Cisco UCS Director Open Automation Cookbook

Release 1.0
Published: April, 2015

Cisco Systems, Inc.
www.cisco.com

Cisco has more than 200 offices worldwide.
Addresses, phone numbers, and fax numbers are listed on the Cisco website at:
www.cisco.com/go/offices.

Table of Contents

1 Getting Started

The ability to let developers contribute to the Cisco UCS Director platform is made possible by Cisco UCS Director inherent modular architecture of the platform and its features. A developer can take advantage of this feature and provide customization and extension by adding several components to Cisco UCS Director.

1.1 Importing the SDK Bundle Project into the Eclipse IDE

You can find the Open Automation SDK with all other Cisco UCS Director software in the download area on Cisco.com. To download the SDK, click the link to the UCS Director SDK. Unpack the sample SDK project zip file in your file system.

To import the SDK bundle project into the Eclipse IDE, do the following:

1. Obtain the SDK Bundle archive and extract the contents to an appropriate folder.
2. Launch Eclipse.
3. Choose **File > Import**.
4. In the **Import** dialog box, choose **General > Existing Projects into Workspace**.
5. Click **Next**.
6. Choose **Select root directory** and browse to the location where you extracted the project.
7. Click **Finish**.

Everything should automatically compile without issue. You may or may not need to replace the Cisco UCS Director SDKs under the /open-auto-sdk/lib path.

1.2 Creating FOO Module

Run the Ant target of the build.xml file.
After successful completion of build, an Foo Module zip file is created.

To rename the module, perform the following steps:

Step 1: Rename the module class name.
Step 2: Modify the name in the module.properties file as:
Name=<new module name>
Step 3: Build the zip file by running ANT target.

---

2 Managing Modules

A module is the top-most logical entry point into Cisco UCS Director. To add or extend any functionality, a module needs to be developed and deployed on the Cisco UCS Director.

2.1 Creating a Module

To create a module, do the following:

1. Create a module by implementing a class that extends AbstractCloupiaModule.
2. Create tasks by implementing necessary task interfaces.
3. Package the module by running ANT.

Sample: Create a Module

```
public class FooModule extends AbstractCloupiaModule
{
       // To Register task
       @Override
       public AbstractTask[] getTasks() {
       //You can implement for the Tasks.
       }
        // we registering all top level reports
       @Override
       public CloupiaReport[] getReports() {
              CloupiaReport[] reports = new CloupiaReport[];
              ...
              return reports;
       }
       @Override
       public void onStart(CustomFeatureRegistry cfr) {
        }
  }
```

2.2 Publishing a Module

To expose a module to the platform runtime, a file named module.properties is provided along with the module. The properties file defines certain properties of the module.

The module.properties file which is provided by Cisco UCS Director should never be altered by a developer. Cisco UCS Director validates the properties file to prevent tampering.

2.2.1 Contents of the module.properties File

#id–The unique identifier for the module (recommendation: restrict this to 3 to 5 lowercase ASCII alphabet characters).

```
moduleID=foo
```

#version–The current version of the module.

```
version=1.0
```

#ucsdVersion–The version of Cisco UCS Director that this version of the module will work best with.

```
ucsdVersion=5.3.0.0
```

#category–The path where all your tasks are placed.

```
category=/foo
```

#format–The version of this module is formatted.

```
format=1.0
```

#name–A user friendly string to identify the module in the open automation reports.

```
name=Foo Module
```

#description–The short descrtion that describes what the module does.

```
description=UCSD Open Automation Sample Module
```

#contact–An email address that consumers of the module can use to request support.

```
contact=support@cisco.com
```

#key–An encrypted key that Cisco UCS Director Open Automation group will provide for validating the module.

```
key=5591befd056dd39c8f5d578d39c24172
```

2.3 Deploying a Module on Cisco UCS Director

The user interface (UI) of Cisco UCS Director 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.
Ensure that you have Shell admin access. The Shell admin access is required to restart Cisco UCS Director services for deploying or deactivate a module.

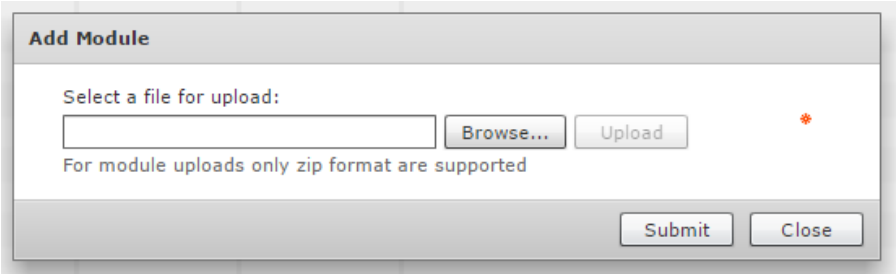
To deploy a module, perform the following steps:

Step 1. In Cisco UCS Director, choose **Administration > Open Automation**.
The **Modules** tab appears and 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	Displays which version of Cisco UCS Director best supports this module.
Contact	The contact information of the person responsible for technical

Column	Description
	support for the module.
Upload Time	The time at which the module is uploaded.
Status	The status of the module. The status includes: Enable, Disable, Active, and Inactive. A user 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 it entirely. A module is Active if Cisco UCS Director is able to successfully initialize it without 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 not included in the module.
Validated	Indicates whether the module is validated.

