProcessInteraction, and UpdateInteraction.

Developing the Custom Code

ProcessInteraction triggers events that result in a call to custom code that automates service configurations. ProcessInteraction makes the following assumptions regarding your custom code:

- The custom code must know what to do with the XML payload based on the domain-specific business rules and models.
- The custom code needs to handle the creation or deletion of any dependent resources.
- The custom code needs to handle auto-design for new orders and auto-redesign for change orders.
- The custom code should assume that the service and service configuration are already created, and that the purpose of the custom logic is to manage the resources and characteristics.
- When modifying a subservice with parent input only:

  The business interaction attachment typically may not contain specific change request information for a subservice that was created when fulfilling the requested service. For example, a voice mail service created by UIM to fulfill the request for a Mobile GSM service with a voice mail feature. In this scenario, the voice mail service is a subservice assigned to the Mobile GSM service. When the subservice requires a change, the change request and service action are often submitted for the parent service, and not for the subservice. In such scenarios, the Web service operation has to identify that the change action is for the subservice, and process the change for the subservice. As a result, if the custom code needs to act on a subservice, it must build a request based on the subservice, call CaptureInteraction, and recursively call ProcessInteraction until it returns the no action action.

Understanding ProcessInteraction Through an Example

The following list describes some of the project content you may need to run ProcessInteraction.

- AUTOMATE_MY_CONFIGURATION.ruleset
  This is a custom ruleset that is the entry point into the custom code. The ruleset calls the AutomateMyConfiguration() method, which is defined in a custom Java class. In this example, the custom Java class is named MyConfigurationManagerImpl.java, which is also described in this list.

- AUTOMATE_MY_CONFIGURATION_EXT.rst
  This is a custom ruleset extension point that associates the AUTOMATE_MY_CONFIGURATION custom ruleset to the UIM-provided BaseConfigurationManager_automateConfiguration extension point and configures the custom ruleset to run instead of the method that the extension point defines (the BaseConfigurationManager.automateConfiguration() method).

- MAP_MY_SERVICE_ACTION.ruleset
This is a custom ruleset that evaluates the custom service action specified in the request and maps it to an entity action that is recognizable to UIM. In this example, there are five custom service actions, so this ruleset evaluates the five custom service actions and maps each one to the appropriate entity action. The entity actions are defined in the Service Fulfillment Web service code, as described in "Customizing ProcessInteraction".

Table 9–2 provides an example of mapping custom service actions to UIM entity actions.

<table>
<thead>
<tr>
<th>Custom Service Action</th>
<th>UIM Entity Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMyService</td>
<td>create</td>
</tr>
<tr>
<td>updateMyService</td>
<td>change</td>
</tr>
<tr>
<td>changeAddToMyService</td>
<td>change</td>
</tr>
<tr>
<td>disconnectMyService</td>
<td>disconnect</td>
</tr>
<tr>
<td>suspendMyService</td>
<td>suspend</td>
</tr>
</tbody>
</table>

- **MAP_MY_SERVICE_ACTION_EXT.rst**
  This is a custom ruleset extension point that associates the MAP_MY_SERVICE_ACTION custom ruleset to the UIM-provided BusinessInteractionManager::getEntityAction extension point and configures the custom ruleset to run instead of the method that the extension point defines (BusinessInteractionManager.getEntityAction).

- **MyConfigurationManagerImpl.java**
  This is custom Java code that contains a series of if else statements that evaluate the mapped entity action. For each entity action, the code calls another method within the same class. Within each of these methods, the finite set of parameters that are valid for the specific service action that was mapped to the entity action is evaluated.

  From there, the custom code calls various API methods to perform the work required to realize any service in UIM.

- **Numerous custom specifications and characteristics**
  When the custom code calls API methods, the methods may require a specification or characteristic as input to realize a service in UIM. So, your project may also have to define any needed specifications and characteristics. For example, you may define the **My Service** Service specification and the **My Service** Configuration Service Configuration specification. You may also define such characteristics as **mySpecificServiceData1** and **mySpecificServiceData2**.

- **Numerous custom rulesets and ruleset extension points**
  When the custom code calls API methods, the existing API functionality may need to be extended to realize a service in UIM. So, your project may also have to define any needed rulesets that can be configured to run before or after the API methods that the custom code calls.
About the ProcessInteraction Response and Faults
The ProcessInteraction response returns a varying level of information based on the `<responseLevel>` value the request specifies. See "About the ResponseLevel Element" for more information.

ProcessInteraction returns an error to the calling system when:
- It cannot find the business interaction specified by the calling system.
- The calling system specifies an input item entity other than Service.
- Any errors thrown by the custom code that ProcessInteraction calls.

GetInteraction
The GetInteraction operation enables external systems to retrieve a business interaction based on an external ID or business interaction ID. The data returned in the response depends on when GetInteraction is called and on the `<responseLevel>` value the GetInteraction request specifies.

When GetInteraction is called before ProcessInteraction, the response returns only the business interaction data. In this scenario, service data is not returned because ProcessInteraction has not yet processed the business interaction into current inventory, so there is no service data in UIM yet.

When GetInteraction is called after ProcessInteraction, the response returns the business interaction data and service data. In this scenario, service data is returned because ProcessInteraction has processed the business interaction into current inventory, so there is service data in UIM to retrieve. The level of detail of service data returned by the response depends on the `<responseLevel>` value the GetInteraction request specifies. See "About the ResponseLevel Element" for more information.

About the GetInteraction Flow
Figure 9–7 shows what occurs when the GetInteraction operation is called. A business interaction is represented as BI in the figure.

Figure 9–7  GetInteraction

(Call to the GetConfiguration Web service operation)
About the GetInteraction Response and Faults

The GetInteraction response returns a varying level of information based on when the operation is called and on the <responseLevel> value the request specifies. See "About the ResponseLevel Element" for more information.

GetInteraction returns an error when:

- The request does not specify an external ID or business interaction ID upon which to base the retrieval
- It cannot find the business interaction specified in the request

UpdateInteraction

The UpdateInteraction operation enables external systems to transition UIM business entities to specific life-cycle states within the context of a business interaction.

When calling UpdateInteraction, the request must specify an external ID or business interaction ID and a business interaction action of APPROVE, ISSUE, CANCEL, or COMPLETE. Actions are defined by the BusinessInteractionActionEnum enumeration in the BusinessInteraction.xsd schema file. While this enumeration defines several actions, only the APPROVE, ISSUE, CANCEL, or COMPLETE actions are valid for UpdateInteraction.

UpdateInteraction uses the business interaction ID to find the service and service configuration, and performs the specified action for the service and service configuration. For example, if the action is APPROVE, it approves the service and service configuration associated to the business interaction and performs the action recursively to any child business interactions.

UpdateInteraction does not cascade to child services assigned to configuration items. For example, if the business interaction is associated to a service configuration, and the service configuration has a service configuration item with a child service assigned to it, the UpdateInteraction does not apply the action to the service configuration item child service status.

About the UpdateInteraction Flow

Figure 9–8 shows what occurs when the UpdateInteraction operation is called. A business interaction is represented as BI in the figure.
About the UIM Service Fulfillment Web Service

**Figure 9–8 UpdateInteraction**

About the UpdateInteraction Response and Faults

The UpdateInteraction response returns a varying level of information based on the `<responseLevel>` value the request specifies. See "About the ResponseLevel Element" for more information.

UpdateInteraction returns an error when:

- It cannot find the business interaction specified by the calling system
- The request specifies a value for `<item>` other than `<service>` or `<connectivity>`

GetConfiguration

The GetConfiguration operation retrieves the service configuration based on the search option specified in the request. The search options, which are defined in the `ConfigurationMessages.xsd` schema file, are listed and described in Table 9–3.

<table>
<thead>
<tr>
<th>Search Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceConfigurationSearch ByConfigId</td>
<td>If the request specifies a service configuration ID, GetConfiguration retrieves the service configuration that matches the specified service configuration ID.</td>
</tr>
<tr>
<td>ServiceConfigurationSearch ByEntityId</td>
<td>If the request specifies an entity ID (service ID), GetConfiguration retrieves the latest active service configuration (any state other than CANCELLED). If there is only one service configuration, GetConfiguration retrieves it.</td>
</tr>
</tbody>
</table>
You specify the search option in the GetConfiguration request, as shown in Example 9–3. The bolded line in the example shows the option of ServiceConfigurationSearchByConfigId. The element below the bolded line shows the configurationId to search for.

**Example 9–3 GetConfiguration Request**

```
<con:getConfigurationRequest>
  <con:searchOptions xsi:type="con:GetServiceConfigurationType"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <con:responseLevel>ENTITY_CONFIGURATION_EXPANDED</con:responseLevel>
    <con:configSearchOption
      xsi:type="con:ServiceConfigurationSearchByConfigId">
      <con:configurationId>123456</con:configurationId>
    </con:configSearchOption>
  </con:searchOptions>
</con:getConfigurationRequest>
```

**About the GetConfiguration Flow**

Figure 9–9 shows what occurs when the GetConfiguration operation is called.

Depending on which search option is specified, the flow may start with the Get Configuration box, or it may start with the Get Service box. For example, when the search option is ServiceConfigurationSearchByConfigId, the entry point to the flow is the Get Configuration box. When the search option is any other option, which are all based on the service ID, the entry point to the flow is the Get Service box.
About the GetConfiguration Response and Faults

The GetConfiguration response returns a varying level of information based on the <responseLevel> value the request specifies. See "About the ResponseLevel Element" for more information.

GetConfiguration returns an error when:

- The request specifies a search option other than the valid search options listed in Table 9–3, "GetConfiguration Search Options".
- The request does not specify the data that the search option needs to perform the search.
- It cannot find the configuration ID or service ID specified in the request.

UpdateConfiguration

The UpdateConfiguration operation transitions the state of the service and service configuration.

To transition the service, the request must specify the service action and service ID.

The valid service actions are:

- COMPLETE
- CANCEL
- DISCONNECT
- SUSPEND
- RESUME
To transition the service configuration, the request must specify the service configuration action and one of the following:

- Service ID
- Service configuration ID
- Service ID and service configuration version number

If the first option is specified (service ID), the operation transitions the latest active service configuration.

The valid service configuration actions are:

- APPROVE
- ISSUE
- CANCEL
- COMPLETE

**Note:** To update the service or service configuration in a manner other than just transitioning the state, use the UIM Reference Web Service updateServiceConfiguration operation. See "UpdateServiceConfiguration" in Appendix B, Reference Web Service for more information.

**About the UpdateConfiguration Response and Faults**

The UpdateConfiguration response includes a success or failure message regarding the update to transition the service or service configuration. The response returns a varying level of information based on the `<responseLevel>` value the request specifies. See "About the ResponseLevel Element" for more information.

UpdateConfiguration returns an error when:

- The request specifies an invalid service action or service configuration action.
- The request specifies invalid data for service ID, service configuration ID, or service configuration version number.

**Understanding the UIM Service Fulfillment Web Service**

This example describes the steps that occur when a CRM sends an order to UIM through an OSS provisioning system for a new prepaid service subscription. The provisioning system sends order information to UIM, such as subscriber name and address, SIM Card, and services. UIM captures the subscriber and service information, and configures the service by assigning resources such as a telephone number (TN), and assigning registrars such as a home location register (HLR) and a voice mail service (VMS). UIM then sends data back to the provisioning system, and the provisioning system uses the data to call an OSS activation system to activate the services.

**Note:** This example describes using the business interaction ID in the request and the response. However, the Web service also provides the ability to use an external object ID instead of the business interaction ID. When opting to use the external object ID, it is up to the user to ensure uniqueness.
1. Provisioning sends a CaptureInteraction request to UIM.
   The request contains a business interaction specification, a business interaction action, and a list of items. Each of the items contains an action and an entity (both necessary to design the service), as well as optional parameters such as name and description.
   In this example, the request specifies a business interaction action of CREATE. Based on this action, CaptureInteraction creates an instance of the business interaction specification and populates the instance with the input.
   The response returns the business interaction ID for the business interaction instance that was created by the request.

2. Provisioning sends a ProcessInteraction request to UIM to configure the service.
   The request contains the business interaction ID, so that ProcessInteraction knows which business interaction to process.
   The response returns the same business interaction ID that was contained in the request.

3. Provisioning sends an UpdateInteraction request to UIM to change the status of the service.
   The request contains the business interaction ID, and the status to which to update the business interaction. In this example, the request specifies a status of Approve.
   UIM updates the business interaction and service configuration to Approve.
   The response returns business interaction ID and the action taken in the form of an enumeration.

4. Provisioning sends a GetInteraction request to UIM to get the business interaction data.
   The request contains the business interaction ID, which UIM uses to retrieve the business interaction data. The response returns the retrieved business interaction data.

5. Provisioning sends a GetConfiguration request to UIM to get the service and configuration data.
   The request contains the service configuration ID, which UIM uses to retrieve the service and service configuration data. The response returns the retrieved service and service configuration data.

6. Provisioning calculates the delivery actions based on the differences between the retrieved business interaction data and the retrieved service and service configuration data.

7. Provisioning sends a request to the activation system to activate the subscriber on the primary and backup HLR and to activate the subscriber on a VMS.

8. Provisioning sends an UpdateInteraction request to UIM to change the status of the service.
   The request contains the business interaction ID and the status to which to update the business interaction and service configuration. In this example, the request specifies a status of Issue.
   UIM updates the business interaction and service configuration to Issue.
   The response returns business interaction ID and the action taken in the form of an enumeration.
Using the UIM Service Fulfillment Web Service

9. Provisioning sends a request to the activation system to activate the service registers.
   The service is now considered delivered.

10. Provisioning sends an UpdateInteraction request to UIM to update the status of the service.
    The request contains the business interaction ID and the status to which to update the business interaction and service configuration. In this example, the request specifies a status of Complete.
    UIM updates the business interaction and service configuration to Complete.
    The response returns business interaction ID and the action taken in the form of an enumeration.

Using the UIM Service Fulfillment Web Service

When using the UIM Service Fulfillment Web service, see the following sections:

■ Locating the WSDL and Schema Files
■ Deploying the Web Service
■ Testing the Web Service
■ Securing the Web Service

Locating the WSDL and Schema Files

The UIM Service Fulfillment Web service is defined by the UIMServiceFulfillment.wsdl file and is supported by several schema files. The WSDL file and supporting schema files are located in the UIM_Home/webservices/schema_servicefulfillment_webservice.zip file.

About the WSDL File

Within ZIP file, the WSDL file is located in the ora_uim_sf_webservice/wsdl directory. The WSDL file defines the CaptureInteraction, ProcessInteraction, UpdateInteraction, GetInteraction, GetConfiguration, and UpdateConfiguration operations. Each Web service operation defines a request, a response, and the possible faults that can be thrown. For example, the WSDL file defines the following for the CaptureInteraction operation:

■ captureInteractionRequest
■ captureInteractionResponse
■ captureInteractionFault
■ inventoryFault
■ validationFault

The request, response, and faults each define an XML structure that is defined in the supporting schema files. The following excerpts show how an XML structure defined in the WSDL correlates to the supporting schema files.

For example, the WSDL file defines and references the biws namespace (bolded):

```xml
xmlns:biws="http://xmlns.oracle.com/communications/inventory/webservice/businessinteraction"
```
targetNamespace. . .
<xsd:import
  namespace="http://xmlns.oracle.com/communications/inventory/webservice/businessinteraction" schemaLocation="./schemas/InteractionMessages.xsd"/>

. . .
<wsdl:message name='CaptureInteractionRequest'>
  <wsdl:part name="captureInteractionRequest" element="biws:captureInteractionRequest"/>
</wsdl:message>

This tells you that the captureInteractionRequest XML structure is defined in the
schema file that defines the specified namespace as its target namespace. A search for
the specified namespace reveals that InteractionMessages.xsd defines the referenced
namespace as its target namespace.

After you determine which schema file defines the XML structure that the WSDL file
references, you can navigate through the schema files to determine child XML
structures and elements.

About the Schema Files
There are several schema files that support the UIM Service Fulfillment Web service.
These schemas are categorized as reference schemas, Web service schemas, and
business schemas.

Reference Schemas
Within the ZIP file, the reference schemas are located in the ora_uim_sf_webservice/wsdl/referenceSchemas directory. The reference schemas define
common elements used by more than one operation. So, the elements are defined in
one place and then referenced.

The reference schemas are:

- InventoryCommon.xsd
- InventoryFaults.xsd
- FaultRoot.xsd

Note: The reference schemas use the Inventory.xsdconfig file for
XML namespace to Java package mapping.

Web Service Schemas
Within the ZIP file, the Web service schemas are located in the ora_uim_sf_webservice/wsdl/schemas directory. The Web service schemas define elements
specific to the Web service, such as the request structures, the response structures, and
any fault structures.

The Web service schemas are:

- InteractionMessages.xsd
- ConfigurationMessages.xsd

Note: The Web service schemas use the wsdl-mapping.xsdconfig file for XML namespace to Java package mapping.
Business Schemas
Within the ZIP file, the business schemas are located in the `ora_uim_business/schemas` directory. Each Web service operation wraps a call (or multiple calls) to the UIM business layer, which is exposed through APIs. The wrapped APIs are the same APIs that the UIM UI calls in response to user input. The business layer APIs are based on functional area, as are the business schemas.

The business schemas are:
- BusinessInteraction.xsd
- Characteristic.xsd
- Configuration.xsd
- CustomObject.xsd
- Entity.xsd
- InventoryGroup.xsd
- LogicalDevice.xsd
- MediaStream.xsd
- Network.xsd
- NetworkAddress.xsd
- Number.xsd
- Party.xsd
- PhysicalDevice.xsd
- Pipe.xsd
- Place.xsd
- Role.xsd
- Service.xsd
- Specification.xsd

**Note:** The API schemas use the `xmlbeans-mapping.xsdconfig` file for XML namespace to Java package mapping.

Deploying the Web Service
The UIM Service Fulfillment Web service is packaged in the `UIMServiceFulfillment.war` file, which is packaged in the `inventory.ear` file. So, when you deploy the `inventory.ear` file, you also deploy the UIM Service Fulfillment Web service.

For instructions on how to deploy the `inventory.ear` file, see *UIM System Administrator’s Guide*.

Verifying the Deployment
You can verify that any UIM Web service is deployed by viewing it in the WebLogic Server Administration Console.

To verify that a UIM Web service is deployed:
1. Log in to the WebLogic Server Administration Console.
Using the UIM Service Fulfillment Web Service


3. In the Deployments table, expand oracle.communications.inventory.


5. Under Web Services, click the link that represents the name of the Web service. For example, click the oracle.communications.inventory.sfws.UIMServiceFulfillmentPortImpl link.

6. Click the Testing tab.

7. In the Deployment Tests table, expand the name of the Web service. For example, expand oracle.communications.inventory.sfws.UIMServiceFulfillmentPortImpl.

8. Under the expanded Web service, click the WSDL link.

The WSDL file appears. Here, you can view the Web service operations that are deployed.

Testing the Web Service

Testing the Web service is done after you deploy the inventory.ear file, which automatically deploys the Service Fulfillment Web service.

Web services can be tested by using any software designed to test Web services, such as:

- LISA for testing SOAP XML through HTTP or JMS
- SoapUI for testing SOAP XML through HTTP
- HermesJMS for testing SOAP XML through JMS

Test Input XML

The UIM installation provides the GSM 3GPP Technology Pack and the Cable TV Technology Pack, and both technology packs use the UIM Service Fulfillment Web service. The technology packs provide test input XML that you can use to test the UIM Service Fulfillment Web service operations. For additional information on these technology packs, see UIM GSM 3GPP Technology Pack Implementation Guide and UIM Cable TV Technology Pack Implementation Guide.

You can also generate your own test input XML by using any software that generates XML based on schema, such as XML Spy, LISA, SoapUI, and so forth.

Preconfiguration for Testing

Prior to running the UIM Service Fulfillment Web service operations, you must have the UIM base cartridges deployed into your UIM environment. The base cartridges are located in the UIM_Home/cartridges/base directory. For additional information on the base cartridges, see UIM Cartridge and Technology Pack Guide.

Securing the Web Service

The Service Fulfillment Web service has security enabled upon installation. Specifically, the HTTP and JMS Web service ports are associated to the default
Using the UIM Service Fulfillment Web Service

WebLogic security policy file, **Auth.xml**. As a result, a user name and password must be sent in clear text over a secure tunnel (HTTPS/t3s).

---

**Note:** The user name and password, and the payload, are not encrypted to avoid significant performance impacts.

---

**About Policy Files**

A policy file can be associated to a port, or to a specific operation defined for the port. When a policy file is associated to a port, it automatically secures all operations defined for the Web service. When a policy file is not associated to a port, a policy file can be associated to one or more operations. If necessary, each operation can specify a different policy file. If no policy file is associated to the port, or to any operations, the Web service is unsecured and no security validations are performed.

Upon installation of UIM, the WebLogic default policy file, **Auth.xml**, is associated to UIMServiceFulfillmentHTTPPort and UIMServiceFulfillmentJMSPort. So, all operations are automatically secured, and all operations under each port require a user name and password in the SOAP message header. **Example 9–4** shows a SOAP message header with a user name and password specified.

