Cisco UCS Director Open Automation Cookbook, Release 6.6

First Published: 2018-05-04

Last Modified: 2018-07-10

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

Preface

Audience

This guide is intended primarily for data center administrators who use Cisco UCS Director and who have responsibilities and expertise in one or more of the following:

- Server administration
- Storage administration
- Network administration
- Network security
- Virtualization and virtual machines

Conventions

Text Type	Indication
GUI elements	GUI elements such as tab titles, area names, and field labels appear in **this font**. Main titles such as window, dialog box, and wizard titles appear in **this font**.
Document titles	Document titles appear in *this font*.
TUI elements	In a Text-based User Interface, text the system displays appears in `this font`.
System output	Terminal sessions and information that the system displays appear in `this font`.

Text Type	Indication
CLI commands	CLI command keywords appear in **this font**. Variables in a CLI command appear in *this font*.
[ ]	Elements in square brackets are optional.
{x | y | z}	Required alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.
< >	Nonprinting characters such as passwords are in angle brackets.
[ ]	Default responses to system prompts are in square brackets.
!, #	An exclamation point (!) or a pound sign (#) at the beginning of a line of code indicates a comment line.

Note Means *reader take note*. Notes contain helpful suggestions or references to material not covered in the document.

Caution Means *reader be careful*. In this situation, you might perform an action that could result in equipment damage or loss of data.

Tip Means *the following information will help you solve a problem*. The tips information might not be troubleshooting or even an action, but could be useful information, similar to a Timesaver.

Timesaver Means *the described action saves time*. You can save time by performing the action described in the paragraph.

Warning IMPORTANT SAFETY INSTRUCTIONS

This warning symbol means danger. You are in a situation that could cause bodily injury. Before you work on any equipment, be aware of the hazards involved with electrical circuitry and be familiar with standard practices for preventing accidents. Use the statement number provided at the end of each warning to locate its translation in the translated safety warnings that accompanied this device.

SAVE THESE INSTRUCTIONS

Related Documentation

Cisco UCS Director Documentation Roadmap

For a complete list of Cisco UCS Director documentation, see the *Cisco UCS Director Documentation Roadmap* available at the following URL: http://www.cisco.com/en/US/docs/unified_computing/ucs/ucs-director/doc-roadmap/b_UCSDirectorDocRoadmap.html.

Cisco UCS Documentation Roadmaps

For a complete list of all B-Series documentation, see the *Cisco UCS B-Series Servers Documentation Roadmap* available at the following URL: http://www.cisco.com/go/unifiedcomputing/b-series-doc.

For a complete list of all C-Series documentation, see the *Cisco UCS C-Series Servers Documentation Roadmap* available at the following URL: http://www.cisco.com/go/unifiedcomputing/c-series-doc.

Note The *Cisco UCS B-Series Servers Documentation Roadmap* includes links to documentation for Cisco UCS Manager and Cisco UCS Central. The *Cisco UCS C-Series Servers Documentation Roadmap* includes links to documentation for Cisco Integrated Management Controller.

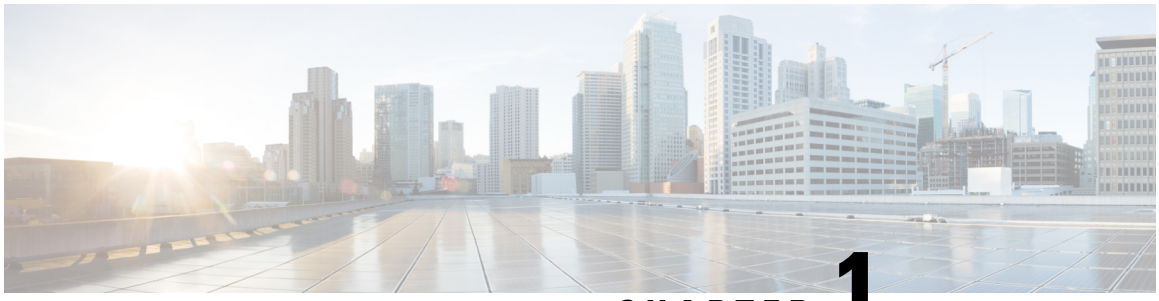
Documentation Feedback

To provide technical feedback on this document, or to report an error or omission, please send your comments to ucs-director-docfeedback@cisco.com. We appreciate your feedback.

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, using the Cisco Bug Search Tool (BST), submitting a service request, and gathering additional information, see What's New in Cisco Product Documentation.

To receive new and revised Cisco technical content directly to your desktop, you can subscribe to the . RSS feeds are a free service.



CHAPTER 1

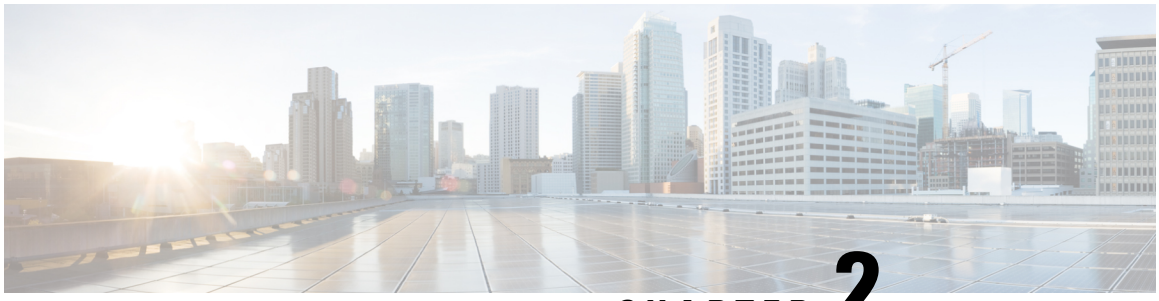
New and Changed Information for this Release

New and Changed Information

The following table provides an overview of the significant changes to this guide for the current release. The table does not provide an exhaustive list of all changes, or of all new features in this release.

Table 1: New Features and Changed Behavior in Cisco UCS Director, Release 6.6

Feature	What's New	Where Documented
Report context ID	User has to set the report context value to the `<<Pod Name>>@<<Ip Address>>`.	Converged Stack Builder, on page 18



CHAPTER 2

Managing Modules

This chapter contains the following sections:

Modules

A module is the top-most logical entry point into Cisco UCS Director.

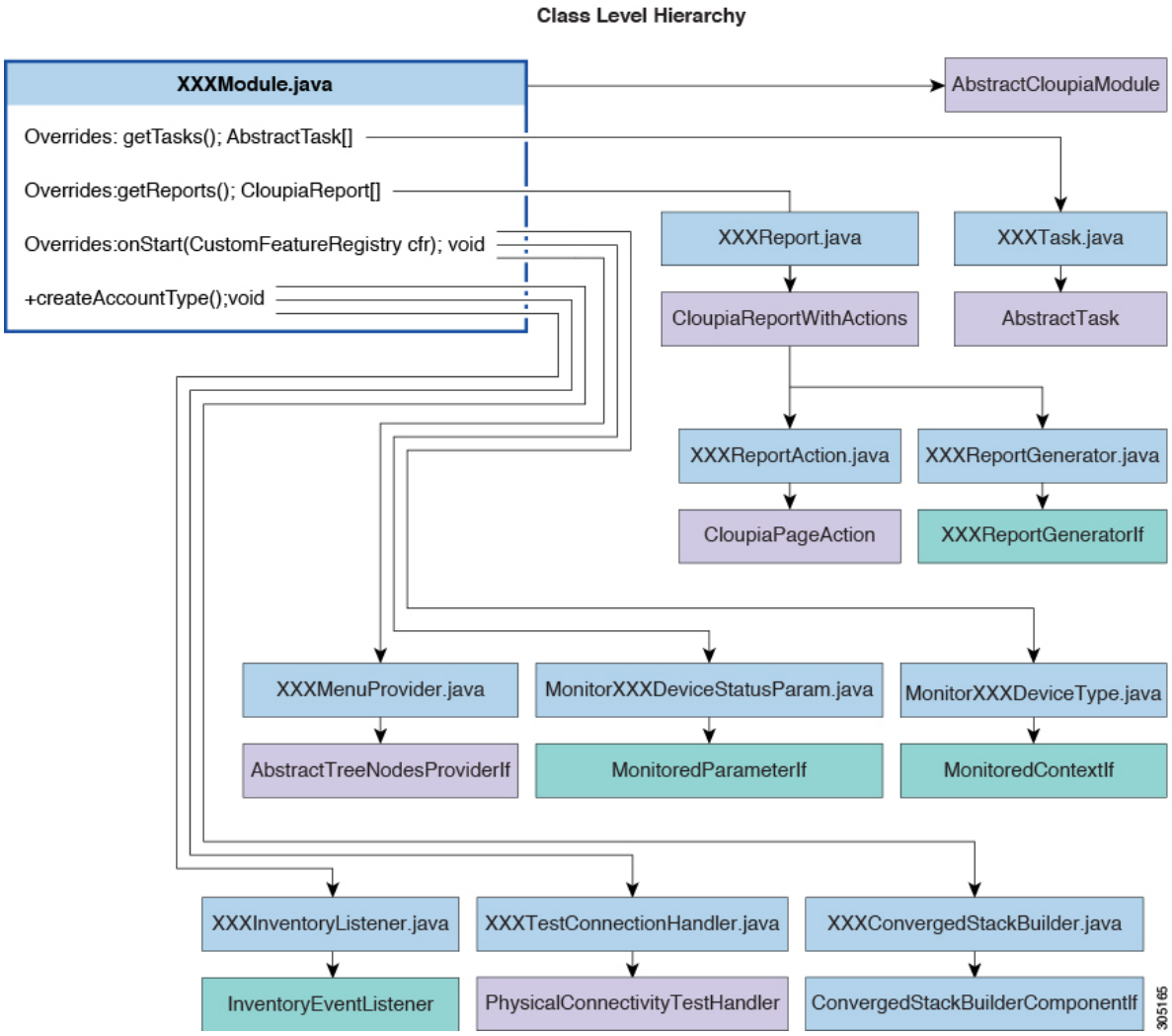
A module can include the following components:

Component	Description
Task	A Workflow Task that can be used as part of a Workflow.
Report	A report that appears in the Cisco UCS Director UI. Reports may (but are not required to) contain clickable actions.
Trigger	A condition that, once satisfied, can be associated with some action. Examples: shutdown VM, start VM, and so on.

Creating a Module

The following items must be in place for your custom module to work:

- A class extending `AbstractCloupiaModule`.
- Override the `OnStart` method in the Module Class that extends the `AbstractCloupiaModule`.
- A `.feature` file specifying your dependent jars and module class.
- A `module.properties` file is required in the custom module.



Before you begin

Refer to **FooModule** in the sample project of the Open Automation SDK bundle.

Step 1 Extend the `AbstractCloupiaModule` class and register all your custom components in this class.

Step 2 Create a `.feature` file that specifies the dependent jars and module class.

This file must end with an extension of `.feature`; see foo.feature for reference. The best practice is to name this file with your module ID. For more details about the `.feature` file, see Packaging the Module, on page 7.

Step 3 Add the necessary custom jar files to the `lib` folder.

Step 4 Package the properties file at the root level of your module jar.

Cisco UCS Director provides you with a `properties` file for validation purposes. The SDK sample provides you with a build file that handles the packaging process.

Note The content of the `module.properties` file is described in Understanding the module.properties File, on page 5.

Step 5 In the `module.properties` file, replace the moduleID with the ID of the custom module.

Step 6 From the Eclipse IDE package explorer, right-click the `build.xml` file and run the ANT target build. This action generates the module `zip` file and save the file to the base directory of your project.

The Open Automation SDK Bundle

The following files and directories are supplied in the Open Automation SDK Bundle. The listed files and directories are for an example Compute module. The Open Automation SDK Bundle also includes three other modules:

- A Storage module
- A Network module
- A dummy example module called "foo."

These other modules have similar file and path names, differing only in the module type (compute, storage, network, or foo).

`.classpath`
The `.classpath` file lists the project's source directory, output directory, and classpath entries such as other projects in the workspace, JAR files, and so on.

`.project`
The `.project` file is maintained by the core Eclipse platform. It describes the project from a generic, plugin-independent Eclipse view.

`src`
The `src` directory contains the Java source files and internationalization resource bundles for compilation in Eclipse.

`resources`
The `resources` directory contains the images to build in the `zip` file.

`compute.feature`
The `compute.feature` file defines Open Automation feature metadata for the compute project.

`lib`
The `lib` directory contains libraries needed for Eclipse to compile the Java source files.

`Poddefinition`
This directory contains the `pod.xml`file.

`cloud sense`
The `cloud sense` directory contains the `compute.report` and `compute.xml` files. These files are required to build the `zip` file.

`moresources`
the `moresources` directory contains the `.mo` and `.properties` REST API files.

Understanding the module.properties File

The `module.properties` file exposes the module to the platform runtime. This file defines properties of the module.

Here is a sample `module.properties` file:

```
moduleID=foo
```

```
version=1.0
ucsdVersion=6.5.0.0
category=/foo
format=1.0
name=Foo Module
description=UCSD Open Automation Sample Module
contact=support@cisco.com
key=5591befd056dd39c8f5d578d39c24172
```

The contents are described in the following table:

Table 2: New Module.Properties (module.properties)

Name	Description
moduleID	The unique identifier for the module. This property is mandatory. Example: `moduleID=foo` **Tip** We recommend that you restrict this ID to a string of 3 to 5 lowercase alphabetic ASCII characters.
version	The current version of your module. This property is mandatory. Example: `version=1.0`
ucsdVersion	The version of Cisco UCS Director designed to support your module (with which your module works best). This property is mandatory. Example: `ucsdVersion=6.5.0.0`
category	The path (/location) where all your tasks must be placed. This property is mandatory. Example: `category=/foo` **Note** The category parameter is the full path to the location where your tasks are placed. If the tasks module is not validated, the path is set to `Open Automation Community Tasks/Experimental`. If the tasks module is validated, the tasks are placed relative to the root folder. For example, you can use `/Physical Storage Tasks/foo`, `/Open Automation Community Tasks/Validated/foo`, or `/foo`. In the last case, there is a folder at root level called `foo`. This feature enables developers to place tasks in categories that are not under Open Automation or in its categories.

Name	Description
format	The version of the format of this module. This property is mandatory. By default, 1.0 version is set for the custom module. Example: `format=1.0` **Restriction** `1.0` is the only acceptable value here.
name	A user-friendly string that identifies your module in the Open Automation reports. Example: `name=Foo Module`
description	A user-friendly description of what your module does. Example: `description=UCSD Open Automation Sample Module`
contact	An email address that consumers of your module can use to request support. Example: `contact=support@cisco.com`
key	An encrypted key that the Cisco UCS Director Open Automation group provides for validating the module. Example: `key=5591befd056dd39c8f5d578d39c24172`

Note Modifying any mandatory properties invalidates your module. If you change any of the mandatory properties, you must request validation again. The name, description, and contact values, which are not mandatory, can be modified or omitted without revalidation.

Packaging the Module

A module is packaged with all the necessary classes, dependent JAR files, a `module.properties` file, and a `.feature` (pronounced "dot-feature") file. The `.feature` file is placed in the same folder as the root of the project. The `.feature` file shows the JAR associated with this module and the path to the dependent JAR files. The name of the `.feature` file is `<moduleID>.feature`.

Note The package name must start with com.cloupia.feature. For example, `Package com.cloupia.feature.oabc.OABCModule`.

The following example shows the content of a `.feature` file:

```
{
      jars:  [ "features/feature-chargeback.jar",
      "features/chargeback/activation-1.1.jar",
      "features/chargeback/axis2-jaxbri-1.5.6.jar",
      "features/chargeback/bcel-5.1.jar",
      "features/chargeback/jalopy-1.5rc3.jar",
      "features/chargeback/neethi-2.0.5.jar",
      "features/chargeback/antlr-2.7.7.jar",
      "features/chargeback/axis2-jaxws-1.5.6.jar",]
      features: [ "com.cloupia.feature.oabc.OABCModule" ]
}
```

Before you begin

We recommend that you use the Apache ANT build tool that comes with Eclipse. You can use any build tool or create the build by hand, but you must deliver a package with the same characteristics as one built with ANT.

SUMMARY STEPS

1. If your module depends on JARs that are not provided with the sample source code, include the jars in the `build.xml` file so that they are packaged in the zip file.
2. From the `build.xml` file, run the ANT target build.

DETAILED STEPS

	Command or Action	Purpose

Step 1	If your module depends on JARs that are not provided with the sample source code, include the jars in the `build.xml` file so that they are packaged in the zip file.	The following example shows a module layout with a third-party JAR:
		`feature-oabc`
		`  feature oabc.jar`
		`  oabc`
		`  lib`
		`  flex`
		`  flex-messaging-common.jar`
		`  oabc.feature`
		The module jar and `.feature` are at the top level of the zip file. We recommend that you put the third-party jars under the `/moduleID/lib` folder path, then any other sub-directories you may want to add.
		`{`
		`  jars:  [ 'features/feature-oabc.jar", features/oabc/lib/flex-messaging-common.jar ],`
		`  features: [ "com.cloupia.feature.oabc.OABCModule" ]`
		`}`
		When you list the jars in the `.feature` file, ensure that the jars start with `features/`; this is mandatory. This convention enables you to include the path to the jar. The path of each jar must be the same path that is used in your zip file. We recommend that you put your module jar first, followed by its dependencies, to ensure that your module loads.

	Command or Action	Purpose
Step 2	From the `build.xml` file, run the ANT target build.	The zip file is generated and saved to the base directory of your project. (We recommend that you create your own project directory for your module. For convenience, in this example we assume that the sample project is the base directory for your project.)

Deploying a Module on Cisco UCS Director

The Cisco UCS Director user interface provides **Open Automation** controls that you can use to upload and manage modules. Use these controls to upload the `zip` file of the module to Cisco UCS Director.

Note Only `zip`-formatted files can be uploaded using the **Open Automation** controls.

Before you begin

Acquire shell administrator access on the Cisco UCS Director VM. You can get this access from your system administrator. To use the Cisco UCS Director Shell Menu as a shell administrator, use SSH to access Cisco UCS Director, using the login **shelladmin** with the password that you got from the administrator.

For SSH access in a Windows system, use PuTTY (see http://www.putty.org/). On a Mac, use the built-in terminal application's SSH utility.

Step 1 Choose **Administration** > **Open Automation**.

Step 2 On the **Open Automation** page, click **Modules**.

The **Modules** page displays the following columns:

Column	Description
ID	The ID of the module.
Name	The name of the module.
Description	The description of the module.
Version	The current version of the module. The module developer must determine how to administer versioning of the module.
Compatible	Which version of Cisco UCS Director best supports this module.
Contact	The contact information of the person responsible for technical support for the module.
Upload Time	The time at which the module was uploaded.

Column	Description
Status	The status of the module. Possible statuses are: Enabled, Disabled, Active, and Inactive. You can control whether a module is enabled or disabled. If enabled, Cisco UCS Director attempts to initialize the module; if disabled, Cisco UCS Director ignores the module. A module is set to the Active state only when Cisco UCS Director is able to successfully initialize the module without throwing an exception. **Note** Active does not necessarily mean that everything in the module is working properly; it merely indicates that the module is up. Inactive means that when Cisco UCS Director tried to initialize the module, a severe error prevented it from doing so. Typical causes for the Inactive flag are: the module is compiled with the wrong version of Java, or a class is missing from the module.
Validated	Indicates whether the module is validated or not.

Note To enable module activation on upload, ensure that the `.feature` file in your module is named after your module ID. For example: If moduleId is `myFeatureName`, then name your feature file `myFeatureName.feature`.

The Cisco UCS Director framework identifies and loads the `.feature` file by name, based on the module ID. If the name of the `.feature` file and the module ID are different, the `.feature` file does not load and the module is not activated. If you choose to give the module ID and the .feature file different names, you must restart Cisco UCS Director to activate the module.

Step 3 Click **Add** to add a new module.

The **Add Modules** dialog box appears.

Step 4 Choose the module zip file from your local files and click **Upload** to upload the module zip file.

Step 5 Enable the module by choosing the module in the **Modules** table and clicking **Enable**.

Step 6 Wait while Cisco UCS Director activates the module.

Note Restarting Cisco UCS Director is not required to add and enable a module. Whereas, restarting Cisco UCS Director is required to disable, delete, or modify a module.

What to do next

Once the module is active, you can test the module.

Enabling Modules

Restarting Cisco UCS Director is no longer required to enable a module. However, you must restart Cisco UCS Director to modify, disable, or delete the new module.

An Open Automation module is enabled when you upload it to Cisco UCS Director. In previous versions of Cisco UCS Director, enabling a new Open Automation module required restarting the Cisco UCS Director server.

This is a short technical description of how a module is enabled when you upload the module to Cisco UCS Director.

When you upload an Open Automation module, the following events occur:

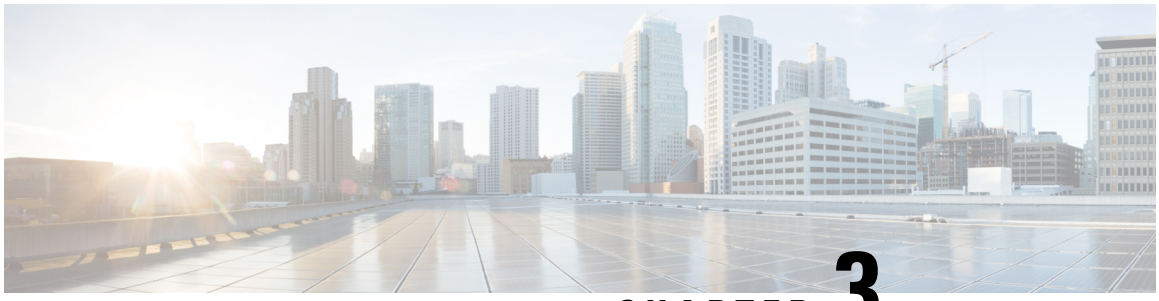
- The module class and its resources are loaded using a new `URLClassLoader`. The new `URLClassLoader` is used for all classes loaded into the JVM.
- Components of the Open Automation module are entered into various registries without restarting Cisco UCS Director.
- The `FeatureFileUploadEntry` table is updated with the `zip` file name, timestamp, and status.
- The system reloads the Pod Definition and Menu xml files, if they are available.
- The system reloads the REST API resource files, recreates the MoPointer objects, and reloads the MoPointers into their respective collections.
- The system recreates reports, workflow tasks, LOVs, system tasks, and other objects.