Step 2. Click **Add** to add a new module.
The **Add Module** dialog box appears.



Step 3. Click **Browse** to choose location of the module zip file to upload.
Step 4. Provide the required information and click **Submit**.

Important: To load the newly added module, you must enable the module and restart Cisco UCS Director services. If enabled, Cisco UCS Director attempts to initialize the module; if it is disabled, the module is ignored.

Step 5. Enable the module by choosing the module and clicking **Enable**.

Step 6. Activate the module by choosing the module and clicking **Add**.
The **Activate Module** dialog box appears.
Step 7. Click **Submit** to activate the module.
In the event of an error, a clear error message appears indicating why the module was not acceptable. A module is Active if Cisco UCS Director is able to successfully initialize the module without exception.

> Note: Active does not necessarily mean that everything in the module is working properly; it merely indicates that the module came up. Inactive indicates that when Cisoc UCS Director tried to initialize the module, a sever error prevented it from doing so. Typical causes for the Inactive flag are: the module was compiled with the wrong version of Java or a class is not included in the module.

Step 8. Stop and restart the Cisco UCS Director services:

a. Open the **Shell Menu** with the shell admin access. The **Cisco UCS Director Shell Menu** opens, prompting you to: **Select a number from the menu below**.
b. At the SELECT prompt, enter the number to stop services and enter the appropriate number to start services again.

Cisco UCS Director takes a couple of minutes to restart. When the system is ready, the module is loaded and the tasks are visible in the UI.

2.3.1 Deactivating a Module

For your changes to take effect, deactivate a module. To deactivate a module, you must stop and restart the Cisco UCS Director services in order.

To deactivate a module, perform the following steps:

Step 1. Choose the module you need to deactivate in the **Modules** tab, 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.

3 Working with a Module

A module can include the following components:

- Task—A workflow task that can be used while defining a workflow.
- Reports—Reports that appear in the Cisco UCS Director UI. Reports may contain action buttons.

- Wizard—A UI component that serves to collect inputs from the user to perform a certain action or actions.
- Trigger—A condition that, once satisfied, can be associated with some action(s). Examples: shutdown VM, start VM, and so on.

Each of the major components listed above can make use of one or more of the following sub-components:

- Object Store
- Annotations
- Lists
- Tables
- Connectors
- Logs

3.1 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 process simplifies and speeds up the development with respect to data persistence. The Object Store documentation includes sections that show how CRUD operations are realized using JDO.

3.2 Annotations

Annotations are one of the most crucial parts of module development. Most of the artifacts are driven by annotations. The annotations make the development effort easy and convenient.
Annotations are used for persistence, report generation, wizard generation, and tasks.

3.2.1 Persistence Annotations

For information about the annotations that are used for persistence, see the Marking a Class as Persistable Object section.

3.2.2 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, a drop-down list, or a check box. For more information, see the Task section.

3.3 Lists of Values (LOVs)

Lists represent the drop-down Lists of Values (LOVs) 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 a lot of prebuilt list providers that the modules can readily use to prompt input from the user.

For an example that illustrates how to use one of the list providers, see the Defining Your Own List Provider and Tasks sections.

3.3.1 Defining Your Own List Provider

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 getLOVs( ) method.

Sample: Return Array of FormLOVPair objects

```
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;
       }
}
```

3.4 Tables (Tabular Reports)

Use the existing tabular reports that are available for user selection. For more information about the tabular reports, see the Tabular Reports section.

3.5 Tasks

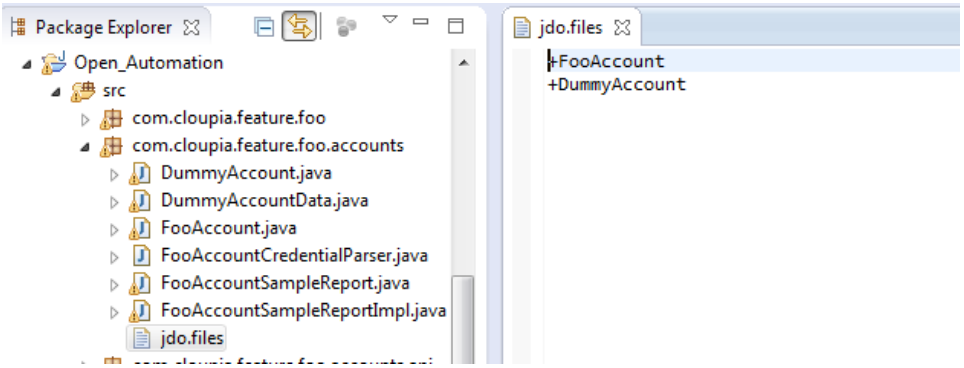
Tasks are used in workflow definition. For more information about tasks, see the Task section.