**Example 9–4  SOAP Message Header**

```xml
<soapenv:Envelope
xmlns:com="http://xmlns.oracle.com/communications/inventory/webservice/common"
xmlns:ser="http://xmlns.oracle.com/communications/inventory/webservice/service"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken wsu:Id="UsernameToken-1"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
        <wsse:Username>uimuser1</wsse:Username>
        <wsse:Password
          Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">Welcome@123</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <ser:captureInteractionRequest>
      . . .
    </ser:captureInteractionRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

**Modifying Web Service Security**

You can modify the default security settings through the WebLogic Server Administration Console.

To modify the default Web service security settings, see the following:

- **Accessing Security**
- **Associating a Policy File**
Disassociating a Policy File

Modifying the Deployment Plan

Accessing Security
To access security:

1. Log in to the WebLogic Server Administration Console.
2. On the Home page, under Domain Structure, click the Deployments link.
   The Summary of Deployments page appears.
3. In the Deployments table, expand oracle.communications.inventory.
5. Under Web Services, click the link that represents the name of the Web service.
   For example, click the oracle.communications.inventory.sfws.UIMServiceFulfillmentPortImpl link.
6. Click the Configuration tab, then click the WS-Policy tab.
   The WS-Policy tab lists the policy files associated with the Web service. Upon installation, this page lists:
   - UIMServiceFulfillmentHTTPPort with the Auth.xml policy file associated
   - UIMServiceFulfillmentJMSPort with the Auth.xml policy file associated
7. Expand either port.
   All operations are listed under the port.

Associating a Policy File
You can associate a policy file to a port, or to a specific operation defined for the port.
To associate a policy file:

1. Access security for the Web service.
   See "Accessing Security" for more information.
2. Click the port or a specific operation.
   The available policy files are listed on the left, and the policy files associated with the port or operation are listed on the right.
3. In the left side, select an available policy file to associate to the port or operation.
4. Click the right arrow, which moves the available policy file to the list of associated policy files.
5. Click OK.

Disassociating a Policy File
You can disassociate a policy file from a port or from a specific operation defined for the port.
To disassociate a policy file:

1. Access security for the Web service.
   See "Accessing Security" for more information.
2. Click the port or a specific operation.
The available policy files are listed on the left, and the policy files associated with the port or operation are listed on the right.

3. In the right side, select the policy file to disassociate from the port or operation.

4. Click the left arrow, which moves the associated policy file to the list of available policy files.

5. Click **OK**.

### Modifying the Deployment Plan

If you choose to modify the default security settings for the Service Fulfillment Web service, then you also need to modify the deployment plan for the Service Fulfillment Web service.

The deployment plan is located in the `UIM_Home/app/plan/Plan.xml` file.

When you install UIM, the deployment plan contains the following:

```xml
<variable-definition>
  <variable>
    <name>WsPolicy_policy:Auth.xml_Direction_13075993400140</name>
    <value>inbound</value>
  </variable>
</variable-definition>

<module-descriptor external="false">
  <root-element>webservice-policy-ref</root-element>
  <uri>WEB-INF/weblogic-webservices-policy.xml</uri>
  <variable-assignment>
    <name>WsPolicy_policy:Auth.xml_Direction_13075993400140</name>
    <xpath>/webservice-policy-ref/port-policy/[port-name="UIMServiceFulfillmentHTTPPort"]/ws-policy/[uri="policy:Auth.xml"]/direction</xpath>
  </variable-assignment>
  <variable-assignment>
    <name>WsPolicy_policy:Auth.xml_Direction_13075993400140</name>
    <xpath>/webservice-policy-ref/port-policy/[port-name="UIMServiceFulfillmentJMSPort"]/ws-policy/[uri="policy:Auth.xml"]/direction</xpath>
  </variable-assignment>
</module-descriptor>
```

If you modify the default security settings through the WebLogic Server Administration Console, the `<value>` element (bolded in the example) gets set to **both**, and needs to be reset back to **inbound**.
Developing Custom Web Services

This chapter provides information on extending Oracle Communications Unified Inventory Management (UIM) by developing custom Web services using the UIM Reference Web service.

Before you begin writing custom Web services, you should understand the approach to developing Web services. When writing custom Web services, you should follow the specified guidelines and patterns. This chapter provides information on both of these topics.

This chapter contains the following sections:

- About the Reference Web Service
- About the WSDL-First Approach to Developing Custom Web Services
- Guidelines and Patterns for Developing Custom Web Services
- Developing and Running Custom Web Services
- Testing Custom Web Services
- Securing Custom Web Services

About the Reference Web Service

The Reference Web service is located in the UIM_Home/webservices/reference_webservice.zip file.

The reference_webservice.zip file contains several files, as described in the following sections. These files can be viewed in Oracle Communications Design Studio by importing the archive ZIP file into Design Studio.

---

**Note:** This chapter assumes you are using Design Studio to develop custom Web services; however, you can use the reference_webservice.zip file to develop custom Web services in any integrated development environment (IDE). This ZIP file contains two Eclipse-specific files, .classpath and .project, which you can ignore if you are using a different IDE.

---

ReferenceUim.wsdl

The ReferenceUim.wsdl file defines the Web service operations described in Appendix B, "Reference Web Service". Each operation defines an input, an output, and the possible faults that can be thrown.
For information on this file, see "Locating the WSDL and Schema Files" in Appendix B, *Reference Web Service*.

**Schema Files**

The schema files that support the Reference Web service define XML structures that are the inputs, outputs, and faults of the Web service operations.

For information on these files, see "Locating the WSDL and Schema Files" in Appendix B, *Reference Web Service*.

**Java Source Files**

The Java source files contain additional information in the form of comments (not Javadoc). These files are listed and described in "Developing the Web Service". The information on the Java source files is presented in "Developing the Web Service", which also provides information on which files need to be modified or created.

**build.xml File**

The *build.xml* file defines several Ant targets that you can run to build a custom Web service. An Ant target is a set of executable tasks that can be run using Ant. For information on running Ant targets within Design Studio, see "Running Ant Targets".

**build.xml File Ant Targets**

Table 10–1 describes the Ant targets defined in the *build.xml* file. "Developing and Running Custom Web Services" contains instructions that tell you when to run these Ant targets.

<table>
<thead>
<tr>
<th>Ant Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clean</td>
<td>Cleans the generated, temporary, and deliverable files and directories.</td>
</tr>
<tr>
<td>all</td>
<td>Performs the complete build process. Calls the <em>build.full</em> Ant target.</td>
</tr>
<tr>
<td>copyResources</td>
<td>Copies the properties files that store localized error messages to the appropriate UIM deployment directory. These properties files are located within the ZIP file, within the <code>&lt;config&gt;/resources/logging</code> directory, and are copied to the <code>&lt;UIM_Home&gt;/config/resources/logging</code> directory.</td>
</tr>
<tr>
<td>wspolicy</td>
<td>Updates the WAR file with the WS Policy files, which describe the Authentication and Encryption mechanism for Web service calls.</td>
</tr>
<tr>
<td>build.full</td>
<td>Performs the complete build process. Calls the following Ant targets in the specified order: <code>clean</code>, <code>generate-from-wsdl</code>, <code>build-service</code>.</td>
</tr>
<tr>
<td>build.full.http</td>
<td>Similar to the <em>build.full</em> Ant target, except it calls the <code>build-service-http</code> Ant target instead of the <code>build-service</code> Ant target. This generates the HTTP Web service WAR file.</td>
</tr>
<tr>
<td>build.full.jms</td>
<td>Similar to the <em>build.full</em> Ant target, except it calls the <code>build-service-jms</code> Ant target instead of the <code>build-service</code> Ant target. This generates the JMS Web service WAR file.</td>
</tr>
</tbody>
</table>
Table 10–1 (Cont.) build.xml Ant Targets

<table>
<thead>
<tr>
<th>Ant Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build-service</td>
<td>Builds the Web service WAR file for HTTP and JM, and stores it in the webarchive directory. The name of the WAR file is wsdl_name.war, where wsdl_name is the name specified by the WSDL_NAME parameter in the COMPUTERNAME.properties file.</td>
</tr>
<tr>
<td>build-service-http</td>
<td>Builds the Web service WAR file for HTTP and stores it in the webarchive directory. The name of the WAR file is wsdl_nameHTTP.war, where wsdl_name is the name specified by the WSDL_NAME parameter in the COMPUTERNAME.properties file.</td>
</tr>
<tr>
<td>build-service-jms</td>
<td>Builds the Web service WAR file for JMS and stores it in the webarchive directory. The name of the WAR file is wsdl_nameJMS.war, where wsdl_name is the name specified by the WSDL_NAME parameter in the COMPUTERNAME.properties file.</td>
</tr>
<tr>
<td>build.deliverable</td>
<td>Builds the Web service cartridge JAR file and stores it in the deliverables directory. Calls the build.full Ant target first to get a complete build for the WAR file.</td>
</tr>
<tr>
<td>generate-from-wsdl</td>
<td>Performs WSDL-to-Java conversions and generates object representations of the schemas. Calls the get-framework-files Ant target.</td>
</tr>
<tr>
<td>get-framework-files</td>
<td>Extracts the framework schema files InventoryCommon.xsd and InventoryFaults.xsd from the uim-webservices-framework.jar file stored in the directory specified by APP_LIB parameter defined in the COMPUTERNAME.properties file.</td>
</tr>
<tr>
<td>extract.ear</td>
<td>Extracts the application.xml file from the EAR file specified by the EAR_PATH parameter defined in the COMPUTERNAME.properties file into the reference_webservice_home/META-INF directory, where reference_webservice_home is the location of the extracted reference_webservice.zip file. The application.xml file needs to be edited manually so that the EAR file can be updated for proper deployment of the Web services.</td>
</tr>
<tr>
<td>update.ear</td>
<td>Updates the EAR file specified by the EAR_PATH parameter in the COMPUTERNAME.properties file by adding the generated Web service WAR file and the edited application.xml file in the webarchive directory into the EAR file. The updated EAR file can be deployed to test the Web services.</td>
</tr>
</tbody>
</table>

About the WSDL-First Approach to Developing Custom Web Services

The WSDL-first approach (also known as the top-down approach), is the recommended way to achieve interoperability, platform independence, and WSDL consistency across Web services. Figure 10–1 shows the design and development sequence of the WSDL-first approach.

Figure 10–1  WSDL-First Design and Development Sequence
Write the WSDL and the corresponding schemas (XSD files) to define the operations and data.

- **WSDL-to-Java generation**
  
  Use the `build.xml` Ant targets provided by the Reference Web service to generate Java source files based on the WSDL and schema definitions.

- **Develop Java Web service interface**
  
  Use the Web service development environment and tools provided by the Reference Web service to implement the Web service interface through the creation of Java files.

For example, the UIM Web service module was designed using the WSDL-first approach. This means that:

- The PortImpl Java source file was generated based on the WSDL, resulting in all operations defined in the WSDL being defined in the PortImpl Java source, but with no coding details.
- Within the PortImpl Java source, each operation was modified to call its respective operation in the AdapterRouter class.
- The AdapterRouter class calls the respective operation in each individual Adapter class.
- The build generates the PortImpl interface based on the WSDL.

### Guidelines and Patterns for Developing Custom Web Services

This section describes class diagrams that represent the UIM Web service development classes.

### Class Diagrams

In the following diagrams, `Action` represents a UIM business action such as delete or update, and `Entity` represents a UIM entity such as Party or ServiceConfiguration. For example, `ActionEntity` may represent deleteParty or getServiceConfiguration.

The Reference Web service provides example classes of the class diagrams which includes request types, response types, fault types, adapters, and implementations.

Figure 10–2 shows the class diagram for request types. `ReferenceUim.wsdl` defines `deletePartyRequest` as type `DeletePartyRequestType` (defined in `Party.xsd`), and `getServiceConfigurationRequest` as type `GetServiceConfigurationRequestType` (defined in `Service.xsd`). `DeletePartyRequestType` and `GetServiceConfigurationRequestType` both extend `InventoryRequestType` (defined in `InventoryCommon.xsd`).
Figure 10–2 Request Types

InventoryRequestType

ActionEntityRequestType

Figure 10–3 Response Types

InventoryResponseType

ActionEntityResponseType

Figure 10–3 shows the class diagram for response types. ReferenceUim.wsdl defines deletePartyResponse as type DeletePartyResponseType (defined in Party.xsd), and getServiceConfigurationResponse as type GetServiceConfigurationResponseType (defined in Service.xsd). DeletePartyResponseType and GetServiceConfigurationResponseType both extend InventoryResponseType (defined in InventoryCommon.xsd).

Figure 10–4 shows the class diagram for fault types. ReferenceUim.wsdl defines getServiceConfigurationFault as type GetServiceConfigurationFaultType (defined in Service.xsd), and updateServiceConfigurationFault as type UpdateServiceConfigurationFaultType (defined in Service.xsd). GetServiceConfigurationFaultType and UpdateServiceConfigurationFaultType both extend InventoryFaultType (defined in InventoryFault.xsd). InventoryFaultType defines a sequence of faults, which are defined by ApplicationFaultType in FaultRoot.xsd.
Figure 10–4  Fault Types

InventoryFaultType

ActionEntityFaultType

Figure 10–5 shows the class diagram for adapters. The example adapter files are PartyAdapter.java and ServiceAdapter.java, both of which extend InventoryAdapterRoot.java. The UIM-owned InventoryAdapterRoot.java class extends the Platform-owned AdaptorRoot.java class.

Figure 10–5  Adapters

InventoryAdapterRoot

EntityAdapter

Figure 10–6 shows the class diagram for a Web service implementation. The ReferenceUim.wsdl file is used to generate the ReferenceUimPort.java source file. The ReferenceUimPortImpl.java example file provides a skeleton class that implements the interface generated in the ReferenceUimPort.java source file.

Note:  The order and method signature using the fully qualified class name of the operations defined in the ReferenceUimPort interface and the ReferenceUimPortImpl implementation class are important and must match the generated source, which is based on the definitions in the WSDL file.
WSDL Interface Pattern

ReferenceUim.wsdl defines a single port type (a Web service interface) that defines all of the exposed operations. When developing new Web service operations, create them within this single port.

The current practice in UIM WS is to only use a single port. Multiple ports are not defined. The only time multiple ports are used is when you need to have a port for HTTP and another for JMS. Multiple ports should not be used for categorically grouping operations.

Operation Name Pattern

A single Web service operation defines:

- Request type

  The pattern for defining a request type is operationNameRequestType, where operationName represents an action (such as create, update, delete) and the name of the entity acted upon (such as Equipment, Pipe, TelephoneNumber). A single request type is defined per operation. For example:
  - CreateEquipmentRequestType
  - UpdatePipeRequestType
  - DeleteTelephoneNumberRequestType

- Response type

  The pattern for defining a response type is operationNameResponseType, where operationName represents an action (such as create, update, delete) and the name of the entity acted upon (such as Equipment, Pipe, TelephoneNumber). A single response type is defined per operation. For example:
  - CreateEquipmentResponseType
  - UpdatePipeResponseType
  - DeleteTelephoneNumberResponseType

- Fault types

  The pattern for defining a fault type is businessFaultFaultType, where businessFault represents a specific business fault that may be thrown. Multiple fault types may be defined per operation. For example:
  - EquipmentNotUniqueFaultType
Guidelines and Patterns for Developing Custom Web Services

- EquipmentNotFoundFaultType
- NotAuthorizedFaultType

Operation Signature Pattern

The signature of a single Web service operation is defined as follows:

```
OperationNameResponseType operationName(OperationNameRequestType) throws businessFault1FaultType, businessFault2FaultType, businessFaultNFaultType
```

For example, the deleteParty operation is defined in the `PartyAdapter.java` file:

```java
public DeletePartyResponseType deleteParty(DeletePartyRequestType deletePartyRequest) throws InventoryFaultType, ValidationFaultType
```

In another example, the captureServiceConfigurationInputs operation is defined in the `ServiceAdapter.java` file:

```java
public CaptureServiceConfigurationInputsResponseType captureServiceConfigurationInputs(CaptureServiceConfigurationInputsRequestType captureServiceConfigurationInputsRequestType) throws InventoryFaultType, ValidationFaultType
```

Table 10–2 shows the operation signature pattern on various commonly used actions. In the table, Entity represents the name of the entity (such as Equipment, Pipe, TelephoneNumber) acted upon by the operation.

### Table 10–2 Operation Signature Pattern

<table>
<thead>
<tr>
<th>Action</th>
<th>Operation Signature Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>CreateEntityResponseType createEntity(CreateEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
<tr>
<td>Find</td>
<td>FindEntityResponseType findEntity(FindEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
<tr>
<td>Update</td>
<td>UpdateEntityResponseType updateEntity(UpdateEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
<tr>
<td>Delete</td>
<td>DeleteEntityResponseType deleteEntity(DeleteEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
<tr>
<td>Calculate</td>
<td>CalculateEntityResponseType calculateEntity(CalculateEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
<tr>
<td>Capture</td>
<td>CaptureEntityResponseType captureEntity(CaptureEntityRequestType) throws businessFault1FaultType, businessFault2FaultType</td>
</tr>
</tbody>
</table>

Input Parameter

- Each operation defines only one input parameter: `operationNameRequestType`.

Output Parameter

- Each operation defines only one output parameter: `operationNameResponseType`.
Fault Types
- Fault types are the SoapFaultType or Exception thrown back to the user.
- Each operation may define multiple fault types.
- Other fault types may be based on the business exceptions thrown. One-to-one mapping between thrown business logic exceptions and the defined fault types is required to capture the different exceptions.
- Fault types contain the error codes and stack trace set by the business logic.

Schemas Pattern
The UIM Web services schema is represented by multiple XSD files. These files are based on the modularization principles followed by the UIM APIs. The UIM API level entity definitions closely follow the TMF SID standard. Modeling the XSDs per the UIM APIs gives the advantage of SID standards compliance.

For example, the XSD files are built parallel to business modules such as Service, Party, Logical Device, and so forth. Along with being modular, the advantage in keeping the XSD files separate from the WSDL makes the WSDL independent of Web Services and reusable across other software technologies.

Schema Definition Versus WSDL Definition
Data definitions are defined in the XSD files, not in the WSDL; the WSDL just references the schema definitions. Also, naming standards for the WSDL do not include Type in the name; naming standards for the schema do include Type in the name.

For example, the ReferenceUim.wsdl file defines deletePartyRequest as type DeletePartyRequestType, which is defined in the Party.xsd file. Similarly, the ReferenceUim.wsdl file defines deletePartyResponse as type DeletePartyResponseType, which is defined in the Party.xsd file.

In another example, the ReferenceUim.wsdl file defines captureServiceConfigurationInputsRequest as type CaptureServiceConfigurationInputsRequestType, which is defined in the Service.xsd file. Similarly, the ReferenceUim.wsdl file defines captureServiceConfigurationInputsResponse as type CaptureServiceConfigurationInputsResponseType, which is defined in the Service.xsd file.

Developing and Running Custom Web Services

**Note:** Design Studio is certified to run on a Windows client, and UIM is certified to run on UNIX. Developers often install UIM on Windows because development work is often done in Design Studio. You can install UIM on Windows for development work, but be aware that it is not certified, and therefore, not supported.

Alternatively, scripted builds are certified on Linux, and building a custom Web service is done through provided scripted builds. So, it is not required that you develop custom Web services on Windows; but, for writing purposes, it is assumed that you are developing custom Web services in Design Studio.
Custom Web services are developed in Design Studio, with the end result being the creation of a WAR file that is then imported into a deployable EAR file. This is a manual process, and this section provides instructions to guide you through the process.

In describing this process, it is assumed that you are working in Design Studio, and therefore working in a Windows environment. Based on this assumption, the location of all required UIM and Oracle WebLogic Server files are described using Windows paths.

---

**Important:** Oracle recommends that you perform the instructions to import, configure, and run the deleteParty Web service operation before introducing any custom code for a new Web service. A successful test of deleteParty ensures that your project is configured properly prior to the start of your custom Web service development.

---

To create a custom Web service, perform the work described in the following sections. The end result of this work is the deployment of an EAR file that contains a new WAR file that defines the Web service.

Pre-development work:

- Configuring Your Work Environment
- Importing the Reference Web Service Project
- Configuring the Imported Project

Development work:

- Locating the API Method Signature in the Javadoc
- Developing the Web Service

Post-development work:

- Generating Java Source Based on the WSDL
- Creating the WAR File
- Extracting and Updating the application.xml File
- Importing the WAR File into the EAR File
- Deploying the EAR File
- Testing Custom Web Services
- Securing Custom Web Services

---

**Configuring Your Work Environment**

Before you begin developing a custom Web service, configure your work environment.

**WebLogic Server**

You must install Oracle WebLogic Server locally. The installation provides the correct version of the JDK and several WebLogic files, both of which are required to compile the Web services project you are building. Running WebLogic Server locally is optional; the installation is required strictly to have the correct version of the JDK, and to have access to specific files for compilation.
Developing and Running Custom Web Services

UIM
You must have access to the following UIM files:

- reference_webservice.zip
- custom.ear or inventory.ear
- WebLogic Server patch files, if applicable

You can copy these files from a UIM installation on a UNIX machine to your machine, or you can install UIM locally. The entire UIM installation is not required; you only need access to UIM files listed above.

If you are copying the files from a UIM installation, or if you installed UIM locally, the location of the files is:

- UIM_Home/webservices/reference_webservice.zip
- UIM_Home/app/custom.ear or inventory.ear
- UIM_Home/lib/*.jar

**Note:** The Reference Web Service schema files are not compatible with the Service Fulfillment Web Service schema files. Due to this, the schema_service_fulfillment_webservice.zip and schema_web_service.zip files cannot reside in your work environment with the reference_webservice.zip. See "Creating Schema Files" for more information.

Design Studio
Install and configure Design Studio to work with the Reference Web service, and to develop new custom Web services. See "Installing Design Studio" and "Configuring Design Studio" for more information.

**Caution:** Configure Design Studio to use the correct version of JDK as specified by the WebLogic Server installation. See UIM System Administrator’s Guide for version information. If not configured to use the correct version of JDK, problems may be encountered that are difficult to trace.

You must also set the ANT_HOME system variable. See "Installing Ant" for more information.

Importing the Reference Web Service Project
Import the reference_webservice.zip file into Design Studio. For instructions on how to import projects into Design Studio using archive files, see the Design Studio Help.

The reference_webservice.zip file contains the directories and XML file listed below. To see these directories and XML file in Design Studio, import the archive ZIP file and open the Java perspective Navigator view.

- codegen

The codegen directory contains files that are generated from the WSDL and schema files. This directory is empty upon importing the contents of the reference_webservice.zip file.
config
The config directory contains a properties file that defines localized error messages used by the Web services module.

etc
The etc directory contains the COMPUTERNAME.properties file. See "Configuring the COMPUTERNAME.properties File" for more information.

security
The security directory contains security-related files that you can use to define authentication and authorization rules for the Reference Web service.

csrc
The src directory contains the Java source files that define the Reference Web service.

test
The test directory contains input test XML files used to test the Reference Web service.

webarchive
The webarchive directory contains the generated WAR file.

WEB-INF
The WEB-INF directory contains the web.xml file, which is a web application deployment descriptor for the Web service.

wsdl
The wsdl directory contains the ReferenceUim.wsdl file, which defines all of Reference Web service operations. This directory also contains schema files that support the WSDL definition inputs, outputs, and faults.

.classpath
The .classpath file is an Eclipse-specific file that is provided with the imported project.

.project
The .project file is an Eclipse-specific file that is provided with the imported project. This file defines the project library list, which lists JAR files that are required to successfully build the project.

build.xml
The build.xml file defines several Ant targets that you can run to build a custom Web service, as described previously in Table 10–1, "build.xml Ant Targets".

---

Note: After importing the archive ZIP file into your workspace, unresolved errors appear in Design Studio until you configure the project. See "Configuring the Imported Project" for more information.
Configuring the Imported Project

Configuring the imported project involves the following actions:

- Configuring the COMPUTERNAME.properties File
- Configuring the Project Library List

Configuring the COMPUTERNAME.properties File

To configure the reference_webservice/etc/COMPUTERNAME.properties file:

1. Copy and rename the COMPUTERNAME.properties file to reflect the name of the computer on which you have Design Studio installed. You can determine your computer name by running the following DOS command:
   
   ```
   echo %COMPUTERNAME%
   ```

2. Update the parameter values defined within the file to reflect the information appropriate to the computer on which you are developing any custom Web services.

   The file defines the following parameters:

   - **WSDL_NAME**=ReferenceUim
     
     This is the name of the WSDL file without the file extension. It is also used for deriving the context path and service URI for the generated Web services WAR file. For example, in this case the Web service context path and URI for the HTTP protocol is
     
     `/ReferenceUim/ReferenceUimHTTP` and for JMS protocol is
     
     `/ReferenceUim/ReferenceUimJMS`.

   - **QUEUE_NAME**=inventoryWSQueue
     
     This is the name of the JMS Web Service Queue. It matches the name of the queue used in the WSDL for the SOAP <address> element for the service port.

   - **MODULE_NAME**=reference_webservice
     
     This is the name of the Web service module. The name is used for creating the distributable Web service cartridge. It is also the name of the directory where the generated Web service WAR file is stored.

   - **FMW_HOME**=/opt/fmw_11gR1PS4
     
     This is the Fusion Middleware WebLogic Server installation path.

   - **WL_HOME**=${FMW_HOME}/wlserver_10.3
     
     This is the WebLogic Server installation path that incorporates the FMW_HOME parameter.

   - **DOMAIN_HOME**=/opt/uim/domains
     
     **DOMAIN_NAME**=uim720

---

*Note:* Reference schemas **InventoryCommon.xsd**, **InventoryFault.xsd**, and **InventoryFaultRoot.xsd** reside in the **uim_webservices_framework.jar** file and are automatically copied to the **wsdl/referenceSchemas** directory when you run the provided **get-framework-files** Ant target later in the process.
UIM_HOME=${DOMAIN_HOME}/${DOMAIN_NAME}/UIM
These parameters collectively specify the UIM installation path.

- **APP_LIB**=${UIM_HOME}/lib
  This is the working directory to which dependent JAR files are extracted from the *inventory.ear* file. This working directory is automatically created for you based on the name provided here.

- **EAR_PATH**=${UIM_HOME}/app/custom.ear
  This is the directory where the *custom.ear* file is located.

- **POMS_ROOT**=/opt/uim/opt/OracleCommunications/POMSClient/lib
  This is the location of the POMS JAR file.

- **PLATFORM**=/opt/uim/opt/OracleCommunications/commsplatform/ws
  This is the location of the Platform Web service JAR file.

- **PATCH_CLASSPATH**=
  This is the path to any WebLogic patch files, if applicable. You must replicate this parameter for each WebLogic patch file to specify the path and specific patch file name.

### Configuring the Project Library List
For instructions on how to configure the project library list, see the Design Studio Help.

*Figure 10–7* shows the imported project library list, which includes the JAR files needed to compile the project.

*Figure 10–7  Project Library List Before Configuring*

![Project Library List Before Configuring](image)

The required JAR files can be categorized into three groups:

- WebLogic files (FMW_LIB)
- Platform files (POMS_LIB and POMS_PLIB)
- UIM files (UIM_LIB)

The project library list of JAR files does not indicate the location of the files, so you must configure the project library list to point to the location of the JAR files. To do this, you need to add new variables named FMW_LIB, POMS_LIB, POMS_PLIB, and UIM_LIB that point the specified directory listed in Table 10–3.

### Table 10–3 Location of JAR Files

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Directory Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMW_LIB</td>
<td>FMW_Home</td>
</tr>
<tr>
<td>POMS_LIB</td>
<td>Oracle_Home/commsplatform/ws</td>
</tr>
<tr>
<td>POMS_PLIB</td>
<td>Oracle_Home/POMSClient/lib</td>
</tr>
<tr>
<td>UIM_LIB</td>
<td>UIM_Home/lib</td>
</tr>
</tbody>
</table>

### Result of Configuring Project Library List

Figure 10–8 shows the project library list after the variables are added. Notice that the library list now includes the location of the JAR files, not just the JAR file names.

### Figure 10–8 Project Library List After Configuring

JARs and class folders on the build path:

- FMW_LIB: C:\Oracle\Middleware\modules\FMW\lib\FMW_Java.jar
- POMS_LIB: C:\Oracle\Middleware\modules\commsplatform\ws\commsplatformJRE.jar
- POMS_PLIB: C:\Oracle\Middleware\modules\commsplatform\ws\commsplatformJRE.jar
- UIM_LIB: C:\Oracle\Middleware\user_projects\domains\base_domain\UIM\lib
- UIM_LIB\characteristic_caps.jar
- UIM_LIB\consumable_caps.jar
- UIM_LIB\groupenabled_caps.jar
- UIM_LIB\group-enabled_caps.jar
- UIM_LIB\jre.framework.jar
- UIM_LIB\jreLib.jar
- UIM_LIB\jreLib\jreLib.jar
- UIM_LIB\jreLib\jreLib\jreLib.jar

**Note:** Adding the variables is one way to configure the library list; you can also write down the names of the required files, remove them from the library list, click Add External JARS, navigate to their location, and add them directly to the list. Either way, the end result is the same: The library list has the location to the files needed to compile the project.

### Locating the API Method Signature in the Javadoc

When creating a new Web service, you need to wrap an existing API method. To locate a particular API method:
1. Access the Javadoc.
   For instructions on how to access the Javadoc, see "Javadoc Documentation".

2. Perform a wild card search for *Manager.class.
   All manager class names end in Manager.class, such as
   TelephoneNumberManager.class, EquipmentManager.class, and so forth.

3. Open the appropriate manager class.
   All exposed methods are defined in manager classes; so, look for a manager class
   with a name similar to the functional area that may contain the method you plan
   to wrap.

4. Locate the method you plan to wrap.

**Information to Capture**

The following information is needed to create a new Web service. This information is
available in the Javadoc after locating the method you plan to wrap.

- Class name that defines the method to wrap
- Package in which the class resides
- Method signature information:
  - Method name
  - Input parameters
  - Return values
  - Exceptions thrown

For example, the deleteParty Web service operation wraps the deleteParty() API
method. The following information was used to define this Web service in the
PartyAdapter.java file:

- PartyManager is the class that defines the deleteParty() method.
- PartyManager resides in the package oracle.communications.inventory.api.party.
- The method signature information includes:
  - Method name: deleteParty
  - Input parameters: Collection of Party objects
  - Return values: none (void)
  - Exceptions thrown: ValidationException

**Developing the Web Service**

Developing a new Web service involves creating a new WSDL file, new schema files,
and new Java source files. Creating each of these files is further explored in the
following sections.
Creating the WSDL File

**Note:** The ReferenceUim.wsdl file is written to be independent of the application server. However, the generate-from-wsdl Ant target in the build.xml file is specific to generating the required source files for deployment into a WebLogic Server environment.

The imported project contains the ReferenceUim.wsdl file, which defines all the Web service operations. Model your custom WSDL file after the ReferenceUim.wsdl file. For more information see the W3C Web Services Description Language Web site:

http://www.w3.org/TR/wsdl

and the w3schools.com WSDL Tutorial Web site:

http://www.w3schools.com/wsdl/default.asp

WSDL Naming Conventions

The ReferenceUim.wsdl file uses WSDL_NAME variable in the COMPUTERNAME.properties file for naming its various SOAP elements. This naming convention allows the build.xml Ant targets to parse these elements consistently, and to generate the correct source files for the Web service interfaces and implementation. The following list of naming conventions is assumed in the WSDL file:

- **ReferenceUim**
  This is the name of the WSDL file without the file extension as set by the WSDL_NAME variable in the COMPUTERNAME.properties file. This name is also used to automatically set other important variables in the build.xml file, such as SERVICE_NAME and PORT_NAME. This name is assumed to be the name of the root definitions element in the WSDL file. This name identifies the name of the following files, which are generated later in the process: ReferenceUimPort.java, ReferenceUimPortImpl.java, ReferenceUim.war, ReferenceUimHTTP.war, and ReferenceUimJMS.war.

- **ReferenceUimPort**
  This is the name of the PortType that is generated for the implementation later in the process. It is used by the generated source ReferenceUimPort.java and ReferenceUimPortImpl.java.

- **ReferenceUimHTTPSoapBinding**
  This is the name of the SOAP binding for Web service operations that are exposed through the HTTP transport protocol. The list of operations identified in this binding element can be a subset of the operations identified in the <PortType> element. The list of operations can be the same as or different from the JMS protocol operations.

- **ReferenceUimJMSSoapBinding**
  This is the name of the SOAP binding for Web service operations that are exposed through the JMS transport protocol. The list of operations identified in this binding element can be a subset of the operations identified in the <PortType> element. The list of operations can be the same as or different from the HTTP protocol operations.

- **ReferenceUimHTTPPort**
This is the name of the HTTP transport Port used in the UIMReference service definition. It references the ReferenceUimHTTPSoapBinding identified earlier. Also, the SOAP address location uses the following for the context path:

http://localhost:7001/ReferenceUim/ReferenceUimHttp

- ReferenceUimJMSPort
  This is the name of the JMS transport Port used in the UIMReference service definition.
  It references the ReferenceUimJMSSoapBinding identified earlier. Also, the SOAP address location uses the following for the context path:
  jms://localhost:7001/ReferenceUim/ReferenceUimJMS?URI=inventoryWSQueue

For example, if you create a new file named InventoryWs.wsdl, the naming conventions result in:

- InventoryWsPort
- InventoryWsHTTPSoapBinding
- InventoryWsJMSSoapBinding
- InventoryWsHTTPPort
- InventoryWsJMSPort

Creating Schema Files

---

**Note:** The Reference Web service schema files are written to be independent of the application server. However, the `generate-from-wsdl` Ant target in the `build.xml` file is specific to generating the required source files for deployment into a WebLogic Server environment.

---

The imported project provides supporting schemas for the Reference Web service operations. The schemas define the inputs, outputs, and faults of the wrapped methods. The schemas are used to generate the Java representation of the incoming/outgoing XML, which can then be mapped to an internal Java entity class (see "EntityMapper.java"). The Java representation is generated by the `generate-from-wsdl` Ant target.

For a new Web service, new schemas must be written that reflect the inputs and outputs of the wrapped method.

Avoiding Schema Incompatibilities
The Reference Web Service schema files are not compatible with the Service Fulfillment Web Service schema files because several of the Reference Web Service schema files share the same name of the business schema files that reside in the `schema_servicefulfillment_web_service.zip` and `schema_web_service.zip` files. Namely,

- Party.xsd
- Place.xsd
- Role.xsd
- Service.xsd
Specification.xsd

However, the file content is not the same: One set was written for the Reference Web Service, and one set was written for the Service Fulfillment Web Service.

When creating a custom Web service that needs to use, for example, the MediaStream.xsd schema file from the schema_webservice.zip, you need to create your own version of the schema file. You can extract the MediaStream.xsd schema file from the schema_webservice.zip file and use it in your work environment, but you cannot import the entire schema_webservice.zip into your work environment to reference the MediaStream.xsd file due to incompatibility.

Modifying the Mapping File
The imported project provides the type-mapping.xsdconfig mapping file. This file is used for XML namespace-to-Java package mapping. For a new Web service, modify the mapping file to update the namespace-to-Java package mappings.

Creating Java Source Files
The imported project provides the supporting Java code for numerous Reference Web service operations. The Java code contains additional information in the form comments within each class (not Javadoc).

- **ReferenceUimPortImpl.java**

  The ReferenceUimPortImpl class is the entry point into the Web service logic. This class calls the AdapterRouter class.

  ReferenceUimPortImpl.java is a generated source file with the content based on the ReferenceUim.wsdl file. This file is generated by the generate-from-wsdl Ant target and is placed in the codegen/WebServiceImpl/oracle/communications/inventory/webservice/ws directory. Copy this file to the src/oracle/communications/inventory/webservice/ws directory as a starting point for the correct implementation of the Web service calling the respective adapter classes. The Reference Web service implementation is already copied into this source directory.

  This class must be modified to call the AdapterRouter for each new Web service. Because this is a generated file, the modifications are based on the WSDL file.

- **AdapterRouter.java**

  The AdapterRouter class routes the call to a specific adapter. If the input from the external source requires mapping, the corresponding mapper class is the input/output for this class.

  This class must be modified for each new Web service.

- **EntityAdapter.java**

  Adapter classes extend the InventoryAdapterRoot class, which extends the Platform-owned AdapterRoot class. Adapters wrap the calls to the UIM API methods. Typically, one adapter class exists per manager class, such as EntityAdapter.java. However, one adapter class can wrap multiple methods from different manager classes.

  Oracle recommends that adapters be as thin as possible. They should simply contain a call to the Manager API or to other helper classes.

  An adapter calls EntityValidator and, if validations pass, calls the business layer API method.
An existing adapter class must be modified, or a new adapter class written, for each new Web service.

- **EntityValidator.java**
  
  Validator classes define an input validation method per Web service. The adapter classes call the corresponding input validation method prior to calling the wrapped API method.

  For cases where input data is passed in, a new validator class is needed per entity.

- **EntityUtils.java**
  
  Utils classes define common utility methods used by the Reference Web service operations.

  The existing EntityUtils class may be extended or a new utils class written, as needed during the development of a new Web service.

- **EntityWorker.java**
  
  Worker classes define methods used by the Reference Web service operations.

  These existing EntityWorker classes may be extended or a new worker class written, as needed during the development of a new Web service.

- **EntityMapper.java**
  
  Mapper classes map the generated object representation of the schemas (external) to the Java entity class (internal) on the way in, and map the Java entity class (internal) to the generated object representation of the schemas (external) on the way out. One mapper class maps a single entity, but a mapper class may be shared across methods in an adapter if the methods use the same entity.

  For cases where the source code references the entity data, a new mapper class is needed per entity.

- **EntityException.java**
  
  Exception classes define exceptions specific to a Web service.

  The existing EntityException classes may be extended or a new exception class written, as needed during the development of a new Web service.

- **FaultFactory.java**
  
  The FaultFactory class maps Exception objects thrown by the API method to InventoryFaultType objects returned by the Web service.

  You may need to modify this class for a new Web service; it depends on whether the API method introduces any new Exception objects that are not already mapped.

**Generating Java Source Based on the WSDL**

The imported project contains the build.xml file, which defines the generate-from-wsdl Ant target. The generate-from-wsdl Ant target copies the latest framework schema files into the Web services project and generates the Java source based on the input WSDL file and supporting schemas. You can run the generate-from-wsdl Ant target to automatically copy the framework files and generate the Java source. For instructions on how to run an Ant target such as the generate-from-wsdl Ant target, see "Running Ant Targets".

The generated package structure and generated files include:
Developing and Running Custom Web Services

- **codegen/src/oracle/communications/inventory/webservice**
  This package contains the generated Java source files. Figure 10–9 shows the generated Java source files, as based on the provided Reference Web service.

- **codegen/WebServiceImpl/oracle/communications/inventory/webservice/ws**
  This package contains the generated `wsdl_namePortImpl.java` source file. Figure 10–10 shows the generated `ReferenceUimPortImpl.java` source file, as based on the provided Reference Web service.

- **codegen/WebServiceInterface**
  This package contains the generated `wsdlName_wsdl.jar` file. Figure 10–10 shows the generated `ReferenceUim_wsdl.jar` file, as based on the provided Reference Web service.

---

**Figure 10–9 Generated Java Source Files**

![Package Explorer](image)

- `ora_uim_webservice_reference`
- `codegen/src`
  - `oracle/communications/inventory/webservice/characteristic`
    - `CharacteristicActionEnum.java`
    - `CharacteristicDataType.java`
    - `CharacteristicType.java`
  - `oracle/communications/inventory/webservice/framework.common`
    - `EntityKey.java`
    - `EntityValueType.java`
    - `InventoryRequestType.java`
    - `InventoryResponseType.java`
  - `oracle/communications/inventory/webservice/framework.fault`
    - `ApplicationFaultType.java`
    - `InventoryFaultType.java`
    - `ValidationFaultType.java`
  - `oracle/communications/inventory/webservice.party`
    - `DeletePartyRequestType.java`
    - `DeletePartyResponseType.java`
    - `PartyActionEnum.java`
    - `PartyDataType.java`
    - `PartyKey.java`
    - `PartyType.java`
  - `oracle/communications/inventory/webservice.place`
  - `oracle/communications/inventory/webservice.role`
  - `oracle/communications/inventory/webservice.service`
  - `oracle/communications/inventory/webservice.specification`
  - `oracle/communications/inventory/webservice.system`
  - `oracle/communications/inventory/webservice.wc`
After the source is generated, the project workspace has access to all the dependent files needed to compile the project. The compiled classes are stored in the `out` directory, as shown in the Navigator view in Figure 10–11. The figure shows the compiled classes for the generated source files shown in Figure 10–9. Class files compiled from Java source files that are part of the original imported project are also placed in the `out` directory, such as the class files within the `out/oracle/communications/inventory/webservice/adapter` and `mapper` directories.
Creating the WAR File

The WAR file contains the compiled classes from the developed custom Web service, plus the JAR file containing the UIM API method that the Web service wraps.

The imported project contains the build.xml file, which defines the build-service Ant target. The build-service Ant target builds the WAR file for HTTP and JMS. (The build.xml file also defines the build-service.http Ant target, which builds the WAR file for HTTP, and the build-service.jms Ant target, which builds the WAR file for JMS.) You can run any of these build-service Ant targets to automatically build the WAR file. For instructions on how to run an Ant target such as the build-service Ant target, see “Running Ant Targets”.

Figure 10–12 shows the created ReferenceUim.war file, which resides in the webarchive/reference_webbservice directory, as based on the provided Reference Web
Developing and Running Custom Web Services

The created WAR file name is `wsdl_name.war`, where `wsdl_name` is the name specified by the WSDL_NAME parameter in the `COMPUTERNAME.properties` file. The WAR file resides in the `webarchive/module_name` directory, where `module_name` is the name specified by the MODULE_NAME parameter in the `COMPUTERNAME.properties` file.

The WAR file contains the following:

- Compiled generated source files (WSDL and XML object representations)
- Compiled developed source files (contents of the `src` directory)
- JAR file that contains the class that defines wrapped method (UIM business services logic)

Figure 10–12 shows the generated directory structure, and the `ReferenceUim.war` file, as based on the provided Reference Web service.

Figure 10–12  `webarchive/WebServiceReference Directory`

```
<table>
<thead>
<tr>
<th>Package Explorer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Extracting and Updating the application.xml File

Every EAR file contains an `application.xml` file, which defines the WAR files that comprise the EAR file. When developing custom Web services, you have the option of importing the custom Web service WAR file into:

- The `custom.ear` file

  You may wish to deploy your custom Web service separately from the UIM application, in which case you would import the custom Web service WAR file into the `custom.ear` file.

- Any custom EAR files

  You may develop several different custom Web services, and wish to deploy each one separately. In this scenario, you would import each custom Web service WAR file into a separate EAR file.
If using this option, each EAR file must contain the `APP-INF/lib/uim-entities.jar` file, and a reference to `oracle.communications.inventory.corelib` within the `META-INF/weblogic-application.xml` file. For an example to emulate, see the `custom.ear` file content.

- **The `inventory.ear` file**

  You may wish to include the WAR file in the `inventory.ear` so that your custom Web service automatically deploys when you deploy the UIM application.

Regardless of which option you choose, the custom Web service WAR file needs to be included in the EAR file, so the `application.xml` file must be updated to include the name of the custom Web service WAR file.

### Extracting the File

If you choose the `custom.ear` or `inventory.ear` file, use the `extract.ear` Ant target to automatically extract the `application.xml` file from the EAR file specified by the EAR_PATH parameter in the `COMPUTERNAME.properties` file into the `reference_webservice_home/META-INF` directory, where `reference_webservice_home` is the location of the extracted `reference_webservice.zip` file.

If you choose to create your own custom EAR file, extract the `application.xml` file from the `custom.ear` file and modify as needed.

The `extract.ear` Ant target is provided in the `build.xml` file. For instructions on how to run an Ant target such as the `extract.ear` target, see "Running Ant Targets".

### Updating the File

**Example 10–1** shows the `application.xml` file from the `custom.ear` file. Whichever `application.xml` file you are updating, add a `<module>` element to identify the custom Web service WAR file and the `<context-root>` element, as shown in **Example 10–2**. The value of the `<context-root>` element is the same as the name of the WSDL file as assumed by the generated files.

**Example 10–1  application.xml**

```xml
<?xml version = '1.0' encoding = 'windows-1252'?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javae">
  <display-name>oracle.communications.inventory.customear</display-name>
  <module>
    <java></java>
  </module>
</application>
```

**Example 10–2  Updated application.xml**

```xml
<?xml version = '1.0' encoding = 'windows-1252'?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javae">
  <display-name>oracle.communications.inventory.customear</display-name>
  <!-- Custom Web Service WAR -->
  <module>
    <web>
      <web-uri>UIMReference.war</web-uri>
    </web>
  </module>
</application>
```
Importing the WAR File into the EAR File

After you determine which EAR file is to contain the custom Web service WAR file, import the WAR file into the appropriate EAR file.

The imported project contains the build.xml file, which defines the update.ear Ant target. The update.ear Ant target updates the EAR file by adding the custom Web service WAR file and the edited application.xml file. The update.ear Ant target determines the location of the EAR file to be updated by using the path you specified in the COMPUTERNAME.properties EAR_PATH parameter. Run the update.ear Ant target to automatically perform these updates to the EAR file. For instructions on how to run an Ant target such as the update.ear Ant target, see "Running Ant Targets".

Deploying the EAR File

The imported project contains the build.xml file, which defines the copyResources Ant target. The copyResources Ant target copies the referenceWS.properties file from the imported project to the UIM_Home/config/resources/logging directory. Prior to deploying the updated EAR file for the first time, run the copyResources Ant target. Unless you change the referenceWS.properties file, you only need to run this Ant target one time. For instructions on how to run an Ant target such as the copyResources Ant target, see "Running Ant Targets".

If your UIM environment resides on another machine, you must copy the updated EAR file to that machine prior to deploying.

For instructions on how to deploy an EAR file, see UIM System Administrator’s Guide.

   Tip: After you have gone through all the steps in this chapter once, you only need to run the clean, all, and update.ear Ant targets to rebuild the EAR file prior to deploying.

Verifying the Deployment

After you have deployed the updated EAR file, verify that the deployment includes the custom service by viewing the Web services in the WebLogic Server Administration Console. See "Verifying the Deployment" in Chapter 9, Integrating UIM through Web Services.

Specifying a Deployment Plan

If you placed your custom Web service in the custom.ear file, or in any custom EAR file, you must specify a deployment plan for the updated EAR file. If you placed your custom Web service in the inventory.ear file, the deployment plan is already specified.

Specifying a deployment plan enables the EAR file to retrieve property values from the UIM_Home/app/AppFileOverrides/platform/runtime-poms.properties file, which defines property values that are used by the persistence framework for cache coordination.

To specify a deployment plan:

1. Log in to the WebLogic Server Administration Console.
2. On the Home page, under Domain Structure, click the Deployments link.
The Summary of Deployments page appears.

3. Under Change Center, click **Lock & Edit**.

4. In the Deployments table, select the updated EAR file that contains your custom Web service, and click **Update**.
   The Update Application Assistant page appears.

5. Click **Change Path**.

6. Change the path to `UIM_Home/app/plan`.

7. Choose `Plan.xml`, and click **Next**.

8. Choose **Redploy this application using the following deployment files**, and click **Finish**.

### Testing Custom Web Services

Test the custom Web service after you deploy the updated EAR file. Custom Web services can be tested by using any software designed to test Web services, such as:

- LISA for testing through HTTP or JMS
- SoapUI for testing through HTTP
- HermesJMS for testing through JMS

### Generating Test Input XML

You can generate your own test input XML by using any software that generates XML based on schema, such as XML Spy, SoapUI, and so forth.

### Preconfiguration for Testing

Be aware of any preconfigurations that must be in place prior to testing any custom Web services.

### Securing Custom Web Services

When you create a new Web service, it is up to you secure the Web service. How you secure the Web service depends upon how you created the Web service. For example, if your custom Web service deploys with the `custom.ear` file, you need to create your own deployment plan; if your custom Web service deploys with the `inventory.ear` file, you need to modify the `inventory.ear` deployment plan that is part of the UIM installation (`UIM_Home/app/plan/Plan.xml` file).

To secure a custom Web service:

1. Access security for the custom Web service.

2. Associate a security policy to the custom Web service.
   See "Associating a Policy File" in Chapter 9, *Integrating UIM through Web Services*.
   You can use the security policy that comes with the UIM installation (`Auth.xml`), or the security policy that comes in the Reference Web service ZIP file (`SampleAuth.xml`), or create your own security policy file.
When you associate a security policy to the custom Web service, a deployment plan is generated in the form of a **Plan.xml** file.

3. Associate the generated deployment plan with the custom Web service by redeploying the EAR file that contains the custom Web service; the redeploy prompts you to supply the path to the EAR file, and to supply the name of the deployment plan (**Plan.xml**).
   
a. The prompt to supply the name of the deployment plan may also prompt you to select **Inbound** or **Both**: Select **Inbound**.

4. Ensure that the deployment plan reflects **Inbound**. See "Modifying the Deployment Plan" in Chapter 9, *Integrating UIM through Web Services*.

See "Securing the Web Service" in Chapter 9, *Integrating UIM through Web Services* for more information.
This chapter provides information on customizing the Oracle Communications Unified Inventory Management (UIM) user interface (UI), which is written using Oracle Application Development Framework (ADF) and Platform Common User Interface (CUI). The information in this chapter describes statically customizing the UI, which can result in backwards compatibility issues. See “Backwards Compatibility” for the implications regarding this type of extension.

UIM UI customizations are made in JDeveloper. After installing JDeveloper, you customize the UIM UI by importing the `UIM_Home/build/dist/inventory.ear` file into JDeveloper and making the desired customizations. You then update the `inventory.ear` file with the customizations and redeploy it for testing.