Deactivating a Module

To deactivate a module you must stop and restart the Cisco UCS Director services for your change to take effect.

Step 1 Choose the module you need to deactivate in the **Modules** table, then click the **Deactivate** control.

Step 2 Stop and restart the Cisco UCS Director services. Follow the same procedure that you use after activating a module.



CHAPTER 3

Managing Accounts

This chapter contains the following sections:

Accounts

You can use Open Automation to add a new custom Account type to Cisco UCS Director.

A custom account type provides new Cisco UCS Director data infrastructure that enables you to work with accounts in new ways. For example, a new account type allows you to manage and report on Cisco UCS Director managed elements in new ways.

Adding a Custom Account Type

To add an account type, perform these tasks:

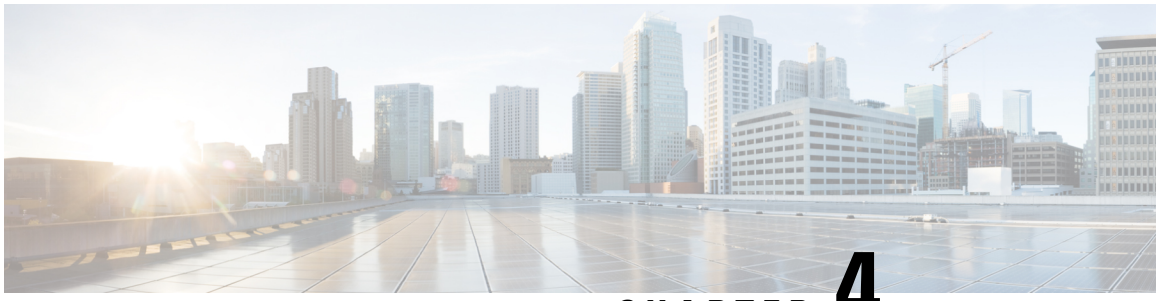
- Extend the **com.cloupia.lib.connector.account.AbstractInfraAccount** class.
- Define values for AccountType and AccountLabel
- Assign mandatory parameters for the AccountTypeEntry. These include setPodTypes, setAccountClass , setAccountType and setAccountLabel
- Complete implementation for Connectivity Test

The code snippet below demonstrates how to provide new Account details.

```
//This class is used to register a new connector into the UCSD.
AccountTypeEntry entry=new AccountTypeEntry();

//Set implementation class for Account type
entry.setCredentialClass(FooAccount.class);

//Account Type
entry.setAccountType(FooConstants.INFRA_ACCOUNT_TYPE);

//Account Label which will be shown in UI
entry.setAccountLabel(FooConstants.INFRA_ACCOUNT_LABEL);
```

```
//Account Category like Compute, Storage, Network or Multi-Domain.
entry.setCategory(InfraAccountTypes.CAT_NETWORK);

//This is mandatory , report generation on context Level for the new account type
entry.setContextType(ReportContextRegistry.getInstance().
getContextByName(FooConstants.INFRA_ACCOUNT_TYPE).getType());

//Account class like Physical-1, Virtual-2, Network-5, Multi-domain-3
//or Other-4
entry.setAccountClass(AccountTypeEntry.PHYSICAL_ACCOUNT);

// This will be used along with the account name to show
// in the System Tasks Report.
entry.setInventoryTaskPrefix("Open Automation Inventory Task");

//Set time frequency to collect account Inventory
entry.setInventoryFrequencyInMins(15);

//Supported POD types for this connector.
entry.setPodTypes(new String[]{"FooStack"});

//To add this account type entry
PhysicalAccountTypeManager.getInstance().addNewAccountType(entry);

// This is mandatory, to test the connectivity of the new account. The
// Handler should be of type PhysicalConnectivityTestHandler.
entry.setTestConnectionHandler(new FooTestConnectionHandler());
// This is mandatory, we can implement inventory listener according to
// the account Type , collect the inventory details.
entry.setInventoryListener(new FooInventoryListener());

//This is mandatory , to show in the converged stack view
entry.setConvergedStackComponentBuilder(new DummyConvergedStackBuilder());

//This is required to show up the details of the stack view in the GUI
entry.setStackViewItemProvider(new DummyStackViewProvider());

// This is required credential.If the Credential Policy support is
// required for this Account type then this is mandatory, can implement
// credential check against the policyname.
entry.setCredentialParser(new FooAccountCredentialParser());
```

Important Refer to the SDK samples. See `com.cloupia.feature.foo.accounts.FooAccount` and `com.cloupia.feature.foo.FooModule` for implementations.



CHAPTER 4

Collecting Account Inventory

This chapter contains the following sections:

About the Inventory Collector

You can introduce support for new devices by implementing your own Inventory Collector using the collector framework. When you are adding support for new devices, you must implement your Inventory Collector to handle collection and persistence of data in the database.

You can use the Inventory Collector framework reports to display the data. For more information about these reports, see Reports, on page 31.

Guidelines for Developing a Module

When you develop a module to support new devices, follow these guidelines:

- Develop for a device family so that you have only one module to support all devices in the family.
- Develop a single module to support only devices within the same category. A module should handle only compute devices, network devices, or storage devices. For example, do not develop a module that supports both a network switch and a storage controller. Instead, develop one module for the network switch and one module for the storage controller.
- Ensure that the devices supported by the same module are similar.
- The same device may come in different models that are meant for distinct purposes. In such cases, it may be appropriate to use different modules to support them.

Creating an Account Type Entry

You must create an AccountTypeEntry class for each account type to register a new Inventory Collector in the system.

The following code snippet explains how to create a new AccountTypeEntry class:

```
// This is mandatory, holds the information for device credential details
entry.setCredentialClass(FooAccount.class);

// This is mandatory, type of the Account will be shown in GUI as drill
// down box
entry.setAccountType(FooConstants.INFRA_ACCOUNT_TYPE);

// This is mandatory, label of the Account
entry.setAccountLabel(FooConstants.INFRA_ACCOUNT_LABEL);

// This is mandatory, specify the category of the account type ie.,
// Network / Storage / Compute
entry.setCategory(InfraAccountTypes.CAT_STORAGE);

//This is mandatory for setting report context for the new account type.
//Ensure that prior to this step the specified report context has been registered in
//module initialization i.e onStart method
//Refer to Registering Report Context section
entry.setContextType(ReportContextRegistry.getInstance().getContextByName(FooConstants.INFRA_ACCOUNT_TYPE).getType());


// This is mandatory, it associates the new account type with either physical or
// virtual account
entry.setAccountClass(AccountTypeEntry.PHYSICAL_ACCOUNT);

// Optional, prefix for tasks associated with this connector
entry.setInventoryTaskPrefix("Open Automation Inventory Task");

// Optional ,configurable inventory frequency in mins
entry.setInventoryFrequencyInMins(15);

// Supported POD types for this connector. The new account type will be associated
// with this pod. Note that this account type will be appended to list of account
// types defined in pod definition XML. Refer to section “Adding a Pod Type” for pod //
definition XML
entry.setPodTypes(new String[] { "FooPod" });

// This is mandatory, to test the connectivity of the new account. The
// Handler should be of type PhysicalConnectivityTestHandler. Account creation is
// is successful if this returns true.
entry.setTestConnectionHandler(new FooTestConnectionHandler());

// This is mandatory, associate inventory listener .Inventory listener will be called //
before and after inventory is done
entry.setInventoryListener(new FooInventoryListener());

// Set device icon path
entry.setIconPath(“/app/images/icons/menu/tree/cisco_16x16.png”);

// set device vendor
entry.setVendor(“Cisco”);

// This is mandatory, in order to properly display your device in the Converged tab // of
the UI
entry.setConvergedStackComponentBuilder(new DummyConvergedStackBuilder());
```

```
// If the Credential Policy support is
// required for this Account type then this is mandatory, can implement
// credential check against the policy name.
entry.setCredentialParser(new FooAccountCredentialParser());

// This is mandatory. Register Inventory Collectors for this account type.
// Refer to section "Creating Inventory Collectors" for more detail.
ConfigItemDef item1 = entry.createInventoryRoot("foo.inventory.root",
FooInventoryItemHandler.class);

// Register the new account entry with the system.
PhysicalAccountTypeManager.getInstance().addNewAccountType(entry);
```

Creating an Inventory Collector

Inventory Collector performs the core tasks of collecting, persisting, and deleting inventory data. Using the collector framework, you can introduce support for new devices by implementing your own Inventory Collector. When adding support for new devices, you must implement your Inventory Collector to handle collection and persistence of data in the database. The inventory collection tasks are embedded in collection handlers for each inventory object.

Inventory Collection Handlers

Inventory collection handlers enable collection of inventory data. You must register inventory collection handlers for inventory collection. These handlers must extend the AbstractInventoryItemHandler class.

The following code snippet registers an inventory collector and enables inventory collection for a specific model object:

```
ConfigItemDef item1 = entry.createInventoryRoot("foo.inventory.root",
FooInventoryItemHandler.class);
```

where

- `foo.inventory.root` is a unique registration ID.
- `FooInventoryItemHandler.class` is the handler class that implements methods for collecting inventory and cleaning inventory.

You must register separate implementation of the `AbstractInventoryItemHandler` class for each object that needs inventory collection. For more information, see the `FooModule.java` and `FooInventoryItemHandler.java` documents.

Inventory Listener

You can define an inventory listener that will be called before and after the inventory collection so that you can plug in your code before or after the inventory collection. This implementation is use case-based. For more information, see `FooInventoryListener.java class`.

Registering Collectors

You must register the collectors as follows:

```
PhysicalAccountTypeManager.getInstance().addNewAccountType(entry);
```

Registering a Report Context

You must define and register a main report context for an account type. The top level reports of the account type are associated with this context.

The following code snippet shows how to register a report context:

```
ReportContextRegistry.getInstance().register(FooConstants.INFRA_ACCOUNT_TYPE,
FooConstants.INFRA_ACCOUNT_LABEL);
```

The top level reports might require you to implement a custom query builder to parse context ID and generate query filter criteria. In such a case, the following code is required in reports:

```
this.setQueryBuilder (new FooQueryBuilder ());
```

For more information about how to build custom query builder, see the FooQueryBuilder.java class. You can register various report context levels for drill-down reports. For more information, see the Developing Drillable Reports, on page 36.

Converged Stack Builder

In the **Converged** tab of the user interface (UI), Cisco UCS Director displays the converged stack of devices for a data center. When you are developing a new connector, if you want to display your device in the Converged UI, you must supply your own ConvergedStackComponentBuilderIf, a device-icon mapping file, and the icons you would like to show.

Note For Network Connectors, you have to ensure that the report contexts are registered with the `<<Pod Name>>@<<Ip Address>>` by setting the report context value to the `<<Pod Name>>@<<Ip Address>>`.

Here is an example of a report context definition:

```
ConvergedStackComponentDetail detail = new ConvergedStackComponentDetail();
detail.setContextValue("Testflex@10.10.2.1");
```

Where, Testflex is the pod name and 10.10.2.1 is the IP address.

Before you begin

Ensure that you have the files in the sample code, including:

- device_icon_mapping.xml
- com.cloupia.feature.foo.inventory.DummyConvergedStackBuilder
- The resources folder that contains all the images

Step 1 Provide an implementation of ConvergedStackComponentBuilderIf.

Extend the abstract implementation:
com.cloupia.service.cIM.inframgr.reports.contextresolve.AbstractConvergedStackComponentBuilder.

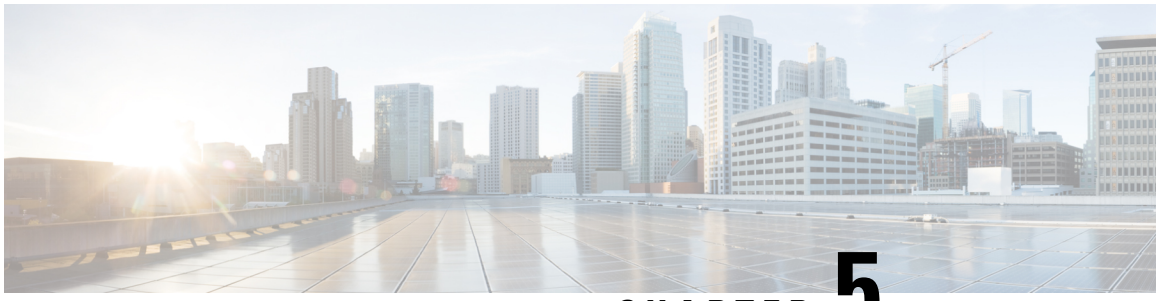
Step 2 Supply a device icon mapping file.

This XML file is used to map the data supplied by your ConvergedStackComponentBuilderIf to the actual images to be used in the UI. This XML file must be named as device_icon_mapping.xml and it must be packaged inside your jar.

Important For each entry in the XML file, the DeviceType must match the model in the ComponentBuilder and the vendor must match the vendor in the ComponentBuilder. The framework uses the vendor and model to uniquely identify a device and to determine which icon to use. Also, in the XML file, the IconURL value should always start with `/app/uploads/openauto`. All of your images will be dumped into this location.

Step 3 Package the images in a module.zip file and place the zip file in the **resources** folder.

The framework copies all your images in the **resources** folder and places them in an uploads folder.



CHAPTER 5

Managing Pods

This chapter contains the following sections:

Defining Pods and Pod Elements

A Pod holds Physical or Virtual accounts. It provides support for different device categories, including compute, storage, and network. The following stages are involved in creating a new pod type using Open Automation module:

- Create the pod definition XML configuration file in the /<Open_Automation>/pod definition directory where <Open_Automation> is the module project. More details are provided later in this section. Refer also to `foo.xml` in the samples provided with the Open Automation SDK.
- Upon deploying the module, the pod definition configuration file is copied to the appropriate Cisco UCS Director location for processing.
- The new pod definition is available in the **Type** dropdown list of values in the **Add Pod** form once the module is enabled and services are restarted.
- Customize the new pod type, as appropriate. For information about customization, refer to information about Converged Stack Builder in Collecting Account Inventory, on page 15 and elsewhere in the SDK documentation.
- If you delete the module, the new pod type created for it (as described in the first step above) will be deleted.

Defining Pod Types and Elements - Examples

Following is a line by line explanation of a pod definition.

The "pod-definition" is the root element. The type should be a string that uniquely identifies the pod type. The label should be what is shown in the UI for this pod type. In the following example, the pod being defined is a Flex Pod:

```
< pod-definition type="FlexPod"label="FlexPod">
```

Next inside the pod-definition are multiple pod-elements. A pod-element describes the device associated with the pod type:

- category specifies the device category the element belongs in 1 (compute), 2 (storage), 3 (network).
- name is the name of device type, this is mostly for readability purposes.
- count is the max number of this device type that can be used in one pod.
- account-types is a comma separated string of all account type IDs that collect data for this device type.

```
 < pod-element category="1" name="Cisco UCS" code="-1"
count="1" account-types="11">
```

The example above shows a typical Cisco UCS pod-element. The category is 1, so it's compute category. The count is 1, so there can only be one Cisco UCS in a Flex Pod. The Cisco UCS collector has an account type ID of 11, which means it is internal. (For a list of IDs for available the available collectors, ask a lead.)

```
< device-model vendor="[cC]isco"version=".*"model="UCSM"/>
```

The device-model provides the details on how UCSD will perform pod compliance checks. The vendor, version, and model strings will be checked against the values you provided when you added the account through the UI. Note that the use of regular expressions is allowed, so in this example, if you enter "cisco" or "Cisco", it is still acceptable.

```
</ pod-definition >
```

Finally, be sure to properly close the pod-definition element.

Complete Examples

Following is an example of a Nexus switch pod definition:

```
<pod-element category="3" name="NXOS" count="6" code="81" account-types="nxos">
<device-model vendor="[cC]isco" version=".*" model="Nexus[\s]*[157].*" />
</pod-element>
```

Here is an example of a NetApp storage device pod definition:

```
<pod-element category="2" count="2" code="77" account-types="12,14">
  <device-model vendor="[nN]et[aA]pp" version=".*"
model="FAS.*|.*Cluster.*|.*OnCommand.*|.*DFM.*" />
 </pod-element>
```

Adding a Pod Type

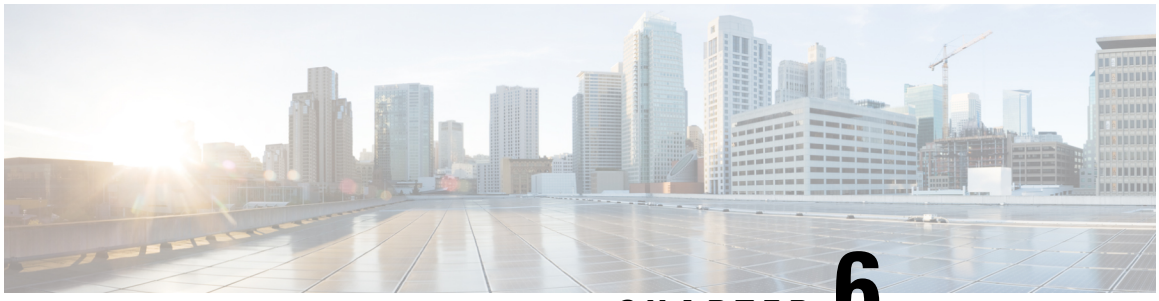
Inside the pod definition are multiple pod elements. A pod element describes the device associated with the pod type:

- category specifies the device category the element belongs in 1 (compute), 2 (storage), 3 (network).
- name is the name of device type, this is mostly for readability purposes.
- count is the max number of this device type that can be used in one pod.
- account types is a comma separated string of all account type IDs that collect data for this device type.
- The following shows a typical Cisco UCS pod element.

The following shows a typical Cisco UCS pod element.

```
< pod-element category="1" name="Cisco UCS" code="1"
count="1" account types="11">
Category is 1 = compute category.
Count is 1 = only one Cisco UCS in a Flex Pod. The Cisco UCS collector has an account type
 ID
of 11, which means it is internal.
< device model vendor="[cC]isco"version=".*"model="UCSM"/>
```

The device model provides the details on how UCSD will perform pod compliance checks. The vendor, version, and model strings will be checked against the values you provided when you added the account through the UI. Note that the use of regular expressions is allowed, so in this example, if you enter"cisco" or "Cisco", it is still acceptable. Make sure to close the pod definition element properly `</ pod-definition >`.



CHAPTER 6

Managing Objects

This chapter contains the following sections:

Object Store

The Object Store provides simple APIs for database persistence. A module that needs to persist objects into the database typically uses the Object Store APIs to perform all the CRUD (Create, Read, Update, and Delete) operations.

Cisco UCS Director uses MySQL as its database. The platform runtime makes use of the Java Data Object (JDO) library provided by DataNucleus to abstract all the SQL operations through an Object Query representation. This simplifies and speeds up the development with respect to data persistence. The Object Store documentation include sections that show how CRUD operations are realized using JDO.

Note This documentation uses the acronym POJO (Plain Old Java Object) to refer to a java class that does not extend any other class or implement any interfaces.

Marking a Class for Persistence

A POJO class that needs to be persisted in the database has to be defined and marked with suitable JDO annotations. The class shown below is marked for JDO persistence.

In this class, note that

- `foo_netapp_filer` is attached on top of the class declaration.
- The `table` attribute specifies the name of the table to be used.
- `foo` is the name of the module.
- `@Persistent` is attached to the field that needs persistence.

```
package com.cloupia.lib.cIaaS.netapp.model;
```

```
@PersistenceCapable(detachable = "true", table = "foo_netapp_filer")
public class  NetAppFiler
{
    @Persistent
    private String filerName;

    @Persistent
    private String accountName;

    @Persistent
    private String dcName;
}
```

The above class has two annotations: `@PersistenceCapable` and `@Persistent`. These are defined in the JDO, and the Cisco UCS Director Platform runtime expects all persistent classes to be marked with these two annotations. Cisco UCS Director uses a flat schema, so creating a nested schema, though possible and allowed in JDO, is not recommended in a Module.

What to do next

The persistence class is now ready for CRUD operations against the database.

Publishing the Persistence Class

A class that is marked with suitable JDO annotation has to be published so that the Platform Runtime can pick up the class.

SUMMARY STEPS

1. Create a file with the name `jdo.files` in the same directory (package) as that of the persistence class.
2. Add the name of the class to the file as follows:

DETAILED STEPS

Step 1 Create a file with the name `jdo.files` in the same directory (package) as that of the persistence class.

Step 2 Add the name of the class to the file as follows:

Example:

```
Linux# cat jdo.files

// Copyright (C) 2010 Cisco Inc. All rights reserved.
//
// Note: all blank lines and lines that start with // are ignored
//
// Each package that has Persistable Objects shall have a file called jdo.files
// Each line here indicates one class that represents a persistable object.
//
// Any line that starts with a + means package name is relative to current package
// If a line starts without +, then it must be complete fully qualified java class name
// (for example: com.cloupia.lib.xyz.MyClass)

+NetAppFiler

Linux#
```

Performing CRUD Operations on the Persistence Class

When a persistence class is ready for CRUD operations against the database, you can perform the different operations available, as shown in the following examples.