4 Marking a Class as a Persistable Object

The POJO class that needs to be persisted in the database must be defined and marked with suitable JDO annotations. After developing the JDO class, you need to:

- Implement JDO enhancement for the persistence.
- Add config class to the jdo.files.

Note: The jdo.files must be in same package as task is developed (refer the following image).



Sample:Marking Persistable Object

```
@PersistenceCapable (detachable = "true", table =
“sampleAccount”)
public class  SampleAccount
{
    @Persistent
    private String AccountName;
      @Persistent
    private String userName;
      @Persistent
    private String password;
}
```

4.1 Storing and Retrieving the Persistence Capable Object

A module that wants to persist objects into the database shall primarily make use of the Object Store APIs to perform all the CRUD operations. The platform runtime makes use of the Java Data Object (JDO) library provided by DataNucleus to abstract all the SQL operations and Object Query representation.

4.1.1 Code to Store Data

```
SampleAccount obj = new SampleAccount();
obj.setAccountName("openauto-account");
obj.setUserName("Admin");
obj.setPassword("password");
ObjStore store = ObjStoreHelper.getStore(SampleAccount.class);
store.insert(obj);
```

4.1.2 Code to Retrieve Data

```
//retrieving all objects
ObjStore store = ObjStoreHelper.getStore(SampleAccount.class);
List list = store.queryAll();

//retrieving objects with query
ObjStore store = ObjStoreHelper.getStore(SampleAccount.class);
String query = "dcName == 'Default Datacenter'";
List filerList = store.query(query);
```

5 Task

Workflow tasks provide the necessary artifacts to contribute to the task library maintained by Cisco UCS Director. The tasks are used in a workflow definition.

5.1 Content of Task

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

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

5.1.1 TaskConfigIf

The class that implements the TaskConfigIf interface contains all the input field definitions annotated for prompting the user.

Sample: Implement a Task

```
@PersistenceCapable(detachable = "true", table =
"foo_helloworldconfig")
public class HelloWorldConfig implements TaskConfigIf {
       // you can define your task implementation
}
```

5.1.2 AbstractTask

A task implementation extends the AbstractTask abstract class and provides implementation for all the abstract methods. This abstract class 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.

Sample: Abstract a Task

```
public class HelloWorldTask extends AbstractTask
{
@Override
public void executeCustomAction(CustomActionTriggerContext
context, CustomActionLogger  actionLogger) throws Exception
{
	long configEntryId =
context.getConfigEntry().getConfigEntryId();
//retrieving the corresponding config object for this handler
	HelloWorldConfig config = (HelloWorldConfig)
context.loadConfigObject();
		}
	@Override
	public TaskConfigIf getTaskConfigImplementation() {
		return new HelloWorldConfig();
	}
	@Override
	public String getTaskName() {
		return HelloWorldConfig.displayLabel;
	}
	@Override
	public TaskOutputDefinition[] getTaskOutputDefinitions() {
		return null;
	}

}
```

Register a task in the module class as shown:

```
public class FooModule extends AbstractCloupiaModule {
@Override
	public AbstractTask[] getTasks() {
		AbstractTask task1 = new HelloWorldTask();
		AbstractTask[] tasks = new AbstractTask[1];
		tasks[0] = task1;
		return tasks;
	}
...
}
```

6 Reports

In Cisco UCS Director, the Open Automation reports are used to display the data and to retrieve the data in the UI for the uploaded module.
The two ways to develop a report are:

- Plan Old Java Object (POJO)-and-Annotation approach. For more information, see the POJO-and-Annotation Approach section.
- Implementing TabularReportGeneratorIf interface. For more information, see the Tabular Reports section.

6.1 Registering a Report

After developing a report, you must register the report into the system.

Sample:Register a Report

```
public class FooModule extends AbstractCloupiaModule {

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

6.2 POJO-and-Annotation Approach

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.

To develop a report, perform the following 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.

Sample: Annotated POJO

```
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 the **com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyReportWithAction** or **com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyDrillableReport** class and 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).

- CloupiaEasyReportWithActions—Use this class when you need to assign action to report.
- CloupiaEasyDrillableReport—Use this class when you need to implement drill down report.

Both classes are used to create a report using the POJO-and-Annotation method.