This chapter contains the following sections:

- Installing JDeveloper
- Extracting the inventory.ear File into JDeveloper
- Customizing the User Interface
- Deploying User Interface Customizations
- Testing User Interface Customizations

### Installing JDeveloper

**Note:** Before installing JDeveloper, you must install the Sun Java Development Kit (JDK). For information on installing JDK, see *UIM Installation Guide*.

To install JDeveloper:

1. Go to the Oracle Software Delivery Cloud.
2. From the Select a Product Pack list, select Oracle Fusion Middleware.
3. From the Platform list, select Microsoft Windows (32-bit).
4. Click Go.
5. From the Results list, select Oracle Fusion Middleware 11g Media Pack for Microsoft Windows (32-bit) for release 11.1.1.5.0, as shown in Figure 11-1.
6. Click **Continue**.
   The download options for the selected media pack appears.

7. Click **Download** for Oracle JDeveloper Part 1 and for Oracle JDeveloper Part 2, as shown in Figure 11–2:

8. Save the downloaded ZIP files to a local directory, such as *JDev_Home*.

9. Navigate to *JDev_Home*, open the downloaded ZIP files, and extract the contents into *JDev_Home*.
   After the extraction, *JDev_Home* contains the *jdevstudio11115install.jar* file.

10. From a command line, navigate to *JDev_Home* and run the following command:
    ```
    java -jar jdevstudio11115install.jar
    ```
    This initiates the JDeveloper installer.

11. On the Welcome window, click **Next**.
    The Choose Middleware Home Directory window appears.

12. Select **Create a new Middleware Home**, enter a middleware home directory name, and click **Next**.
    **Note:** The remainder of this chapter refers to the middleware home directory you entered as *JDev_Home*.

13. Select **Complete** and click **Next**.
    The JDK Selection window appears.

14. Click **Browse** and navigate to your local installation of the JDK, and click **Next**.
    The Confirm Product Installation Directories appears.

15. Take the defaults and click **Next**.
    The Choose Shortcut Location window appears.

16. Take the defaults and click **Next**.
    The Installation Summary window appears.

17. Take the defaults and click **Next**.
    The installation begins.
18. When the installation completes, click **Done**.

**Extracting the inventory.ear File into JDeveloper**

To extract the `inventory.ear` file into JDeveloper:

1. Create a local directory, such as `tempEar_Home`.
2. Copy the `UIM_Home/build/dist/inventory.ear` file to `tempEar_Home`.
3. Double-click the `JDev_Home/jdeveloper/jdeveloper.exe` file. The Select Role window appears.
4. Select **Default Role** and click **OK**. The Configure File Type Associations window appears.
5. Click **Cancel**.
6. From the JDeveloper menu, select **File**, then select **New**. The New Gallery window appears.
7. On the **Available Items** tab, under **Categories**, expand **General**, and select **Applications**.
8. Under **Items**, select **Application from EAR File**, and click **OK**. The Create Application from EAR File window appears. This window has three parts: **Location**, **Ear Modules**, and **Finish**. **Location** appears first.
9. Next to the **EAR File** field, click **Browse** and navigate to the `tempEar_Home/inventory.ear` file. Selecting the `inventory.ear` file automatically populates the fields on this window as follows:
   - **EAR File** defaults to `tempEar_Home/inventory.ear`, based on the selected EAR file.
   - **Application File** defaults to `inventory`, based on the name of the selected EAR file.
   - **Directory** defaults to `C:/JDeveloper/mywork/inventory`. You can change the defaulted directory to any directory you prefer. The directory specified gets created by the process you are about to initiate.
   - **Source Roots** defaults to `tempEar_Home`.
10. Leave the **Copy Files to Application** check box deselected, and click **Next**. **Ear Modules** appears.
11. Accept the default module names and project names, and click **Next**. **Finish** appears, showing the `inventory.ear` file location, and the location of the projects that JDeveloper is about to build based on the modules in the `inventory.ear` file.
12. Click **Finish**. JDeveloper does the following:
   - Creates a workspace. The workspace directory name and location are based on the directory name and location specified in the **Directory** field on the Location window.
■ Creates an application in the workspace. The application name (inventory) is based on the imported EAR file name.

■ Creates several projects within the inventory application, as shown in Figure 11–3. Each project name is based on a module name from the selected inventory.ear file.

![Figure 11–3 Inventory Application Projects](image)

13. Delete all of the projects except the inv project by doing the following:
   a. Select all of the projects except the inv project.
   b. Right-click on the group of selected projects and select Delete Project.
      The Confirm Delete Project window appears.
   c. Select Remove projects from application, and click Yes.

Customizing the User Interface

Customizations can be in the form of new files or additions to existing files. If you are deleting files or modifying existing files with changes or deletions, be aware of the errors this may cause. These types of errors are logged by Oracle WebLogic Server when you deploy the updated inventory.ear file.

---

**Note:** You cannot customize the UIM home page.

About the UI Files

UIM UI customizations involve several types of files, such as JSFF, XML, Java, and XLF files, as described in the following sections.

**JSFF and XML Files**

Each page in the UIM UI is defined by a JSFF and XML file. For example, the UIM Party Summary page is defined by the PartySummary.jsff and...
PartySummaryPageDef.xml, and the UIM Party Maintenance page is defined by the PartyEdit.jsff and PartyEditPageDef.xml files.

These files are located within the inventory application inv project, in the Web Content/oracle/communications/inventory/ui/functionalArea/page directory, where functionalArea is a UIM functional area such as equipment, number, service, and so forth.

Within each functionalArea/page directory, the JSFF and XML file names follow the naming convention shown in Figure 11–4. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The XML page file names end with PageDef.

**Figure 11–4  Page File Naming Conventions**

![Application Navigator](image)

**XML Files**

Each page in the UIM UI has a specific task flow defined by a an XML file. For example, the UIM Party Summary page task flow is defined by the PartySummaryFlow.xml file, and the UIM Party Maintenance page task flow is defined by the PartyEditFlow.xml file.

These files are located within the inventory application inv project, in the Web Content/WEB-INF/oracle/communications/inventory/ui/functionalArea/flow directory, where functionalArea is a UIM functional area such as equipment, number, service, and so forth.
Within each *functionalArea/flow* directory, the XML file names follow the naming convention shown in Figure 11–5. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The XML task flow file names end with *Flow*.

**Figure 11–5  Task Flow File Naming Conventions**

![Application Navigator](image)

**Java Files**

The functionality of each page in the UIM UI is driven by logic in a Java source file that is compiled into a Java class file. For example, the UIM Equipment Summary page is driven by the `EquipmentSummaryBean.class` file, and the Equipment Maintenance page is driven by the `EquipmentEditBean.class` file.

These files are located within the inventory application inv project, in the `WebContent/WEB-INF/classes/oracle/communications/inventory/ui/functionalArea/bean/page` directory, where *functionalArea* is a UIM functional area such as equipment, number, service, and so forth.

Within each *functionalArea/bean/page* directory, the Java file names follow the naming convention shown in Figure 11–6. For example, each file name contains the entity name (Place, Party, and so forth), and the Web page (Search, List, Summary, Edit, and so forth). The Java file names end with *Bean*. 
XLF Files

XLF files define text values that display throughout the UIM UI. XLF files also define formats that are used to display the text values in a specific way.

The `InventoryUIBundle.xlf` file, which defines text values, is located within the inventory application inv project, in the `Web Content/WEB-INF/classes/oracle/communications/inventory/ui/common/bundle` directory.

The `Preferences.xlf` file, which defines the DATE_FORMAT, is located within the inventory application inv project, in the `Web Content/WEB-INF/classes/oracle/communications/platform/ui` directory.

**Note:** If you customize the DATE_FORMAT in the `Preferences.xlf` file, you must also change the `system.dateFormat` specified in the `UIM_Home/config/resources/logging/system.properties` file.

When entity managers throw informational, warning, or error messages that contain a date, the message date is not formatted using the DATE_FORMAT specified in the XLF file. Rather, the message date is formatted using the `system.dateFormat` specified in the `system.properties` file. So, if you customize the date format, you must change it in both files.
DCX File

The DataControls.dcx file defines the registry for all the delegates, which are defined as data controls. If your customizations require a new delegate, this file needs to be updated to include the new delegate.

This file is located within the inventory application inv project, in the Web Content/oracle/communications/inventory/ui directory.

The following Web sites are useful when working with DCX files to customize the UIM UI:

- The ADF Tasks virtual book provides ADF information and is available at: http://www.oracle.com/pls/as111130/vbook_subject?subject=adf
- Additional information can be found at: http://download.oracle.com/docs/cd/E14571_01/index.htm

Displaying Custom Attributes on a Web Page

Custom attributes are any attributes that you have added to an existing entity. You can display custom attributes by editing the JSFF files for the entity’s functional area. For example, if you add the subscriberId attribute to the Service entity, you can display subscriberId on the UIM Service Summary page by editing the inv/oracle/communications/inventory/ui/service/page/ServiceSummary.jsff file.

To display the value of the subscriberId attribute, add the following component to the JSFF file:

<af:outputText
value="#{pageFlowScope.ServiceSummaryBean.entityObject.subscriberId}"/>

Adding Custom Input Fields to a Web Page

You can edit the value of custom attributes in the UIM UI by adding an input field to the JSFF file for the entity’s maintenance page.

By convention, maintenance page file names end with Edit. For example, EquipmentEdit.jsff maintains an equipment entity, and ServiceEdit.jsff maintains a service entity. Maintenance pages operate in two modes:

- **New**: For creating a new instance of the entity
- **Edit**: For modifying an existing instance of the entity

A managed bean exists for every entity, and the bean contains all of the attributes defined for the entity. For example, for an equipment entity, the page is EquipmentEdit.jsff and the Java class is EquipmentEditBean.class. Similarly, for a service entity, the page is ServiceEdit.jsff and the Java class is ServiceEditBean.class.

If the type attribute is added to the Equipment entity, type can be displayed on the UIM Equipment Summary page and edited on the UIM Equipment Maintenance page.

To do this, edit the following files:

- In the InventoryUIBundle.xlf file, add the following to define the text for the type attribute as it displays in the UI:
In the `EquipmentSummary.jsff` file, add the following ADF component:

```xml
<af:panelLabelAndMessage label="#{inventoryUIBundle.Type}" id="plam2">
    <af:outputText value="#{pageFlowScope.EquipmentSummaryBean.entityObject.type}" id="ot3"/>
</af:panelLabelAndMessage>
```

In the `EquipmentEdit.jsff` file, add the following ADF component to edit the field:

```xml
<af:inputText value="#{pageFlowScope.EquipmentEditBean.entityObject.type}" label="#{inventoryUIBundle.Type}" id="it1"></af:inputText>
```

### Adding Conditional Components to a Web Page

Components are Web page building blocks. For example, the OutputText component is used for displaying entity attribute values on a Web page, and the InputText component is used for editing entity attribute values on a Web page. Conditional components are components that may or may not be rendered on a Web page, depending upon the outcome of an expression that can be evaluated. You can make a component a conditional component with custom logic.

To make a component a conditional component, edit the JSFF page file and map the component to a Java method, which evaluates an expression. The expression can be implemented with a custom logic class that extends the original bean class. For example, to make the `Activate/Deactivate` check box attribute on the UIM Equipment Maintenance page conditional upon an active condition being true, make the following changes:

- In the `EquipmentEdit.jsff` file:

  ```xml
  <af:selectBooleanCheckbox value="#{pageFlowScope.EquipmentEditBean.active}" text="#{inventoryUIBundle.Active}" disabled="#{!(pageFlowScope.EquipmentEditBean.active)}" id="it7">
  </af:selectBooleanCheckbox>
  ```

- Create a new Java class that extends `EquipmentEditBean.java`, and have the class define the following method:

  ```java
  public boolean getActive()
  {
      if(this.getEquipment() != null && this.getEquipment() instanceof Equipment)
      {
          Equipment equipment = (Equipment)this.getEquipment();
          InventoryState inventoryState = equipment.getAdminState;
          return(inventoryState != null !inventoryState.equals(InventoryState.END_OF_LIFE):true);
      }
      return false;
  }
  ```

### Disabling an Input Field on a Web Page

You can disable InputText components based on a condition. For example, to make the `type` attribute on the UIM Equipment Maintenance page conditional upon an active condition being true, make the following changes to the `EquipmentEdit.jsff` page file:

```xml
<af:inputText value="#{pageFlowScope.EquipmentEditBean.entityObject.type}"
```
Adding a Custom Action to a Web Page

You can add a custom action to a Web page by editing the JSFF page file to include a link or a button to call a custom listener method on the page. For example, to add a button to the UIM Service Summary page, which calls the generateReport() method, make the following changes to the `<ServiceSummary.jsff>` file:

```xml
    <af:outputText value="Text value of the link"/>
</af:commandButton>
```

A custom class needs to implement this method, and the custom class needs to run in place of the original. This is done using rulesets and extension points. For information on rulesets and extension points, see Chapter 8, "Extending UIM through Rulesets".

If the custom class is `com.foo.ServiceSummary.class`, the `ServiceSummary.java` source file would reside in the `<inv/Web Content/WEB-INF/src/com/foo>` directory and would contain the following:

```java
package com.foo;
public abstract class ServiceSummary
    extends oracle.communications.inventory.ui.service.bean.page.ServiceSummaryBean
{
    public void generateReport(ActionEvent event)
    {
        // perform the custom logic here
        System.out.println(getService().toString());
    }
}
```

Next, the `<inv/Web Content/WEB-INF/oracle/communications/inventory/ui/service/flow/ServiceSummaryFlow.xml>` task flow must be edited to add the new `com.foo.ServiceSummary.class` to the pageFlowScope.

Adding a Custom Search Field

You can add a custom search field to existing search criteria. For example, you can add the `Grade` field to the Telephone Number Search criteria. To do this, you must extend the API that the UI calls, as well as the UI.

Extending the API

To add a custom search field to existing search criteria and extend the API to take this new field into account:

1. Create a class named `CustomTNSearchCriteriaImpl` that extends `TelephoneNumberSearchCriteriaImpl`. Add the `CriteriaItem` for `Grade` to this class.
2. Write a ruleset to override the `PersistenceHelper.makeTelephoneNumberSearchCriteria()` method to create and return an instance of `CustomTNSearchCriteriaImpl` instead of the `TelephoneNumberSearchCriteriaImpl`.

See Chapter 8, "Extending UIM through Rulesets" for information on how to write a ruleset.
3. Write a ruleset to extend the TelehoneNumberManager.findTelephoneNumber() method. In the extension code, add your business-specific code to use this new search criteria to restrict the result set.

See Chapter 8, "Extending UIM through Rulesets" for information on how to write a ruleset.

Extending the UI

**TelephoneNumberSearch.jsff** renders **InventoryQuery.jsff** to build the query criteria on the page. To add the Grade field:

1. Create a new custom class, such as **TNQueryBean.java**, that extends the oracle.communications.inventory.ui.number.bean.query.TelephoneNumberQueryBean class. To add a new field, the getAttributeDescriptors() method needs to be overridden.

2. Your custom class, **TNQueryBean.java**, needs to have something as shown in the following example. (In the getAttributeDescriptors() method, the fields in the queryAttributes String Array are rendered on the UI as search fields.)

   ```java
   public class TNQueryBean extends TelephoneNumberQueryBean {
       public static final String GRADE = "GRADE";

       public TNQueryBean() { super(); }

       protected List<AttributeDescriptor> getAttributeDescriptors() {
           List<AttributeDescriptor> attributeDescriptors = super.getAttributeDescriptors();

           AttributeDescriptorImpl attributeDescriptor = null;
           attributeDescriptor = this.createAttributeDescriptorImpl
               ("GRADE", "GRADE", Constants.STRING_TYPE,null,
               ComponentType.inputText);

           String[] queryAttributes =
               {TELEPHONE_NUMBER, RANGE_FROM, Constants.SPECIFICATION, RANGE_TO,
               SERVICE_SPECIFICATION, INVENTORY_GROUP, Constants.INVENTORY_STATUS,
               CONDITION_TYPE, Constants.RESOURCE_ASSIGNMENT_STATUS, GRADE};
           this.setQueryAttributes(queryAttributes);

           return attributeDescriptors;
       }
   }
   ```

3. Change the **TelephoneNumberSearchResultsFlow.xml** file to add your custom class (**TNQueryBean**) in pageFlowScope, as shown below:

   ```xml
   <managed-bean>
4. Create a new custom class, such as `TNDelegate.java`, that extends the `TelephoneNumberDelegate` class. In the custom class, override the `getSearchCriteria()` method to pass the `Grade` field to API. On the API side, extend `oracle.communications.inventory.api.number.TelephoneNumberSearchCriteria`, and define the `Grade` field as a member. This is shown in the following example:

```java
import oracle.communications.inventory.ui.common.utils.CriteriaContainer;
import oracle.communications.inventory.api.number.TelephoneNumberSearchCriteria;

public class TNDelegate extends TelephoneNumberDelegate {
    public TNDelegate() { super(); }

    protected CriteriaContainer getSearchCriteria() {
        CriteriaContainer container = super.getSearchCriteria();
        TelephoneNumberSearchCriteria criteriaTNObj = (TelephoneNumberSearchCriteria)container.getCriteria();
        // Set the GRADE field in CriteriaTNObj and pass it to the container.
        return container;
    }
}
```

5. Update the `DataControls.dcx` file to include the new `TNDelegate`, as shown below:

```xml
<AdapterDataControl id="TelephoneNumberDelegate"
    FactoryClass="oracle.communications.inventory.ui.framework.datacontrol.Inventor
yDataControlFactoryImpl"
    ImplDef="oracle.adf.model.adapter.beanBeanDefinition"
    SupportTransactions="false"
    SupportsSortCollection="true"
    SupportsResetStates="false"
    SupportsRangesize="false"
    SupportsFindMode="false"
    SupportsUpdates="true"
    Definition="oracle.communications.inventory.ui.number.delegate.TNDelegate"
    BeanClass="oracle.communications.inventory.ui.number.delegate.TNDelegate"
    xmlns="http://xmlns.oracle.com/adfm/datacontrol">
```
6. Copy `TelephoneNumberDelegate.xml`, paste it in the same directory, and rename it `TNDelegate.xml`. Afterward, open `TNDelegate.xml` and change all occurrences of `TelephoneNumberDelegate` to `TNDelegate`.

### Deploying User Interface Customizations

To deploy your UIM UI customizations:

1. In JDeveloper, create the `inv.war` file:
   a. In the Application Navigator, right-click on the `inv` project, select **Deploy**, then select `inv`.

      The Deploy inv window appears. This window has two parts: **Deployment Action**, and **Summary**. **Deployment Action** appears first.

   b. Select **Deploy to WAR** and click **Next**.

      **Summary** appears, showing the location of `inv.war` file after JDeveloper builds it.

   c. Click **Finish**.

   ![Note: Neither the created `inv.war` file nor the created `deploy` directory in which it resides displays in JDeveloper, even after a refresh. To see it, navigate to your JDeveloper workspace in Windows Explorer.]

2. Update the `inventory.ear` file to include the updated version of the `inv.war` file you just created:
   a. Outside of JDeveloper, navigate to `tempEar_Home`.
   b. Open the `inventory.ear` file.
   c. Add the `inv.war` file to the `inventory.ear` file, replacing the existing `inv.war` file with the `inv.war` file that contains your UI customizations.
   d. Save and close the `inventory.ear` file.
   e. Copy the updated `inventory.ear` file from `tempEar_Home` to the `UIM_Home/build/dist` directory.

3. Deploy the updated `inventory.ear` file.

   For instructions on how to deploy the `inventory.ear` file, see *UIM System Administrator’s Guide*.

### Customizing Logos

When customizing the UI, you can also customize logos. Customizing logos involves a different set of files, so there is a separate procedure for customizing them.

To customize logos:

1. Open the `UIM_Home/lib/comms-platform-webapp.war` file and extract the `WEB-INF/lib/comms-platform-ui.jar` file to a local directory, such as `tempDir`.

2. Open the `tempDir/comms-platform-ui.jar/images` directory and add your custom logo file.
3. Open the `tempDir/comms-platform-ui.jar/oracle/communications/platform/templates/CommsUIShell.jspx` file, and modify the file as follows:
   a. Locate the text:
      ```html
      <af:image id="oracleImage"
      source="/afr/logo-oracle-red.png"
      clientComponent="true" shortDesc="Oracle"/>
      ```
   b. Change the text that defines the source to:
      ```html
      <af:image id="oracleImage"
      source="/images/customLogoFileName"
      clientComponent="true" shortDesc="Oracle"/>
      ```
      where `customLogoFileName` is the name of your custom logo file that you previously added to the `tempDir/comms-platform-ui.jar/images` directory. The `customLogoFileName` includes the file type extension, such as `.gif`, `.jpg`, or `.png`.

4. Save and close the `tempDir/comms-platform-ui.jar` file.

5. Repackage the WAR file by doing the following:
   a. Open the `UIM_Home/lib/comms-platform-webapp.war/WEB-INF/lib` directory.
   b. Replace the `comms-platform-ui.jar` file with the `tempDir/comms-platform-ui.jar` file that contains your customizations.
   c. Save and close the `UIM_Home/lib/comms-platform-webapp.war` file.

6. Log in to the WebLogic Server Administration Console.

7. Stop the UIM application:
   b. Select the check box for `oracle.communications.inventory`, and click Stop.
   c. Choose Force Stop Now, and click Yes. The UIM application stops.

8. Delete the UI library:
   a. On the Summary of Deployments page, select the check box for `oracle.communications.platform.cui.webapp`, and click Delete. The library is deleted.

9. Open a command line.

10. Navigate to the `UIM_Home/servers/serverName/tmp/_WL_user` directory:
    ```
    cd UIM_Home/servers/serverName/tmp/_WL_user
    ```

11. Delete the `oracle.communications.inventory` directory.
    ```
    rm -rf oracle.communications.inventory
    ```
12. Close the command line and return to the WebLogic Server Administration Console, Summary of Deployments page.

13. Install the UI library:
   a. Click Install, and select comms-platform-ui.jar located in UIM_Home/lib.
      The library is installed.

14. Select the check box for oracle.communications.inventory, and click Update.
    This redeployes the inventory.ear file and starts the UIM application.

Testing User Interface Customizations

You can test your UIM UI customizations by running UIM and navigating to the customized pages or new pages to validate that the customizations are working correctly. If customizations included changes or deletions to existing files, regression testing is required to ensure the customizations did not break existing UIM UI functionality.

Note: If working in a clustered environment, delete the oracle.communications.inventory directory from the tmp/_WL_user directory for each of the servers.
This chapter provides information on localizing the Oracle Communications Unified Inventory Management (UIM) user interface (UI), and on localizing the UIM Help. Localization is the process of translating a UI or Help system from the original language in which it was written into a different language for use in a specific country or region. For example, the UIM UI and UIM Help are written in English. If your company is based in France and you purchase UIM, you may want to localize UIM to display the UI and Help in French.

Localizing UIM involves modifying a specific set of files that UIM uses to display text in the UI and in the Help.

This chapter contains the following sections:

- Setting the Language Preference in Internet Explorer
- Determining the Locale ID
- Localizing UIM
- Localizing UIM Help

---

**Note:** Before localizing your UIM environment, you must identify a strategy for maintaining future localizations. Oracle does not provide a file that lists the details of what changed between releases.

---

**Setting the Language Preference in Internet Explorer**

For a localized version of UIM to display correctly in Internet Explorer, users need to configure language preferences.

To configure language preferences in Internet Explorer:

1. From the **Tools** menu, select **Internet Options**.
   
   The Internet Options window appears.

2. Click **Languages**.
   
   The Language Preference window appears.

3. The language you plan to use must display at the top of the list to have priority.
   
   If the language you plan to use is listed:
   a. Select the language.
   b. Click **Move Up** to move the language you plan to use to the top of the list.
   
   If the language you plan to use is not listed:
a. Click Add.
   The Add Language window appears.
b. Select a language.
c. Click OK.
   The Language Preference window returns.
d. Select the language you have added, and click Move Up to move it to the top of the list.

4. Click OK.

Determining the Locale ID

A locale ID is a standardized ID that represents a language and region in which the language is spoken. For example, **fr-CA** is the locale ID for French spoken in Canada, and **es-MX** is the locale ID for Spanish spoken in Mexico.

Localizing UIM involves copying and renaming existing files to include a locale ID. The renamed files that include a locale ID become the translated version of the original files.

To determine the locale ID in Internet Explorer:

1. From Tools menu, then select Internet Options.
   The Internet Options window appears.
2. Click Languages.
   The Language Preference window appears.
3. Click Add.
   The Add Language window appears.
   Languages are listed alphabetically. Several languages are spoken in more than one country, so the locale ID reflects the language and the country in which the language is spoken.
4. Locate the language to which you are localizing and note the locale ID.
5. Close the Add Language, Language Preference, and Internet Option windows.

Localizing UIM

Localizing the UIM UI involves working with a UIM-provided cartridge that you import into Oracle Communications Design Studio, modify, and deploy. Design Studio also provides various editors, such as an XML editor and an HTML editor, that you can use to translate files for localization.

The following sections describe localizing UIM:

- About the UI-Specific Files
- Localizing the UI-Specific Files
- Deploying the Cartridge Containing the Localized Files
- Testing the UIM UI Localization
About the UI-Specific Files

The UI-specific files are a set of .xlf and .properties files that contain localizable text strings that define labels and messages. You modify the text string within these files to localize UIM.

- .xlf files
  The UIM UI was written using Application Development Framework (ADF). ADF-specific files use the .xlf file extension. XLF files contain localizable text strings for labels that display in the UI.

- .properties files
  The UIM UI calls UIM API methods, which may result in an information, warning, or error message displaying in the UI. Properties files contain localizable text strings for API messages that display in the UI.

Localizing the UI-Specific Files

Localizing the UI is accomplished by modifying the text strings in XLF and properties files that display in the UI.

To localize the UI-specific files, perform the work described in the following sections:

- Importing the Localization Archive File into Design Studio
- Locating the UI-Specific Files within the Project
- Copying and Renaming the UI-Specific Files
- Editing the UI-Specific Files

Importing the Localization Archive File into Design Studio

Note: Within Design Studio, you must be in the Studio Design perspective Cartridge view.

The $UIM_HOME/cartridges/sample/ora_uim_localization_reference_cartproj.zip file contains an Inventory project with all of the UI-specific files that you can import into Design Studio to localize.

For instructions on how to import projects using archive files, see the Design Studio Help.

Locating the UI-Specific Files within the Project

Note: Within Design Studio, you must be in the Java perspective Package Explorer view.

The localization archive file that you imported into Design Studio contains the ora_uim_localization_reference project. The UI-specific files are contained within the project.

XLF Files
The UI-specific XLF files are located in the ora_uim_localization_reference project, within the
model/content/inventory.ear/inv.war/WEB-INF/classes/oracle/communications
directory. The communications directory contains the following subdirectories, which
contain the UI-specific XLF files:

- inventory/ui/common/bundle/InventoryUIBundle.xlf
- inventory/ui/framework/bundle/InventoryOHWBundle.xlf
- platform/ui/CommsUIShell.xlf
- platform/ui/Preferences.xlf

Properties Files
The UI-specific properties files are located in the ora_uim_localization_reference
project, within the model/content/product_home/config/resources/logging
directory.

Copying and Renaming the UI-Specific Files
Copying and renaming the UI-specific files ensures that the default file is always in
place to use for display if needed. Adding the locale to the file name differentiates
your localized files from the default files, which simplifies upgrades. If files are edited
for localization without being renamed to reflect the locale, all localization efforts are
lost when you upgrade because the files are overwritten.

To copy and rename the files within the Design Studio Java perspective Package
Explorer view:

1. Right-click on the file and select Copy.
2. Right-click on the parent directory of the copied file and select Paste.
   The Name Conflict dialog box appears.
3. Modify the file name to include the appropriate locale ID.
   For example, rename InventoryUIBundle.xlf to InventoryUIBundle_fr_ca.xlf and
   rename equipment.properties to equipment_fr_ca.properties for French spoken
   in Canada.
   See “Determining the Locale ID” for more information.
4. Click OK.

Note: On the Add Language window shown in “Determining the Locale ID”, the locale ID is separated by a dash. When renaming
the XLF and properties files, use an underscore in place of the dash.

Note: If you copy and paste the file, and then try to rename it, the
Rename menu option is not available when right-clicking on the file
in the Java perspective. You can, however, copy and paste the file and
rename by selecting File from the menu, and then selecting Rename.

Editing the UI-Specific Files
To edit the UI-specific files, perform the work described in the following sections:

- Editing the XLF Files
- Editing the Properties Files
Editing the XLF Files
To edit the XLF files within Design Studio:

1. Open the Java perspective.
2. Open the Package Explorer view.
3. Within the imported project, locate the XLF files.
   See "Locating the UI-Specific Files within the Project" for more information.
4. Right-click on the file and select **Open With**, then select **Text Editor**.

   **Caution:** If you double-click on the file, Design Studio may open the file for editing outside of Design Studio.

5. Edit the value of the `<source>` elements, which define text that displays in the UI.
   Example 12–1 is an excerpt from the `InventoryUIBundle.xlf` file that shows numerous `<source>` elements. Edit only the value of the `<source>` elements: for example **UIM Home Page**, **Inventory**, **Home**, and **Products**.

   **Example 12–1  InventoryUIBundle.xlf**
   
   ```xml
   <trans-unit id="LANDING_PAGE_TITLE">
   <source>UIM Home Page</source>
   <target/>
   </trans-unit>
   <trans-unit id="MENU_INVENTORY">
   <source>Inventory</source>
   <target/>
   </trans-unit>
   <trans-unit id="MENU_HOME">
   <source>Home</source>
   <target/>
   </trans-unit>
   <trans-unit id="MENU_PRODUCT">
   <source>Products</source>
   <target/>
   </trans-unit>
   </trans-unit>
   ```

   **Note:** The `Preferences.xlf` file defines a date format. If you want to localize the date format, see "XLF Files" for more information.

Editing the Properties Files
To edit the properties files within Design Studio:

1. Open the Java perspective.
2. Open the Package Explorer view.
3. Within the imported project, locate the properties files.
   See "Locating the UI-Specific Files within the Project" for more information.
4. Right-click on the file and select **Open With**, then select **Text Editor**.

   **Caution:** If you double-click on the file, Design Studio may open the file for editing outside of Design Studio.
5. Edit the text strings that define API messages that display in the UI.

Example 12–2 is an excerpt from the `party.properties` file that shows two messages. Each message is defined by two lines: the first line defines the message ID, and the second line defines the message text that displays in the UI. Edit only the message text; for example, `Party Id {0} already exists` and `The party with Id {0} was successfully deleted`.

Example 12–2 also shows that messages are not necessarily error messages; the `partyDeleted` message is an informational message.

```
Example 12–2  party.properties

party.alreadyExists.id=230002
party.alreadyExists=Party Id {0} already exists.
party.partyDeleted.id=230009
party.partyDeleted=The party with Id {0} was successfully deleted.
```

Deploying the Cartridge Containing the Localized Files

Localized files are modified as part of a project. After the modifications are complete, build the project to create the cartridge that can be deployed into UIM. Every cartridge should be cleaned and rebuilt prior to deploying.

For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge into UIM using the Cartridge Deployer Tool, see `UIM System Administrator’s Guide`.

---

**Note:** When a cartridge containing localizable XLF files is deployed into UIM, the `inventory.ear` file automatically redeployed, resulting in the localization changes being applied to the UI.

---

Testing the UIM UI Localization

You can test your UIM UI localization by running UIM and navigating from page to page to validate that the pages are displaying the localized text.

Localizing UIM Help

The following sections describe localizing UIM Help:

- About UIM Help
- Localizing the UIM Help Files
- Deploying the Localized Help System
- Testing the UIM Help Localization

About UIM Help

The UIM Help uses Oracle Help for the Web. Oracle Help is a browser-based Help system that runs as a Web application based on a Java servlet. You do not need specialized knowledge of Oracle Help to localize UIM Help; you can use the information in this chapter, supplemented by the Oracle Help documentation. See `Oracle Fusion Middleware Developer’s Guide for Oracle Help` for more information:

```
http://docs.oracle.com/cd/E16162_01/doc.1112/el6280/toc.htm
```
About the Oracle Help Configuration File
The Oracle Help configuration file, ohwconfig.xml, is located in the UIM_home/app/inventory.ear/inv.war/WEB-INF/help directory. The ohwconfig.xml file contains references to each Help system deployed into an application. Upon installation, ohwconfig.xml references the default UIM Help system (English) deployed into UIM. This file requires configuration for localization.

About the UIM Help Files
The UIM Help files are located in the UIM_home/app/inventory.ear/inv.war/WEB-INF/help/helpsets/uimoh_help.jar file, which contains the following Help files:

- **.htm files**: Each HTML file is a separate Help topic. The text in all of the HTML files requires translation.
- **uimoh.hs**: This file describes the Help system. When UIM Help is initiated through the UIM user interface, uimoh.hs is the starting point. This file does not require translation.
- **toc.xml**: This file defines the Table of Contents that appears in the left pane of the Oracle Help window. The text in this file requires translation.
- **map.xml**: This file associates Help IDs with the HTML file names. The toc.xml file uses the IDs to link entries to Help topics. This file does not require translation.
- **search.idx**: This file is used when you perform a text search of the Help content. The file defines a search index that searches the Help content in the HTML files. After the HTML files are translated, the search index must be regenerated using the Java-based Help Indexer. For more information, see "Regenerating the Search Index File".
- **target.db**: This file contains cross-reference information used for navigating between Help topic headings. This file does not require translation.
- **dcommon/html/cpyr.htm**: This file defines the Help copyright page and requires translation. (The dcommon directory contains standard Oracle support files, including a CSS file, several graphics files, and the Help copyright page, but only the Help copyright page requires translation.)

Localizing the UIM Help Files
To localize UIM Help, perform the work described in the following sections:

- Extracting the Help Files
- Translating the Help Files
- Regenerating the Search Index File
- Creating the Localized Help JAR File
- Configuring the Oracle Help File

**Extracting the Help Files**
Use the default Help files installed with UIM as the starting point for your localization. To extract the Help files:

1. Copy the UIM_home/app/inventory.ear/inv.war/WEB-INF/help/helpsets/uimoh_help.jar file to tempDir, where tempDir is a local directory.
2. Open the tempDir/uimoh_help.jar file.
3. Extract all the objects in the uimoh_help.jar file into tempDir.
4. Click the File column heading in tempDir, which sorts the objects by file type.

You should see the following directories and files in tempDir:

- dcommon directory
- img directory
- META-INF directory
- target.db
- uimoh_help.jar
- uimoh.hs
- numerous .htm files
- search.idx
- map.xml
- toc.xml

Translating the Help Files
To translate the Help files, perform the work described in the following sections:

- Translating the Copyright Page
- Translating the Help Topics
- Translating the Table of Contents

Translating the Copyright Page
To translate the copyright page:

1. In Windows Explorer, navigate to the tempDir/dcommon/html directory.
2. Open the cpyr.htm file.
3. Translate the content of the <title>, <h1> through <h6>, and <p> elements to the local language.

For example, translate the bolded content in Example 12–3:

Example 12–3 Excerpt from cpyr.htm

<title>Oracle Legal Notices</title>
<link rel="stylesheet" href="../css/blafdoc.css" type="text/css" />
</head>
<body>
<h1>Oracle Legal Notices</h1>
<h2>Copyright Notice</h2>
<p>Copyright © 1994-2012, Oracle and/or its affiliates. All rights reserved.</p>

Translating the Help Topics
To translate the Help topics:

1. In Windows Explorer, navigate to the tempDir directory.
The Help topics text is defined in the numerous .htm files within this directory. Each .htm file must be translated.

2. Open an .htm file.

3. Translate the content of the <title>, <h1> through <h6>, <p>, and <td> elements to the local language.

For example, translate the bolded content in Example 12–4. Elements that are not text, such as the HTML tags themselves, should not be changed.

Example 12–4  Excerpt from tel_nbr_info_work_area.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta name="OAC_IGNORE_SKIP_NAV" content="true" />
<meta http-equiv="Content-Type" content="text/html; charset=us-ascii" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<meta http-equiv="Content-Script-Type" content="text/javascript" />
<title>Telephone Number - Information Work Area</title>
<meta name="generator" content="Oracle DARB XHTML Converter (Mode = ohj/ohw) - Version 5.1.2 Build 073" />
<meta name="date" content="2012-09-17T22:25:55Z" />
<meta name="robots" content="noarchive" />
<meta name="doctitle" content="Telephone Number - Information Work Area" />
<meta name="relnum" content="Release 7.2.2" />
<meta name="partnum" content="E36042-0" />
<meta name="topic-id" content="telephoneInfo" />
<link rel="copyright" href="/dcommon/html/cpyr.htm" title="Copyright" type="text/html" />
<link rel="stylesheet" href="/dcommon/css/blafdoc.css" title="Oracle BLAFDoc" type="text/css" />
<link rel="contents" href="toc.htm" title="Contents" type="text/html" />
</head>
<body>
<p><a id="CHDCJEIG" name="CHDCJEIG"></a><a id="telephoneInfo" name="telephoneInfo"></a></p>
<div class="sect1"><!-- infolevel="all" infotype="General" -->
<h1>Telephone Number - Information Work Area</h1>
<p>You use the <span class="gui-object-title">Telephone Number - Information</span> work area to edit the information that appears in the Summary work area <span class="gui-object-title">Information</span> panel. Some data elements, such as the ID, cannot be changed after the entity is created.</p>
</div>
</body>
```

4. Repeat steps 2 and 3 for each .htm file in the tempDir directory.

The fields that appear in this work area are determined by the entity specification definition used to create the entity. The specification is created in Design Studio. The fields defined below for this entity are common among most specifications.
Translating the Table of Contents
To translate the Table of Contents:

1. In Windows Explorer, navigate to the tempDir directory.
2. Open the toc.xml file.
   Each item in the Table of Contents is defined by a <tocItem> element.
3. Translate the content of each <tocItem> to the local language.
   For example, translate the bolded content of the text attribute in Example 12–5. Do not change the content of the target attribute.

Example 12–5 Excerpt from toc.xml
<tocitem target="uim_help_interface.htm" text="Getting Started with Unified Inventory Management">

Note: Oracle Help automatically translates the Help window menu options; field names; and informational, warning, and error messages. The translation is based on the locale defined in the ohwconfig.xml file.

For example, if the only language preference specified is English, and the ohwconfig.xml file defines a single locale of French, Oracle Help translates the Help window menu options, field names, and messages to French.

Oracle recommends that the language preference with the highest priority be the same language defined as the locale in the ohwconfig.xml file.

Regenerating the Search Index File
After translating the Help files, regenerate the search index file to reflect the content of the translated files. This is accomplished using Oracle Help Indexer.

Note: Using Oracle Help Indexer requires that you have Java installed.

To install Oracle Help Indexer:

1. Go to the Oracle Technology Network Web site:
   http://www.oracle.com/technetwork/topics/utilsoft-085729.html
2. Download the help-indexer.jar file to tempDir, where tempDir is a local directory.

To regenerate the search index file using Oracle Help Indexer:

1. Open a Windows command prompt.
2. Change the directory to tempDir by entering the following command:
   cd tempDir
3. Enter the following command, which creates a new search.idx file, and overwrites the existing search.idx file:
   java -mx64m -classpath pathToJarFile/help-indexer.jar oracle.help.tools.index.Indexer -l=locale -e=charset pathToHelpFiles search.idx
where:

- `pathToJarFile` is the directory path to the `help-indexer.jar` file.
- `locale` is the standardized locale ID that represents the localized language. See "Determining the Locale ID" for more information.
  
  If you do not specify a locale, the system’s default locale is used.
- `charset` is the Java-supported character set encoding.
- `pathToHelpFiles` is the directory path to the Help files.

For example:

```java
java -mx64m -classpath C:\tempDir\help-indexer.jar
oracle.help.tools.index.Indexer -l=fr_CA -e=8859_1 C:\tempDir search.idx
```

See "Using the Text Search Indexer" in Oracle Fusion Middleware Developer’s Guide for Oracle Help for more information:

http://docs.oracle.com/cd/E15523_01/doc.1111/e14149/oha_gen_fts.htm

### Creating the Localized Help JAR File

After translating the Help files and regenerating the search index, create a new JAR file containing the localized Help files.

To create the new JAR file:

1. In Windows Explorer, navigate to the `tempDir` directory.
   
   The `tempDir` directory contains the `uimoh_help.jar` file, the translated Help files, and the regenerated search index file.

2. Copy the `uimoh_help.jar` file to `tempDir` to create a second copy of the `uimoh_help.jar` file in `tempDir`.

3. Select the copied version of the `uimoh_help.jar` file and rename it `uimoh_help.locale.jar`, where `locale` is the standardized ID that represents a language and region in which the language is spoken. For example, `fr-CA` is the locale for French spoken in Canada, and `es-MX` is the locale for Spanish spoken in Mexico.

   For more information, see "Determining the Locale ID".

   **Note:** On the Add Language window shown in "Determining the Locale ID", the locale ID is separated by a dash. When renaming the JAR file, use an underscore in place of the dash.

4. Open the `uimoh_help.locale.jar` file.

5. Delete all of the objects in the JAR file.

6. Add the localized Help files to the `uimoh_help.locale.jar` file. (This includes all of the directories and all of the files in `tempDir`, with the exception of `uimoh_help.jar` and `uimoh_help.locale.jar`.

7. Save and close the `uimoh_help.locale.jar` file.

You can verify that you included all of the directories and files by checking the number of objects in the `uimoh_help.jar` file and in the `uimoh_help.locale.jar` file; the two JAR files should contain the same number of objects. To determine the number of objects in
Localizing UIM Help

each JAR file, select all of the objects in each JAR file; this provides a count of all objects selected.

Configuring the Oracle Help File

After translating the Help files, regenerating the search index, and creating a localized Help JAR file, configure the ohwconfig.xml file to reflect the localized Help JAR file.

To configure the ohwconfig.xml file:


   The file defines the default Help system (English):

   `<locales>
   <!-- English: -->
   <locale language="en">
   <books>
   <helpSet id="uimoh_help"
     jar="/helpsets/uimoh_help.jar"
     location="uimoh.hs"/>
   </books>
   </locale>
   </locales>

2. Update the `<locale>` element to reflect the localized Help system:

   `<locales>
   <!-- French Canadian: -->
   <locale language="fr">
   <books>
   <helpSet id="uimoh_help_fr_ca"
     jar="/helpsets/uimoh_help_fr_ca.jar"
     location="uimoh.hs"/>
   </books>
   </locale>
   </locales>

You do not need to change the location attribute value, which is the name of the file that resides in the specified JAR file.