Create a New Instance of the Object

```
NetAppFiler filer = new NetAppFiler();
filer.setAccountName("netapp-account");
filer.setDcName("Default Datacenter");
filer.setfilerName("filer0");
filer.setIpAddress("192.168.0.1");

ObjStore store = ObjStoreHelper.getStore(NetAppFiler.class);
    store.insert(filer);
```

Modify a Single Instance of the Object

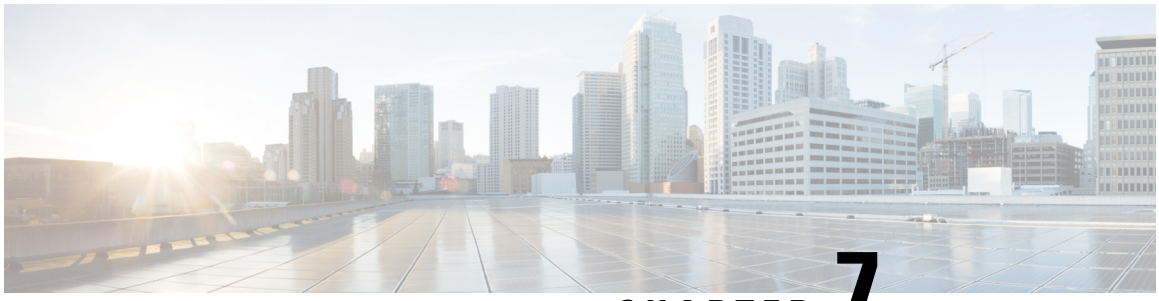
```
ObjStore store = ObjStoreHelper.getStore(NetAppFiler.class);
String query = "filerName == 'filer0'";
//Use Java field names as parameter,

    // can use && , || operators in the query.
    store.modifySingleObject(query, filer);
```

Querying All the Instances from the Database

```
ObjStore store = ObjStoreHelper.getStore(NetAppFiler.class);

List filerList = store.queryAll();
```

Querying the Instances with a Filer Query

```
ObjStore store = ObjStoreHelper.getStore(NetAppFiler.class);

String query = "dcName == 'Default Datacenter'";
List filerList = store.query(query);
```



CHAPTER 7

Managing Annotations and LOVs

This chapter contains the following sections:

Annotations

Annotations are one of the most crucial parts of Module development. Most of the artifacts are driven by annotations. This makes the development effort all the more easy and convenient.

Annotations are used for persistence, report generation, wizard generation, and tasks.

Persistence Annotations

See Marking a Class for Persistence, on page 25, for information about the annotations that are used for persistence.

Task Annotation

When a task is included in a Workflow, the user is prompted for certain inputs. The user is prompted for an input when a field of the class representing the task is marked with an annotation. The FormField annotation determines what type of UI input field to show to the user: a text field, or a dropdown list, or a checkbox, etc. For more information, see Tasks, on page 53.

Lists of Values (LOVs)

Lists represent the drop-down LOVs (Lists of Values) that are displayed to the user to facilitate getting the correct inputs for a task. You can reuse an existing list or create your own list to show in the Task UI.

Cisco UCS Director defines over 50 prebuilt List providers that the modules can readily use to prompt input from the user. For more information, see Appendix A, on page 87.

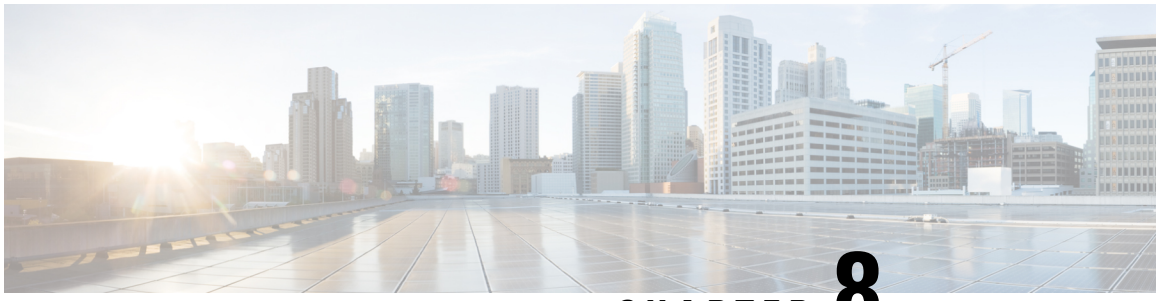
For an example that shows how to use one of the list providers, see Defining Your Own List Provider, on page 30, and Tasks, on page 53.

Defining Your Own List Provider

You can define your own list provider and ask the Platform Runtime to register it with the system.

A list provider class implements the **LOVProviderIf** interface and provides implementation for the single method **getLOVs( )**. See the following example:

```
class MyListProvider implements LOVProviderIf
{

 /**
  * Returns array of FormLOVPair objects.  This array is what is shown
  * in a dropdown list.
  * A FormLOVPair object has a name and a label. While the label is shown
  * to the user, the name will be used for uniqueness
  */
 @Override
 public FormLOVPair[] getLOVs(WizardSession session) {

  // Simple case showing hard-coded list values

  FormLOVPair http = new FormLOVPair("http", "HTTP");
  // http is the name, HTTP is the value
  FormLOVPair https = new FormLOVPair("https", "HTTPs");

  FormLOVPair[] pairs = new FormLOVPair[2];
  pairs[0] = http;
  pairs[1] = https;
  return pairs;
 }
}
```



CHAPTER 8

Managing Reports

This chapter contains the following sections:

Reports

The Open Automation reports are used to display and to retrieve the data in the UI for the uploaded module.

You can develop your own reports in two ways. The simplest way is to use the Plain Old Java Object (POJO)-and-Annotation approach. The more advanced approach is to implement the `TabularReportGeneratorIf` interface programmatically.

You can develop POJO-based reports with the following classes:

- CloupiaEasyReportWithActions
- CloupiaEasyDrillableReport

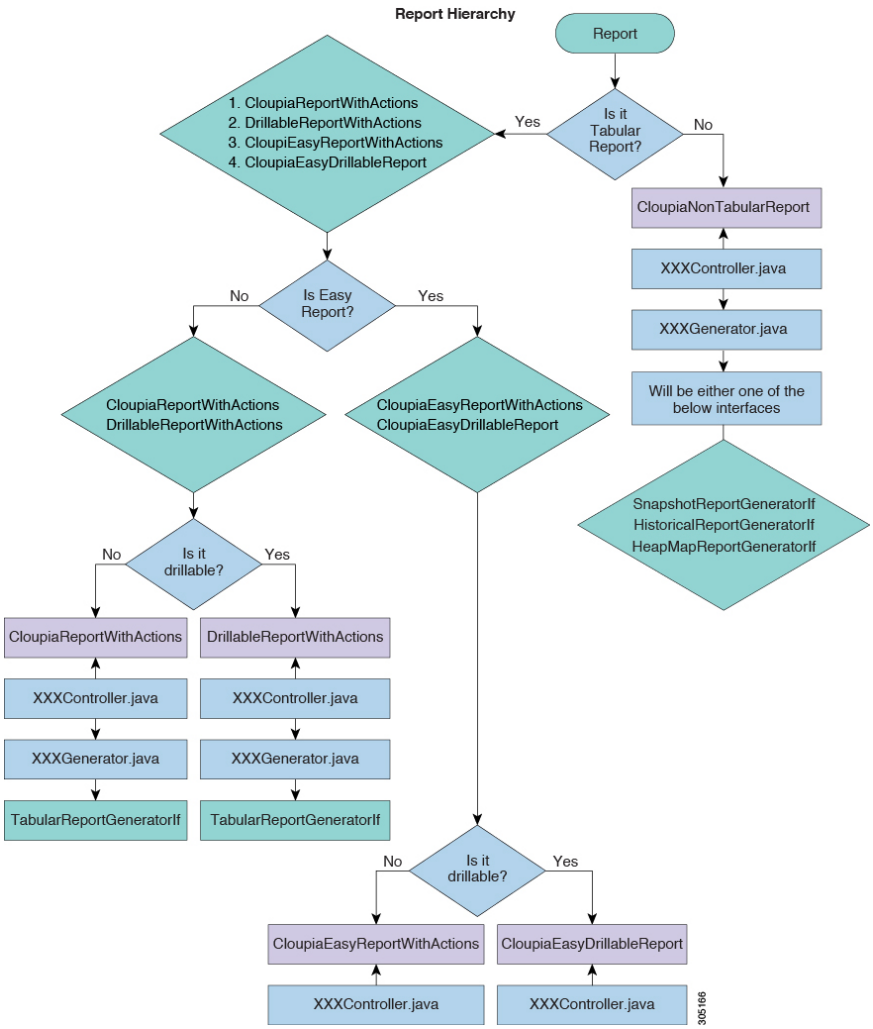
You can develop programmatic reports with the following classes:

- CloupiaReportWithActions
- DrillableReportWithActions

When you develop reports, you must decide whether to use the POJO-based approach or whether you should generate the report programmatically. You must also decide whether to include drill-down reports (which are possible with either the POJO or the programmatic approach).

The Open Automation documentation about creating your own reports includes instructions for creating both *tabular* and *non-tabular* reports. Non-tabular reports in this context include bar chart, line chart, pie chart, heat map, and summary reports; and also a "form report". A form report is a form that occupies the space of a report (that is, the space of an entire tab in the UI).

Figure 1: Report Flow



Note The information about tabular reports is fundamental; the procedures that you use to create a tabular report form the basis for developing non-tabular reports.

Developing Reports Using POJO and Annotations

You can develop a POJO□based report using the following classes:

- CloupiaEasyReportWithActions
- CloupiaEasyDrillableReport

To develop a report, use the Java Data Object (JDO) POJOs that are developed for persistence and add some annotations. The report is ready for display in the UI.

SUMMARY STEPS

1. Implement the com.cloupia.service.cIM.inframgr.reports.simplified.ReportableIf interface in data source POJO. Use the getInstanceQuery method in the ReportableIf interface to return a predicate that is used by the framework to filter out any instances of the POJO that you do not want to display in the report.
2. For each field in the POJO that needs to be displayed in the report, use the @ReportField annotation to mark it as a field to include in the report.
3. Extend one of the following classes. Both classes are used to create a report using the POJO□and□Annotation method. Provide the report name (to uniquely identify this report), the label of this report (to be displayed to the user), and the data source (the POJO that you just created).

DETAILED STEPS

Step 1 Implement the com.cloupia.service.cIM.inframgr.reports.simplified.ReportableIf interface in data source POJO. Use the getInstanceQuery method in the ReportableIf interface to return a predicate that is used by the framework to filter out any instances of the POJO that you do not want to display in the report.

Step 2 For each field in the POJO that needs to be displayed in the report, use the @ReportField annotation to mark it as a field to include in the report.

Example:

```
public class SampleReport implements ReportableIf{
@ReportField(label=”Name”)
@Persistent
private String name;
public void setName(String name){ this.name=name;
}
public String getName(){ return this.name;
}
@Override
public String getInstanceQuery() { return "name == '" + name+ "'";
}
}
```

This POJO can be referred to as the data source.

Step 3 Extend one of the following classes. Both classes are used to create a report using the POJO□and□Annotation method. Provide the report name (to uniquely identify this report), the label of this report (to be displayed to the user), and the data source (the POJO that you just created).

- com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyReport WithAction

  Use this class when you need to assign action to report.

- com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyDrillabl eReport

  Use this class when you need to implement drill down report.

Example

Implementing ReportableIf

The DummySampleImpl class implements the ReportableIf interface as you use the getInstanceQuery method which returns the predicate and it is used by framework to filter out any instances of the POJO that you do not want to display in the report.

```
@PersistenceCapable(detachable = "true")
public class DummySampleImpl implements ReportableIf {
@Persistent
private String accountName;
@ReportField(label="Name")
@Persistent
private String name;
}
```

Extending CloupiaEasyReportWithActions

Extend the CloupiaEasyReportWithActions class and provide the report name (that should be unique to fetch the report), data source (which is pojo class), and report label (that is displayed in the UI) to get a report. You can assign the action to this report by returning action object from the getActions() method.

```
public class DummySampleReport extends CloupiaEasyReportWithActions {
//Unique report name that use to fetch report, report label use to show in UI
and dbSource use to store data in CloupiaReport object.
private static final String name = "foo.dummy.interface.report";
private static final String label = "Dummy Interfaces"; private static final
Class dbSource =
DummySampleImpl.class;
public DummySampleReport() { super(name, label, dbSource);
}
@Override
public CloupiaReportAction[] getActions() {
// return the action objects,if you don’t have any action then simply return
null.
}
}
```

Register the DummySampleReport report with the module class in the getReport section of the UI.

Developing Tabular Reports

Before you begin

See the com.cloupia.feature.foo.reports.DummyVLANsReport and com.cloupia.feature.foo.reports.DummyVLANsReportImpl for examples.

SUMMARY STEPS

1. Create an instance of TabularReportInternalModel which contains all the data you want to display in the UI.
2. Extend one of the following classes. Both classes are used to create a report using the POJO and Annotation method.
3. Implement the Tabular-ReportGeneratorIF.
4. Provide the report name (to uniquely identify this report), the label of this report (to be displayed to the user), and the data source (the POJO that you just created).
5. Specify the implementation of the data source and make sure that the isEasyReport() method returns false.

DETAILED STEPS

Step 1 Create an instance of TabularReportInternalModel which contains all the data you want to display in the UI.

Step 2 Extend one of the following classes. Both classes are used to create a report using the POJO and Annotation method.

- com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyReport WithAction

  Use this class when you need to assign action to report.

- com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyDrillabl eReport

  Use this class when you need to implement drill down report.

Step 3 Implement the Tabular-ReportGeneratorIF.

Step 4 Provide the report name (to uniquely identify this report), the label of this report (to be displayed to the user), and the data source (the POJO that you just created).

Step 5 Specify the implementation of the data source and make sure that the isEasyReport() method returns false.

Tabular Report

The DummyReportImpl class implements the TabularReportGeneratorIf interface. If you need more granular control over how you display the data in a report, use this approach to create report by implementing TabularReportGeneratorIf interface.

```
public class DummyReportImpl implements TabularReportGeneratorIf
{
private static Logger logger = Logger.getLogger(DummyReportImpl.class);
@Override
public TabularReport getTabularReportReport(ReportRegistryEntry reportEntry,
ReportContext context) throws Exception {
TabularReport report = new TabularReport();
// current system time is taking as report generated time, setting unique
report name and the context of report
report.setGeneratedTime(System.currentTimeMillis());
report.setReportName(reportEntry.getReportLabel());
report.setContext(context);
//TabularReportInternalModel contains all the data you want to show in report
TabularReportInternalModel model = new TabularReportInternalModel();
model.addTextColumn("Name", "Name"); model.addTextColumn("VLAN ID", "VLAN
ID"); model.addTextColumn("Group", "Assigned To Group");
model.completedHeader(); model.updateReport(report);
return report;
```

```
}
}
public class DummySampleReport extends CloupiaReportWithActions {
private static final String NAME = "foo.dummy.report"; private static final
String LABEL = "Dummy Sample";
//Returns the implementation class
@Override
public Class getImplementationClass() { return DummyReportImpl.class;
}
//Returns the report label use to display as report name in
UI
@Override
public String getReportLabel() { return LABEL;
}
//Returns unique report name to get report
@Override
public String getReportName() { return NAME;
}
//For leaf report it should returns as false
@Override
public boolean isEasyReport() { return false;
}
//For drilldown report it should return true
@Override
public boolean isLeafReport() { return true;
}
}
```

Register the report into the system to display the report in the UI.

Developing Drillable Reports

Reports that are nested within other reports and are only accessible by drilling down are called drillable reports. Drillable reports are applicable only for the tabular reports.

The report data source must be implemented through the POJO and Annotation approach. It is mandatory to override the isLeafReport API to return false. The report should extend thecom.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyDrillableReport class. The report data source must be implemented using the TabularReportGeneratorIf interface.The report should extend thecom.cloupia.service.cIM.inframgr.reports.simplified.DrillableReportWithA ctions class.Both classes require you to provide instances of the reports that will be displayed when the user drills down on the base report. Each time the getDrillDownReports() method is called, it should return the same instances. You should initialize the array of reports and declare them as member variables, as in com.cloupia.feature.foo.reports.DummyAccountMgmtReport.

To manage context levels in drill-down reports, do the following:

1. Add report registries for the drill-down context. For more information, see Registering Report Contexts, on page 65.

   Example:

   ```
   ReportContextRegistry.getInstance().register(FooConstants. DUMMY_CONTEXT_ONE_DRILLDOWN,
    FooConstants. DUMMY_CONTEXT_ONE_ DRILLDOWN _LABEL);
   ```

2. In the parent report, override the `getContextLevel()` class to return the drill-down context (for example, DUMMY_CONTEXT_ONE_ DRILLDOWN) that is defined in the report registry as in the previous step.

Example:

```
@Override
 public int getContextLevel() {
DynReportContext context =
ReportContextRegistry.getInstance().getContextByName(FooConstants. DUMMY_CONTEXT_ONE_
DRILLDOWN);
  logger.info("Context " + context.getId() + " " + context.getType());
  return context.getType();
}
```

3. In the drill-down child reports, override the `getMapRules()` class to refer the drill-down context (For example, DUMMY_CONTEXT_ONE_ DRILLDOWN)) that is defined in the report registry.

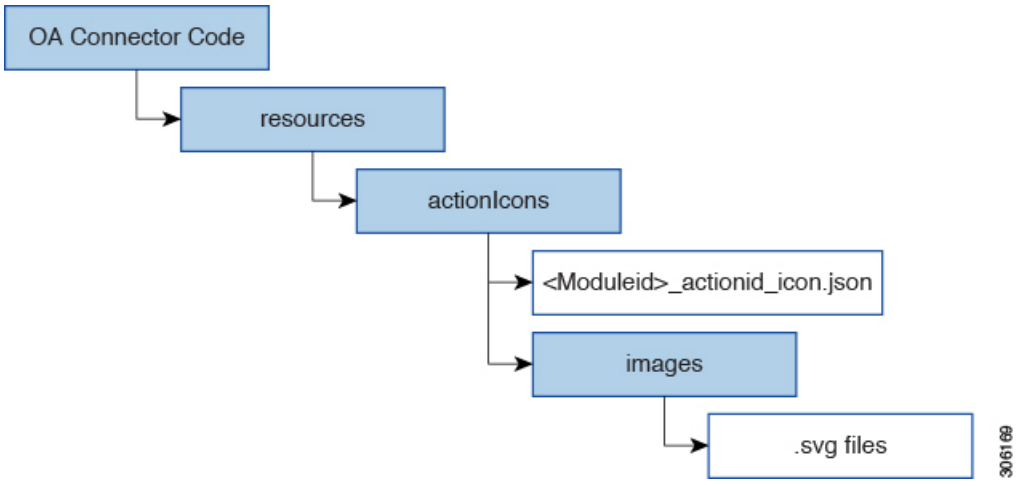
Example:

```
@Override
 public ContextMapRule[] getMapRules() {

DynReportContext context =
ReportContextRegistry.getInstance().getContextByName(FooConstants. DUMMY_CONTEXT_ONE_
DRILLDOWN);

  ContextMapRule rule = new ContextMapRule();
  rule.setContextName(context.getId());
  rule.setContextType(context.getType());

  ContextMapRule[] rules = new ContextMapRule[1];
  rules[0] = rule;

  return rules;
}
```

Integrating Action Icons

You can integrate custom icons to be deployed with your module in Cisco UCS Director.

To deploy custom action icons with Cisco UCS Director, save the icons in your resources folder as in the following illustration.



To add action icons to your module, do the following:

Step 1 Create the icons in scalable vector graphics (SVG) format. The images must be of size 27 x 27.

Step 2 Save the icon `.svg` files in the images folder as shown in the illustration.

Step 3 Reference the icons from the `<MODULE_ID>_actionid_icon.json` file in the `actionIcons` folder as shown in the illustration.

For example, here are two entries from the `<MODULE_ID>_actionid_icon.json` file for action icons for a module named `compute`.

```
[
  {
    "id": "compute - Custom Enable OA Module",
    "iconName": "compute_EnableOpenAutomation.svg",
    "defaultIconName": "compute_EnableOpenAutomation.svg"
  },
  {
    "id": "compute - Custom Disable OA Module",
    "iconName": "compute_DisableOpenAutomation.svg",
    "defaultIconName": "compute_DisableOpenAutomation.svg"
  }
]
```

The values of "iconName" and "defaultIconName" do not have to be unique.

The value of the "id" field must be unique, not only within the Open Automation module but throughtout all the modules in Cisco UCS Director. To systematically assign unique names, we recommend you define "id" using the pattern “<ModuleId> - <Action Name>”; for example, “compute - Custom Enable OA Module”.

Step 4 Reference the action icons in the module code using the unique id value. For example:

```
public class SimpleDummyAction extends CloupiaPageAction {

 private static Logger logger = Logger.getLogger(SimpleDummyAction.class);

 // Provide a unique strings to identify this form and action (note: prefix is the module id; good
practice)
 private static final String formId = "compute.simple.dummy.form";
 private static final String ACTION_ID = "compute - Custom Enable OA Module";

 // This is the label shown in the UI for this action. This label should match the “id” column in
the
 // <ModuleId>_actionid_icon.json file, if you are using custom action icons.
 private static final String label = "compute - Custom Enable OA Module";
```

Step 5 Integrate the action with the report. See Developing Form Reports, on page 48 and Developing Reports Using POJO and Annotations, on page 33.

What to do next

Build the project and upload the module to Cisco UCS Director. View the custom action in the Open Automation built forms and reports by navigating to **Open Automation** > **Modules**.

Note The Open Automation <MODULE_ID>_actionid_icon.json file is stored as oa_<MODULE_ID>_actionid_icon.json on the Cisco UCS Director server and is merged with actionid_icon.json, the file containing all of the action icons references in Cisco UCS Director.

The actionid_icon.json file can become corrupted in scenarios where the server is shutdown while the upload is happening. If that happens, retrieve the backup file backup_actionid_icon.json from the location /opt/infra/web_cloudmgr/apache-tomcat/webapps/app/ux/resources on the Cisco UCS Director server and restore the actionid_icon.json file. Restart the server, then upload the module again.

Registering Reports

The final step in developing reports is registering all the components you have developed in AbstractCloupiaModule. You must implement createAccountType(() and getReports(). If you instantiate and return new instances of the reports, they will be registered into the system.

```
public class FooModule extends AbstractCloupiaModule {
@Override
public CloupiaReport[] getReports() {
CloupiaReport[] reports = new
CloupiaReport[2];
}
}
reports[0] = new SampleReport(); reports[1] = new FooAccountSampleReport
();
return reports;
```

Registering a Report Context

You must define and register a main report context for an account type. The top level reports of the account type are associated with this context.

The following code snippet shows how to register a report context:

```
ReportContextRegistry.getInstance().register(FooConstants.INFRA_ACCOUNT_TYPE,
FooConstants.INFRA_ACCOUNT_LABEL);
```

The top level reports might require you to implement a custom query builder to parse context ID and generate query filter criteria. In such a case, the following code is required in reports:

```
this.setQueryBuilder (new FooQueryBuilder ());
```

For more information about how to build custom query builder, see the FooQueryBuilder.java class. You can register various report context levels for drill-down reports. For more information, see the Developing Drillable Reports, on page 36.