Sample:POJO-based Report

Step 1: 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;
}
```

Step 2: 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 into the system to display the report in the UI. For more information, see the Registering a Report section.

6.3 Tabular Reports

Develop a tabular report using the TabularReportGeneratorIf interface.
The approach is to create an instance of TabularReportInternalModel which contains all the data you want to display in the UI.
To create a tabular report, perform the following steps:

Step 1. Create TabularReportInternalModel, populate the header, and add text values for each column to fill out a row.

Step 2. Extend the **com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaReportWithActions** or **com.cloupia.service.cIM.inframgr.reports.simplified.DrillableReportWithActions** class.

- CloupiaReportWithActions—Extend this class to create an Open Automation report with or without action and without drill-down.
- DrillableReportWithActions—Extend this class to create an Open Automation report with or without action and with drill-down.

Step 3. Specify the implementation of the data source and make sure that the ***isEasyReport***() method returns false.

Sample: 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. For more information, see the Registering a Report section.

6.4 Drillable Reports

Reports that are nested within other reports and are only reachable by drilling down are called drillable report. Drillable reports are appliacable only for the tabular reports.

Key points for creating drillable reports are:

- The report data source must be implemented through the POJO-and-Annotation approach. The report should extend the **com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaEasyDrillableReport** class.
- The report data source must be implemented using the TabularReportGeneratorIf interface. The report should extend the **com.cloupia.service.cIM.inframgr.reports.simplified.DrillableReportWithActions** 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. The best way to do this is to initialize the array of reports. For more information, see the Registering a Report section.

Sample: Drillable Reports

```
public class DummyOneSampleReport extends
DrillableReportWithActions  {
```

```
        ...
                @Override
                public CloupiaReport[] getDrilldownReports() {
                        CloupiaReport[] ddReports = new
CloupiaReport[1];
                        ddReports[0] = new SampleReport();
                        return ddReports;
                }
}
```

6.5 Report with Action

When developing reports, you can also include actions to go with those reports. The following things are required to develop a report action:

- Form Object—The form object is the POJO with JDO and form field annotation. The form object must be similar to configuration object that you developed for the report.
- Form Handler—You must place the logic for report action in the handler.

Extend the **com.cloupia.service.cIM.inframgr.reports.simplified.CloupiaPageAction** class and override the following methods in the class:

- definePage—Defines layout of the form.
- loadDataToPage—Handles loading of data in the form.
- validatePageData—Handles whatever logic you want to execute when user clicks the submit button.

Sample: Report with Action

```
Public class DummyForm{
        @FormField(label = "Name", help = "Name")
       private String name;

        @FormField(label = "Id", type =
FormFieldDefinition.FIELD_TYPE_NUMBER)
        private boolean id;
        }
```

For every form, the **Action** class needs to be implemented by extending CloupiaPageAction.

```
 public class DummyAction extends CloupiaPageAction{
       private static Logger logger =
  Logger.getLogger(DummyAction.class);
                private static final String formId =
       "foo.simple.dummy.form";
              private static final String ACTION_ID =
       "foo.simple.dummy.action";
                private static final String label = "Dummy Action";
       // it returns the action id when you will click the action button
                  @Override
```

```
	public String getActionId() {
		return ACTION_ID;
	}

// the form id will return when you will submit the form.
	public String getFormId()
	{
		return formId;
	}

//Returns report label to display in UI as report name
	@Override
	public String getLabel() {
		return label;
	}

//it returns the action type like how the form should appear from
UI.
	@Override
	public int getActionType() {
		return ConfigTableAction.ACTION_TYPE_POPUP_FORM;
	}
//if this returns false then no need to select report to appear
action button if it returns as true then
//We need to select the report to appear the action button in UI
in the registred context.
	@Override
	public boolean isSelectionRequired() {
		return false;
	}
//if this method returns as false then no need to do double click
to get report if it returns true then
//we have to do double click for getting report.
	@Override
	public boolean isDoubleClickAction() {
		return false;
	}
 // If this returns true then after clicking on action button it
will navigate to next level of the report.
	@Override
	public boolean isDrilldownAction() {
		return false;
	}
//define the form object like how the form should appear into UI.
	@Override
	public void definePage(Page page, ReportContext context) {
		page.bind(formId, DummyForm.class);
	}

//Loading data that you might want to show in the form when it is
first displayed to the user.
	@Override
public void loadDataToPage(Page page, ReportContext context,
WizardSession session) throws     Exception {
```

```
                String query = context.getId();
                      SimpleDummyForm form = new DummyForm();
                form.setName("dummy name - hardcoded for demo
purposes");
                session.getSessionAttributes().put(formId, form);
                page.marshallFromSession(formId);
                String dir =
FileManagementUtil.getDir(page.getSession());
                page.setSourceReport(formId + ".uploadFileName",
dir);
        }

// validating the data when the user presses the submit button .
@Override
public int validatePageData(Page page, ReportContext context,
WizardSession session) throws Exception {
//for page data validation put your code .
        }
        }
//Returns the label of action
        @Override
        public String getTitle() {
                return label;
        }
```

For registering an action to the report class, see the Tabular Reports section.

6.6 Pagination Report

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

1. Report class which extends CloupiaReportWithActions
2. Report source class which provides data to be displayed in the table
3. Pagination report handler class

To achieve the pagination reports, perform the following steps:

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;
}
}
```

6.7 Specifying the Report Location

To specify the exact location where your report need to appear in the UI, you must provide the following information:

- UI menu location ID
- Context map rule that corresponds to the report context of the location

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

To specify the report location, perform the following steps:

Step 1. Enable the developer menu for your session.
   1) In Cisco UCS Director, click your login name in the upper right.
   2) In the **User Information** dialog box, click the **Advanced** tab.
   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 2. 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.