About Multiple Locales

Oracle Help can support multiple locales. For multiple locales, each localized Help system is configured with a `<locale>` element in the ohwconfig.xml file. For example, the following results in both French and Spanish Help systems being available in UIM upon redeployment:

```
<locales>
 <!-- French: -->
 <locale language="fr">
   <books>
     <helpSet id="uimoh_help_fr_ca"
       jar="/helpsets/uimoh_help_fr_ca.jar"
       location="uimoh.hs"/>
   </books>
 </locale>
</locales>
<locales>
 <!-- Spanish: -->
 <locale language="es">```
When multiple locales are defined, the language preference for all locales must be set. If not set, only the first locale defined in the ohwconfig.xml file displays in UIM Help. See "Setting the Language Preference in Internet Explorer" for more information.

When multiple locales are defined, the <parameters> element configuration values are applied:

- **<combineBooks>**

  To merge Help systems, set the value of <combineBooks> to **true**. The Help navigational views behave as a single, integrated Help system.

  To use separate Help systems, set the value of <combineBooks> to **false**. The separate Help navigational views are accessed based on the language preference with the higher priority.

  Regardless of the <combineBooks> value, each locale that is defined in the ohwconfig.xml file must be specified as a language preference. See "Setting the Language Preference in Internet Explorer" for more information.

  **Note:** Oracle Help automatically translates the Help window menu options; field names; and informational, warning, and error messages. The translation is based on the first locale defined in the ohwconfig.xml file.

  For example, if the only language preference specified is English, and the ohwconfig.xml file defines the locales of French and Spanish, Oracle Help translates the Help window menu options, field names, and messages to French.

  However, when multiple locales are defined, the language preference for all locales must be specified. Otherwise, only the first locale defined in the ohwconfig.xml file displays in UIM Help. So, when the language preferences are set, Oracle Help translates the Help window menu options, field names, and messages to the language preference with the highest priority.

- **<useLabelInfo>**

  If <useLabelInfo> is set to **true**, author-defined labels are used for the navigators of merged Help systems.

  If <useLabelInfo> is set to **false**, default labels such as Contents, Index, and Search are used for the navigators of merged Help systems.

- **<cacheSize>**

  Note:
<cacheSize> indicates the number Help systems kept in memory at one time. The default value is 3.

For more information, see "Oracle Help for the Web Configuration File" in Oracle Fusion Middleware Developer’s Guide for Oracle Help, which you can find here:

http://docs.oracle.com/cd/E16162_01/doc.1112/e16280/ohff_ohwconfig.htm

Deploying the Localized Help System

The default Help system is deployed when you deploy the inventory.ear file.

To deploy the localized Help system:

1. Repackage the UIM_Home/app/inventory.ear file to include the localized Help files by doing the following:
   b. Copy the tempDir/uimoh_help_locale.jar file to the UIM_Home/app/inventory.ear/inv.war/WEB-INF/help/helpsets directory.

   **Note:** If your UIM Help is supporting multiple locales, each JAR file defined by each <locale> element in the ohwconfig.xml file must be present in the UIM_Home/app/inventory.ear/inv.war/WEB-INF/help/helpsets directory.

2. Deploy the repackaged inventory.ear file.

   For instructions on how to deploy the inventory.ear file, see UIM System Administrator’s Guide.

Testing the UIM Help Localization

After you deploy the localized Help system, test your UIM environment to verify that the localized Help system is working correctly.

In UIM, open the Help and do the following:

- Navigate to several topics from links in the Table of Contents to ensure that the correct topics appear and display correctly.
- Test several links within Help topics to ensure they are working.
- Search for several terms and verify that you get the expected results.
- If testing multiple locales that function as a single Help system, verify translations for all locales.
- If testing multiple locales that function as separate Help systems, change the language preference priority to verify translations for each locale.
This chapter provides information about optimizing Oracle Communications Unified Inventory Management (UIM) concurrent resource allocation. The information describes how UIM telephone number entities use row locking to optimize concurrent resource allocation, and how you can extend additional UIM entities to use row locking in your UIM environment. For example, if your UIM environment heavily uses equipment entities, you can extend UIM to have equipment entities use row locking to optimize concurrent resource allocation of equipment entities.

This chapter contains the following sections:

- About Concurrent Resource Allocation
- About Row Locking
- Understanding How Row Locking Works for Telephone Numbers
- Extending UIM Entities to Use Row Locking

**About Concurrent Resource Allocation**

Concurrent resource allocation occurs when multiple clients simultaneously select the same resource from the same resource pool and then try to reserve or assign that resource.

The inherent problem with concurrent resource allocation is that only one of the clients can assign the resource; the other clients fail and must re-try the operation. When the client re-tries the operation, the resources are queried a second time, and again only one client succeeds. As this scenario unfolds, new clients may query the same resource pool. Multiple clients simultaneously selecting from the same resource pool, failing to reserve or assign a resource, and re-trying the operation can continue indefinitely, causing performance issues.

For example, when two clients simultaneously search by ZIP code for an available telephone number to assign, the search results can return the same telephone number to both clients. If both clients try to assign the same telephone number, one client succeeds, and one client fails. The client that fails must re-run the search to get a new set of available telephone numbers from which to assign.

In most UIM environments, telephone number entities are heavily used. Due to this, telephone number entities use row locking to optimize concurrent resource allocation for assigning and reserving telephone numbers. See "About Row Locking" for more information.
About Row Locking

This section builds upon the information presented in Chapter 4, "Extending the Data Model".

Row locking is a data model pattern that can be statically applied to a UIM entity through the metadata files, and enabled and used through custom code. When the row locking is in place for an entity and an entity finder API called, lock policy details may be provided to the entity finder API. If lock policy details are provided, selected entity rows are locked. If no lock policy details are provided, selected entities are returned to the client with no row locks on them.

For example, when two clients simultaneously search by ZIP code for an available telephone number to reserve or assign, the search results do not return the same telephone number to both clients because each telephone number entity that meets the search criteria is locked prior to returning the search results. The locking is achieved by updating the TelephoneNumberRowLock table with the telephone numbers returned in the search. If a telephone number is already present in the TelephoneNumberRowLock table, the telephone number is filtered from the search results prior to returning the search results to the client. When telephone number locks are released, the telephone numbers are removed from the TelephoneNumberRowLock table.

Row locking removes the potential of the same telephone number being returned to both clients. If the same number is not returned to both clients, the same number cannot be selected by both clients, which eliminates the possibility of one client failing to assign the telephone number.

Row locking is applied by setting the appropriate LockPolicy values during the search. See "About the LockPolicy Object" for more information.

---

Note: Implementing the row lock pattern achieves the same result as database row-locking. However, traditional database row-locking is not used.

---

Understanding How Row Locking Works for Telephone Numbers

In UIM, TelephoneNumber entities are used for resource reservation and resource assignment. Upon UIM installation, the TelephoneNumber entity is the only entity enabled for row locking through the entity definition in the metadata. UIM search logic performs row-locking operations based on the lock policy details provided to the search.

For example, when the entity finder API is called to find telephone numbers for resource reservation, the entity finder API logic row-locks the telephone numbers. If the row-lock operation is successful, telephone numbers are reserved and the row-locks are released. If the row-lock operation fails (row-locks for all the telephone numbers are not obtained), an error message displays and no attempt is made to reserve the telephone numbers.

Similarly, when the entity finder API is called to find telephone numbers for resource assignment, the entity finder API logic row-locks the telephone numbers. If the row-lock operation is successful, telephone numbers are assigned to a configuration item and the row-locks are released. If the row-lock operation fails (row-locks for all the telephone numbers are not obtained), an error message displays and no attempt is made to assign the telephone numbers.
About the RowLock Pattern

The RowLock pattern is statically applied to the TelephoneNumber entity in the metadata. Additionally, a LockPolicy object is used in the telephone number search criteria. When searching for telephone numbers, the telephone numbers returned in the search results are locked based on the details specified in the LockPolicy object. In this manner, concurrent resource allocation attempts return different sets of telephone numbers.

About Releasing Locked Rows

Locked rows are released in the following ways:

- When an entity finder API is called with a LockPolicy, the entity finder API logic calls a method at the end of the operation that releases the locks.
- A timer listener automatically releases row-locked telephone numbers at specified intervals.
- A database administrator can manually release row-locked telephone numbers.

About the LockPolicy Object

The LockPolicy object defines the attributes listed in Table 13–1. LockPolicy attribute values are set as part of the search criteria, which is passed to the entity finder API, and which affects the search behavior.

### Table 13–1  LockPolicy Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numberOfResources</td>
<td>long</td>
<td>0</td>
</tr>
<tr>
<td>expirationTimeStamp</td>
<td>Date or int</td>
<td>Value specified for the lockPolicy.defaultRowLockExpirationDuration property in the system-config.properties file.</td>
</tr>
<tr>
<td>filterExistingLocks</td>
<td>boolean</td>
<td>false</td>
</tr>
</tbody>
</table>

**numberOfResources**

The numberOfResources value tells the entity finder API the number of entity rows to lock. For example, when a value of zero is specified, no entity rows are locked; when a value greater than zero is specified, the number of entity rows specified by the value are locked.

If the search criteria specifies a range, and the numberOfResources value is greater than zero, the entity finder API ignores the range and returns the number of entities specified by the numberOfResources value. For example, if the range specifies 0-20 and the numberOfResources values specifies 50, 50 entities are returned. This occurs because the find-by-range feature is disabled when row locking is used when calling and entity finder API.

**expirationTimeStamp**

The expirationTimeStamp value is used by the entity finder API when creating the locked row. The expirationTimeStamp value can be set as a date or as a duration. When the value is a date, the value is used to set the lock expiration date and time. When the value is a duration, the value is added to the current timestamp to calculate the lock expiration date and time.
filterExistingLocks
The filterExistingLocks value indicates whether or not the entity finder API filters out existing locked entities from the search result. When the value is true, existing locks are not included in the search results. When the value is false, existing locks are included in the search results. When a client search results are for view-only purposes, the value must be false.

Example LockPolicy Attribute Combinations
The entity finder API finds entities based on the specified search criteria and lock policy. Table 13–2 summarizes the different LockPolicy attribute value combinations and the affect each has on the entity finder API search results.

When numberOfResources is zero, the expirationTimeStamp value is not applicable because if no locks are applied, there is no need to set when the locks expire. When numberOfResources is greater than zero, the expirationTimeStamp value does not affect the outcome. As a result, the expirationTimeStamp attribute is not included in the table.

<table>
<thead>
<tr>
<th>numberOfResources</th>
<th>filterExistingLocks</th>
<th>Entity Finder API Search Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
<td>The search results exclude row-locked entities. From the search results, no entities are locked. The search results are returned to the client.</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>The search results include row-locked entities. From the search results, no entities are locked. The search results are returned to the client.</td>
</tr>
<tr>
<td>n</td>
<td>true</td>
<td>The search results exclude row-locked entities. From the search results, n entities are locked, time stamped, and returned to the client.</td>
</tr>
<tr>
<td>n</td>
<td>false</td>
<td>The search results include row-locked entities. From the results, n entities are locked, time stamped, and returned to the client.</td>
</tr>
</tbody>
</table>

Extending UIM Entities to Use Row Locking
Extending UIM entities to use row locking involves:

- Statically Extending the Data Model
- Enabling Row Locking
- Using Row Locking with Entity Finder APIs

Statically Extending the Data Model
This section builds upon the information presented in Chapter 4, "Extending the Data Model".

To statically extend the data model.

1. Open Oracle Communications Design Studio.
2. Import the UIM_Home/cartridges/tools/ora_uim_entity_sdk_cartproj.zip file.
3. Create a new uim-*entities.xml file in the ora_uim_entity_sdk/src directory.
4. In the XML file, add an *EntityName*RowLock entity definition and sequence definition, where *EntityName* is an entity name such as TelephoneNumber or Equipment.

   Example 13–1 is an excerpt from the uim-number-entities.xml metadata file that shows the TelephoneNumberRowLock entity definition and sequence definition.

   **Example 13–1  uim-number-entities.xml**

   ```xml
   <entity type="ocim:TelephoneNumberRowLock"
   interface="oracle.communications.inventory.api.entity.TelephoneNumberRowLock"
   timeBound="true" entityIdsSequenceGenerator="RowLockEntitySeqGen">
   <implements interface="java.lang.Cloneable"/>
   <implements interface="oracle.communications.inventory.api.EntityRowLock"/>
   <attribute name="lockedEntityId" index="unique"/>
   </entity>
   <entityIdsSequenceGenerator name="RowLockEntitySeqGen"
   sequence="ROWLOCKENTITY_SEQ" initialValue="0" allocationSize="1"/>
   
   Example 13–2 shows what you would need to add to the uim-equipment-entities.xml metadata file to extend Equipment entities to use row locking.

   **Example 13–2  uim-equipment-entities.xml**

   ```xml
   <entity type="ocim:EquipmentRowLock"
   interface="oracle.communications.inventory.api.entity.EquipmentRowLock"
   timeBound="true" entityIdsSequenceGenerator="RowLockEntitySeqGen">
   <implements interface="java.lang.Cloneable"/>
   <implements interface="oracle.communications.inventory.api.EntityRowLock"/>
   <attribute name="lockedEntityId" index="unique"/>
   </entity>
   <entityIdsSequenceGenerator name="RowLockEntitySeqGen"
   sequence="ROWLOCKENTITY_SEQ" initialValue="0" allocationSize="1"/>
   
   Example 13–3 is an excerpt from the uim-number-types.xsd metadata file that shows the TelephoneNumberRowLock complexType definition, which defines the lockedEntityId attribute for the TelephoneNumberRowLock entity.

   **Example 13–3  uim-number-types.xsd**

   ```xml
   <xs:complexType name="TelephoneNumberRowLock">
   <xs:sequence>
   <xs:element name="lockedEntityId" type="xs:long">
   <xs:simpleType>
   <xs:restriction base="xs:long">
   <xs:maxLength value="19"/>
   </xs:restriction>
   </xs:simpleType>
   </xs:element>
   </xs:sequence>
   </xs:complexType>
   ```

5. Create a new uim-*-types.xsd file in the ora_uim_entity_sdk/src directory.

6. In the XSD file, add an *EntityName*RowLock complexType definition.

   Example 13–3 is an excerpt from the uim-number-types.xsd metadata file that shows the TelephoneNumberRowLock complexType definition, which defines the lockedEntityId attribute for the TelephoneNumberRowLock entity.

   **Example 13–3  uim-number-types.xsd**

   ```xml
   <xs:complexType name="TelephoneNumberRowLock">
   <xs:sequence>
   <xs:element name="lockedEntityId" type="xs:long">
   <xs:simpleType>
   <xs:restriction base="xs:long">
   <xs:maxLength value="19"/>
   </xs:restriction>
   </xs:simpleType>
   </xs:element>
   </xs:sequence>
   </xs:complexType>
   ```
Example 13–4 shows what you would need to add to the uim-equipment-types.xsd metadata file to extend Equipment entities to use row locking.

Example 13–4  uim-equipment-types.xsd
<xs:complexType name="EquipmentRowLock">
  <xs:sequence>
    <xs:element name="lockedEntityId" type="xs:long">
      <xs:simpleType>
        <xs:restriction base="xs:long">
          <xs:maxLength value="19"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

See "Extending the Data Model Through the Metadata Files" for more information.

7. Generate the data model to include any new row lock entities. See "Generating, Compiling, and Packaging the Entity Source Files" for more information.

Enabling Row Locking
To enable row locking:
1. Create a custom timer listener class that does the following:
   - Implements the TimerListener interface
   - Overrides the timerExpired method
   - Contains logic in the timerExpired method that deletes expired row locks

Example 13–5 shows a custom timer listener class that deletes expired equipment row locks. To delete other entity row locks, replace the equipment-specific listener class name and code (bolded) with the respective listener class name and entity names.

Example 13–5  Custom Timer Listener Class
package com.package.entityrowlock;
import javax.transaction.UserTransaction;
import oracle.communications.inventory.api.framework.logging.Log;
import oracle.communications.inventory.api.framework.logging.LogFactory;
import oracle.communications.inventory.api.framework.security.UserEnvironment;
import oracle.communications.inventory.api.framework.timer.InventoryTimer;
import oracle.communications.inventory.api.framework.timer.TimerListener;
import oracle.communications.inventory.api.util.Utils;
import oracle.communications.platform.persistence.Finder;
import oracle.communications.platform.persistence.PersistenceHelper;
import oracle.communications.platform.persistence.PersistenceManager;

public class EquipmentRowLockExpiryListener implements TimerListener {
    private static final Log log =
        LogFactory.getLog(EquipmentRowLockExpiryListener.class);
    /**
     * This method deletes all the expired equipment row locks.
     *
     * private void timerExpired (InventoryTimer timer)
Extending UIM Entities to Use Row Locking

```java
{
    PersistenceManager pm = null;
    UserTransaction userTransactionInstance = null;
    UserEnvironment userEnvironment = null;
    Finder finder = null;
    try {
        userEnvironment = Utils.startUserEnvironment();
        pm = PersistenceHelper.makePersistenceManager();
        finder = PersistenceHelper.makeFinder();
        StringBuilder filterStr = new StringBuilder();
        filterStr.append("delete from EquipmentRowLock o where o.endDate <= sysdate");
        userTransactionInstance = pm.getTransaction();
        userTransactionInstance.begin();
        finder.executeUpdateNativeSQL(filterStr.toString());
        userTransactionInstance.commit();
    } catch (Throwable t) {
        if (userTransactionInstance != null) {
            try {
                userTransactionInstance.rollback();
            } catch (Exception notUsed) { }
        }
        finally {
            try {
                Utils.endUserEnvironment(userEnvironment);
            } catch (Exception e) {
                log.error("", e);
            }
            if (finder != null)
                finder.close();
        }
    }
}
```

2. Configure the `timer.properties` file so the custom timer listener class is called at regular intervals by defining the `firstTime`, `period`, and `listener` properties, as shown in Example 13–6. The `listener` property value is the fully qualified custom listener class.

**Example 13–6  timer.properties File**

```plaintext
# Time in milliseconds to invoke the timer after the server is started
ldRowLockExpiration.firstTime=600

# Time in milliseconds interval to invoked the time.
ldRowLockExpiration.period=600

# The timer class which has the implementation logic to purge the expired locks
ldRowLockExpiration.listener=com.package.entityrowlock.EntityNameRowLockExpiryListener
```

**Note:** In Example 13–6, `ldRowLockExpiration` can be any text, but the `firstTime`, `period`, and `listener` property names cannot be changed.

3. Configure the `system-config.properties` file to set the default values for the `defaultRowLockExpirationDuration` and `maxSupportedRowLocks` properties, as shown in Example 13–7.
Extending UIM Entities to Use Row Locking

Example 13–7  system.config.properties File

```
# The default row locks expiration duration in milliseconds for the entity.
# This value should be defined to be less than the transaction time out.
lockPolicy.defaultRowLockExpirationDuration=30000

# Default maximum number of entities to be row locked.
# This should be less than the default range defined in application.
lockPolicy.MaxSupportedRowLocks =100
```

Using Row Locking with Entity Finder APIs

You can write custom code to use row locking with entity finder APIs for any heavily-used entities in your UIM environment.

Understanding How UIM Uses Row Locking

Figure 13–1 shows the flow of an entity finder API call for all UIM entities except telephone numbers, which use row locking. When a UIM user initiates an entity search, an entity finder API is called, and the search results are returned to the client.

**Figure 13–1  Flow of Entity Finder API without Row Locking**

Figure 13–2 shows the flow of a telephone number entity finder API call, which uses row locking. When a UIM user initiates a telephone number search, the LockPolicy attributes are set before calling the entity finder API, and the locked search results are returned to the client.

**Figure 13–2  Flow of TelephoneNumber Entity Finder API with Row Locking**

Writing Custom Code to Use Row Locking

You can write custom code to use row locking with entity finder APIs for any heavily-used entities in your UIM environment. The custom code must set the LockPolicy attributes and call the entity finder API. This can be accomplished through:

- Custom Rulesets
- Custom Web Services
Custom Rule sets

In the following scenario, the custom code resides in an Inventory cartridge, within a custom Java class that is called by a custom ruleset.

Figure 13–3 shows the flow of an entity finder API call for rowlock-enabled entities that you use with row locking through custom code. After the customizations are in place, when a UIM user initiates an entity search, an entity finder API is called. However, before the entity finder API runs, the method is intercepted by the custom extension point, which is configured to run the custom ruleset instead of the entity finder API. The custom ruleset calls the custom code, which sets the LockPolicy attributes and calls the same entity finder API. Based on the LockPolicy attributes specified, the search results are locked and returned to the client.

**Figure 13–3  Flow of Entity Finder API Using a Custom Ruleset**

The following procedure provides detailed steps and example custom code that you can use to create the customized flow of an entity finder API call that uses row locking.

To use row locking with entity finder APIs:

1. Create a custom Java class.
Example 13–8 shows a custom Java class that finds equipment entities using row locking, but you can write similar logic for any entity. To accomplish this, the custom logic must:

- Create a LockPolicy object
- Set the LockPolicy attributes
- Set the entity-specific search criteria object with the LockPolicy object
- Call the appropriate entity finder API method, passing in the appropriate entity search criteria object that is populated with the LockPolicy
- Set the ruleset return value to the row-locked results from the entity finder API call

Example 13–8  Custom Java Class

```java
package oracle.communications.custom;

import java.util.*;
import oracle.communications.platform.persistence.PersistenceHelper;
import oracle.communications.inventory.api.entity.Equipment;
import oracle.communications.inventory.api.entity.EquipmentManager;
import oracle.communications.inventory.api.entity.EquipmentSearchCriteria;
import oracle.communications.inventory.api.framework.LockPolicy;
import oracle.communications.inventory.extensibility.extension.util.ExtensionPointRuleContext;

public class CustomEquipmentSearch
{
    public void main(EquipmentSearchCriteria criteria, ExtensionPointRuleContext context) throws Exception
    {
        LockPolicy lockPolicy = PersistenceHelper.makeLockPolicy();
        lockPolicy.setNumberOfResources(20);
        lockPolicy.setExpiration(5000);
        lockPolicy.setFilterExistingLocks(true);
        criteria.setLockPolicy(lockPolicy);

        EquipmentManager equipmMgr = PersistenceHelper.makeEquipmentManager();
        List<Equipment> equipObjs = equipmMgr.findEquipment(criteria);

        context.setReturnValue(equipObjs);
    }
}
```

2. Create a custom ruleset.

Example 13–9 shows a custom ruleset that calls the custom Java class shown in Example 13–8. You can write a similar custom ruleset to call any custom Java class.

Example 13–9  Custom Ruleset

```java
package oracle.communications.inventory.rules

import oracle.communications.custom.CustomEquipmentSearch;
import oracle.communications.inventory.api.entity.EquipmentSearchCriteria;
import oracle.communications.inventory.extensibility.extension.util.
```
ExtensionPointRuleContext;

rule "EquipmentSearch"
    salience 0
when
    criteria : EquipmentSearchCriteria()
    context : ExtensionPointRuleContext()
then
    CustomEquipmentSearch customClass = new CustomEquipmentSearch();
    customClass.main(criteria, context);
end

---

**Note:** The criteria local variable is based on the argument that the custom extension point defines, which is EquipmentSearchCriteria, as shown in Example 13–10. The context local variable of ExtensionPointRulesetContext is made available to all rulesets by the extensibility framework, which appends this argument to the list of arguments defined by the custom extension point. See "ExtensionPointContext and ExtensionPointRuleContext" for more information.

---

3. Create a custom extension point.

Example 13–10 shows the custom extension point method signature for the findEquipment API method. You can define a similar extension point signature for any entity finder API, such as findEquipmentHolder, findLogicalDevice, findPipe, and so forth.

**Example 13–10 Custom Extension Point Signature**

```java
public abstract interface java.lang.String
    oracle.communications.inventory.api.equipment.EquipmentManager.findEquipment(oracle.communications.inventory.api.entity.equipment.EquipmentSearchCriteria)
```

4. Create a custom ruleset extension point.

The custom ruleset extension point configures the custom ruleset to run at the custom extension point. In this scenario, the placement of the custom ruleset must be **Instead** of the method defined by the custom extension point. In this manner, the custom ruleset calls the entity finder API and returns the row-locked entities to UIM.

**Custom Web Services**

**Note:** This section builds upon the information presented in Chapter 10, "Developing Custom Web Services", and assumes you have an understanding of Web services and how to develop them.

---

In this scenario, the custom code resides in a custom Web service. Figure 13–4 shows the flow of an entity finder API call for rowlock-enabled entities that you use with row locking through custom code. After the customizations are in place, the Web service is initiated by an external system through a request. The Web service custom code sets the LockPolicy attributes and calls the entity finder API. Based on the LockPolicy attributes specified, the search results are locked and returned.
to the Web service. The Web service then sends the locked search results back to the external system through a response.

**Figure 13–4 Flow of Entity Finder API Using a Custom Web Service**

To use row locking with entity finder APIs through a custom Web service, the Web service must contain a Java class similar to the one shown in Figure 13–8.

### Using Row Locking Without Entity Finder APIs

You can also use row locking without using entity finder APIs. Example 13–11 shows a custom Java class that locks a Collection of entities. To accomplish this, the custom logic must:

- Create a Collection of entities
- Create a LockPolicy object
- Set the LockPolicy attributes
- Call the RowLockManager.lock method, passing in the entity rows to be locked and the LockPolicy used to lock them

**Example 13–11 Custom Java Class**