Enabling the Developer Menu

Step 1 In Cisco UCS Director, click your login name in the upper right.

Step 2 In the User Information dialog box, click the Advanced tab.

Step 3 Check the Enable Developer Menu (for this session) check box and close the User Information dialog box. The Report Metadata option becomes available in the report views opened in the session.

Step 4 Navigate to a tabular report in the same location where you want your report to appear and click Report Metadata to see the Information window. See the Report Context section at the top of that window.

Step 5 Find the integer value assigned to the uiMenuTag.

Step 6 The uiMenuTag tells you what your report's getMenuID should return.

Step 7 Find the value assigned to type. The type provides the UI menu location ID that you need to build the context map rule, which in turn tells you what your report's getMapRules must return.

Step 8 Get the context map rule that is necessary to build the context map from the report metadata. The first column provides the type of report context and the second column provides the name of the report context. Given that you have the type, you can locate the name. For example, 0 maps to global. When you have both information (the context name and the context type), you can build your context map rule.

Step 9 Initiate a context map rule with details similar to those in the following code sample:

```
ContextMapRule rule = new ContextMapRule(); rule.setContextName("global");
rule.setContextType(0);
ContextMapRule[] rules = new ContextMapRule[1]; rules[0] = rule;
```

Note This sample uses the plain constructor. Do not use another constructor. The plain constructor serves the purpose and explicitly sets these values.

Specifying the Report Location

To specify the exact location where your report will appear in the user interface, you must provide two pieces of information:

- The UI menu location's ID
- The Context Map Rule that corresponds to the report context of the location.

To gather these pieces of information, start by using the metadata provided by Cisco UCS Director. The metadata includes data for the report nearest to the place where you want your report to appear, and you can use this data to start constructing the report specifications that you need.

Step 1 Enable the developer menus for your session.

a) In Cisco UCS Director, click your login name in the upper right.
b) In the **User Information** dialog box, click the **Advanced** tab.
c) Check the **Enable Developer Menu (for this session)** check box and close the User Information dialog box.

The **Report Metadata** option becomes available in the report views opened in the session.

Step 2 Navigate to a tabular report in the same location where you want your report to appear, then click on **Report Metadata** to see the **Information** window. See the **Report Context** section at the top of that window.

a) Find the integer value assigned to the **uiMenuTag**.

The **uiMenuTag** tells you what your report's `getMenuID` should return.

The MenuID default values are:

- Physcial -> Storage -> LH Menu Tree Provider is 51.
- Physcial -> Compute -> LH Menu Tree Provider is 50.
- Physcial -> Network -> LH Menu Tree Provider is 52.

b) Find the value assigned to the **type**.

The **type** tells you the *first* piece of information you need to build the context map rule, which in turn tells you what your report's `getMapRules` should return.

Step 3 Get the second piece of information necessary to build the context map from the `reportContexts.html` file. See a copy in Appendix B, on page 91.

The `reportContexts.html` file lists every report context registered in the system. The first column provides the **type** of report context and the second column provides the **name** of the report context. Given that you have the **type**, you can locate the name. For example, 0 maps to "global".

When you have both pieces of information (the context name and the context type) you can build your context map rule.

Step 4 Instantiate a Context Map Rule with details similar to those in the following code sample.

Example:

```
ContextMapRule rule = new ContextMapRule();
rule.setContextName("global");
rule.setContextType(0);

ContextMapRule[] rules = new ContextMapRule[1];
rules[0] = rule;
```

Note that this sample uses the plain constructor. Do NOT use another constructor. The plain constructor serves the purpose and explicitly sets these values.

If your report specification code has properly set these new values OR overridden the methods to return these values, you should be able to view the report in the expected location.

Tip All the new report samples will show up under **Physical > Network > DummyAccount** tab. Find a report by drilling down in one of the rows.

Developing Bar Chart Reports

Open Automation enables you to develop non-tabular reports such as Bar Charts. Developing a bar chart is similar to developing a plain tabular report, and you should follow the same basic procedures. For the bar chart report, data can be provided by the source class. Override the getSnapshotReport method and provide the data source. It is mandatory to override the getReportType and getReportHint APIs to return corresponding values.

Step 1 Extend CloupiaNonTabularReport by following the example provided here:

Example:

```
public class BarChartReport extends CloupiaNonTabularReport  {

    private static final string NAME = "foo.dummy.bar.chart.report";
    private static final string LABEL = "Dummy Bar Chart";
```

Step 2 Override getReportType() and getReportHint(). Refer to this code snippet:

Example:

```
@Override
public int getReportType()
{
    return ReportDefinition.REPORT_TYPE_SNAPSHOT;
}

@Override
public int getReportHint()
{
    return ReportDefiniton.REPORT_HINT_BARCHART;
}
```

Step 3 Implement your own bar chart by following the example provided in this code:

Example:

```
public class BarChartReportImpl implements SnapshotReportGeneratorIf {

private final int NUM_BARS = 2;
private final String BAR_1 = "bar1";
private final String BAR_2 = "bar2";
```

Step 4 To build a bar chart and register it to a category, follow the example provided in this section of code:

Example:

```
ReportNameValuePair[] rnv1 = new ReportNameValuePair [NUM_BARS];
rnv1[0] = new ReportNameValuePair(BAR_1, 5);
rnv1[1] = new ReportNameValuePair(BAR_2, 10);

SnapshotReportCategory cat1 = new SnapshotReportCategory();
cat1.setCategoryName("cat1");
cat1.setNameValuePairs(rnvr1);
```

Bar Chart

```
public class SampleBarChartReportImpl implements SnapshotReportGeneratorIf {
//In this example , defines the number of bars should be in chart as bar1 nd
bar2 like shown in above snapshot
private final int NUM_BARS = 2; private final String BAR_1 = "bar1"; private
final String BAR_2 = "bar2";
@Override
public SnapshotReport getSnapshotReport(ReportRegistryEntry reportEntry,
ReportContext context) throws Exception
{
SnapshotReport report = new SnapshotReport(); report.setContext(context);
report.setReportName(reportEntry.getReportLabel());
report.setNumericalData(true); report.setValueAxisName("Value Axis Name");
report.setPrecision(0);
chart
```

```
// setting the report name value pair for the bar
ReportNameValuePair[] rnv1 = new
ReportNameValuePair[NUM_BARS];
rnv1[0] = new ReportNameValuePair(BAR_1, 5); rnv1[1] = new
ReportNameValuePair(BAR_2, 10);
// setting category of report SnapshotReportCategory cat1 = new
SnapshotReportCategory();
cat1.setCategoryName("cat1"); cat1.setNameValuePairs(rnv1);
});
report.setCategories(new SnapshotReportCategory[] { cat1
return report;
}
}
The Report class extends CloupiaNonTabularReport to override the
getReportType() and getReportType() methods to make the report as bar chart.
public class SampleBarChartReport extends CloupiaNonTabularReport
{
private static final String NAME = "foo.dummy.bar.chart.report"; private
static final String LABEL = "Dummy Bar Chart";
// returns the implementation class
@Override
public Class getImplementationClass() { return SampleBarChartReportImpl.class;
}
//The below two methiods are very important to shown as Bar cahrt in the GUI.
//This method returns the report type for bar chart shown below.
@Override
public int getReportType() {
return ReportDefinition.REPORT_TYPE_SNAPSHOT;
}
//This method returns the report hint for bar chart shown below
@Override
public int getReportHint()
{
return ReportDefinition.REPORT_HINT_BARCHART;
}
//bar charts will be display in summary if it returns true
@Override
public boolean showInSummary()
{
return true;
}
}
```

Developing Line Chart Reports

Open Automation enables you to develop non-tabular reports such as line charts. Line chart is a trending report. The HistoricalDataSeries Class provides historical information, where DataSample array is the set of values within the given time frame (fromTime, toTime).

Developing a line chart is similar to developing a plain tabular report, and you should follow the same basic procedures.

Step 1 Extend CloupiaNonTabularReport . Override getReportType and return *REPORT_TYPE_HISTORICAL*.

Step 2 Implement HistoricalReportGeneratorIf. For the line chart report, data can be provided by the source class.

```
public class SampleLineChartReportImpl implements HistoricalReportGeneratorIf
{
@Override
```

```
public HistoricalReport generateReport(ReportRegistryEntry reportEntry,
ReportContext repContext,String durationName, long fromTime, long toTime)
throws Exception {
HistoricalReport report = new HistoricalReport();
report.setContext(repContext); report.setFromTime(fromTime);
report.setToTime(toTime); report.setDurationName(durationName);
report.setReportName(reportEntry.getReportLabel());
int numLines = 1; HistoricalDataSeries[] hdsList = new
HistoricalDataSeries[numLines];
HistoricalDataSeries line1 = new HistoricalDataSeries();
line1.setParamLabel("param1");
line1.setPrecision(0);
// createDataset1() this method use to create dataset. DataSample[] dataset1 =
createDataset1(fromTime, toTime); line1.setValues(dataset1);
hdsList[0] = line1; report.setSeries(hdsList); return report;
}
//implementation for method createDataset1()
private DataSample[] createDataset1(long start, long end) { long interval =
(end □ start) / 5;
long timestamp = start; double yValue = 1.0;
DataSample[] dataset = new DataSample[5]; for (int i=0; i<dataset.length; i++)
{
DataSample data = new DataSample(); data.setTimestamp(timestamp);
data.setAvg(yValue);
timestamp += interval; yValue += 5.0;
dataset[i] = data;
}
return dataset;
}
}
```

The line chart report extends the CloupiaNonTabularReport class and overrides the getReportType() method.

```
public class SampleLineChartReport extends CloupiaNonTabularReport {
// report name and report label is defined. private static final String NAME =
"foo.dummy.line.chart.report";
private static final String LABEL = "Dummy Line Chart";
//Returns implementation class
@Override
public Class getImplementationClass() { return
SampleLineChartReportImpl.class;
}
//This method returns report type as shown below
@Override
public int getReportType() {
return ReportDefinition.REPORT_TYPE_HISTORICAL;
}
}
```

Developing Pie Chart Reports

Open Automation enables you to develop non-tabular reports such as pie charts. A single Open Automation pie chart is not generally suited to handling more than one category, so be aware that the instructions and sample code provided here are intended to create a pie chart featuring only one category. The data set generated below for the pie chart represents five slices, each slice's value is specified as (i+1) * 5.

Developing a pie chart is similar to developing a plain tabular report, and you should follow the same basic procedures.

Note A single Open Automation pie chart is not generally suited to handling more than one category. The instructions and sample code provided here create a pie chart featuring one category and five slices.

Step 1 Extend CloupiaNonTabularReport.

Example:

Step 2 Override getReportType(), and return *REPORT_TYPE_SNAPSHOT*.

Step 3 Override getReportHint(), and return *REPORT_HINT_PIECHART*.

Example

```
public class SamplePieChartReport extends CloupiaNonTabularReport
{
//Returns implementation class
@Override
public Class getImplementationClass() { return SamplePieChartReportImpl.class;
}
//Returns report type for pie chart as shown below
@Override
public int getReportType() {
return ReportDefinition.REPORT_TYPE_SNAPSHOT;
}
//Returns report hint for pie chart as shown below
@Override
public int getReportHint()
{
return ReportDefinition.REPORT_HINT_PIECHART;
}
}
```

```
public class SamplePieChartReportImpl implements SnapshotReportGeneratorIf {
@Override
public SnapshotReport getSnapshotReport(ReportRegistryEntry reportEntry,
ReportContext context) throws Exception { SnapshotReport report = new
SnapshotReport(); report.setContext(context);
report.setReportName(reportEntry.getReportLabel());
report.setNumericalData(true); report.setDisplayAsPie(true);
report.setPrecision(0);
//creation of report name value pair goes ReportNameValuePair[] rnv = new
ReportNameValuePair[5]; for (int i = 0; i < rnv.length; i++)
{
(i+1) * 5);
}
rnv[i] = new ReportNameValuePair("category" + i,
//setting of report category goes SnapshotReportCategory cat = new
SnapshotReportCategory();
cat.setCategoryName(""); cat.setNameValuePairs(rnv);
report.setCategories(new SnapshotReportCategory[] { cat
});
return report;
}
}
```

Developing Heat Map Reports

A heat map represents data with cells or areas in which values are represented by size and/or color. A simple heat map provides an immediate visual summary of information.

The instructions provided in this section show how to create a heat map report showing three sections, each of which is split into four equal "child" sections, where `i` sets the size up to 25. Developers can continue to split sections into sections by extending the approach described here.

Developing a heat map report is similar to developing a plain tabular report, and you should follow the same basic procedures. There are a few important differences. To create a heat map, you must:

Step 1 Extend CloupiaNonTabularReport by following the example provided here:

Example:

```
public class BarChartReport extends CloupiaNonTabularReport  {

    private static final string NAME = "foo.dummy.heatmap.report";
    private static final string LABEL = "Dummy Heatmap Chart";
```

Step 2 To create a heat map report with three sections, with each section split further into four sections, follow the example provided in this code:

Example:

```
for (int i=0; i<3; i++) {
  String parentName = "parent" + i;
  HeatMapCell root = new HeatMapCell();
  root.set.Label(parentName);
  root.setUnUsedChildSize(0.0);

  //create child cells within parent cell
  HeatMapCell[] childCells = new HeatMapCell[4];
  for (int j=0; j<4; j++) {
    HeatMapCell child = new HeatMapCell();
    child.setLabel(parentName + "child" + j);
    child.stValue((j+1)*25); //sets color, the color used
    //for each section is relative, there is a scale in the UI
    child.setSize(25);  //sets weight
    childCells[j] = child;
  }
  root.setChildCells(childCells);
  cells.add(root);
}
```

For additional examples of successful heatmap code, refer to com.cloupia.feature.foo.heatmap.DummyHeatmapReport and com.cloupia.feature.foo.heatmap.DummyHeatmapReportImpl .

Developing Summary Reports

Open Automation enables you to develop your own Summary reports. The summary report is considered a non-tabular report. Although it is a summary report in function, you can determine whether or not to display this report in the summary panel.

Developing a summary report is similar to developing a plain tabular report, and you should follow the same basic procedures. There are a few important differences. To create a summary report, you must:

Before you begin

Step 1 To extend CloupiaNonTabularReport, follow the example provided here:

Example:

```
public class DummySummaryReport extends CloupiaNonTabularReport  {

    private static final string NAME = "foo.dummy.summary.report";
    private static final string LABEL = "Dummy Summary";
```

Step 2 Override getReportType() and getReportHint(), using this code snippet:

Example:

```
@Override
public int getReportType()
{
    return ReportDefinition.REPORT._TYPE_SUMMARY;
}

/**
* @return report hint
*/
@Override
public int getReportHint()
{
    return ReportDefiniton.REPORT_HINT_VERTICAL_TABLE_WITH_GRAPHS;
}
```

Step 3 Define how data will be grouped together.

Example:

```
model.addText("table one key one", "table one property one", DUMMY_TABLE_ONE);
model.addText("table one key two", "table one property two", DUMMY_TABLE_ONE);

model.addText("table two key one", "table two property one", DUMMY_TABLE_TWO);
model.addText("table two key two", "table two property two", DUMMY_TABLE_TWO);
```

Step 4 Optional: To display a Graph or Chart in a summary panel, follow the example code provided here.

Use this code in the summary chart report if you want the chart to appear in the summary panel; the default is NOT to display the report in this panel. Refer to the Bar Chart topic for more detail.

Example:

```
//NOTE: If you want this chart to show up in a summary report, you need
//to make sure that this is set to true; by default it is false.
@Override
public boolan showInSummary()
{
    return true;
}
```

For additional examples of successful summary report code, refer to com.cloupia.feature.foo.summary.DummySummaryReport and com.cloupia.feature.foo.summary.DummySummaryReportImpl.

Developing Form Reports

You can utilize the Open Automation form framework to build a form that occupies the space of a report. Such form reports, which consume the space of an entire tab in the UI (normally reserved for reports) are also called "config forms". The form report is considered a non-tabular report. To a developer, it resembles a report action.

Developing a form report is similar to developing a plain tabular report, and you should follow the same basic procedures. There are a few important differences.

Step 1 To extend CloupiaNonTabularReport, follow the example provided here:

Example:

```
public class DummyFormReport extends CloupiaNonTabularReport  {

    private static final string NAME = "foo.dummy.form.chart.report";
    private static final string LABEL = "Dummy Form Report";
```

Step 2 Set up getReportType and isManagementReport, referring to this code snippet:

Make sure that isManagementReport returns true. If you return false, the UI will not show your form.

Example:

```
@Override
public int getReportType()
{
    return ReportDefinition.REPORT_TYPE_CONFIG_FORM;
}

@Override
public boolean isManagementReport()
{
    return true;
}
```

Step 3 Extend the **CloupiaPageAction** class to define an action that will trigger the form layout.

For the form report, the Report implementation class will be different from other report implementations.

Example:

```
@override
```

```
public void definePage(Page page, ReportContext context) {
  //This is where you define the layout of your action.
  //The easiest way to do this is to use this "bind" method.
  //Since I already have my form object, I just need to provide
  //a unique ID and the POJO itself. The framework will handle all the other details.
  page.bind(formId, DummyFormReportObject.class);
  //A common request is to hide the submit button which normally comes for free with
  //any form. In this particular case, because this form will show as a report,
  //I would like to hide the submit button,
  // which is what this line demonstrates
  page.setSubmitButton("");
}
```

When the user clicks the Submit button in the UI, the method validatePageDate (shown in the following step) is called.

Step 4 Set up validatePageDate as shown in this code example:

Example:

```
@Override
public int validatePageData(Page page, report Context context,
WizardSession session) throws exception {
  return PageIf.STATUS_OK;
}
```

For additional examples of successful form report code, refer to:

- com.cloupia.feature.foo.formReport.DummyFormReport
- com.cloupia.feature.foo.formReport.DummyFormReportAction
- com.cloupia.feature.foo.formReport.DummyFormReportObject.

Managing Report Pagination

Cisco UCS Director provides the CloupiaReportWithActions and PaginatedReportHandler classes to manage data split across several pages, with previous and next arrow links.

To implement the pagination tabular report, implement the following three classes:

- Report class which extends CloupiaReportWithActions
- Report source class which provides data to be displayed in the table
- Pagination report handler class

Step 1 Extend CloupiaReportWithActions.java in the Report file and override the getPaginationModelClass and getPaginationProvider methods.

```
//Tabular Report Source class which provides data for the table
@Override
public Class getPaginationModelClass() { return DummyAccount.class;
}
//New java file to be implemented for handling the pagination support.
@Override
public Class getPaginationProvider() { return FooAccountReportHandler.class;
```

```
}
Override the return type of the isPaginated method as true.
@Override
public boolean isPaginated() { return true;
}
```

Step 2 Override the return type of the getReportHint method as ReportDefinition REPORT_HINT_PAGINATED_TABLE to get the pagination report.

```
@Override
public int getReportHint(){
return ReportDefinition.REPORT_HINT_PAGINATED_TABLE;
}
```

Step 3 Extend PaginatedReportHandler.java in the FooAccountReportHandler handler and override the appendContextSubQuery method.

- Using the Reportcontext, get the context ID.
- Using the ReportRegistryEntry, get the management column of the report.
- Using the QueryBuilder, form the Query.

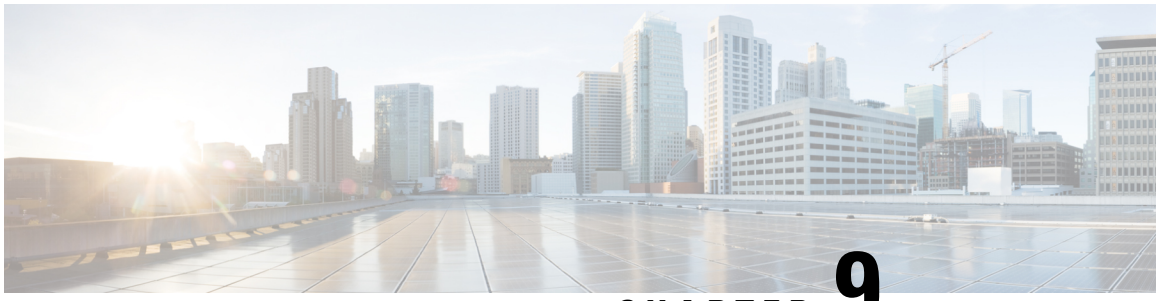
```
 @Override
public Query appendContextSubQuery(ReportRegistryEntry
entry,TabularReportMetadata md, ReportContext rc, Query query)
{
logger.info("entry.isPaginated()::::::"+entry.isPaginated())
;
String contextID = rc.getId();
if (contextID != null && !contextID.isEmpty()) { String str[] =
contextID.split(";"); String accountName = str[0];
logger.info("paginated context ID = " + contextID); int mgmtColIndex =
entry.getManagementColumnIndex(); logger.info("mgmtColIndex :: " +
mgmtColIndex); ColumnDefinition[] colDefs = md.getColumns(); ColumnDefinition
mgmtCol = colDefs[mgmtColIndex]; String colId = mgmtCol.getColumnId();
logger.info("colId :: " + colId);
//sub query builder builds the context id sub query (e.g. id = 'xyz')
QueryBuilder sqb = new QueryBuilder();
//sqb.putParam()
sqb.putParam(colId).eq(accountName);
//qb ands sub query with actual query (e.g. (id = 'xyz') AND ((vmID = 36) AND
//(vdc = 'someVDC')))
if (query == null) {
//if query is null and the id field has actual value, we only want to return
//columnName = value of id
Query q = sqb.get();
return q;
} else {
QueryBuilder qb = new QueryBuilder(); qb.and(query, sqb.get());
return qb.get();
}
} else {
return query;
}
}
```

Querying Reports using Column Index

Step 1 Extend PaginatedReportHandler.java in the FooAccountReportHandler handler.

Step 2 Override the appendContextSubQuery method:

```
@Override
public Query appendContextSubQuery(ReportRegistryEntry entry,
TabularReportMetadata md, ReportContext rc, Query query)
```



CHAPTER 9

Managing Tasks

This chapter contains the following sections:

Tasks

Workflow Tasks provide the necessary artifacts to contribute to the Task library maintained by Cisco UCS Director. The task can be used in a Workflow definition.

At a minimum, a task should have the following classes:

- A class that implements the TaskConfigIf interface.
- A class that extends and implements methods in the AbstractTask class.