   1) Find the integer value assigned to the **uiMenuTag**.
      The **uiMenuTag** tells you what your report's *getMenuID* should return.
   2) 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 3. 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 4. Instantiate 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.

6.8 Non-tabular Report

To implement the non-tabular report, implement the following two classes:

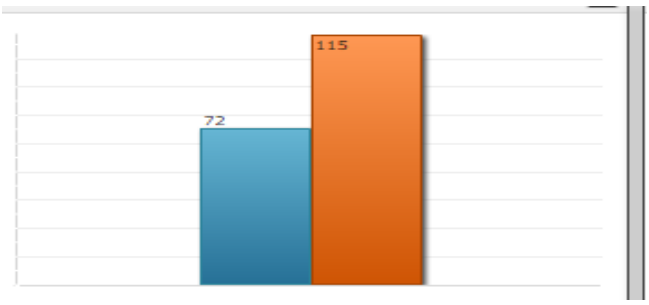
1) Report class which extends CloupiaNonTabularReport
2) Report source class

6.8.1 Bar Chart Report

To create a bar chart, you must:

1) Extend the CloupiaNonTabularReport class.
   a) Use getReportType and return REPORT_TYPE_SNAPSHOT.
   b) Use getReportHint and return REPORT_HINT_BARCHART.
2) Implement SnapshotReportGeneratorIf.
   a) For the bar chart report, data can be provided by the source class. Override the getSnapshotReport method and provide the data source. For more information, see the following sample.

The following image illustrates the graphical representation of bar chart report in the UI.



Sample: 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);

             // setting the report name value pair for the bar
chart
             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.

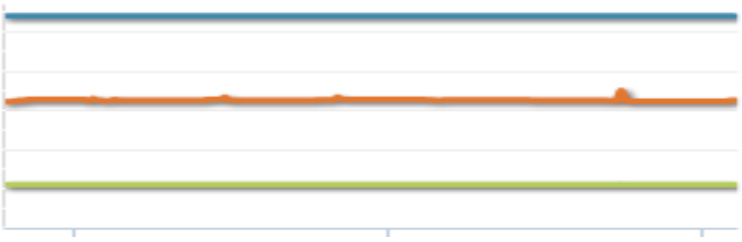
```
public class SampleBarChartReport extends CloupiaNonTabularReport
{

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;
        }

    }
```

Register the report into the system to display the report in the UI. For more information, see the Registering a Report section.

6.8.2 Line Chart Report

Line chart is the graphical representation of the report as shown in the following image.



To implement the line chart report, you must:

1) Extend the CloupiaNonTabularReport class.
   a) Use getReportType and return REPORT_TYPE_HISTORICAL.
2) Implement HistoricalReportGeneratorIf.
   a) For the line chart report, data can be provided by the source class. Override the generateReport method and provide the data source. For more information, see the following sample.

Sample: Line Chart Report

```
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;
      }
      }
```

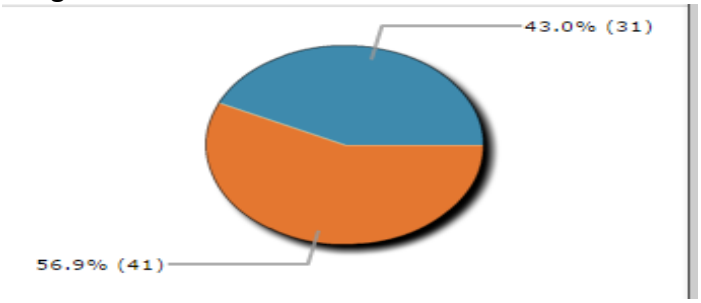
The line chart report extends the CloupiaNonTabularReport class and override the getReportType() method.

```
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;
       }

}
```

Register the report into the system to display the report in the UI. For more information, see the Registering a Report section.

6.8.3 Pie Chart Report

Pie chart is the graphical representation of the report as shown in the following image.



Developing a pie chart report is similar to developing a bar chart report, and you should follow the same basic procedures. For more information, see the Bar Chart Report section.

The important difference for creating a pie chart is:

1) Extend the CloupiaNonTabularReport class.
   a) Use getReportType and return REPORT_HINT_PIECHART.