```java
package oracle.communications.custom;

import java.util.*;
import oracle.communications.platform.persistence.PersistenceHelper;
import oracle.communications.inventory.api.common.RowLockManager;
```
import oracle.communications.inventory.api.framework.LockPolicy;

public class CustomClass
{
    public void main() throws Exception
    {
        Collection myCollection = new Collection();

        // Populate myCollection with like entities, such as Equipment entities,
        // Pipe entities, LogicalDevice entities, etc.
        . . .

        // Create a LockPolicy and populate the attributes
        LockPolicy lockPolicy = PersistenceHelper.makeLockPolicy();
        lockPolicy.setNumberOfResources(20);
        lockPolicy.setExpiration(5000);
        lockPolicy.setFilterExistingLocks(true);

        // Call RowLockManager.lock to lock entities in myCollection
        RowLockManager rowLockMgr = PersistenceHelper.makeRowLockMgr();
        Collection myLockedCollection = rowLockMgr.lock(myCollection, lockPolicy);
    }
}

**Note:** You can use rulesets and extension points to run the custom code shown in Example 13–11. See “Using Row Locking with Entity Finder APIs” for an example.
This chapter provides information about the Oracle Communications Unified Inventory Management (UIM) cooperation framework, which enables UIM to work with other systems, such as Oracle Communications Internet Name and Address Management (INAM) and Oracle Communications MetaSolv Solution (MSS).

Understanding the cooperation framework is necessary when extending UIM to work with other systems through the use of the following reference cooperation cartridges:

- **ora_uim_ipaddress_cooperation**
  
  In the IP address cooperation, UIM works with INAM. For example, UIM manages services, and INAM manages IP addresses. When a service resource assignment requires an IP address, INAM is the IP address resource repository from which UIM finds a resource to assign.

- **ora_uim_vlanid_cooperation**
  
  In the VLAN ID cooperation, UIM works with MSS. For example, MSS manages service configurations, and UIM manages VLAN IDs. When a service resource assignment requires a VLAN ID, UIM is the VLAN ID resource repository from which MSS finds a resource to assign.

- **ora_uim_connectivity_cooperation**
  
  In the connectivity cooperation, UIM works generically with an external system to manage connectivity resources. For example, UIM sends a Work Order to an external system, such as MSS, to request a lease on connections.

See Appendix C, "Reference Cooperation Cartridges" for more information.

This chapter contains the following sections.

- **About Cooperation and Data Federation**
- **About Transaction-Based and Order-Based Cooperations**
- **About Externally Enabled Entities**

### About Cooperation and Data Federation

Cooperation is working together to achieve a common goal. In development terms, cooperation is different systems working together to achieve a common industry-specific goal, such as service fulfillment and connectivity design.

Data federation enables different systems to cooperate while presenting a common user experience. In a data federation arrangement, specific data access, data management tasks, and processes are transparently delegated to other systems. For example, UIM manages services, and INAM manages IP addresses, and these two
systems cooperate through the use of the ora_uim_ipaddress_cooperation cartridge. When this cooperation is in place, and you are working in UIM on a service such as assigning a resource that requires an IP address, it is transparent to you that UIM is cooperating with INAM to supply the IP address.

The different ways in which systems cooperate are called external arrangements. Table 14–1 lists all of the external arrangements used by the IP address, VLAN ID, and connectivity cooperations.

**Table 14–1 External Arrangements**

<table>
<thead>
<tr>
<th>External Arrangement</th>
<th>IP Address Cooperation</th>
<th>VLAN ID Cooperation</th>
<th>Connectivity Cooperation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEDERATED_VIEW</td>
<td>UIM views IP addresses from INAM.</td>
<td>UIM views network system and product catalog from MSS.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>LEASED_IN</td>
<td>UIM leases in IP address from INAM for service assignment.</td>
<td>Not applicable.</td>
<td>UIM leases in a connection from an external system for service trail enablement.</td>
</tr>
<tr>
<td>LEASED_OUT</td>
<td>Not applicable.</td>
<td>UIM leases out VLAN ID to MSS for service assignment.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>SHARED</td>
<td>Not applicable.</td>
<td>MSS shares service catalog and network system entities to relate to UIM VLAN domains.</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>

**About Transaction-Based and Order-Based Cooperations**

The cooperation framework supports four types of cooperative external arrangements: federated, leased in, leased out, and shared. See Table 14–1, ”External Arrangements” for more information. Each of these external arrangements are either:

- **Transaction-Based**
- **Order-Based**

**Transaction-Based**

Transaction-based cooperations are point-to-point integrations between UIM business logic and an external system. Transaction-based cooperations typically revolve around a simple resource, such as a telephone number or IP address.

For a transaction-based cooperation to work, the external system must support a synchronous API that UIM can call. The synchronous transaction has a beginning and an end through the use of the startTransaction method and the endTransaction method.

The IP address cooperation and the VLAN ID cooperation are transaction-based cooperations that use data federation.

**Order-Based**

Order-based cooperations are schema-based integrations between UIM and an external order management system, such as Oracle Communications Order and Service Management (OSM). Order-based cooperations involve order requests from
UIM to an external system that creates, designs, assigns, activates, and tests resources. The external system then provides a response back to UIM. Order-based cooperations typically revolve around a multi-phased design and delivery process, such as an OSM order flow. Within the multi-phased process, the external order management system may send requests to UIM to lease data, such as pipe-related data for connectivity.

For an order-based cooperation to work, UIM must support the ability to create an order request and send it asynchronously to an external system. Additionally, UIM must support the ability to listen for, and handle, the asynchronous order response from the external system.

The connectivity cooperation is an order-based cooperation that calls asynchronous APIs provided by external systems to lease connectivity resources. The connectivity cooperation does not use data federation.

---

**Note:** The following subsections build upon the information in Chapter 9, "Integrating UIM through Web Services", which describes a Service Order, and how the Service Order is saved as a business interaction attachment.

---

**Work Order**

A Service Order is a type of Business Interaction request from an external system for UIM to perform various actions on a Service entity. The actions can affect a service, service configuration, and the life cycles of supporting service configuration item resources. Similarly, a Work Order is a type of Business Interaction request from UIM to an external system to perform various actions on inventory entities in external systems, such as network resources, connections, devices, or services. The work order is used within the context of an order-based cooperation.

The schema used for external systems to communicate with UIM is consistent and extensible. For example, the schema:

- Provides a consistent way to organize and group the items and entities related to the order
- Supports actions with corresponding parameters or properties at the order, item, and entity level
- Defines one structure that is used for both requests and responses, regardless of which system is requesting or responding
- Defines the <parameter> element, which makes the schema readily extensible through custom parameter names and corresponding custom parameter values

**Business Interaction Attachment**

When a Service Order request is received by UIM from an external system, the XML is saved as business interaction attachment. Similarly, when a Work Order request is sent by UIM to an external system, the XML is saved as a business interaction attachment.

The BusinessInteractionAttachment entity defines the following attributes:

- name
  - The name attribute is used for identifying the entity in UIM.
- content
The content attribute supports any generic content for a request or response. The content attribute data type is a BLOB, so the entity attachments can contain formats other than XML requests and responses.

- **category**
  The category attribute in an enumeration that distinguishes the different attachment categories. The enumeration values are REQUEST and RESPONSE.

- **parentAttachment and childAttachments**
  The parentAttachment and childAttachments attributes make it possible to receive multiple responses per request, such as relating the request to its responses in a hierarchical relationship. As a result, the parentAttachment and childAttachments attributes create a parent-child relationship for the attachments. The childAttachments attribute is an ordered list.

---

**About Externally Enabled Entities**

This section builds upon the information presented in Chapter 4, "Extending the Data Model".

Externally enabled entities are entities that are designated as being part of a cooperation. To support cooperation in UIM, several entities are defined as externally enabled in the metadata through the use of the <externalEnabled> element. The externally enabled entities are:

- BusinessInteraction
- CustomNetworkAddress
- CustomObject
- Pipe
- Service

---

**External Identification**

To support a consistent way to identify an external entity in UIM, the entity identity as known by the external system must be maintained. The entity external identity may or may not have similar properties to UIM entity identity. Either way, the identities must be correlated for both systems to operate on the same intended entity in any interactions. In addition, the same entity may have other types of identity. For example, the NativeEMS domain presents another identity that is typically found for network-facing entities.

So, it is possible for an entity to have multiple identities, depending on the perspective used to refer to the entity. This perspective is known as the entity management domain. The entity management domain is the context in which the entity identity is commonly known and used, which is typically the owner of the entity identity.

It is also possible to have a one-to-many relationship from the entity to multiple identities. However, some of the more commonly used identities are defined as attributes on the main entity to improve performance and to support application logic. For example, the application logic that supports cooperative inventory is dependent on these identities.

Externally enabled entities have the following generated attributes:

- externalObjectId
The externalObjectId attribute provides a public unique identity for a business entity within the context of the domain specified by externalManagementDomain.

- **externalName**

  The externalName attribute provides a business-meaningful name of the business entity (identified by externalObjectId) within the context of the domain specified by externalManagementDomain.

- **externalManagementDomain**

  The externalManagementDomain attribute identifies an external system, domain name, party, or participant in a cooperation.

---

**Note:** externalManagementDomain is not the entity owner. Entity ownership can refer to technical or system ownership, such as MSS or INAM; and entity ownership can refer to business or ownership, such as AT&T or East Region. These two types of entity ownership are independent of each other, as is the type of entity ownership that refers to entity identification management. An ownership attribute is not supported.

---

- **externalArrangement**

  The externalArrangement attribute is an enumeration that identifies the cooperation model between UIM and the external party for the given entity. The valid enumerated values are:

  - **FEDERATED**
    
    Describes when the resource is temporarily retrieved from an external system into UIM views. For example, Network System, Product Catalog, and IP Address; before Network System, Product Catalog, and IP Address are shadowed into UIM.

  - **LEASED_IN**
    
    Describes when data is leased by UIM from an external system, such as an IP address or a connection.

  - **LEASED_OUT**
    
    Describes when data is leased by UIM to an external system, such as VLAN ID.

  - **SHARED**
    
    Describes when data is managed cooperatively between UIM and an external system. For example, Network System and Product Catalog data are shadowed into UIM. That is, the data is stored in both Network System and in UIM.
Note: In UIM, the availability of a leased resource, such as connectivity (Pipe) or VLAN ID (Custom Network Address), is based on the entity’s Inventory State attribute value of INSTALLED or UNAVAILABLE. The leasing terms for the resources, such as effective dates, are not managed using additional entity attributes. Rather, this is supported by cooperation with the external system. For example, UIM is responsible for initiating or terminating the lease of the resources, along with a corresponding update to the resource inventory state; updates to the entity start and end date are not necessary.
This appendix provides information on Oracle Communications Unified Inventory Management (UIM) sample cartridges, which you can use when extending UIM.

This appendix contains the following sections:

- About the Sample Cartridges
- Using the Sample Cartridges

### About the Sample Cartridges

The UIM sample cartridges reside in the `UIM_Home/cartridges/sample` directory and include the following cartridges:

- **ora_uim_servicetopology_sample**
  
  The topology is a graphical representation of the spatial relationships and connectivity among your related inventory entities. The `ora_uim_servicetopology_sample` cartridge contains characteristics, specifications, extension points, and rulesets that collectively provide a working example of extending the topology of a specific service.

- **ora_uim_pathanalysis_sample**
  
  The topology uses a specific set of entities, and a specific algorithm to determine the path between any two entities. This algorithm is called the path analysis. The `ora_uim_pathanalysis_sample` cartridge contains characteristics, specifications, extension points, and rulesets that collectively provide a working example of extending the path analysis of a specific service.

- **ora_uim_geocoder_sample**
  
  Geocoding is the process of associating geographic coordinates, such as latitude and longitude, with other geographic data, such as street addresses or postal codes. The `ora_uim_geocoder_sample` cartridge contains characteristics, specifications, extension points, and rulesets that collectively provide a working example for geocoding a US address.

### Using the Sample Cartridges

To use these sample cartridges, you can:

- Deploy the sample cartridge into a UIM test environment to analyze the data and give you ideas for implementing a customized version in a production environment.
- Import the sample technology pack into Oracle Communications Design Studio to analyze the characteristics, specifications, extension points, and rules to understand how to implement a customized version in a production environment.

The sample cartridges are dependent on the ora_uim_baseextpts cartridge. So, if you import a sample cartridge into Design Studio, you must first import the ora_uim_baseextpts cartridge; and if you deploy a sample cartridge into UIM, you must first deploy the ora_uim_baseextpts cartridge into UIM.

**Note:** Oracle recommends that you deploy all of the base cartridges into UIM.

For instructions on how to import a sample cartridge into Design Studio, see the Design Studio Help. When sample cartridges are imported into Design Studio, the project within the cartridge is compiled. If compiler errors are present, you must configure the project library list. See “Configuring the Project Library List” for more information.

For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge into UIM using the Cartridge Deployer Tool, see *UIM Cartridge and Technology Pack Guide*.

**Note:** To use the ora_uim_geocoder_sample cartridge, you must license Oracle Spatial 11g. In addition, you must have geocoding data in Oracle Spatial format, such as that purchased from a third-party vendor like NAVTEQ.
This appendix provides information on the Oracle Communications Unified Inventory Management (UIM) Reference Web service. The Reference Web service defines operations that enable an external system to create new services and service configurations in UIM, and to retrieve or update existing ones.

**Note:** The Reference Web service is deprecated and is being replaced by the UIM Service Fulfillment Web service.

The Reference Web service operations differ from the Service Fulfillment Web service operations described in Chapter 9, "Integrating UIM through Web Services" in that the operations do not function at the business interaction level. Rather, the Reference Web service operations function at the service and service configuration level. By using business interactions, an external system can manage groupings of services, manage relationships between services, and manage the resources created to fulfill the services. Without business interactions, managing groupings of services, managing relationships between services, and managing the resources created to fulfill the services is delegated to the solution designer.

The Reference Web service also differs from the Service Fulfillment Web service described in Chapter 9, "Integrating UIM through Web Services" in that the Reference Web service WAR file is not packaged in the `inventory.ear` file. When you deploy the `inventory.ear` file, the Reference Web service does not automatically deploy. Rather, you must manually import the provided `ReferenceUim.war` file into the `inventory.ear` file, `custom.ear` file, or a custom EAR file to deploy. Instructions on how to do this are provided.

This appendix contains the following sections:

- About the Reference Web Service Operations
- Understanding the Reference Web Service Operations Through Examples
- Using the Reference Web Service
About the Reference Web Service Operations

**Note:** Before you begin reading about the Reference Web service, it is important that you have an understanding of the following subjects described in *UIM Concepts*:

- Services, service configurations, and service configuration items
- Life cycle management

**Note:** Depending on your platform, ensure the `UIM_HOME/config/castor.properties` file is set as follows:

- For Linux/Solaris:
  ```
  org.exolab.castor.xml.serializer.factory=org.exolab.castor.xml.
  XercesJDK5XMLSerialzerFactory
  ```

- For AIX:
  ```
  org.exolab.castor.xml.serializer.factory=org.exolab.castor.xml.
  XercesXMLSerializerFactory
  ```

The Reference Web service defines the following operations:

- **CaptureServiceConfigurationInputs**
- **AutomateServiceConfiguration**
- **GetServiceConfiguration**
- **UpdateServiceConfiguration**
- **ChangeServiceConfigurationState**
- **CalculateConfigurationDifferences**
- **DeleteParty**

The Reference Web service is deprecated and Oracle recommends that you replace the use of the Reference Web service with the UIM Service Fulfillment Web service. Comparable operations between the two are listed in Table B–1.

### Table B–1 Comparability Between Reference Web Service and UIM Service Fulfillment Web Service Operations

<table>
<thead>
<tr>
<th>Reference Web Service Operation</th>
<th>UIM Service Fulfillment Web Service Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaptureServiceConfigurationInputs</td>
<td>CaptureInteraction</td>
</tr>
<tr>
<td>AutomateServiceConfiguration</td>
<td>ProcessInteraction</td>
</tr>
<tr>
<td>GetServiceConfiguration</td>
<td>GetConfiguration</td>
</tr>
<tr>
<td>UpdateServiceConfiguration</td>
<td>This operation was not replaced.</td>
</tr>
<tr>
<td>ChangeServiceConfigurationState</td>
<td>UpdateConfiguration</td>
</tr>
<tr>
<td>CalculateConfigurationDifferences</td>
<td>This operation was not replaced.</td>
</tr>
<tr>
<td>DeleteParty</td>
<td>This operation was not replaced.</td>
</tr>
</tbody>
</table>
All of the Reference Web service operation requests define the `<expandedResponse>` element, which is defined as a boolean in the Service.xsd schema file. Depending on the boolean value specified in the request, the level of information returned by the response can vary:

- false (default option)
  Returns the entity and configuration information.
- true
  Returns the entity, configuration, and any child configurations.

**CaptureServiceConfigurationInputs**

This operation captures the order information that is used by subsequent Web service calls. This operation may also perform any of the following actions, if required, based on the input XML payload:

- Create the service
- Create the service configuration version
- Create the party
- Create the place
- Associate the party or place, or both, to the service instance

The service is created only for an **Add** order, along with the service configuration version 1 (if not already created). For a **Change** or **Disconnect** order, only the new service configuration version is created (if not already created).

---

**Note:** Regarding the input XML when calling the CaptureServiceConfigurationInputs operation, the `<previousVersion>` element is not required by the XML to validate because it is not used by the operation.

---

**Note:** Regarding the input XML when calling the CaptureServiceConfigurationInputs operation, the `<version>` element is required by the XML to validate, but is not used when `<orderType>` is **new** and `<orderAction>` is **change**. That is because for change orders:

- When the latest service configuration version status is **completed**, a new service configuration version is created and the version number is incremented.
- When the latest service configuration version status is **In Progress**, **Designed**, or **Issued**, a new service configuration version is not created; the existing service configuration is updated.

---

During order processing on an external system, an amendment order may be sent to UIM using the CaptureServiceConfigurationInputs operation. In this scenario, the new XML content for the amendment order is saved with the service configuration version for subsequent processing, but a new service configuration version is not created.
Each service configuration version has zero or more XML payloads. If the service configuration version is created through UIM, there is no XML payload. Otherwise, there is at least one.

**AutomateServiceConfiguration**

This operation provides the ability to customize the service configuration auto-design and auto-redesign logic using the extensibility framework. (Auto-design logic supports the initial order submission, and auto-redesign logic supports order amendments.) For information on the extensibility framework, see Chapter 8, "Extending UIM through Rulesets".

This operation wraps the ServiceConfigurationManager.autoAllocateServiceConfig() method, and the UIM installation provides an enabled extension point for this method. The enabled extension point name is `InventoryConfigurationSpec_ServiceConfigurationManager_autoAllocateServiceConfig.rstp` and it is located in the `ora_uim_baseextpts.jar` file.

To automate service configurations particular to your business requirements, do the following, using Oracle Communications Design Studio:

- Write a ruleset that automatically designs and redesigns service configurations based on customer requirements. Additional information on this task is described below in "Using Technology Pack Rulesets" and "Developing a Ruleset".
- Create a ruleset extension point that associates the ruleset to the provided extension point. For instructions on how to define a ruleset extension point, see the Design Studio Help.
- Deploy the cartridge that contains the ruleset, extension point, and ruleset extension point into UIM. For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge into UIM using the Cartridge Deployer Tool, see UIM Cartridge and Technology Pack Guide.

**Using Technology Pack Rulesets**

Several of the UIM Technology Packs provide additional rulesets specific to the technology. Check the documentation on the various Technology Packs to determine if there is an existing ruleset that can be used in place of developing a new ruleset or used as a starting point for developing a new ruleset.

**Developing a Ruleset**

The following are some guidelines when developing a ruleset to automate the service configuration:

- The ruleset logic needs to know what to do with the XML payload based on the domain-specific business rules and models.
- The ruleset logic needs to handle the creation or deletion of any dependent resources.
- The ruleset logic needs to handle auto-design for new orders and auto-redesign for amendment orders.
- The ruleset logic should assume that the service configuration version is already created and has to manage the resources and characteristics. (For amendment orders, the logic needs to unassign and delete irrelevant characteristics).
GetServiceConfiguration

This operation retrieves the service configuration and its configuration items based on the input service ID and version number. It also retrieves the statuses and dates on the configuration.

UpdateServiceConfiguration

The UpdateServiceConfiguration operation updates the service configuration with assignments, characteristics, and child configuration items.

You can specify a particular configuration item to update by using the <path> element, as shown in Example B–1. The bolded line in the example shows the <path> element. The syntax for the text of the <path> element is explained following the example.

Example B–1  UpdateConfiguration Request

```xml
<ser:updateServiceConfigurationRequest>
  <com:requestContext>
    <com:messageConfirmation>ignore</com:messageConfirmation>
  </com:requestContext>
  <expandedResponse>true</expandedResponse>
  <orderData>
    <orderHeaderData>
      <orderId>1</orderId>
      <orderDate>2009-03-11T12:00:00-05:00</orderDate>
      <orderType>new</orderType>
    </orderHeaderData>
    <orderDetailData>
      <orderAction>add</orderAction>
      <party>
        <id>wsTest1_Party</id>
        <services xsi:type="ser:ServiceDataType">
          <id>wsTest1_service</id>
          <specification>
            <name>wsMPLSVPNService</name>
            <entityType>Service</entityType>
            <description>MPLSVPNService</description>
          </specification>
          <configurations>
            <version>1</version>
            <id>Se_wsTest1_1</id>
            <configSpec>
              <name>wsMPLSVPNServiceConfiguration</name>
              <entityType>Service Configuration</entityType>
              <description>MPLSVPNServiceConfiguration</description>
            </configSpec>
            <configurationItems>
              <!-- Path of the config item is used to uniquely identify a config item in a config item tree. Use index only if there are duplicate config item names. Index starts from 0. For example, level1/level2[1] indicates second occurrence of a config item with name level2 and child of config item with name level1. -->
              <path>level1/level2[1]</path>
            </configurationItems>
            <resourceAssignment xsi:type="ser:ResourceAssignmentDataType">
              <resource>
                <id>wsTest1_site</id>
                <type>GEOSITE</type>
                <specification>
```
The syntax for the text of the <path> element is configuration item name[index]. For example, here is a configuration with three configuration items:

- Configuration
  - Configuration item 1, name = TN
  - Configuration item 2, name = TN
  - Configuration item 3, name = TN

Indexes start with zero. So, the index for configuration item 1 is zero, the index for configuration item 2 is 1, and the index for configuration item 3 is 2. Using this configuration example, if you wanted to update configuration item 3, the <path> element would reflect TN[2].

Life Cycle Management
The update operation cannot be used to perform life cycle management operations on parties, places, services, and resources that support the service configuration. Before an entity can be utilized by the configuration, a life cycle management operation must be performed to bring the entity into existence before calling the UpdateServiceConfiguration operation. If an entity is no longer utilized by the configuration, and the life of the entity is bound to the service, then a life cycle management operation must be performed to end the life of the entity, before or after calling the UpdateServiceConfiguration operation.

ChangeServiceConfigurationState
This operation transitions the life-cycle state of a service configuration and its associated resource assignments through its defined transition states. This operation can also be used to cancel a service configuration.

CalculateConfigurationDifferences
This operation calculates the differences between the target configuration relative to another configuration, such as the current version implemented in the network. This operation is used to get the changes in resource assignments and characteristic values to send to field work for activation and deactivation.
DeleteParty

This operation deletes one or more existing Party entities based on the input party IDs.

This operation is included in the Web service to enable you to test external system integration with UIM by calling a simple Web service operation that does not require a lot of set up in UIM prior to calling the operation. To test the integration, create a Party entity using the UIM application interface, and then delete that Party entity using the DeleteParty operation.

Understanding the Reference Web Service Operations Through Examples

This section provides examples involving the UIM Reference Web service operations. To understand the examples, you must first understand the terminology used regarding the possible types of orders:

- New order
  
  A new order is any order that is not an amendment order. New orders handle adding a new service, changing an existing service, or disconnecting an existing service. A new order that changes an existing service may involve adding, updating, or disconnecting service items on an existing service.

- Amendment order
  
  An amendment order is an order that amends a previously created new order prior to the new order being completed. An amendment order may involve adding to, updating, or deleting from the order content.

**Figure B–1  Example of Web Service Operation Calls**

Several of the following examples refer back to Figure B–1.

**New Order/Add Example**

Figure B–1 shows an example of a new order/add, with the exception of the UpdateServiceConfiguration description: For a new order/add, the action and action parameters reflect only:
Assign resource
Add characteristics

Unassigning resources and removing characteristics are not applicable for a new order/add.

Another example of a new order/add is to call just the CaptureServiceConfigurationInputs and AutomateServiceConfiguration operations. Still another example of a new order/add is the flow shown in Figure B–1, but with multiple calls to UpdateServiceConfiguration. Examples may also involve the UI, such as a Web service creating the service and a user performing other tasks on that service through the UI.

**New Order/Change Example**

Figure B–1 shows an example of a new order/change, with the exception of the CaptureServiceConfigurationInputs description: For a new order/change, the description reflects only:

- Create service configuration 2
- Save XML payload

For a new order/change, the service already exists, so it is not created again. To apply the change, a new version of the service configuration is created. The UpdateServiceConfiguration action and action parameters may reflect several different scenarios: Changing an existing characteristic (which equates to the removal of the existing characteristic and adding a new one), assigning another resource, removing an existing characteristic, and so forth.

Another example of a new order/change is to call just the CaptureServiceConfigurationInputs, AutomateServiceConfiguration, and ChangeServiceConfigurationState operations. Examples may also involve the UI, such as a Web service creating the service and a user performing other tasks on that service through the UI.

**Amendment Order Example**

An example of an amendment order is to call just the CaptureServiceConfigurationInputs operation to save the XML payload. For an amendment order, the service being amended is not yet in service. A new version of the service configuration is not created for an amendment order.

Another example of an amendment order is shown in Figure B–1, where the UpdateServiceConfiguration action and action parameters may reflect several different scenarios: Changing a previously submitted characteristic (which equates to the removal of the submitted characteristic and adding a new one), assigning another resource, removing a previously submitted characteristic, and so forth.

**New Order/Disconnect Example**

Figure B–1 shows an example of a new order/disconnect, with the exception of the CaptureServiceConfigurationInputs description: For a new order/disconnect, the description reflects only:

- Create service configuration 3
- Save XML payload

To apply the disconnect, a new version of the service configuration is created.
Another exception is the UpdateServiceConfiguration description. For a new order/disconnect, the action and action parameters reflect only:

- Unassign resource
- Remove characteristics

Assigning resources and adding characteristics are not applicable for a new order/disconnect.

Another example of a new order/disconnect is to call just the CaptureServiceConfigurationInputs and ChangeServiceConfigurationState operations. Examples may also involve the UI, such as a user disconnecting the service through the UI and a ruleset calling the ChangeServiceConfigurationState operation.

### Using the Reference Web Service

When using the Reference Web service, see the following sections:

- Locating the WSDL and Schema Files
- Deploying the Reference Web Service
- Testing the Reference Web Service
- Securing the Reference Web Service

### Locating the WSDL and Schema Files

The Reference Web service operations are defined by the `ReferenceUim.wsdl` file, and supported by several schema files. The WSDL file and supporting schema files are located in the `UIM_Home/webservices/reference_webservice.zip` file.

### About the WSDL File

Within the `reference_webservice.zip` file, the WSDL file is located in the `wsdl` directory. When you locate the WSDL file, you can see that it defines the Reference Web service operations: CaptureServiceConfigurationInputs, AutomateServiceConfiguration, GetServiceConfiguration, UpdateServiceConfiguration, ChangeServiceConfigurationState, CalculateConfigurationDifferences, and DeleteParty. You can also see that each Web service operation defines a request, a response, and the possible faults that can be thrown. For example, the WSDL file defines the following for the CaptureServiceConfigurationInputs operation:

- `captureServiceConfigurationInputsRequest`
- `captureServiceConfigurationInputsResponse`
- `captureServiceConfigurationInputsFault`

The request, response, and faults each define an XML structure that is defined in the supporting schema files. The following excerpts are provided to show you how an XML structure defined in the WSDL correlates to the supporting schema files.

For example, the WSDL file defines:

```xml
<wSDL:message name="CaptureServiceConfigurationInputsRequest">
    <wSDL:part name="captureServiceConfigurationInputsRequest">
        <wSDL:element name="invsvc:captureServiceConfigurationInputsRequest"/>
    </wSDL:part>
</wSDL:message>
```
The above excerpt references the invsvc namespace (bolded in the example), which the WSDL file also defines at the top of the file:

xmlns:invsvc="http://xmlns.oracle.com/communications/inventory/webservice/service"

This tells you that the captureServiceConfigurationInputsRequest XML structure is defined in the schema file that defines the specified namespace as its target namespace. A quick search for the namespace reveals that the Service.xsd file defines the referenced namespace as its target namespace:

targetNamespace="http://xmlns.oracle.com/communications/inventory/webservice/service"

After you determine which schema file defines the XML structure that the WSDL file references, you should be able to readily traverse among the schema files to determine child XML structures and elements.

About the Schema Files
There are several schema files that support the Reference Web service operations. These schemas are categorized as reference schemas and Web service schemas, as described in the following sections.

Reference Schemas
The reference schemas define common elements used by all of the UIM Web services, not just by the Reference Web service. So, the elements are defined in the framework and then referenced.

The reference schemas are:

- InventoryCommon.xsd
- InventoryFaults.xsd
- FaultRoot.xsd

The reference schemas are contained in the uim-webservices-framework.jar, and you can readily copy them into your workspace using the get-framework-files Ant target defined in the build.xml file. The build.xml file is contained in the reference_webservice.zip file. See "build.xml File" in Chapter 10, Developing Custom Web Services.

Note: The reference schemas use the Inventory.xsdconfig file for XML namespace to Java package mapping.

Web Service Schemas
Within the reference_webservice.zip file, the Web service schemas are located in the wsdl/schemas directory. The Web service schemas define elements specific to the Web service, such as the request structures, the response structures, and any fault structures.

The Web service schemas are:

- Characteristic.xsd
- Party.xsd
- Place.xsd
Deploying the Reference Web Service

The Reference Web service is are part of the UIM installation, in the form of a ZIP file, which contains the `ReferenceUim.war` file. To deploy the Reference Web service, you must manually import the `ReferenceUim.war` file into the `inventory.ear` file, `custom.ear` file, or a custom EAR file. Regardless of which EAR file you choose to use, each EAR file contains an `application.xml` file that must be updated to include the `ReferenceUim.war` file. Afterwards, when you deploy the appropriate EAR file, the Reference Web service deploys and is ready to use.

The work required to run the Reference Web service is a subset of the work required to develop and run custom Web services. See "Extracting and Updating the application.xml File" and "Importing the WAR File into the EAR File" in Chapter 10, Developing Custom Web Services.

Verifying the Deployment

After you have deployed the updated EAR file, you can verify that the deployment includes the Reference Web service by viewing the Web services in the Oracle WebLogic Server Administration Console. See "Verifying the Deployment" in Chapter 9, Integrating UIM through Web Services for more information.

Testing the Reference Web Service

Web services can be tested by using any software designed to test Web services, such as SoapUI for testing through HTTP, or HermesJMS for testing through JMS. Testing is done after the EAR file is deployed.

Test Input XML

The `reference_webservice.zip` file provides numerous test input XML files used to test the Reference Web service operations. These files are located in the `reference_webservice.zip` directory.

The test input XML files used with HTTP testing tools have a suffix of `_http.xml`, and the test input XML files used with JMS testing tools have a suffix of `_jms.xml`. Two test files are provided for each Reference Web service operation. For example, `automateServiceConfiguration_http.xml` and `automateServiceConfiguration_jms.xml`.

Generating Test Input XML

You may also generate your own test input XML by using any software that generates XML based on schema, such as XML Spy, SoapUI, and so forth.
Preconfiguration for Testing
Prior to running the Reference Web service, UIM requires some preconfiguration which is provided for you in the JAR files located in the reference_webservice.zip/test/testCartridge directory:

- The wsSampleCartSource.jar file contains the Design Studio source preconfigurations that can be imported into Design Studio for viewing or modifying.
- The wsSampleCart.jar file contains the compiled preconfigurations that can be deployed directly into UIM. For instructions on how to deploy a cartridge into UIM from Design Studio, see the Design Studio Help. For instructions on how to deploy a cartridge into UIM using the Cartridge Deployer Tool, see UIM Cartridge and Technology Pack Guide.

The following sections briefly describe the preconfiguration required for each Reference Web service operation. See the readme.txt file for a description of the provided preconfigurations.

DeleteParty A party must exist in the database before it can be deleted by the DeleteParty operation.

CaptureServiceConfigurationInputs The service specification upon which CaptureServiceConfigurationInputs is based must exist in the database before the operation runs. The service specification upon which the CaptureServiceConfigurationInputs operation is based can be imported into UIM through the MPLS VPN cartridge, or you can create your own specification in Design Studio and import your created cartridge into UIM.

AutomateServiceConfiguration This operation performs service automation using the extension point provided in the ora_uim_baseextpts cartridge. A service ID is required, and either a serviceConfigurationId or serviceConfigurationVersionId is required to process the AutomateServiceConfiguration operation. The caller must set expandedResponse to true to get the complete service tree as a response.

GetServiceConfiguration The service configuration must exist in the database before it can be retrieved by the GetServiceConfiguration operation.

UpdateServiceConfiguration This operation updates a service configuration and service characteristics. A service ID is a required, and either a serviceConfigurationId or serviceConfigurationVersionId is required to process the UpdateServiceConfiguration operation. The caller must use ServiceConfigurationItemDataType instead of ServiceConfigurationItemType, ResourceAssignmentDataType instead of ResourceAssignmentType, and CharacteristicDataType instead of CharacteristicType using xsi:type in the request. The caller must set expandedResponse to true to get the complete service tree as a response. If the request contains more than one ServiceType to be updated, only the first ServiceType in the array will be updated.

CalculateConfigurationDifferences The two service configuration versions must exist in the database for an existing service to produce the difference between the service configuration versions.

ChangeServiceConfigurationState The service configuration must exist in the database before the service configuration state can be transitioned by the ChangeServiceConfigurationState operation.
Securing the Reference Web Service

Oracle recommends that you set up security after you have successfully tested the Reference Web service.

The Reference Web service does not include security requirements to authenticate the user. Therefore, username and password information is not required in the SOAP header for any Reference Web service call. Example B–2 shows an unsecured Reference Web service request, and Example B–3 shows the successful response.

Example B–2  Unsecured Reference Web Service Request

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <par:deleteParty>
      <partyKey>
        <partyId>jdoe</partyId>
      </partyKey>
    </par:deleteParty>
  </soapenv:Body>
</soapenv:Envelope>
```

Example B–3  Unsecured Reference Web Service Response (Successful)

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
      <com:requestContext xsi:nil="true" xmlns:com="http://xmlns.oracle.communications.com/communications/inventory/webservice/common">
        <partyKey>
          <partyId>jdoe</partyId>
        </partyKey>
      </com:requestContext>
    </par:deletePartyResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

When you use the Reference Web service, it is up to you secure it. How you secure the Web service depends on the EAR file into which you imported the Reference Web service WAR file. For example, if the Reference Web service deploys with the custom.ear file, you need to create your own deployment plan; if the Reference Web service deploys with the inventory.ear file, you need to modify the inventory.ear deployment plan that is part of the UIM installation (UIM_Home/app/plan/Plan.xml file).

To secure a custom Web service:

1. Access security for the custom Web service.
   See "Accessing Security" in Chapter 9, Integrating UIM through Web Services.

2. Associate a security policy to the custom Web service.
   See "Associating a Policy File" in Chapter 9, Integrating UIM through Web Services. You can use the security policy that comes with the UIM installation (Auth.xml), or the security policy that comes in the Reference Web service ZIP file (SampleAuth.xml), or create your own security policy file.
When you associate a security policy to the custom Web service, a deployment plan is generated in the form of a Plan.xml file.

3. Associate the generated deployment plan with the Reference Web service by redeploying the EAR file that contains the Reference Web service; the redeploy prompts you to supply the path to the EAR file, and to supply the name of the deployment plan (Plan.xml).

   a. The prompt to supply the name of the deployment plan may also prompt you to select Inbound or Both: Select Inbound.

4. Ensure that the deployment plan reflects Inbound. See "Modifying the Deployment Plan" in Chapter 9, Integrating UIM through Web Services.

After securing the Reference Web service, send another Web service request, still without a security header, and it now fails because security validations are in place. Example B–4 shows the unsecured Web service request, and Example B–5 shows the failed response stating the error, "No security header in message but required by policy."

**Example B–4  Unsecured Web Service Request**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
<soapenv:Header/>
<soapenv:Body>
  <par:deleteParty>
    <partyKey>
      <partyId>jdoe</partyId>
    </partyKey>
  </par:deleteParty>
</soapenv:Body>
</soapenv:Envelope>
```

**Example B–5  Secured Web Service Response (Failed)**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
<soapenv:Header/>
<soapenv:Body>
  <soapenv:Fault>
    <faultcode>soapenv:Server</faultcode>
    <faultstring>No security header in message but required by policy.
```

Note: The Auth.xml file is automatically available for selection to associate with the Reference Web service. Oracle recommends that you use the provided Auth.xml file secure the Reference Web service.

If you are using a security policy other than the Auth.xml file, there is an additional step required to get the security policy file to be available for selection to associate to the Reference Web service: The security policy file must reside in the Reference Web service WAR file. Then, when you deploy the EAR file that contains the Reference Web service WAR file, the security policy in the WAR file becomes available for selection to associate the Reference Web service.

To get the security policy into the WAR file, you must place the security policy in the policies directory of the project that creates the WAR file, and then you must recreate the WAR file. See Chapter 10, "Developing Custom Web Services".
Add the security header to the request, as shown in Example B–6.

**Example B–6  Secured Web Service Request**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken wsu:Id="UsernameToken-3737059" xmlns:wsu="http://docs.oasis-open.org/wss/200401-wss-wssecurity-utility-1.0.xsd">
        <wsse:Username>uimuser</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">Welcome@123</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <par:deleteParty>
      <partyKey>
        <partyId>jdoe</partyId>
      </partyKey>
    </par:deleteParty>
  </soapenv:Body>
</soapenv:Envelope>
```

Retest the Web service with the security header in place within the request and it now passes security validations, as shown in Example B–7.

**Example B–7  Secured Web Service Response (Successful)**

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <par:deletePartyResponse xmlns:par="http:xmlns.oracle.communications.com/communications/inventory/webservice/party">
      <com:requestContext xsi:nil="true" xmlns:com="http:xmlns.oracle.communications.com/communications/inventory/webservice/common">
        <partyKey>
          <partyId>jdoe</partyId>
        </partyKey>
      </com:requestContext>
    </par:deletePartyResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

See "Securing the Web Service" in Chapter 9, *Integrating UIM through Web Services* for more information.
This chapter provides information about using the Oracle Communications Unified Inventory Management (UIM) cooperation cartridges, which enable UIM to work cohesively with other systems, such as Oracle Communications Internet Name and Address Management (INAM) and Oracle Communications MetaSolv Solution (MSS), in handling IP addresses, VLAN IDs, and connectivity.

You can download the cooperation cartridges from the following Web site:

http://www.oracle.com/technetwork/indexes/samplecode/uim-samples-1878903.html

Detailed information about each cooperation cartridge is covered in the following technical white papers, located within each cartridge:

- UIM_Cooperation_Technical_Spec_IP_Address.docx
- UIM_Cooperation_Technical_Spec_VLAN_ID.docx
- UIM_Cooperation_Technical_Spec_Connectivity.docx

Additionally, the connectivity cooperation cartridge contains the following sequence diagrams:

- UIM_Cooperation_Technical_Spec_ConnectivitySeqDiagCreateLease.pdf
- UIM_Cooperation_Technical_Spec_ConnectivitySeqDiagDisconnect.pdf

This chapter contains the following sections:

- About the IP Address Cooperation
- About the VLAN ID Cooperation
- About the Connectivity Cooperation
- Using the Reference Cooperation Cartridges
- Cooperative Solution Considerations

About the IP Address Cooperation

In the IP address cooperation, UIM works with INAM. For example, UIM manages services, and INAM manages IP addresses. When a service resource assignment
About the VLAN ID Cooperation

In the VLAN ID cooperation, UIM works with MSS. For example, MSS manages service configurations, and UIM manages VLAN IDs. When a service resource assignment requires a VLAN ID, UIM is the VLAN ID resource repository from which MSS finds a resource to assign.

The VLAN ID cooperation is delivered in the UIM_Home/cartridges/sample/cooperation/ora_uim_vlanid_cooperation.zip file.

About the Connectivity Cooperation

In the connectivity cooperation, UIM works generically with an external system to manage connectivity resources. For example, UIM sends a Work Order to an external system, such as MSS, to request a lease on connections.

The connectivity cooperation is delivered in the UIM_Home/cartridges/sample/cooperation/ora_uim_connectivity_cooperation.zip file.

Using the Reference Cooperation Cartridges

The IP Address and VLAN ID cooperation cartridges are fully functional and can be deployed with minimal customizations: You must import the cartridge into Oracle Communications Design Studio and:

- Update the properties files to reflect your environment
- Update CooperationConstants to reflect your environment
- Rebuild the cartridge to include your updates

The connectivity cooperation cartridge is also functional; however, in addition to minimal customizations listed above, you must also:

- Configure your UIM environment to specify the cooperating external system

Creating New or Extending Existing Cooperation Cartridges

You can also use a cooperation cartridge as an example to follow when creating a new cooperation cartridge, or you can modify an existing cooperation cartridge to suit your requirements. Whether you are creating a new cartridge or modifying an existing cooperation cartridge, you must avoid deploying cartridges that contain same-named rulesets, extension points, or ruleset extension points. Doing so results in the last cartridge deployed overriding any same-named cartridge content that was previously deployed. See "Avoiding Cooperation Conflicts".

**Note:** Creating new or extending existing cooperation cartridges is done in the same manner as creating new or extending existing technology packs. See UIM Cartridge and Technology Pack Guide for more information.
Cooperative Solution Considerations

When planning a cooperation solution, you need to consider the following actions.

Determining the Cooperative Solution

When planning a cooperative solution, one of the first decisions you need to make is determining which cooperative solution best suits your needs: transaction-based or order-based.

- Transaction-based solution
  
  The IP Address and VLAN ID cooperation cartridges provide examples of transaction-based cooperative solutions. These solutions use the Custom Object and Custom Network Address entities, which have a generic nature that can model virtually any resource from a foreign system. They are simple and avoid complex requests in UIM.

- Order-based solution
  
  The Connectivity cooperation cartridge provides an example of an order-based cooperative solution. This solution uses the Connectivity (Pipe), Business Interaction, and Service entities. A complex resource such as a circuit can be federated, but UIM does not have a native connectivity understanding of the resource; so, in this type of scenario, it is better to relay the work to the external system through a work order.

All of the entities that are used in the provided transaction-based and order-based cooperative solutions are defined in the metadata as externally-enabled entities. When planning a cooperative solution, you can use any these entities in your solution, or you can extend the data model by defining any entity to be externally enabled. See Chapter 4, "Extending the Data Model" for more information.

See "About Transaction-Based and Order-Based Cooperations" for more information. See also the cooperation technical white papers, which provide detailed information about these solution approaches, including examples of good approaches, as well as approaches to avoid.

Avoiding Cooperation Conflicts

Oracle recommends that you deploy only one version of each of the cooperation cartridges into any given UIM environment; all three of the cooperation cartridges can be deployed into one UIM environment, but not multiple versions of the same cooperation cartridge. For example, if you plan to extend a cooperation cartridge, deploy only the extended cartridge; do not deploy the original cartridge. Deploying multiple versions of the same cartridge can cause conflicts with the rulesets and extension points contained in the cartridges.

Oracle also recommends that when extending a cooperation cartridge, all modifications be in the form of additions or changes to existing code in a ruleset, or to existing Java code that a ruleset calls. Any form of deletions, including deleting logic from the ruleset or Java code, and deleting any artifacts from any of the cartridge directories, can cause the cooperation to not work.

Lastly, Oracle recommends that you not rename the project. While it is possible to rename the project by renaming the .project file, and the .inventoryCartridge file, all of the cartridge content still remains the same, and if you deploy both cartridges into UIM, it equates to deploying multiple versions of the same cartridge; in this case, the cartridges just happen to have different names. If you must rename the project, you
can, as long as you only deploy the renamed cartridge, and not both the original cartridge and the renamed cartridge.

Creating New Specifications

If you are extending a cooperation cartridge by creating a new specification, Oracle recommends that you follow the specification naming convention used in the cooperation cartridges. The naming convention dictates that specification names end with (E) to designate the specifications that represent external objects. For example, if you intend to name your new specification:

- INAMIPAdress, instead name it INAMIPAddress(E)
- VLANIDForM6, instead name it VLANIDForM6(E)
- M6NetworkSystem, instead name it M6NetworkSystem(E)

While this naming convention is not required, it makes it possible for end users to immediately recognize which objects are external.

Managing External Identifiers

Your cooperation solution must manage external identifiers (IDs). The IDs in UIM, and the IDs in the cooperating external system, must be evaluated and included in the cooperation solution planning process. During the planning process, consider the following regarding managing external IDs:

- If the external system is to be represented in UIM as a Custom Object, the deployed cooperation cartridge logic must maintain the UIM ID, ensuring the uniqueness across all Custom Objects. The same principle holds true for all the external enabled entities.

- The external system has its own ID for the object. This ID can be used to set the UIM ID for the object. For example, the VLAN ID cooperation, which cooperates with MSS, sets UIM IDs to system-component-externalSystemID, where system is MSS, component is an MSS component, and externalSystemID is the MSS native ID for the object. This results in UIM IDs such as MSS-NS-1234 or MSS-PC-1234.

  Setting UIM IDs using this type of pattern ensures Custom Objects are unique in UIM. Similarly, the IP Address cooperation, which cooperates with INAM, sets the UIM ID to include the unique IP address from INAM. This ensures that the Custom Network Addresses are unique in UIM.

- The UIM externalObjectId attribute stores the cooperating external system’s unique ID for an object. The externalObjectId value must be set for UIM logic, or any rule logic, to correctly determine which objects are external, and which are native, to UIM.

Creating Externally Enabled Entities in UIM

When a deployed cooperation cartridge creates an externally enabled entity in UIM, the logic that creates the entity must also set the entity attributes. This includes the externally-enabled entity attributes of:

- externalObjectId
- externalName (optional)
- externalArrangement
- externalManagementDomain
The methods to set these attributes are defined on the entity. For example,
`EntityName.setExternalObjectId()`, where `EntityName` is any externally-enabled entity such as CustomObject, BusinessInteraction, Service, and so forth. See "External Identification" for more information about these attributes.

In addition to setting the attributes, the cartridge logic must declare the entity as external by calling the `setExternal(true)` method. The `setExternal()` method then calls the `setTemporaryEntityId()` method to generate a temporary ID for the UIM internal entity ID. Whenever you persist the external entity in UIM, you must first call the `unsetTemporaryEntityId()` method to remove the temporary ID for the UIM internal entity ID. You can then safely persist the entity in UIM. These methods are also defined on the entity. For example, `EntityName.setExternal()`, where `EntityName` is any externally-enabled entity such as CustomObject, BusinessInteraction, Service, and so forth.

Accessing a New External System

Accessing a new external system may be part of a new cooperation solution. For example, your cooperation solution may require UIM to have a federated view of system-X. In this scenario, UIM must be able to query system-X and include object references to system-X.

When accessing system-X:

- Determine which items to retrieve from system-X. In this example, component-X has been the determined item to retrieve from system-X.
- Determine how component-X is to be represented in UIM. For example, component-X can be represented by any externally-enabled entity such as CustomObject, CustomNetworkAddress, and so forth.
- Determine the mechanism that is to retrieve the information from the system-X. For example, API calls, direct database queries, and so forth.
- Determine the specification names for these objects.
- Create the necessary Custom Object, Custom Network Address, Pipe, Business Interaction, and Service specifications in Design Studio.
- Create a file that contains any constants you may need for accessing a new external system. For example, see the `CooperationConstants.java` source file. This file is located in each of the cooperation cartridges, in the `src/oracle/communications/inventory/techpack/cooperation/cooperation/common` directory, where `cooperation` is either `ipaddress`, `vlanid`, or `connectivity`, depending on the cooperation cartridge.

In the constants file:

- Add new constants for any new specification names.
- Add new constants for the interface to system-X. For example, host, port, user ID, and password.
- Add new constants for the external system and the component. For example, system-X and component-X.
- Create a file that contains a list of all externally-enabled entities, and that includes mapping information of each entity to the external system and component. For example, see the `ExternalEntitiesRegistry.java` source file. This file is located in each of the cooperation cartridges, in the `src/oracle/communications/inventory/techpack/cooperation/cooperation/common` directory.
mon directory, where cooperation is either ipaddress, vlanid, or connectivity, depending on the cooperation cartridge.

- Create a new Java manager interface that contains the method signatures that support the desired behavior.
- Create a new Java implementation class that contains the logic that supports the desired behavior. Add any new error messages to the cooperationCooperation.properties file, where cooperation is ipaddress, vlanid, or connectivity.
- Modify the ruleset DRL files to call the new Java Manager and its methods. (Logic can be added directly in the DRL file, or in a new Java class that the ruleset DRL file calls. Keep in mind that you cannot debug a DLR file. See Chapter 8, "Extending UIM through Rulesets" for more information.)

Creating Custom Web Services

You can extend a cooperation cartridge by adding custom code in a ruleset, or adding custom Java code that a ruleset calls. The custom logic can call a custom Web service, making custom Web services part of a new cooperation solution. For example, the VLAN ID cooperation that enables cooperation between UIM and MSS required that MSS update the status of UIM objects. This was accomplished by creating a Web service that MSS calls, and the Web service is part of the delivered VLAN ID cooperation solution.

See Chapter 10, "Developing Custom Web Services" for information on creating custom Web services.

You can also extend a cooperation cartridge by using JMS queues to invoke external web services, as demonstrated by the connectivity cooperation solution.