TaskConfigIf

A class that implements this interface becomes a Task's input. That is, a task that wants to accept inputs for its execution shall depend on a class that implements **TaskConfigIf**. The class that implements this interface should also contain all the input field definitions appropriately annotated for prompting the user. The class should also have JDO annotations to enable the Platform runtime to persist this object in the database.

A sample Config class is shown in the sample code.

AbstractTask

A task implementation must extend the **AbstractTask** abstract class and should provide implementation for all the abstract methods. This is the main class where all the business logic pertaining to the task goes. The most important method in this class, where the business logic implementation will be scripted, is **executeCustomAction( )**. The rest of the methods provide sufficient context to the Platform runtime to enable

the task to appear in the Orchestration designer tree and to enable the task to be dragged and dropped in a Workflow.

Developing a TaskConfigIf

To develop a task, you must first implement **TaskConfigIf**. During the process of setting up the task configuration interface, you must determine what data is required to perform your task.

In the following example, **EnableSNMPConfig** exposes details of the process of developing a **TaskConfigIf**. The **Enable SNMP** task is designed to enable SNMP on a Cisco Nexus device.

To proceed, you must have the IP address of the Nexus device, the login, and the password.

You see the annotation at the beginning of **EnableSNMPConfig**.

```
@PersistenceCapable(detachable= "true", table = "foo_enable_snmp_config")
public class EnableSNMPConfig implements TaskConfigIf
{
```

You must provide a `PersistenceCapable` annotation with a table name that is prefixed with your module ID. You must follow this convention; because Cisco UCS Director prevents a task from being registered if you try to use a table name that is not prefixed with your module ID.

Next, see the following fields:

- **handler name**
- **configEntryId**
- **actionId**

```
public static finald String HANDLER_NAME = "Enable SNMP for Nexus";

//configEntryId and actionId are mandatory fields
@Persistent
private long   configEntryId
@Persistent
private long   actionId
```

The handler name is the name of the task. The name should be a unique string; you will create problems if you use the same handler name in multiple tasks.

Each task must have a **configEntryId** and **actionId**, exactly as shown above. You must have corresponding getter and setters for these two fields. These two fields are absolutely mandatory; you must have these fields in your config object.

Next, you see the data actually needed to perform the task:

```
//This is the ip address for the Nexus device on which you want to enable SNMP.
@FormField(label = "Host IP Address", help = "Host AP Address", mandatory = true,
     type = FormFieldDefinition.FIELD_TYPE_EMBEDDED_LOV,
     lovProvider = ModuleConstants.NEXUS_DEVICES_LOV_PROVIDER)
@USerInputField(type = ModuleConstants.NEXUS_DEVICE_LIST)
@Peristent
private String   ipAddress   = "";

@FormField(label = "Login", help = "Login", mandatory = true
@Persistent
private String   login;

@FormField(label = "Password", help = "Password", mandatory = true
```

```
@Persistent
private String   password;
```

As you review the code sample above, note that the developer needs the following:

- The IP address of the device.

  In this example, an LOV is used to get this IP address. See Annotations, on page 29 for more information about annotations and LOVs.

- The login and password, which the user must enter.

  To obtain these, use the form field annotations to mark these fields as data that will be provided by the user.

- Getters and setters for each of these fields.

Once the config object is completed, you must mark it for Java Data Object (JDO) enhancement.

Before you begin

You must have the Cisco UCS Director Open Automation software development kit (SDK).

Step 1 Include a jdo.files file in the same package as your config objects.

See the jdo.files and packaging in the SDK example. Note that the jdo.files must be named exactly in this way.

Step 2 In the jdo.files, specify all the classes that need to go through JDO enhancement.
The build script supplied with the SDK will complete JDO enhancement for you if you have executed this step properly.

What to do next

The handler object is where you actually execute your custom code. A handler object must implement **AbstractTask**. The `executeCustomAction` method enables you to retrieve the corresponding config object that you developed previously to execute your code.

Developing an Abstract Task

When your config object is ready, you must extendAbstractTask to actually use the new config object. This example shows the EnableSNMPTask.

At this point, you should look at this method: `executeCustomAction`.

```
public void executeCustomAction(CustomActionTriggerContext context, CustomActionLogger
actionLogger) throws Exception
{
   long configEntryId = context.getConfigEntry().getConfigEntryID();
   //retrieving the corresponding config object for this handler
   EnableSNMPConfig config = (EnableSNMPConfig) context.loadConfigObject();
```

`executeCustomAction` is where the custom logic takes place. When you call `context.loadConfigObject()`, you can cast it to the config object that you defined earlier. This process allows you to retrieve all the details that you need to perform your task. This example shows that after getting the config object, the SSH APIs are used to execute the enable SNMP commands.

When a workflow is rolled back, a task must provide a way to undo the changes it has made. This example shows the use of a change tracker:

```
//If the user decides to roll back a workflow containing this task,
//then using the change tracker, we can take care of rolling back this task (i.e.,
//disabling snmp)
context.getChangeTracker().undoableResourceAdded("assetType", "idString",
SNMP enabled", "SNMP enabled on " + config.getIpAddress(),
new DisableSNMPNexusTask().getTaskName(), new DisableSNMPNexusConfig(config));
```

The rollback code informs the system that the undo task of Enable SNMP task is the Disable SNMP task. You provide the undo config object and its name. The rest of the arguments are about logging data, which you might or might not want to provide.

DisableConfig actually takes place in the **EnableConfig**. In this case, the enable config contains the device details, so when the `Disable SNMP` task is called, you know exactly which device to disable SNMP on.

You must also implement `getTaskConfigImplementation`. This example instantiates an instance of the config object in returning it:

```
@Override
public TaskConfigIf getTaskConfigImplementation() {
   return new EnbleSNMPConfig();
}
```

Note Make sure that you specify the config object that you intend to use with this task.

What to Do Next: Include this task in your module to make it ready for use in Cisco UCS Director.

About Schedule Tasks

If you need to develop a purge task or aggregation task, or some other kind of repeatable task, you can use the Schedule Task framework, which includes the following components:

- **AbstractScheduleTask**
- **AbstractCloupiaModule**

AbstractScheduleTask

Your task logic should be placed in the `execute()` method of this class. Provide your module ID and a string that describes this task to get started. You must provide your own module ID, or the module will not be registered properly.

For more information, refer the DummyScheduleTask class in the foo module.

```
public DummyScheduleTask(){
super("foo");
}
```

Adding/Removing Schedule Tasks

AbstractCloupiaModule has an add and remove schedule task API. Typically, in the `onStart()` implementation of your **AbstractCloupiaModule**, you would instantiate your tasks and register them with the add method by calling the addScheduleTask method in your module class as follows:

```
addScheduleTask(new DummyScheduleTask());
```

For more information, refer the FooModule.java class.

Registering Custom Workflow Inputs

You can develop your own input types in Cisco UCS Director. For more information, refer to *Cisco UCS Director Orchestration Guide, Release 4.1*. However, they must be prefixed with your module ID. See Developing a TaskConfigIf, on page 54, in which an additional annotation is used to specify a custom workflow input.

```
public static final String NEXUS_DEVICE_LIST = "foo_nexus_device_list";
@UserInputField(type = ModuleConstants.NEXUS_DEVICE_LIST)
```

In this example, ModuleConstants.NEXUS_DEVICE_LIST resolves to `foo_nexus_device_list`.

Before you begin

Develop the required TaskConfigIf and the AbstractTask components for your custom workflow.

What to do next

Register a custom workflow output. See Registering Custom Task Output, on page 57.

Registering Custom Task Output

You can enable a task to add an output.

Before you begin

See the EmailDatacentersTask to see an example of how to create custom task outputs.

SUMMARY STEPS

1. Implement the method `getTaskoutputDefinitions()` in the task implementation and return the output definitions that the task is supposed to return.
2. Set the output from the task implementation.

DETAILED STEPS

Step 1 Implement the method `getTaskoutputDefinitions()` in the task implementation and return the output definitions that the task is supposed to return.

```
@Override
public TaskOutputDefinition[] getTaskOutputDefinitions() {
   TaskOutputDefinition[] ops = new TaskOututDefiniton[1];
```

```
   ops[0] = FooModule.OP_TEMP_EMAIL_ADDRESS;
   return ops;
]
```

Step 2 Set the output from the task implementation.

```
@Override
public void executeCustomAction(CustomActionTriggerContext context,
CustomerActionLogger action Logger) throws Exception
{

long configEntryId = context.getConfigEntry().getConfigEntryId();
//retrieving the corresponding config object for this handler
EmailDatacentersConfig config = (EmailDatacentersConfig context.loadConfigObject();

if (config == null)
{
    throw net Exception("No email configuration found for custom Action"
        + context.getAction().getName
        + "entryId" + configEntryId);
}

|.......
 .......

try
{
    context.saveOutputValue(OutPutConstants.OUTPUT_TEMP_EMAIL_ADDRESS, toAddresses);
```

Consuming Custom Output as Input in Other Tasks

This section describes how output can be used as input in another task. This section uses some aspects of the example in the previous section. The output definition is defined as follows:

```
@Override
public TaskOutputDefinition[] get TaskOutputDefinitions() {
    TaskOutputDefinition[] ops = new TaskOutputDefinitions[1};
//NOTE: If you want to use the output of this task as input to another task. Then the second
 argument
//of the output definition MUST MATCH the type of UserInputField in the config of the task
 that will
//be recieving this output. Take a look at the HelloWorldConfig as an example.
ops[0] = new TaskOutputDefinition(
      FooConstants.EMAIL_TASK_OUTPUT_NAME,
      FooConstants.FOO_HELLO_WORLD_NAME,
      "EMAIL IDs");
return ops;
}
,
```

The example defines an output with the FooConstants.EMAIL_TASK_OUTPUT_NAME name, and with the FooConstants.FOO_HELLO_WORLD_NAME type. To configure another task that can consume the output as input, you must make the types match.

So, in the new task that consumes FooConstants.FOO_HELLO_WORLD_NAME as input, you must enter the following in the configuration object:

```
//This field is supposed to consume output from the EmailDatacentersTask.
//You'll see the type in user input field below matches the output type
```

```
//in EmailDatacentersTasks's output definition.
@FormField(label = "name", help = "Name passed in from a previous task", mandatory = true)
@UserInputField(type = FooConstants.FOO_HELLO_WORLD_NAME)
@Persistent
private String     login;
```

The type in the `UserInputField` annotation matches the type that is registered in the output definition. With that match in place, when you drag and drop the new task in the Cisco UCS Director **Workflow Designer**, you can map the output from one task as input to the other task while you are developing the workflow.

Consuming Output from Existing Tasks as Input

This section shows how to consume output from built-in workflow tasks as input to your custom task. This process is similar to setting up custom outputs to be consumed as input in one important way: the configuration object of your task must have a field whose type is exactly the same as the type of the output that you want.

SUMMARY STEPS

1. Choose **Policies > Orchestration > Workflows**, and then click **Task Library**.
2. Find the task that you want to add, and then choose it to see the information displayed under the heading: **User and Group Tasks: Add Group**.
3. Pick the appropriate Type value from the Outputs table.
4. Specify the Type value in the UserInputField.
5. Configure the mapping as you develop your workflow, using the **User Input Mapping to Task Input Attributes** window as you add an action to the workflow, or edit related information in the workflow.

DETAILED STEPS

Step 1 Choose **Policies > Orchestration > Workflows**, and then click **Task Library**.

Tip Press **Cntl–Find** to locate tasks in the very long list that appears. For example, entering Group takes you directly to **User and Group Tasks**.

Step 2 Find the task that you want to add, and then choose it to see the information displayed under the heading: **User and Group Tasks: Add Group**.

Tip Press**Cntl–Find** to locate tasks in the very long list that appears. For example, entering Group takes you directly to **User and Group Tasks**.

The crucial type data is provided in the Outputs table, the last table provided under the heading.

Table 3: Add Group - Outputs Table

Output	Description	Type
OUTPUT_GROUP_NAM	Name of the group that was created by admin	gen_text_input
OUTPUT_GROUP_ID	ID of the group that was created by admin	gen_text_input

Step 3 Pick the appropriate Type value from the Outputs table.

The goal is to obtain the Type value that will be matched to the Task. In the example, the task consumes the group ID, so you know that the Type is *gen_text_input*.

Step 4 Specify the Type value in the UserInputField.

Example:

```
@FormField(label = "Name", help = "Name passed in from previous task",
mandatory = true)
@UserInputField(type="gen_text_input")
@Persistent
private String       name;
```

Note You could also use *@UserInputField(type = WorkflowInputFieldTypeDeclaration.GENERIC_TEXT)*. This is equivalent to using *@UserInputField(type="gen_text_input")*. You may find it easier to use *type = WorkflowInputFieldTypeDeclaration.GENERIC_TEXT* which uses the constants defined in the SDK.

The last step is to configure the mapping properly when you are developing your workflow.

Step 5 Configure the mapping as you develop your workflow, using the **User Input Mapping to Task Input Attributes** window as you add an action to the workflow, or edit related information in the workflow.

Verifying the Custom Task Is In Place

Assuming that your module is working properly, you can verify that the custom task is in place by opening the Cisco UCS Director Task Library and verifying that the task appears in it.

SUMMARY STEPS

1. In Cisco UCS Director, choose **Policies** > **Orchestration**, and then choose the **Workflows** tab.
2. In the **Workflows** tree directory, navigate to a workflow in which the task appears, and and then choose that workflow row.
3. With workflow selected, click **Workflow Designer**.
4. Verify that the task of interest appears in the list of available tasks and in the graphic representation of the tasks in the workflow.

DETAILED STEPS

Step 1 In Cisco UCS Director, choose **Policies** > **Orchestration**, and then choose the **Workflows** tab.

The **Workflows** tab displays a table that lists all available workflows.

Step 2 In the **Workflows** tree directory, navigate to a workflow in which the task appears, and and then choose that workflow row.

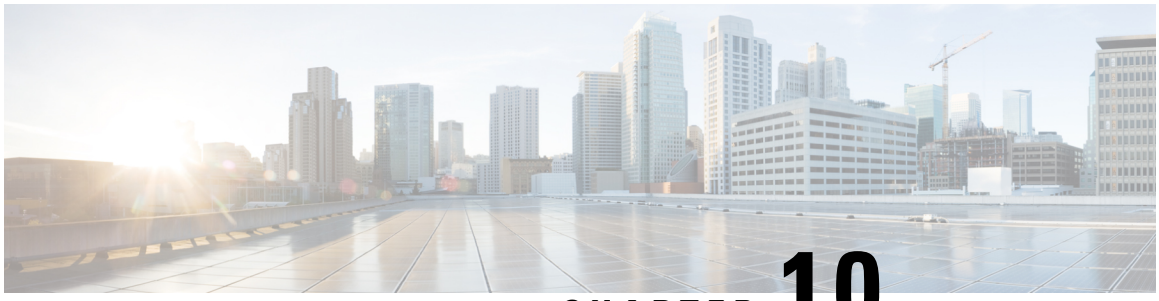
To facilitate navigation, use the Search option in the upper right-hand corner, above the table, to navigate to the workflow.

Additional workflow-related controls appear above the workflows table.

Step 3 With workflow selected, click **Workflow Designer**.

The **Workflow Designer** screen opens, displaying an **Available Tasks** list and the Workflow Design graphic view.

Step 4 Verify that the task of interest appears in the list of available tasks and in the graphic representation of the tasks in the workflow.



CHAPTER 10

Managing Menus

This chapter contains the following sections:

Menu Navigation

Cisco UCS Director uses menu navigation to determine what reports and forms to display in the UI. For more information on the subject of report locations, refer to Specifying the Report Location, on page 40.

The leftNavType field specifies the type of navigation to be used in your menu item.

The value none means that:

- No navigation is required.
- The context map rule associated with the menu item will use type = 10, name = "global_admin". (Important!)

Tip When the leftNavType is set to none, the type value and name value for the context map rule associated with the menu item comes in handy when you need to register your reports to this menu location.

If the leftNavType is backend_provided, you must provide an implementation of com.cloupia.model.cIM.AbstractTreeNodesProviderIf that populates the left hand navigation tree.

Each node of the navigation tree must provide the following elements:

- A label
- The path to an icon to show in the UI (optional)

Note The size of the icon should be 24 x 24 pixels and the format of the icon should be PNG.

- The context type (for more details, see the section about registering report contexts)
- The context ID (this will become the report context ID that you may use when generating tables)

The navigation tree must be associated with a menu ID. When registering the tree provider, use the corresponding menu ID.

Table 4: System Menu ID for Virtual Account

Menu	ID
Compute	0
Storage	1
Network	2

Table 5: System Menu ID for Physical Account

Item	ID
Compute	50
Storage	51
Network	52

Defining a Menu Item

Step 1 Option 1: Add a new menu item underneath an existing folder; in this case, the one called **Virtual**.

When adding a menu item into an existing menu category, you first have to locate the menuid of the category to which you want to add the item. In the example, we add the new menu item under "Virtual", which has the menuid of 1000. Take note of the parent menu item with just the menuid filled in: this is all you need in order to signal that you are placing your menu item into an existing category. The new menu item is placed into the children field.

Example:

```
<menu>
 <!-- this shows you how to add a new menu item underneath virtual -->
 <menuitem>
   <menuid>1000</menuid>
     <children>
       <menuitem>
          <menuid>12000</menuid>
          <label>Dummy Menu 1</label>
          <path>dummy_menu_1/</path>
          <op>no_check</op>
          <url>modules/GenericModule.swf</url>
          <leftNavType>backend_provided</leftNavType>
       </menuitem>
   </children>
 </menuitem>
```

Step 2 Option 2: Add an entirely new menu item into the UI.

If you are defining an entirely new menu item, provide all the details as shown in the example. First provide all the details for the menu category, then add all the child menu items underneath it. The example here shows a menu two levels deep, but in theory you can go as deep as you want. The best practice is to create menus no more than three levels deep.

Example:

```
<!-- entirely new menu -->
<menu>
  <menuitem>
     <menuid>11000</menuid>
     <label>Sample Category</label>
     <path>sample/</path>
     <op>no_check</op>
     <children>
       <menuitem>
         <menuid>11001</menuid>
         <label>Sample Menu 1</label>
         <path>Sample_menu_1/</path>
         <op>no_check</op>
         <url>modules/GenericModule.swf</url>
         <leftNavType>backend_provided</leftNavType>
       </menuitem>
     </children>
  </menuitem>
<menu>
```

What to do next

Register the menus.

Registering a Menu Item

For Open Automation, menu registration is handled automatically. As a developer, you only need to name the xml file of your menu as `menu.xml`, then package it as part of your module. Ensure that the `menu.xml` file is at the top level of the module jar file.

Before you begin

Define a new menu item under either a new or an existing folder.

Registering Report Contexts

This topic focuses on adding new report contexts. When developing new menu items, new report contexts are crucial: you must register new unique contexts, you CANNOT use existing contexts.

The Open Automation documentation about defining menu navigation briefly mentions that you need to provide a report context type when building your left hand navigation tree provider.

Report contexts are used by the system to determine which reports can be displayed at any point in the UI. For more background information, refer to the documentation on specifying report location: Specifying the Report Location, on page 40. See also the list of existing report context data in Appendix B, on page 91.

For open automation, there are APIs in place to auto-generate a new report context. Refer to `com.cloupia.feature.foo.FooModule` for examples on registering report contexts and menu providers.

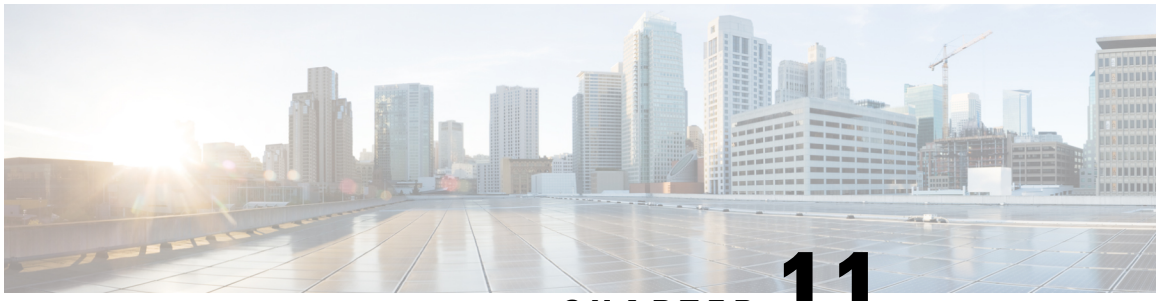
Tip Auto generated report contexts are not portable. This means that if you deploy your module in one instance of UCSD and the same module in another instance of UCSD, the auto-generated report context you get in each instance may have different values. Thus, any code you write that uses those duplicate values will not necessarily work! To avoid such problems, use the ReportContextRegistry to register report contexts and retrieve them.

Use com.cloupia.model.cIM.ReportContextRegistry.register(String name, String label), and take a look at the javadocs and sample code for more detail.

Refer to code samples and the Specifying Report Location document to see how these report contexts ultimately end up being used.

Before you begin

Open Automation developers who need to register report contexts should first talk to a UCSD lead. The UCSD lead can provide you with a block of integers reserved exclusively for your use. This will guarantee that any report contexts you define are unique. When you have your block, you can use ReportContextRegistry.register(int type, String name, String label) to register the new context.



CHAPTER 11

Managing Trigger Conditions

This chapter contains the following sections:

Trigger Conditions

To create a trigger for a specific purpose, you must have a trigger condition that is correctly defined. If a trigger condition does not already exist, you have to implement it. Likewise, if the appropriate and necessary components of the condition are not yet defined, then you can implement them using the information provided here.

In the Create Trigger Wizard (found under **Policies** > **Orchestration** > **Triggers**, at the **Specify Conditions** step, you should have the options available to set up the new trigger condition.

A trigger is composed of two components:

- An implementation of **com.cloupia.service.cIM.inframgr.thresholdmonitor.MonitoredContextIf**.
- At least one implementation of **com.cloupia.service.cIM.inframgr.thresholdmonitor.MonitoredParameterIf**.

The **MonitoredContextIf** is supposed to describe the object that is to be monitored and supply a list of references to the object. When you use the **Edit Trigger** > **Specify Conditions** element of the Wizard, you should see controls and related options that allow you to select the object and the references to it. For example, the **MonitoredContextIf** might be used to monitor the "Dummy Device" objects and to return a list of all the Dummy Devices available.