Sample: Pie Chart Report

```
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++)
            {
                rnv[i] = new ReportNameValuePair("category" + i,
(i+1) * 5);
            }
            //setting of report category goes
            SnapshotReportCategory cat = new
SnapshotReportCategory();
            cat.setCategoryName("");
            cat.setNameValuePairs(rnv);
            report.setCategories(new SnapshotReportCategory[] { cat
});
            return report;
        }

}
```

Extend the CloupiaNonTabularReport class and override the getReportType() and getReportHint() methods.

```
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;
        }

}
```

Register the report into the system to display the report in the UI. For more information, see the Registering a Report section.

6.8.4 Heat Map Report

Open Automation enables you to develop non-tabular reports such as heat maps. 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. For more information, see the Tabular Reports section.

The important difference for creating a heat map report is:

1) Extend CloupiaNonTabularReport.
   a) Use getReportType and return REPORT_TYPE_HEATMAP.

Sample: Heat Map Report

```
public class DummyHeatmapReport extends CloupiaNonTabularReport
{
private static final String NAME = "foo.dummy.heatmap.report";
private static final String LABEL = "Dummy Heatmap";
@Override
                    public int getReportType() {
                          return
ReportDefinition.REPORT_TYPE_HEATMAP;
       }
}

public class DummyHeatmapReportImpl implements
HeatMapReportGeneratorIf{

@Override
       public HeatMapReport getHeatMapReportReport(ReportRegistryEntry
       reportEntry, ReportContext context) throws Exception {
              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);
}

}

}
```

6.8.5 Summary Report

Open Automation enables you to develop summary reports. The summary report is considered as a non-tabular report. Although it is a summary report in function, you can determine whether 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. For more information, see the Tabular Reports section.

The important difference for creating a summary report is:

1) Extend CloupiaNonTabularReport.
   a) Override getReportType() and return REPORT_TYPE_SUMMARY.
   b) Override getReportHint() and return REPORT_HINT_VERTICAL_TABLE_WITH_GRAPHS.
   c) Set isManagementReport to return True.

Sample: Summary Report

```
public class DummySummaryReport extends CloupiaNonTabularReport
{

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

       //Override getReportType() and getReportHint(), using this
code snippet:
@Override
public int getReportType()
{
    return ReportDefinition.REPORT._TYPE_SUMMARY;
}

/**
* @return report hint
*/
@Override
public int getReportHint()
{
```

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

public class DummySummaryReportImpl implements
TabularReportGeneratorIf{

@Override
public TabularReport getTabularReportReport(ReportRegistryEntry
reportEntry, ReportContext context) throws Exception {
TabularReport report = new TabularReport();
  report.setContext(context);
 report.setGeneratedTime(System.currentTimeMillis());
 report.setReportName(reportEntry.getReportLabel());

 //showing how to add two tables to your summary panel
 //the tables in summary panel are always two column tables
 SummaryReportInternalModel model = new
SummaryReportInternalModel();

 //this will be my first table
 //two rows, column one and column two values shown, note the
last value
 //this is how you group what data is grouped together
        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);

        //you'll notice the charts that show in summary panels
aren't mentioned here
        //that's because you'll need to specify in the chart
report whether or not the
        //chart should be displayed in the summary panel or not,
look in the BarChartReport
        //example for more detail

        //finally perform last clean up steps
        model.setGroupOrder(GROUP_ORDER);
        model.updateReport(report);

        return report;
}
```

```
}
```

7 Developing a New Menu

Creating a new menu item involves a series of associated tasks as follows:

- Define your menus.
- Register new menus.
- Register new report contexts.
- As necessary, write a left hand navigation tree provider.
- Write your reports and make sure they point to your new menu location with the proper context map rule.

7.1 Defining a Menu Item

When using Open Automation, the only way to add new menu is by providing a file called menu.xml. The code examples provided in this section, provides guidance regarding the task for defining a menu item.

The following code samples are included in the Open Automation SDK samples. The XML samples show the following two options for introducing menu items:

- How to insert new menu items into existing folders (for example, the Virtual folder)
- How to add entirely new menu items into the UI

Every menu node has a unique integer ID associated with it. When introducing new menu items, be aware that the following menu IDs are reserved as indicated in the table.

Menu	ID
Virtual	1000
Physical	1001
Organizations	1002
Policies	1003
Virtual/Hypervisor Policies	1007
Physical Infrastructure Policies	1008
Administration	1004

The components of a menu item are:

- label—The label displayed in the UI for the menu.
- path—The value used in the URL when navigating to the menu.

> Important: You must include a backslash at the end of this string. For example, dummy_menu_1/ or dummy_menu_2/.

- op—The permission the user must have to access this menu; no_check means everyone can access it.
- url—This should always be modules/GenericModules.swf. This field should only be populated if it is a menu item and not a category.
- leftNavType—Valid values are either none or backend_provided. For more information on lefNavType, see the Defining Menu Navigation section. This field should only be populated if it is a menu item and not a category.
- children—If the menu has sub menus, you need to add them here. The best practice is not to have more than three levels in a menu.

Sample values for the menu item components listed above are provided in the code samples.

Option 1: Add a new menu item under an existing folder.
For instance, to add a new menu item under the existing folder called Virtual, you have to locate the ID of the menu category to which you want to add the item. The ID of the Virtual folder is 1000. Take note of the parent menu item with just the menuid filled in. This is all you need to signal that you are placing your menu item into an existing category. The new menu item is placed into the children field.

Sample