The **MonitoredParamterIf** is used in the definition of a trigger condition as follows:

- It provides the specific parameter to be examined. For example, it could be a parameter representing the status of the particular Dummy Device (for example, ddTwo) as defined by the **MonitorContextIf**.
- It supplies the operations that can be applied to the parameter. Typical operations include, for example:,
  - less than
  - equal to
  - greater than

  (The appropriate operations depend on the implementation.)

- It supplies a list of values, each of which can be logically compared against the parameter to activate the trigger.

So, for example, a trigger condition such as "Dummy Device ddTwo Status is down" can be logically tested as a condition. If the monitored Status parameter renders the statement True, the trigger condition is met.

Adding Trigger Conditions

Before you begin

Refer to the Open Automation javadocs for details on the implementation of the interface.

Step 1 Implement a **MonitoredContextIf** and all the applicable **MonitoredParameterIf**s.

```
public class MonitorDummyDeviceStatusParam implements MonitoredParameterIf {
@Override
public String getParamLabel() {
//this is the label of this parameter shown in the ui
return "Dummy Device Status";
}
@Override
public String getParamName() {
//each parameter needs a unique string, it's a good idea to //prefix each
parameter
//with your module id, this way it basically guarantees //uniqueness
return "foo.dummy.device.status";
}
@Override
public FormLOVPair[] getSupportedOps() {
//this should return all the supported operations that can be //applied to
this parameter
FormLOVPair isOp = new FormLOVPair("is", "is");
FormLOVPair[] ops = { isOp };
return ops;
}
@Override
public int getValueConstraintType() {
return 0;
}
@Override
public FormLOVPair[] getValueLOVs() {
//this should return all the values you want to compare against //e.g.
threshold values
FormLOVPair valueUP = new FormLOVPair("Up", "up");
FormLOVPair valueDOWN = new FormLOVPair("Down", "down");
FormLOVPair valueUNKNOWN = new FormLOVPair("Unknown", "unknown");
FormLOVPair[] statuses = { valueDOWN, valueUNKNOWN, valueUP };
return statuses;
}
@Override
public int getApplicableContextType() {
//this parameter is binded to MonitorDummyDeviceType, so it needs //to return
the same
//value returned by MonitorDummyDeviceType.getContextType()
DynReportContext dummyContextOneType =
ReportContextRegistry.getInstance().getContextByName(FooConstants.DUMMY_CONTEX
T_ONE);
return dummyContextOneType.getType();
}
```

```
@Override
public String getApplicableCloudType() {
return null;
}
@Override
public int checkTrigger(StringBuffer messageBuf, int contextType,
String objects, String param, String op, String values) {
//you want to basically do if (objects.param op values) { //activate } else {
not activate }
//first step, you'd look up what objects is pointing to, usually objects
should be an identifier
//for some other object you actually want
//in this example, objects is either ddOne (dummy device) or ddTwo, for
simplicity's sake, we'll
//say ddOne is always up and ddTwo is always down
if (objects.equals("ddOne")) {
if (op.equals("is")) {
//ddOne is always up, so trigger only gets activated when "ddOne is up"
if (values.equals("up")) {
return RULE_CHECK_TRIGGER_ACTIVATED;
} else {
return RULE_CHECK_TRIGGER_NOT_ACTIVATED;
}
} else {
return RULE_CHECK_ERROR;
}
} else {
if (op.equals("is")) {
//ddTwo is always down, so trigger only gets activated when "ddTwo is not up"
if (values.equals("up")) {
return RULE_CHECK_TRIGGER_NOT_ACTIVATED;
} else {
return RULE_CHECK_TRIGGER_ACTIVATED;
}
} else {
return RULE_CHECK_ERROR;
}
}
}
}
public class MonitorDummyDeviceType implements MonitoredContextIf {
@Override
public int getContextType() {
//each monitored type is uniquely identified by an integer
//we usually use the report context type
DynReportContext dummyContextOneType =
ReportContextRegistry.getInstance().getContextByName(FooConstants.DUMMY_CONTEX
T_ONE);
return dummyContextOneType.getType();
}
@Override
public String getContextLabel() {
//this is the label shown in the ui
return "Dummy Device";
}
@Override
public FormLOVPair[] getPossibleLOVs(WizardSession session)
//this should return all the dummy devices that could potentially be monitored
//in this example i only have two dummy devices, usually the value should be
an identifier you can use
//to reference back to the actual object
FormLOVPair deviceOne = new FormLOVPair("ddOne", "ddOne");
FormLOVPair deviceTwo = new FormLOVPair("ddTwo", "ddTwo");
FormLOVPair[] dummyDevices = { deviceOne, deviceTwo };
```

```
return dummyDevices;
}
@Override
public String getContextValueDetail(String selectedContextValue) {
//this is additional info to display in the ui, i'm just returning a dummy
string
return "you picked " + selectedContextValue;
}
@Override
public String getCloudType(String selectedContextValue) {
// TODO Auto□generated method stub
return null;
}
}
```

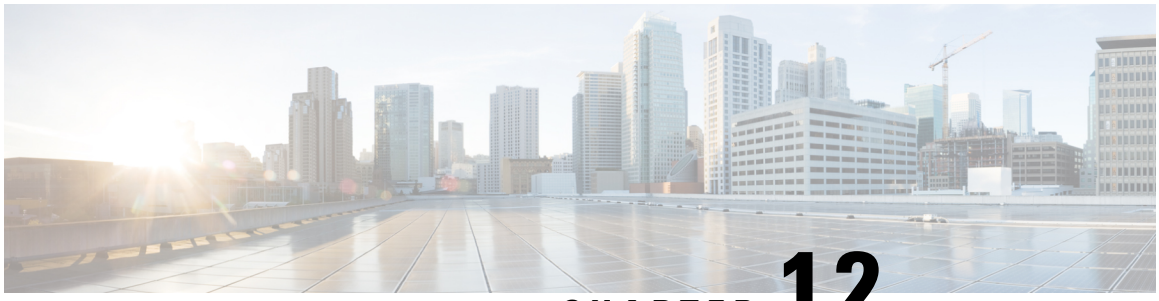
Step 2 Register the trigger condition into the system.

`com.cloupia.service.cIM.inframgr.thresholdmonitor.MonitoringTriggerUtil` has a static method for this purpose.

```
// adding new monitoring trigger, note, these new trigger components
// utilize the dummy context one i've just registered
// you have to make sure to register contexts before you execute
// this code, otherwise it won't work
MonitoringTrigger monTrigger = new MonitoringTrigger(
new MonitorDummyDeviceType(),new MonitorDummyDeviceStatusParam());
MonitoringTriggerUtil.register(monTrigger);
menuProvider.registerWithProvider();
```

a) Group your **MonitoredContextIf** and its **MonitoredParameterIf**s together into a **com.cloupia.service.cIM.inframgr.thresholdmonitor.MonitoringTrigger**.

b) Register the monitoring trigger with the utility.



CHAPTER 12

Managing REST API

This chapter contains the following sections:

The REST API

Once you develop your own Cisco UCS Director features as modules using the Cisco UCS Director Open Automation tool, you can develop and expose REST API support for the modules in the Cisco UCS Director GUI.

The following are terms used when using the Cisco UCS Director REST API:

- MO—The entities are instantiated as managed objects (MOs). The create or update operation must target a specific MO. MOs are exposed through the API.
- Class—Templates that define the properties and states of objects in the management information tree.
- Attribute—An attribute is a persistent piece of information that characterizes all objects in the same class.
- MoReference—An annotation providing the path reference of the MO.

To expose the REST API, create a new MO for the REST API support and register it with the connector. This enable developers to view the registered REST APIs with the respective MO entities in the Cisco UCS Director GUI. Developers can perform CRUD operations on connectors using the REST APIs.

You can execute each API by using the **REST API Browser** or the **REST Client**. See Cisco UCS Director REST API Getting Started Guide for detailed steps.

Identifying Entities

Identify MOs for REST API support from features and the objects' reports.

For example, you can retrieve all the cloud accounts in a particular physical datacenter or all the VDCs created in a particular cloud. Thus datacenter, cloud Account, and VDC are some of the entities modelled as objects that can be retrieved.

Configuring a POJO Class for REST API Support

Step 1 Implement a `TaskConfigIf` interface for each POJO class to indicate that the POJO class is exposed through REST.

Step 2 Annotate each POJO with the `XmlRootElement` tag.

Note Ensure that the POJO name is the same as the one defined for the POJO in the MO resource path. For example:

```
@XmlRootElement(name=”foo”)
```

Step 3 Annotate the identifier field of each POJO with the `@MoReference` tag.

For example,

```
(@MoReference(path=”foo.ID.account.ID.sample)
```

Step 4 Specify the POJO path in the MO definitions file (`<featurename>-api.mo` file) and in the properties file named as `<featurename>-url-mapping.properties`For example:

```
foo=foo.*.account.*.sample.*
```

This path specifies the location of the POJO in the tree hierarchy (for example: `datacenter.ID`, `cloud.ID`, `vdc.ID`).

Note You can use camel-casing in the keywords. Do not use special characters in the keywords.

Step 5 Register a `MoPointer` for each POJO that includes the API resource path, resource class, and adaptor of the MO. For example:

```
FooAdaptor fooAdaptor = new fooAdaptor ();
MoPointer p = new MoPointer("foo", "foo.ID.account.ID.sample",fooAdaptor, Foo.class);

parser.addMoPointer(p);
```

Input Controllers

Every Config file has a *Controller* that can be used to validate specific fields and modify the appearance and behavior of form inputs.

When to Use Controllers

Use controllers in the following scenarios:

- To implement complex show and hide GUI behavior including finer control of lists of values, tabular lists of values, and other input controls displayed to the user.
- To implement complex user input validation logic.

With input controllers you can do the following:

- **Show or hide GUI controls:** You can dynamically show or hide various GUI fields such as checkboxes, text boxes, drop-down lists, and buttons, based on conditions. For example, if a user selects UCS Manager from a drop-down list, you can prompt for user credentials for UCS Manager or change the list of values (LOVs) in the drop-down list to shown only available ports on a server.
- **Validate form fields:** You can validate the data entered by a user. For invalid data entered by the user, errors can be shown. The user input data can be altered before it is persisted in the database or before it is persisted to a device.
- **Dynamically retrieve a list of values:** You can dynamically fetch a list of values from the Cisco UCS Director database or data sources and use them to populate GUI form objects.

Marshalling and Unmarshalling UI Objects

A Controller object is always associated with a Config object. Controllers work in two stages, *marshalling* and *unmarshalling*. Both stages have two substages, *before* and *after*. These four substages correspond to four Action methods in the Controller Object. To use a controller, you marshall (control UI form fields) and unmarshall (validate user inputs) the related GUI form objects using the controller's methods.

The following table summarizes these stages.

Stage	Sub-stage
Marshalling — Used to hide and unhide form fields and for advanced control of LOVs and tabular LOVs.	**beforeMarshall** — Used to add or set an input field and dynamically create and set the LOV on a page (form). **afterMarshall** — Used to hide or unhide an input field.
Unmarshalling — Used for form user input validation.	**beforeUnmarshall** — Used to convert an input value from one form to another form, for example, to encrypt the password before sending it to the database. **afterUnmarshall** — Used to validate a user input and set the error message on the page.

To validate fields with a controller:

1. In the Config file, use an annotation to set validate equal to `true`.
2. Implement one or more Action calls in the Controller.

Following are examples of a Config and its corresponding Controller.

Config:

Following is the start page of an example UI. The file ends with the `Config` method and implements the `TaskConfigIf` interface. `@FormField` annotations are used to design the front end.

```
@FormController(value="com.cloupia.feature.compute.api.config.controller.ComputeAccountSpecificConfigController")
@XmlRootElement(name = "ComputeAccount")
@PersistenceCapable(detachable = "true", table = "compute_account_device")
public class ComputeAccountSpecificConfig implements TaskConfigIf{
public static final String HANDLER_NAME = "Compute Account Controller";
public static final String HANDLER_LABEL = "Compute Account Controller";

public static final String TYPE_STANDARD = "0";
public static final String TYPE_ADVANCED = "1";
public static final String TYPE_CUSTOM = "2";

@Persistent
private long actionId;
@Persistent
private long configEntryId;

private int modelID;
}
```

```
// Create a controller by extending the AbstractObjectUIController.
public class ComputeAccountSpecificConfigController extends AbstractObjectUIController {

private static Logger logger = Logger.getLogger(ComputeAccountSpecificConfigController.class);
@Override
public void beforeMarshall(Page page, String id, ReportContext context, Object pojo) throws
 Exception
{
}

}
```

Controller:

Name the controller file `<ConfigName>Controller`, where <*ConfigName*> is the base name of the configuration file. In this way the framework can fetch only this particular controller.

The `AbstractObjectUIController` has four methods you can override to control UI input. These methods are called before and after both marshalling and unmarshalling, and have different functions with respect to validation.

Before Marshall

Before page data is loaded, the `beforeMarshall` method is called to validate the loaded page data.

Example:

```
Public void beforeMarshall (Page page, String id, ReportContext context, Object pojo)
throws Exception
{
  logger.info ("In Controller before Marshall " + context.getId());
  ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
  logger.info(" before Marshall ");
  page.setEmbeddedLOVs(id + ".chargeDuration", getDurationLOV());
}
```

After Marshall

After page data is loaded, the `afterMarshall` method is called to validate the fields and hide on any fields.

Example:

```
Public void afterMarshall (Page page, String id, ReportContext context, Object pojo)
throws Exception
{
  ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
```

```
  page.setEditable (id + ".accountName", false);
  String accountName = context.getId();
}
```

Before Unmarshall

Before the UI is loaded, the `beforeUnmarshall` method is called.

```
Public void beforeUnmarshall (Page page, String id, ReportContext context, Object pojo)
 throws Exception
{
}
```

After Unmarshall

After form submission, the `afterUnmarshall` method is called. This method does page validation.

Example:

```
Public void afterUnmarshall(Page page, String id, ReportContext context, Object pojo)
throws Exception
{
 ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
 if (page.isPageSubmitted()) {
  String accountName = config.getAccountName();
  String gateWayAddress = ipAddressBlock.getDefGw();
  String Size = ipAddressBlock.getSize();
  String toAddress = calculateIPRange(fromAddress, Size);
  if (!validateAccountName(accountName)) {
   page.setPageMessage("Invalid account name. Please use only characters. Special
characters are not allowed.");
   throw new Exception ("Invalid account name. Please use only characters. Special
characters are not allowed.");
  }
 }
}
```

Implementing a Workflow Task

Each workflow task has a configuration class and a respective task handler for the task. The configuration class and the task handler implement the `TaskConfigIf` and `AbstractTask` classes, respectively.

To implement the workflow tasks, the framework must accommodate two scenarios:

- The existence of a workflow configuration class and its handler in the presence of a front-end POJO that is used to get the database information. For example: `DummyAccount`, `DummyAccountCreateConfig.class`, and `DummyAccountHandler.java`.
- The existence of a workflow configuration class and its handler in the absence of a front-end POJO. For example: `DummyAccountCreateConfig`.

Log Files

The API logs are stored in a `logfile.txt` file located in the `/op/infra/inframgr` directory.

The log file includes the following information:

- `INFO` (the severity keyword)

- The date-timestamp in the UTC format: `yyyy/MM/dd-HH:mm:ss,SSS`.
- The Java class where instrumentation is implemented.
- The name of the instrumented action.

See Cisco UCS Director Open Automation Troubleshooting Guide for scenarios that you may encounter and the recommended ways to handle them.

Examples

Example 1

The following code snippet provides registration for REST API support using an adaptor-based handler:

```
/* A REST adaptor is used to handle the CRUD operations for resources.
* You can extend the adaptor functionality by inheriting the WFTaskRestAdaptor.
* You can override the executeCustomAction, getTaskConfigImplementation, getTaskName and
getTaskOutputDefinitions methods according to need.
*/
WFTaskRestAdaptor restAdaptor = new WFTaskRestAdaptor();
/* MoPointer is a placeholder to register the REST APIs.
* @param0 is a mandatory field, and is used to define a resource name.
* @param1 is a mandatory field, and is used to define a ResourceURL.
* @param2 is a mandatory field, and is used to define a restAdaptor.
* @param3 is a mandatory field, and is used to define the resource config class.
* If you don't want to allow the read operation for an API, use the following constructor:
* MoPointer(String name, String path, MoResourceListener moListener, Class moModel, boolean
 isMoPersistent, boolean isReadAPISupported)
* mopointer must be registered in order for the API to display in REST API browser.
*/
MoPointer p = new MoPointer("ComputeAccount", "ComputeAccount", restAdaptor,
HelloWorldConfig.class);
/*
* The createOARestOperation method is used to register REST API operations through the Open
 Automation connector.
* @param0 is a mandatory field, and is used to define an operation name.
* @Param1 is a mandatory field, and is used to define a resource handler name. The handler
 name is used for handling the REST API operations.
* @param2 is a mandatory field, and is used to define a resource config class. The resource
 config class is required for executing a handler operation. .
*/
p.createOARestOperation("CREATE_OA", ComputeAccountCreateConfig.HANDLER_NAME,
ComputeAccountCreateConfig.class);
p.createOARestOperation("DELETE_OA", ComputeAccountDeleteConfig.HANDLER_NAME,
ComputeAccountDeleteConfig.class);
p.createOARestOperation("UPDATE_OA", ComputeAccountUpdateConfig.MODIFY_HANDLER_NAME,
ComputeAccountUpdateConfig.class);
/*
* Category is used for the REST API browser folder structure.
*/
p.setCategory(ComputeConstants.REST_API_FOLDER_NAME);
/*
* The registered REST APIs are communicated to the framework through a MoParser. Loading
REST APIs in the framework is mandatory.
*/
parser.addMoPointer(p);
```

Example 2

The following code snippet provides registration for REST API support using a Listener-based handler.

```
/** REST Listener is used to handle the CRUD operations for the Resource.
* You can extend the Listener functionality by inheriting the AbstractResourceHandler.
* You can override the methods createResource, updateResource, deleteResource and query
according to need.
*/
ComputeResourceAPIListner nPolcyList = new ComputeResourceAPIListner();
/* MoPointer is a placeholder to register the REST APIs.
* @param0 is a mandatory field, and is used to define a resource name.
* @param1 is a mandatory field, and is used to define a ResourceURL.
* @param2 is a mandatory field, and is used to define a Listener implementation class.
* @param3 is a mandatory field, and is used to define the resource config class. mandatory
 field.
* If you don't want to allow the read operation for an API, use the following constructor:
* MoPointer(String name, String path, MoResourceListener moListener, Class moModel, boolean
 isMoPersistent, boolean isReadAPISupported)
* mopointer must be registered in order for the API to display in REST API browser.
*/
MoPointer p = new MoPointer("ComputeResource", "ComputeResource", new
ComputeResourceAPIListner, ComputeAccountListnerConfig.class);
p.setSupportedOps(MoOpType.CREATE, MoOpType.UPDATE, MoOpType.DELETE);
p.setCategory(ComputeConstants.REST_API_FOLDER_NAME);
parser.addMoPointer(p);
```

Example 3

The following code snippet provides the configuration classes for workflow task actions without the front-end POJO for the read operation.

```
WFTaskRestAdaptor adaptor = new WFTaskRestAdaptor();
boolean isMoPersistent = false;
p = new MoPointer("ComputeAccountConfig" ,"ComputeAccount", adaptor, null, isMoPersistent);

p.createOARestOperation("CREATE_OA", ComputeAccountCreateConfig.class);
parser.addMoPointer(p);
```

Note If you pass the parameter as null, the REST API will not support the READ operation.

The following list suggests URIs for the MO defined in all the provided examples.

- URI for GET—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and `<name>` is the MO pointer name specified during the registration.

  Exception: The GET URI for a configuration class that does not have a front end POJO is handled as a default operation by Cisco UCS Director.

- URI for POST—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and `<name>` is the MO pointer name specified during the registration (`ComputeAccountCreateConfig` in the example). Provide the configuration POJO (`ComputeAccountCreateConfig`) as an XML payload in the HTTP request body.

- URI for UPDATE—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and where `<name>` is the MO pointer name specified during the registration. Specify the XML form of the configuration POJO in the HTTP request body.

- URI for DELETE—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and where `<name>` is the MO pointer name specified during the registration. Specify the XML form of the configuration POJO in the HTTP request body.

Invoking the REST API Using a Python Script

The Cisco UCS Director REST APIs can be invoked from an external system. This section provides an example of how to invoke the REST API from a Python script.

The example showcases only the set of APIs available in Cisco UCS Director after uploading and enabling the Open Automation (OA) compute module zip file. Other Cisco UCS Director REST APIs can be invoked in similar fashion. Cisco UCS Director partners can develop their own OA modules and expose REST APIs for their custom OA module features. The REST API for the custom OA module feature is available after uploading and enabling it on the Cisco UCS Director server.

Finally, REST APIs exposed through such a custom OA module can be executed from a Python script like the one that follows.

Prerequisites

Before running the following script, you must meet the following prerequisites:

On the Cisco UCS Director server, upload and enable the Open Automation compute module `zip` file. See The REST API, on page 71.

On the clent, do the following:

- Install Python version 2, release 2.6 or later
- Install the Python `requests` http library
- Install the Python `lxml` XML library
- Acquire the server IP address and REST API Access Key for your Cisco UCS Director installation

Note The `requests` and `lxml` libraries are not available in the default Python installation, and must be installed separately. The following script uses `requests` to make HTTP REST API calls and `lxml` to construct XML payloads. You can use other libraries instead to handle these tasks.

Python Script

```
# This script demonstrates how to invoke the Cisco UCS Director XML REST API
#
# This script requires Python version 2, release 2.6 or later
# The Python 'requests' HTTP library is used to to invoke the REST API
# Cisco UCSD XML REST API payload and response are in XML format
# The Python 'lxml' XML library is used to construct the XML payload and to parse the XML
response
#
# Python executable

import sys
import requests
```

```
from lxml import etree
import logging

# Use the logging module to log events
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Read the UCSD server IP and REST key from the command line
try:
 number_of_arguments = len(sys.argv)
 if number_of_arguments != 3:
  raise Exception("Invalid number of arguments !! Please run script as 'python
script_file_name.py ipAddress UCSD_REST_KEY'")
except Exception as e:
 logger.error("Exception occurred while executing this script : " +str(e))
 sys.exit(1)
# IP address of UCSD server on which the REST API is to be invoked
IP_address = sys.argv[1]

# Headers for HTTP requests
headers = {}

# Authenticate the user with the REST key
rest_key_value = sys.argv[2]
content_type = "application/xml"
headers["X-Cloupia-Request-Key"] = rest_key_value
headers["content-type"] = content_type

custom_operation_type = None
resource_URL = None
http_request_type = None
resource_complete_URL = None
response = None
payload_data = None

def main():
 # Invoke the HTTP POST REST API operation 'CREATE_COMPUTE_ACCOUNT' to create a ComputeAccount

 # custom_operation_type = CREATE_COMPUTE_ACCOUNT,
 # resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

 custom_operation_type = "CREATE_COMPUTE_ACCOUNT"
 resource_URL = "/cloupia/api-v2/ComputeAccount"
 http_request_type = "POST"
 resource_complete_URL = "https://" + IP_address + resource_URL

 # Construct XML Payload for CREATE_COMPUTE_ACCOUNT
 cuic_operation_request = etree.Element('cuicOperationRequest')
 operation_type = etree.SubElement(cuic_operation_request, 'operationType')
 operation_type.text = custom_operation_type
 payload = etree.SubElement(cuic_operation_request, 'payload')
 compute_account = etree.Element('ComputeAccount')
 account_name = etree.SubElement(compute_account, 'accountName')
 account_name.text = 'sdk_compute'
 status = etree.SubElement(compute_account, 'status')
 status.text = 'On'
 ip = etree.SubElement(compute_account, 'ip')
 ip.text = '1.1.1.1'
 inner_text = etree.tostring(compute_account)
 payload.text = etree.CDATA(inner_text)
 payload_data = etree.tostring(cuic_operation_request)

 logger.info("Executing HTTP POST CREATE_COMPUTE_ACCOUNT REST API...")
 logger.info("payload = %s",payload_data)
```

```
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP GET REST API 'Read' operation to read all ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None
logger.info("Executing HTTP GET REST API to read all ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET
resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None
logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'UPDATE_COMPUTE_ACCOUNT' to update a ComputeAccount

# custom_operation_type = UPDATE_COMPUTE_ACCOUNT,
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

custom_operation_type = "UPDATE_COMPUTE_ACCOUNT"
resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for UPDATE_COMPUTE_ACCOUNT
cuic_operation_request = etree.Element('cuicOperationRequest')
operation_type = etree.SubElement(cuic_operation_request, 'operationType')
operation_type.text = custom_operation_type
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_account = etree.Element('ComputeAccount')
account_name = etree.SubElement(compute_account, 'accountName')
account_name.text = 'sdk_compute'
status = etree.SubElement(compute_account, 'status')
status.text = 'Off'
ip = etree.SubElement(compute_account, 'ip')
ip.text = '2.2.2.2'
inner_text = etree.tostring(compute_account)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST UPDATE_COMPUTE_ACCOUNT REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
```

```
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the ComputeAccount updated with new ip address and status
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'DELETE_COMPUTE_ACCOUNT' to delete a ComputeAccount

# custom_operation_type = DELETE_COMPUTE_ACCOUNT,
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

custom_operation_type = "DELETE_COMPUTE_ACCOUNT"
resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for DELETE_COMPUTE_ACCOUNT

cuic_operation_request = etree.Element('cuicOperationRequest')
operation_type = etree.SubElement(cuic_operation_request, 'operationType')
operation_type.text = custom_operation_type
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_account = etree.Element('ComputeAccount')
account_name = etree.SubElement(compute_account, 'accountName')
account_name.text = 'sdk_compute'
inner_text = etree.tostring(compute_account)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST DELETE_COMPUTE_ACCOUNT REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the deletion of ComputeAccount
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
```

```
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'CREATE' to create a ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource, HTTP request type = POST

resource_URL = "/cloupia/api-v2/ComputeResource"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for ComputeResource@CREATE

cuic_operation_request = etree.Element('cuicOperationRequest')
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_resource = etree.Element('ComputeResource')
account_Name = etree.SubElement(compute_resource, 'accountName')
account_Name.text = 'sdk_compute'
status = etree.SubElement(compute_resource, 'status')
status.text = 'On'
ip = etree.SubElement(compute_resource, 'ip')
ip.text = '1.1.1.1'
inner_text = etree.tostring(compute_resource)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST ComputeResource@CREATE REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the creation of ComputeResource
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP PUT REST API operation 'UPDATE' to update a ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = UPDATE

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "PUT"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for ComputeResource@UPDATE

cuic_operation_request = etree.Element('cuicOperationRequest')
payload = etree.SubElement(cuic_operation_request, 'payload')
```

```
compute_resource = etree.Element('ComputeResource')
account_name = etree.SubElement(compute_resource, 'accountName')
account_name.text = 'sdk_compute'
status = etree.SubElement(compute_resource, 'status')
status.text = 'Off'
ip = etree.SubElement(compute_resource, 'ip')
ip.text = '2.2.2.2'
inner_text = etree.tostring(compute_resource)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP UPDATE ComputeResource@UPDATE REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify ComputeResource account updated with new ip address and status
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP DELETE REST API Read operation - DELETE a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = DELETE


resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "DELETE"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP DELETE REST API to delete a specific ComputeResource resource
...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the deletion of ComputeResource 'sdk_compute'
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET
resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
```

```
 logger.info("API Response HTTP Status Code = %s", response.status_code);
 logger.info("Response content type = %s", response.headers['content-type'])
 logger.info("Response content = %s", response._content)

def invoke_rest_API(complete_URL, payload_data, http_request_type):
 requests.packages.urllib3.disable_warnings()
 response = None
 if http_request_type == "GET":
  response = requests.get(complete_URL, headers=headers, verify=False)
 elif http_request_type == "POST":
  response = requests.post(complete_URL, data=payload_data, headers=headers, verify=False)
 elif http_request_type == "PUT":
  response = requests.put(complete_URL, data=payload_data, headers=headers, verify=False)
 elif http_request_type == "DELETE":
  response = requests.delete(complete_URL, headers=headers, verify=False)
 else:
  raise Exception("Invalid HTTP request type")
 return response

if __name__ == "__main__":
 try:
  main ()
 except Exception as e:
  logger.error("Exception occurred while executing this script : " +str(e))
```

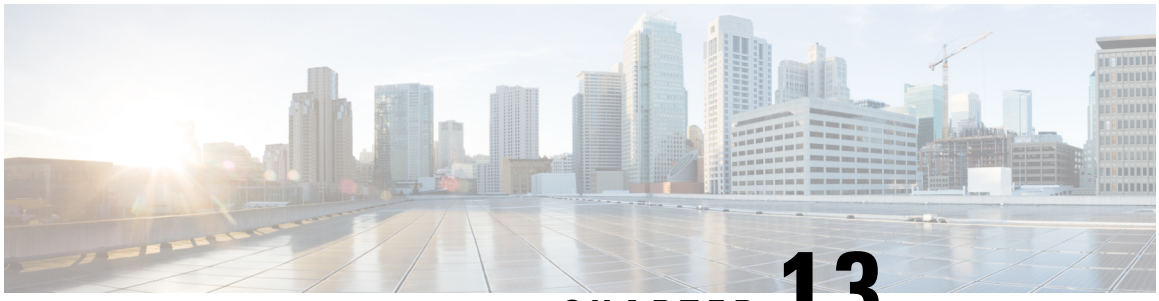
Executing the Script from the Command Line

Run the script from the command line as follows:

```
python scriptfile.py ip_address UCSD_REST_KEY
```

For example:

```
python execute_UCSD_REST_API.py 172.29.110.222 111F3D780A424C73A1C60BDD65BABB0B
```



CHAPTER 13

Change Tracking API

This chapter contains the following sections:

Change Tracking API

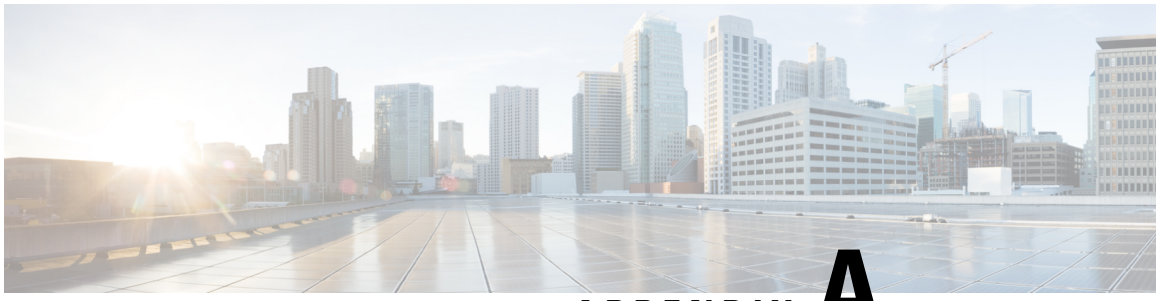
You can use the Change Tracking API to track changes that are made through their module and to record the changes in the database.

The constructor is `ChangeTrackingAPI`.

```
package com.cloupia.feature.foo.scheduledTasks;

import org.apache.log4j.Logger;

import com.cloupia.feature.foo.FooModule;
import com.cloupia.model.cIM.ChangeRecord;
import com.cloupia.service.cIM.inframgr.AbstractScheduleTask;
import com.cloupia.service.cIM.inframgr.FeatureContainer;
import com.cloupia.service.cIM.inframgr.cmdb.ChangeTrackingAPI;

/**
 * This is a simple example demonstrating how to implement a scheduled task.  This task is
 executed
 * every 5 mins and simply makes a logging statement and increments the number of times
it's been
 * executed.  It removes itself from the system once it has been executed twice.  It also
 * demonstrates how you can use the change tracking APIs to track changes made to the system.

 *
 */
public class DummyScheduleTask extends AbstractScheduleTask {

 private static Logger logger = Logger.getLogger(DummyScheduleTask.class);

 private int numTimesExecuted = 0;

 private static final long TWO_MINS = 60*1000*2;
 private static final int MAX_TIMES_EXECUTED = 2;

 public DummyScheduleTask() {
  super("foo");
 }

 @Override
```

```
 public void execute(long lastExecution) throws Exception {
  logger.info("vxvxvxvxvx - dummyTask has been executed " + numTimesExecuted + " times.");
  numTimesExecuted++;

  if (numTimesExecuted == MAX_TIMES_EXECUTED) {
   logger.info("vxvxvxvxvx - removing dummyTask");
   FooModule module = (FooModule) FeatureContainer.getInstance().getModuleById("foo");
   //NOTE: Use getTaskName() and NOT getScheduleTaskName(), it's really important
   //We distinguish the two: getTaskName is used internally by the system, where we do
   //some extra stuff to ensure uniqueness of the task name (prepend moduleID), so we need
 to
   //make sure to use this when removing tasks!
   module.removeScheduleTask(this.getTaskName());
   //use the static ChangeTrackingAPI to create an instance of ChangeRecord, these are just
 values you'd like have
   //tracked and store in the changes DB
   ChangeRecord rec = ChangeTrackingAPI.create("openAutoDeveloper",
ChangeRecord.CHANGE_TYPE_DELETE, "Dummy Task removed from System",
                "foo dummy task");
   //insert the record like so
   ChangeTrackingAPI.insertRecord(rec);
  }
 }

 @Override
 public long getFrequency() {
  return TWO_MINS;
 }

 @Override
 protected String getScheduleTaskName() {
  //usually good idea to name your task something descriptive
  return "dummyTask";
 }

}
```

Tip To view the change tracking records (CMDB) from the Cisco UCS Director GUI, choose **Administration > Integration > Change Records**.



APPENDIX A

Appendix A

This appendix contains the following sections:

Existing List of Value Tables

The following table lists existing tabular reports of available lists of values.

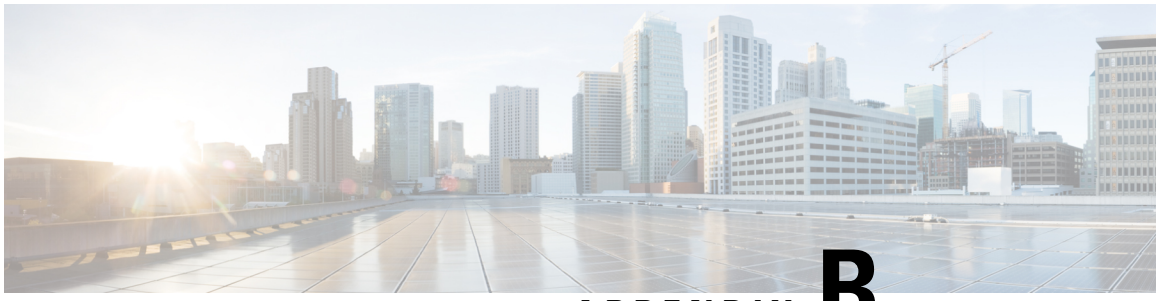
Table 6: LOV Reports for List Providers:

Report: ListProvider Name	Description
vdiMcsCatalogAllocationTypeList	VDI MCS Catalog Allocation Type Selector
netAppClusterPortAssIfGroupProvider	NetApp Cluster Port Associated IfGroup Selector
catalogList	Catalog Selector
DiskSizesList	Disk Size Selector
vdcProfileList	vDC Profile Selector
vmwareCloudNamesList	VMware Account Selector
protocolList	NetApp vFiler Protocol List
MemorySizesList	Memory Size Selector
vdiVdcList	VDI VDC Selector
emcRAIDTypesListProvider	EMC RAID Type Selector
portGroupTypeList	VMware Portgroup Type
ucsNetworkPolicyList	Cisco UCS Network Policy
iGroupTypeList	NetApp Initiator Group Type
vdiCatalogList	VDI Catalog Selector

Report: ListProvider Name	Description
hpServerBootActionTypes	HP Boot Mode
netappAllClusterAssocNonAssocIfGroupList	NetApp Cluster All IfGroup Selector
dfmStorageServiceList	NetApp OnCommand Storage Services
NetworkDevicePortProfileModeList	Networking Port Profile Mode
ucsServiceProfileList	Cisco UCS Service Profile Selector
hostNodeList	Host Node Selector
vdiMcsCatalogTypeList	VDI MCS Catalog Type Selector
netappAllClusterVLANList	NetApp Cluster vLAN Selector
vdcList	vDC Selector
netappAllClusterAggrList	NetApp Cluster Aggregate Selector
NetworkDeviceBasicList	Networking Device
netappAllClusterPortList	NetApp Cluster Port Selector
MSPGroupList	MSP Group
sizeUnitList	NetApp Size Units
amazonVMList	Amazon VM Selector
ec2AccountList	Amazon Cloud Selector
ucsBladeList	Cisco UCS Server
emcSizeUnits	EMC Size Units
vdiMemorySizesList	VDI Memory Size Selector
VMwareDVSUplinkPortgroupList	VMware DVSwitch Uplink Portgroup
hpServerAutoPowerActionTypes	Hewlett Packard (HP) Auto Power Mode
hpServerBootSourceAction	HP Boot Source
hpServerPowerSaverActionTypes	HP Power Saver Mode
hpServerPowerSaverActionTypes	HP Power Saver Mode
osTypeList	NetApp OS Type
vdiCPUsList	VDI CPU Selector
vscResizeDatastoreList	NetApp VSC Resize Datastore Selector
UcsAccountList	Cisco UCS Account

Report: ListProvider Name	Description
containerProvider	Service Container
amazonVMActions	Generic VM Action Selector
dfmGroupList	NetApp OnCommand Groups
hpServerPowerActionTypes	HP Server Power Mode
kvmVMActions	KVM VM Action Selector
netappAllClusterNodeList	NetApp Cluster Node Selector
vdiAccountList	VDI Account Selector
NetworkDevicePortProfileTypeList	Networking Port Profile Type
ec2VolumeSizeList	EC2 Volume Size Selector
userGroupsLOV	User Group
vmwarePortGroupLoVProvider	VMware Port Group
kvmVMList	KVM VM Selector

Appendix A



APPENDIX B

Appendix B

This appendix contains the following sections:

Report Context Types and Report Context Names

About Report Context Data

Use this list to find and to specify the context data of a report. It provides the Report Context Type, a unique ID number associated with each report context, followed by the Report Context Name. For basic information about reports and contexts, see the Cisco UCS Director REST API Guide.

0 global

1 cloud

2 hostnode

3 vm

4 action

5 cluster

6 services

7 group

8 user

9 event

10 global_admin

11 catalog

12 service_request

13 user_workflow

14 amz_dep_policy

15 rackspace_policy

16 compute_chassis

Appendix B

17 compute_server
18 compute_io_card
19 compute_fbi
20 compute_fbi_port
21 vdc
22 vdc_reports
23 datacenter
24 alldatacenters
25 service_profile
26 server_pool
27 vm_snapshot
28 managed_report
29 cloudsense_partner
30 cloudsense_customer
31 cloudsense_partner_admin
32 cloudsense_admin
33 storage_volumes
34 custom_actions
35 storage_aggregates
36 storage_disks
37 ucsm
38 pari
39 pari_device
40 storage_accounts
41 host_account
42 network
43 network_device
44 luns
45 network_switch_port
46 net_device_generic
47 net_device_n1k
48 net_device_fab_ic
49 net_device_n5k
50 ucs_storage_policy

51 ucs_network_policy
52 msp
53 ucs_org
54 ucs_policies
55 ucs_mac
56 ucs_boot_policy
57 netapp_dfm_prov_policy
58 netapp_dfm_prot_policy
59 netapp_dfm_service_policy
60 netapp_dfm_vfiler_template_policy
61 ucs_sp_template
62 netapp_dfm_dataset
63 rhev_datacenter
64 rhev_storage_domian
65 rhev_user
66 rhev_template
67 netapp_filer
68 ucs_wwnn
69 ucs_wwpn
70 ucs_uuid
71 ucs_iomodule
72 ucs_iomodule_port
73 netapp_v_filer
74 netapp_vfiler_volume
75 netapp_vfiler_lun
76 ucs_service_profile
77 ucs_service_profile_template
78 vmware_port_group
79 ucs_portchannel
80 netapp_initiator_group
81 netapp_initiator
82 netapp_vfiler_initiator_group
83 netapp_vfiler_initiator
84 kvm_dep_policy

85 vvd

86 ocap_module

87 hp

88 hp_server

89 ocap_module_report

90 ocap_module_file

91 ocap_module_changelog

92 ocap_module_workflowtasks

93 ocap_module_workflowinout

94 ocap_module_schedjobs

95 ocap_module_cloudsense

96 ucs_mac_block

97 compute_server_adapter_unit

98 ucs_locale

99 netapp_dfm_group

100 ucs_sp_vhba

101 ucs_sp_vnic

102 ucs_vnictemplate

103 net_device_qos_policy

104 netapp_ipspace

105 netapp_interface

106 ucs_vlan

107 vmware_network_vswitch

108 vmware_network_dvswitch

109 vmware_network_dvportgroup

110 vmware_network_vmknic

111 datastore

112 net_device_vlan

113 net_device_vsan

114 net_device_interface

115 net_device_port_profile

116 net_device_zone

117 storage_ip_proto_policy

118 standalone_rack_server_account

119 rack_server_summary
120 cimc_boot_definition
121 net_device_asa
122 net_device_asa_context
123 netapp_cifs_share
124 netapp_vfiler_cifs_share
125 netapp_qtree
126 netapp_vfiler_qtree
127 xenvdi_desktop
128 xenvdi_catalog
129 xenvdi_desktop_group
130 vvd_studio_wizard
131 vvd_studio_wizard-page
132 cimc_server_cpu
133 cimc_server_memory
134 cimc_server_pci_adapters
135 cimc_server_psu
136 cimc_server_fan
137 cimc_server_network_adapter
138 cimc_server_vhba
139 cimc_server_vhba_boottable
140 storage_cluster_controller
141 storage_cluster_account
142 storage_cluster_node
143 storage_cluster_vserver
144 storage_cluster_volume
145 cimc_server_vnic
146 tier3_group
147 emc_vnx_account
148 xenvdi_identity_pool
149 storage_cluster_lun
150 storage_cluster_igroup
151 storage_cluster_logInf
152 storage_cluster_initiator

153 storage_cluster_Qtree
154 emc_vnx_file_system
155 emc_vnx_data_mover
156 emc_vnx_mount
157 emc_vnx_cifs_share
158 emc_vnx_volume
159 emc_vnx_nfs_export
160 emc_vnx_storage_pool
161 scm_workspace
162 scm_config_account
163 emc_vnx_cifs_server
164 emc_vnx_dns_domain
165 emc_vnx_network_interface
166 emc_vnx_logical_net_device
167 net_device_brocade_300
168 vmware_network_nic
169 infra_load_balancer
170 emc_vnx_lun
171 emc_vnx_storage_pool_for_block
172 emc_vnx_raid_group
173 emc_vnx_lun_folder
174 emc_vnx_storage_group
175 emc_vnx_host
176 emc_vnx_host_initiator
177 emc_vnx_storage_processor
178 ipmi_account
179 ipmi_server
180 compute_accounts
181 hyperv_library_server
182 net_device_brocade_nos_vdx
183 vmware_image_context
184 netapp_snapmirror
185 netapp_vfiler_snapmirror
186 netapp_snapmirror_destination