```
<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>
```

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 sample. Provide all the details for the menu category and add all the child menu items under it. The example shows a menu that is two levels deep, but in theory you can go as deep as you want. The best practice is to create menus not more than three levels deep.

Sample

```
<!-- 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>
```

7.2 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* and package the xml file as part of the module. Ensure that the menu.xml file is at the top level of the module jar file.

7.3 Defining Menu Navigation

Cisco UCS Director uses menu navigation to determine what reports and forms to display in the UI. For more information about the report locations, see the Specifying the Report Location section.

The **leftNavType** field specifies the type of navigation to be used in the menu item. The value **none** means that:

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

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 needs to provide the following elements:

- Label
- Path to an icon to show in the UI (optional)
- Context type (for more details, see the registering report contexts section in Cisco UCS Director Open Automation Developer Guide.)

- Context ID (this will become the report context ID that you may use when generating tables)

The navigation tree needs to be associated with a menu ID, so when registering the tree provider, make sure to use the corresponding menu ID.

System Menu ID for Virtual Account	
Menu	**ID**
Compute	0
Storage	1
Network	2
System Menu ID for Physical Account	
Menu	**ID**
Compute	50
Storage	51
Network	52

8 Developing a Trigger Condition

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

In the **Create Trigger** wizard (**Policies > Orchestration > Triggers**), at the **Specify Conditions** screen, you must have the options available to set up the new trigger condition.

A trigger is composed of the following 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 describes the object that is to be monitored and supply a list of references to the object. Choose a trigger, click **Edit Trigger**, and go to the **Specify Conditions** screen of the trigger edit wizard. Use the controls and related options to select the object and the references to it. For example, MonitoredContextIf can 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:

- To provide the specific parameter to be examined. For example, a parameter representing the status of the particular Dummy Device (for example, ddTwo) as defined by the MonitorContextIf.
- To supply 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.)
- To supply a list of values, each of which can be logically compared against the parameter to activate the trigger.

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.

8.1 Adding New Trigger Conditions

To add a new trigger condition, do the following:

1. Implement MonitoredContextIf and all the applicable MonitoredParameterIfs.
2. Register the trigger condition into the system.

Sample

Step 1: Implement MonitoredContextIf and all the applicable MonitoredParameterIfs.

```
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_CONTEXT_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_CONTEXT_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.

```
// 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();
```

9 Developing CloudSense Reports

Each CloudSense report consists of two components:

- An XML descriptor file
- The actual implementation of the report in JavaScript

The XML descriptor file requires the following:

- A unique string to identify the report
- A description of the report that is used to display in the UI

- The name of the implementation file for the report
- The context that must be used when generating the report

The implementation of CloudSense reports in JavaScript is described in detail in the Cisco UCS Director documentation.

You must place all CloudSense report-related files in a folder called cloudsense. The zip file that you upload through Open Automation must contain the cloudsense folder. For more information about the folder structure, see the sample module.

10 Adding an Account

Add an account in Cisco UCS Director to trigger the system to locate a collector for inventory collection. To add an account, define a new account type and its details by referring the AccountTypeEntry of the specific account. You need to specify the unique account type, account category, and the location in the UI where you want the report to be placed.

Adding a sample account includes the following tasks:

1. Creating a Credential Class
2. Registering the account for JDO enhancement
3. Registering an account in a Module

Creating a POJO class to collect data froma user

Define inputs in the POJO class to collect data from user, that must be persistable and also must populate the inputs to report. While adding an account, annotate the input fields with suitable annotations as shown in the following sample:

```
@persistent //this is used to persistence
@ReportField(label="Account Name") //this is used to populate the
inputs to report in the UI.
```

Note: This sample collects data from dummy class.

Sample

```
@PersistenceCapable(detachable = "true")
public class DummyAccount {

	@ReportField(label="Account Name")
	@Persistent
	private String accountName;

	@ReportField(label="Status")
	@Persistent
	private String status;

	@ReportField(label="IP Address")
	@Persistent
	private String ip;
```

```
}
```

10.1 Creating a Credential Class

Create the credential class that extends AbstractInfraAccount implementing the ConnectorCredential interface.

While implementing the ConnectorCredential interface, override the InfraAccount toInfraAccount() method.

Sample

```
public class FooAccount extends AbstractInfraAccount implements
ConnectorCredential{
	//Pojo can implement for the account creation
	@Persistent
	@FormField(label = "Device IP", help = "Device IP",
mandatory = true)
	private String deviceIp;


	@Persistent
@FormField(label = "Protocol", help = "Protocol", type =
FormFieldDefinition.FIELD_TYPE_EMBEDDED_LOV, validate = true, lov
= {
			"http", "https" })
	private String protocol="http";

@FormField(label = "Port", help = "Port Number", type =
FormFieldDefinition.FIELD_TYPE_TEXT)
	@Persistent
	private String port = "8080";

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

	@Persistent
@FormField(label = "Password", help = "Password", type =
FormFieldDefinition.FIELD_TYPE_PASSWORD)
		private String password;
	/**
	 * @return the deviceIp
	 */
	public String getDeviceIp() {
		return deviceIp;
	}

	/**
	 * @param deviceIp
	 *            the deviceIp to set
	 */
	public void setDeviceIp(String deviceIp) {
		this.deviceIp = deviceIp;
	}
```

```
/**
	 * @return the protocol
	 */
	public String getProtocol() {
		return protocol;
	}

	/**
	 * @param protocol
	 *            the protocol to set
	 */
	public void setProtocol(String protocol) {
		this.protocol = protocol;
	}

	/**
	 * @return the port
	 */
	public String getPort() {
		return port;
	}

	/**
	 * @param port
	 *            the port to set
	 */
	public void setPort(String port) {
		this.port = port;
	}

	/**
	 * @return the login
	 */
	public String getLogin() {
		return login;
	}

	/**
	 * @param login
	 *            the login to set
	 */
	public void setLogin(String login) {
		this.login = login;
	}

	/**
	 * @return the password
	 */
	public String getPassword() {
		return password;
	}

	/**
```

```
         * @param password
         *            the password to set
         */
        public void setPassword(String password) {
              this.password = password;
        }

@Override
public InfraAccount toInfraAccount() {
              try {
                    ObjStore<InfraAccount> store = ObjStoreHelper
                                .getStore(InfraAccount.class);
                    String cquery = "server == '"
                                + deviceIp + "' && userID == '"
+ login
                                + "' && transport == '" +
protocol + "' && port == "
                                + Integer.parseInt(port);
                    logger.debug("query = " + cquery);

                    List<InfraAccount> accList =
store.query(cquery);
                    if (accList != null && accList.size() > 0)
                          return accList.get(0);
                    else
                          return null;
              } catch (Exception e) {
logger.error("Exception while mapping DeviceCredential to
InfraAccount for server: "+ deviceIp + ": " + e.getMessage());
              }

              return null;
        }

}
```

10.2 Registering the Account for JDO Enhancement

After implementing the POJO and credential class, register the account for Jdo enhancement. For more information, see the Marking a Class as a Persistable Object.

10.3 Registering an Account in a Module

After implementing the account, register the account in the FooModule class.

Sample

```
private void createAccountType(){
AccountTypeEntry entry=new AccountTypeEntry();
// setting the credential class that holds the device credential
details
```

```
entry.setCredentialClass(FooAccount.class);
//There are three category of account type compute,storage and
network. we are
//setting the category as Storage shown below
entry.setCategory(InfraAccountTypes.CAT_STORAGE);

// we are setting type of account. There are two types of account
available virtual
//account and physical account.
entry.setAccountClass(AccountTypeEntry.PHYSICAL_ACCOUNT);

// we are setting the connection handler. Before adding account
this checks
//whether device is reachable or not by pinging the device ip.
entry.setTestConnectionHandler(new FooTestConnectionHandler());

//according to account type we developed inventory listener for
collecting credential
//details
entry.setInventoryListener(new FooInventoryListener());
//Registring of inventory object goes
registerInventoryObjects(entry);
//Registering of an account goes .
PhysicalAccountTypeManager.getInstance().addNewAccountType(entry)
;
}
```

11 Inventory Collection for an Account

To collect the inventory for the items, register the root name and its handler class in the FooModule class.

Sample

```
//Registering in FooModule clas
private void createAccountType(){
try {
       ...
       // Adding inventory for root objects
       registerInventoryObjects(entry);

       PhysicalAccountTypeManager.getInstance().addNewAccountType(
entry);
}catch(Exception e){
}
```

The RegisterInventoryObjects method is used for collecting the inventory for root or any unique item. Invoke the item handler for the collected inventory to parser and bind the response to the object for populating data in the UI.

```
private void registerInventoryObjects(AccountTypeEntry
fooRecoverPointAccountEntry) {
```

```
ConfigItemDef fooRecoverPointStateInfo =
fooRecoverPointAccountEntry
.createInventoryRoot("foo.inventory.root",FooInventoryItemHandler
.class);
}

public class FooInventoryItemHandler extends
AbstractInventoryItemHandler {

	private static Logger logger =
Logger.getLogger(FooInventoryItemHandler.class);
	//this method use for clean up operation
	@Override
	public void cleanup(String accountName) throws Exception {
		// cleanup implementation goes .

	}

	@Override
	public void doInventory(String accountName,
InventoryContext inventoryCtxt)
			throws Exception {
		//inventory implementation goes
	}
```