187 netapp_vfiler_snapmirror_destination

188 netapp_snapmirror_schedule

189 netapp_vfiler_snapmirror_schedule

190 infra_compute_servers

191 net_device_brocade_nos

192 net_device_brocade_fos

193 datacenter_ucs_account

194 ucs_iqn_pool

195 resource_pool

196 dc_ucs_account_server

197 per_rack_server_account

198 dc_account_hp_server

199 per_account_ipmi_server

200 per_dc_ipmi_account

201 per_dc_hp_account

202 per_dc_cimc_account

203 external_service_request

204 net_device_cisco_nxos

205 net_device_cisco_nxos_n7k

206 storage_cluster_port

207 storage_cluster_ifgroup

208 storage_cluster_vlan

209 external_sr_chargeback

210 resources_chargeback

211 ucs_service_profile_bootpolicy

212 ucs_rack_mount_server

213 ucs_fabric_extender

214 ucs_fan_module

215 work_order

216 resource_alert

217 hyperv_image_context

218 hyperv_snapshot_context

219 storage_cluster_exportrule

220 ucs_fan

221 ucs_io_module_port

222 ucs_psu

223 ucs_server_dce

224 ucs_server_hba

225 ucs_server_nic

226 ucs_server_cpu

227 ucs_server_memory_unit

228 ucs_server_disk

229 ucs_vsan

230 ucs_lan_port_channel

231 ucs_fi_fc_port

232 custom_feature

233 ucs_vhbatemplate

234 ucs_vlan_group

235 emc_vmax_device

236 emc_vmax_account

237 emc_vmax_igroup

238 emc_vmax_portGroup

239 emc_vmax_storageGroup

240 emc_vmax_masking_view

241 ucs_lan_conn_policy

242 ucs_san_conn_policy

243 ucs_storage_conn_policy

244 ucs_storage_iGroup

245 ucs_decommissioned_server

246 hyperv_cluster_csv

247 hyperv_logical_network

248 hyperv_logicalnework_def

249 hyperv_vmnetwork_subnet

250 hyperv_vm_network

251 hyperv_logical_switch

252 hyperv_file_share

253 hyperv_cluster_availble_nodes

254 ucs_sp_rename

255 hyperv_logicalnework_def_subnet
256 hyperv_storage_file_server
257 hyperv_storage_array
258 hyperv_storage_provider
259 hyperv_storage_classifications
260 hyperv_storage_pool
261 hyperv_storage_pool_per_array
262 vnmc_datacenter_account
263 vnmc_account
264 ucs_fault_suppress_task
265 hyperv_host_group
266 vnmc_tenant
267 vnmc_zone
268 vnmc_acl_policy_rule
269 vnmc_vdc
270 vnmc_vapp
271 vnmc_tier
272 net_device_n3k
273 net_device_mds
274 net_device_cisco_nxos_n7k_vdc
275 emc_vmax_director
276 hyperv_native_uplink_pp
277 hyperv_native_vna_pp
278 hyperv_static_ip_pool
279 hyperv_port_classification
280 hyperv_vm_network_adapter
281 emc_vmax_thin_pool
282 emc_vmax_meta_dev
283 storage_cluster_aggregates
284 net_device_ios
285 emc_vmax_fast_policy
286 emc_vmax_storage_tier
287 netapp_cluster_cron_job
288 netapp_cluster_snapshot_policy

289 netapp_cluster_snapshot_policy_schedule
290 ucs_local_disk_config_policy
291 emc_vnx_meta_lun
292 emc_vnx_block_account
293 emc_vnx_file_account
294 netapp_cluster_volume_snapshot
295 netapp_cluster_vserver_volume_cifs
296 ucs_discovered_server
297 ucs_central
298 ucs_central_domain_group
299 ucs_central_compute_system
300 net_device_cisco_nxos_n7k_vdc_storage
301 cluster_vserver_domain
302 net_device_n9k
303 cimc_server_storage_adapter
304 cimc_server_storage_adapter_summary
305 context_type_cimc_server_storage_adapter_physical
306 context_type_cimc_server_storage_adapter_virtual
307 cimc_server_processor_unit
308 cimc_server_pci_adapter_unit
309 cimc_server_network_adapters
310 cimc_server_network_adapters_eth
311 cimc_server_new_psu
312 cluster_vserver_Ip_Host_Mapping
313 rack_server_api_supported
314 vnmc_vsg_policy
315 netapp_cluster_portset
316 cluster_vserver_sis_policy
317 cimc_server_vic_adapter
318 cimc_server_vic_adapter_vhba
319 cimc_server_vic_adapter_vnic
320 cluster_wwpn_alias
321 system_task
322 system_task_history

323 remote_agent
324 agent_tasks
400 xyz_context
425 all_pods_physical_compute
426 ucs_central_org
427 pnsc_cs_profile
428 netapp_cluster_nfs_service
429 all_pods_physical_storage
430 all_pods_physical_network
431 ucs_central_accounts
432 multi_domain_managers
433 hyperv_host_adapter_ln
434 hyperv_vnetwork_hostadapter
435 netapp_cluster_vserver_peer
436 netapp_cluster_snapmirror
437 pnsc_policy_set
438 netapp_cluster_snapmirror_policy
439 pnsc_policy_list
440 ucs_central_chassis
441 ucs_central_server
442 ucs_central_server_storage_controller
443 ucs_central_vnic_template
444 ucs_central_vhba_template
445 ucs_central_service_profile
446 ucs_central_service_profile_templ
447 ucs_central_wwpn
448 ucs_central_wwnn
449 ucs_central_mac
450 ucs_central_uuid
451 ucs_central_ippool
452 ucs_central_server_pool
453 ucs_central_lan_conn_pol
454 ucs_central_san_conn_pol
455 netapp_cluster_job

Appendix B

456 net_device_n6k

457 ucs_central_boot_policy

458 pnsc_compute_firewall

459 net_device_n1110

460 ucs_central_fan_module

461 netapp_cluster_vserver_routing_group

462 netapp_cluster_peer

463 ucs_central_vsan

464 ucs_central_vlan

465 ucs_central_rack_mount_server

466 ucs_iscsi_adapter_policy

467 ucs_network_control_policy

468 ucs_qos_policy

469 ucs_central_fex

470 whiptail_account

471 whiptail_initiator_group

472 whiptail_volume_group

473 whiptail_lun

474 whiptail_interface

475 whiptail_accela

476 whiptail_invicta

477 ucs_central_fc_adapter_policy

478 ucs_central_firmware_policy

479 ucs_central_maintenance_policy

480 ucs_central_server_pool_policy

481 ucs_central_server_pool_policy_qual

482 ucs_central_vnic_vhba_placement_policy

483 ucs_central_vhba_policy

484 ucs_central_vnic_policy

485 ucs_central_storage_policy

486 ucs_central_network_policy

487 ucs_central_local_disk_policy

488 ucs_central_iqn_pool

489 netapp_cluster_export_policy

490 ucs_central_local_service_profile

491 ucs_central_local_service_profile_templ

492 ucs_central_vnic

493 ucs_central_vhba

494 ucs_central_fabric_interconnect

495 network_static_ip_pool_policy

496 hyperv_storage_fileshare

497 hyperv_storage_lun

498 netapp_cluster_disk

499 hyperv_host_group_storage_pool

500 hyperv_host_group_storage_lun

501 pnsc_accounts

502 emc_vmax_datadev

503 emc_vmax_thindev

504 netapp_vlan_interface

90001 collector.data.collecion.policy.report

90002 collector.data.collecion.policy.assciate.report

90003 DummyAccount.generic.infra.report.6000:2

90004 foo.dummy.drilldown.interface.report

90005 VMAX System Devices

90006 System Summary

90007 VMAX Tiers

90008 VMAX Symmetrix Devices

90009 VMAX Thin Devices

90010 VMAX Meta Devices

90011 VMAX Initiator Groups

90012 VMAX Initiators

90013 VMAX Storage Groups

90014 VMAX Port Groups

90015 VMAX Masking Views

90016 VMAX Thin Pools

90017 VMAX Fast Policies

90018 VMAX Fast Controller

90019 VMAX FAST Status

90020 WHIPTAIL System Summary

90021 WHIPTAIL SSR Report

90022 WHIPTAIL Bonds Report

90023 WHIPTAIL Virtual Interfaces Report

90024 WHIPTAIL VLANs Report

90025 WHIPTAIL Physical Interfaces Report

90026 WHIPTAIL iSCSI Settings Report

90027 WHIPTAIL SSNs Report

90028 WHIPTAIL RAID Health Report

90029 WHIPTAIL Volume Groups Report

90030 WHIPTAIL LUNs Report

90031 WHIPTAIL Initiator Groups Report

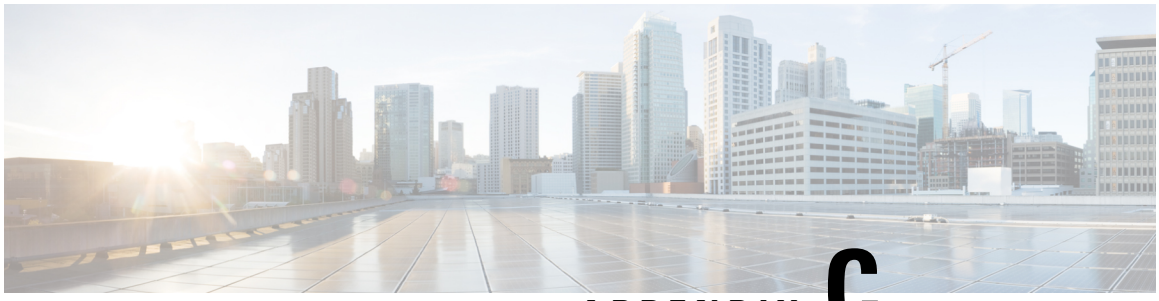
90032 WHIPTAIL FC Report

90033 cluster.system.tasks.policy.report

90034 foo.dummy.context.one

90033 cluster.system.tasks.policy.report

90034 foo.dummy.context.one



APPENDIX C

Appendix C

This appendix contains the following sections:

Form Field Types

This appendix provides a list of form field types that is used to define the type of form fields during form creation in an open automation module. For defining a form field, it is mandatory to provide the label and type of the form field.

1. FIELD_TYPE_TEXT

   The FIELD_TYPE_TEXT defines a field as text field. It is the default field type. If the field type is not defined for a form field annotation, the form field is categorized as text type by default.

   Attributes

   - **maxLength**—Specify the maximum number of character allowed in a text field.
   - **Size**—Set the size of the text field using one of the following values:
     - FIELD_SIZE_SMALL
     - FIELD_SIZE_MEDIUM
     - FIELD_SIZE_LARGE
     - FIELD_SIZE_MEDIUM_SMALL
     - FIELD_SIZE_LARGE_SMALL
     - FIELD_SIZE_SMALL_MEDIUM
     - FIELD_SIZE_LARGE_MEDIUM
     - FIELD_SIZE_SMALL_LARGE
     - FIELD_SIZE_MEDIUM_LARGE

   Sample

```
@FormField(label = "Name", help = "Name",
size =FormFieldDefinition.FIELD_SIZE_SMALL)
private String name;
```

2. FIELD_TYPE_NUMBER

   The FIELD_TYPE_NUMBER defines that a field should contain a numeric value.

   Attributes

   - **minValue**—Specify the minimum acceptable value for the numeric field. For example, 1.
   - **maxValue**—Specify the maximum acceptable value for the numeric field. For example, 65535.

   Sample

```
@FormField(label = "FIELD_TYPE_NUMBER",
type = FormFieldDefinition.FIELD_TYPE_NUMBER, minValue = 1, maxValue = 65535)
private int number;
```

3. FIELD_TYPE_TABULAR

   The FIELD_TYPE_TABULAR defines a field as a table.

   Attributes

   - **table**—Specify a name for the tabular field.
   - **multiline**—This attribute is boolean type. Set as true to allow addition of multiple lines for the table.

   Sample

```
@FormField(label = "FIELD_TYPE_TABULAR",
type = FormFieldDefinition.FIELD_TYPE_TABULAR,
table = SimpleTabularProvider.SIMPLE_TABULAR_PROVIDER, multiline = true)
private String[] plainTabularValues;
```

4. FIELD_TYPE_BOOLEAN

   The FIELD_TYPE_BOOLEAN sets a field as boolean type. If the field is selected, the field value is set as true otherwise the field value is set as false.

   Attributes

   No specific attribute for this field.

   Sample

```
@FormField(label = "FIELD_TYPE_BOOLEAN", type = FormFieldDefinition.FIELD_TYPE_BOOLEAN)
private boolean boolType;
```

5. FIELD_TYPE_LABEL

   The FIELD_TYPE_LABEL argument lets you specify a label for the field.

   Attributes

   - **htmlPopupTag**—Specify the URL that need to be loaded in the popup window.
   - **htmlPopupLabel**—Specify the label for the popup window.
   - **htmlPopupStyle**—Set the popup style for the label using one of the following values:
     - INFO_TAG

- HELP_TAG
- CUSTOM_TAG
- INFO_URL
- HELP_URL
- CUSTOM_URL

Sample

```
@FormField(type = FormFieldDefinition.FIELD_TYPE_LABEL, label = "FIELD_TYPE_LABEL",
htmlPopupTag = "http://www.cisco.com",htmlPopupLabel = "http://www.cisco.com",
htmlPopupStyle = HtmlPopupStyles.CUSTOM_URL)
private String dummyLink;
```

6. FIELD_TYPE_EMBEDDED_LOV

   The FIELD_TYPE_EMBEDDED_LOV defines the field as embedded list of values (LOV) type and allows user to select one of value from the list of values.

   Attributes

   - You can specify either **lov** or **lovProvider** as attribute.

   Sample

```
@FormField(label = "FIELD_TYPE_EMBEDDED_LOV", help = "Value",
type = FormFieldDefinition.FIELD_TYPE_EMBEDDED_LOV, lovProvider =
SimpleLovProvider.SIMPLE_LOV_PROVIDER)
private String value;
```

7. FIELD_TYPE_PASSWORD

   The FIELD_TYPE_PASSWORD sets a field as password. The characters in a password field are masked (shown as asterisks or circles).

   Attributes

   No specific attribute for this field.

   Sample

```
@FormField(label = "FIELD_TYPE_PASSWORD",
type = FormFieldDefinition.FIELD_TYPE_PASSWORD)
private String password;
```

8. FIELD_TYPE_DATE

   The FIELD_TYPE_DATE defines an input field that should contain a date.

   Attributes

   No specific attribute for this field.

   Sample

```
@FormField(label = "FIELD_TYPE_DATE", type = FormFieldDefinition.FIELD_TYPE_DATE)
private long dateLong;
```

9. FIELD_TYPE_DATE_TIME

   The FIELD_TYPE_DATE_TIME defines an input field that should contain a date and time.

Attributes

No specific attribute for this field.

Sample

```
@FormField(label = "FIELD_TYPE_DATE_TIME", type =
FormFieldDefinition.FIELD_TYPE_DATE_TIME)
private long dateTime;
```

10. FIELD_TYPE_MULTI_SELECT_LIST

The FIELD_TYPE_MULTI_SELECT_LIST defines an input field to accept input from the multiple values.

Attributes

- **lovProvider**—Set the list of values that need to be displayed in the input field.

Sample

```
@FormField(label = "FIELD_TYPE_MULTI_SELECT_LIST",
type = FormFieldDefinition.FIELD_TYPE_MULTI_SELECT_LIST, lovProvider =
SimpleLovProvider.SIMPLE_LOV_PROVIDER)
private String listValue;
```

11. FIELD_TYPE_HTML_LABEL

The FIELD_TYPE_HTML_LABEL defines a field as HTML label. The HTML tag are accepted as string.

Attributes

- **size**—Set the size of the HTML label.

Sample

```
@FormField(type = FormFieldDefinition.FIELD_TYPE_HTML_LABEL, label =
"FIELD_TYPE_HTML_LABEL", htmlPopupLabel = "<a href='http://www.cisco.com'>Cisco</a>")
private String dummyLink2;
```

12. FIELD_TYPE_FILE_UPLOAD

The FIELD_TYPE_FILE_UPLOAD defines a field to upload a file.

Attributes

No specific attribute for this field.

Sample

```
@FormField(label = "FIELD_TYPE_FILE_UPLOAD", type =
FormFieldDefinition.FIELD_TYPE_FILE_UPLOAD)
private String uploadFileName;
```

13. FIELD_TYPE_TABULAR_POPUP

The FIELD_TYPE_TABULAR_POPUP defines the field as a tabular popup type.

Attributes

- **table**—Specify the tabular field name (TabularPovider) that has been already registered in the open automation module.

  The following sample code of how the tabular provider is registered in the open automation module:

```
StorageModule.java(Registering  Tabular report)
cfr.registerTabularField(SimpleTabularProvider.SIMPLE_TABULAR_PROVIDER,
SimpleTabularProvider.class, "0", "0");
```

Sample

```
@FormField(label = "FIELD_TYPE_TABULAR_POPUP",
type = FormFieldDefinition.FIELD_TYPE_TABULAR_POPUP, table =
SimpleTabularProvider.SIMPLE_TABULAR_PROVIDER)
private String tabularPopup;
```

14. FIELD_TYPE_EMBEDDED_LOV_RADIO

    The FIELD_TYPE_EMBEDDED_LOV_RADIO defines the field as an embedded LOV radio buttons.

    Attributes

    - You can choose either **lov** or **lovProvider** as attribute.

    Sample

    ```
    @FormField(label = "FIELD_TYPE_EMBEDDED_LOV_RADIO",
    type = FormFieldDefinition.FIELD_TYPE_EMBEDDED_LOV_RADIO, mandatory = true, lov = {
    "Mode 1", "Mode 2", "Mode 3"}, validate = true, group="FIELD_TYPE_EMBEDDED_LOV_RADIO")
    private String modeType = "Select Mode";
    ```

15. FIELD_TYPE_HTML_TEXT

    The FIELD_TYPE_HTML_TEXT defines the field as a HTML text type.

    Attributes

    No specific attribute for this field.

    Sample

    ```
    @FormField(label = "FIELD_TYPE_HTML_TEXT", type =
    FormFieldDefinition.FIELD_TYPE_HTML_TEXT,
    editable = true, size=FormFieldDefinition.FIELD_SIZE_MEDIUM_SMALL)
    private String status = "<h1>FIELD_TYPE_HTML_TEXT</h1>";
    ```

16. FIELD_TYPE_LABEL_WITH_SPACE

    The FIELD_TYPE_LABEL_WITH_SPACE defines the field as a label with space.

    Attributes

    No specific attribute for this field.

    Sample

    ```
    @FormField(label = "FIELD_TYPE_LABEL_WITH_SPACE",
    help = "Ordering of VNICs", type = FormFieldDefinition.FIELD_TYPE_LABEL_WITH_SPACE)
    private String vnicLabel;
    ```

17. FIELD_TYPE_IMAGE_SELECT_LIST

    The FIELD_TYPE_IMAGE_SELECT_LIST defines the field that should accept selection of image from the image select list.

    Attributes

    No specific attribute for this field.

    Sample

```
@FormField(label = "FIELD_TYPE_IMAGE_SELECT_LIST", type =
FormFieldDefinition.FIELD_TYPE_IMAGE_SELECT_LIST, mandatory = false, editable = true)
private String catalogIcon;
```

18. FIELD_TYPE_BUTTON_PANEL

    The FIELD_TYPE_BUTTON_PANEL defines the field as a button panel.

    Attributes

    - **lov**—Specify the registered LOV provider name. Also, you can can directly give the values as Lov ={http,https}. For more information, refer the SimpleLovProvider.java sample LOV provider in the open automation module.

    Sample

```
@FormField(label = "FIELD_TYPE_BUTTON_PANEL",
type = FormFieldDefinition.FIELD_TYPE_BUTTON_PANEL,
lov = {"Discover Servers"}, validate = true, group = "UCSM/CIMC Common", mandatory =
false)
private String discoverServers = "Discover Servers";
```

19. FIELD_TYPE_TEXT_LINE_NUMS

    The FIELD_TYPE_TEXT_LINE_NUMS defines the field as a text field with line numbers.

    Attributes

    - **maxlength**—Specify the maximum number of character allowed in the text field.
    - **multiline**—This attribute is boolean type. Set as true to allow addition of multiple lines for the text field.
    - **size**—Set the size of the text field using one of the following values:
        - FIELD_SIZE_SMALL
        - FIELD_SIZE_MEDIUM
        - FIELD_SIZE_LARGE
        - FIELD_SIZE_MEDIUM_SMALL
        - FIELD_SIZE_LARGE_SMALL
        - FIELD_SIZE_SMALL_MEDIUM
        - FIELD_SIZE_LARGE_MEDIUM
        - FIELD_SIZE_SMALL_LARGE
        - FIELD_SIZE_MEDIUM_LARGE

    Sample

```
@FormField(label = "FIELD_TYPE_TEXT_LINE_NUMS",
help = "Error Text to validate", mandatory = false,
multiline = true, maxLength = 8192,
type = FormFieldDefinition.FIELD_TYPE_TEXT_LINE_NUMS,
size = FormFieldDefinition.FIELD_SIZE_SMALL_LARGE)
private String message;
```

20. FIELD_TYPE_LARGE_FILE_UPLOAD

The FIELD_TYPE_LARGE_FILE_UPLOAD defines a field that should allow user to upload a large file.

Attributes

No specific attribute for this field.

Sample

```
@FormField(label = "", help = "Upload a file",
mandatory = true, validate = true, type =
FormFieldDefinition.FIELD_TYPE_LARGE_FILE_UPLOAD,
annotation = "For module uploads only zip format are supported")
private String uploadFile;
```

21. FIELD_TYPE_COLORPICKER

    The FIELD_TYPE_COLORPICKER defines a field to pick a color.

    Attributes

    No specific attribute for this field.

    Sample

```
@FormField(label = "FIELD_TYPE_COLORPICKER", help = "Color",
mandatory = true, validate = true, type = FormFieldDefinition.FIELD_TYPE_COLORPICKER)
private String color;
```

Common Attributes

This section provides a list of common attributes that you can use along with the form fields to control the field activity. For example, if you want to make Name field as mandatory, you need to pass true as the mandatory attribute value.

1. **validate**—The attribute type is boolean. By default, the attribute value is **false**. If you want to validate a field, you need to pass **true** as the attribute value.
2. **hidden**—The attribute type is boolean. By default, the attribute value is **false**. If you want to hide a field in a form, you need to pass **true** as the attribute value.
3. **mandatory**—The attribute type is boolean. By default, the attribute value is **false**. If you want to make a field as mandatory, you need to pass **true** as the attribute value.
4. **editable**—The attribute type is boolean. By default, the attribute value is **false**. This attribute is only applicable for text field. If you want to make the text field as editable, you need to pass **true** as the attribute value.
5. **group**—The attribute type is string. If you want to define single field or multiple fields in a group, you need to specify the group name as the attribute value.
6. **view**—The attribute type is string. If you have multiple forms in a wizard, you need to mention the page number as the attribute value so that the field will be displayed on the specified page number. If you do not have multiple pages in a wizard, no need to use this attribute.
7. **help**—The attribute type is string. This attribute lets you specify descriptive help text for the field. If you provide help text, the text will be displayed when a user hovers the mouse pointer over the field.

Appendix C