# 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

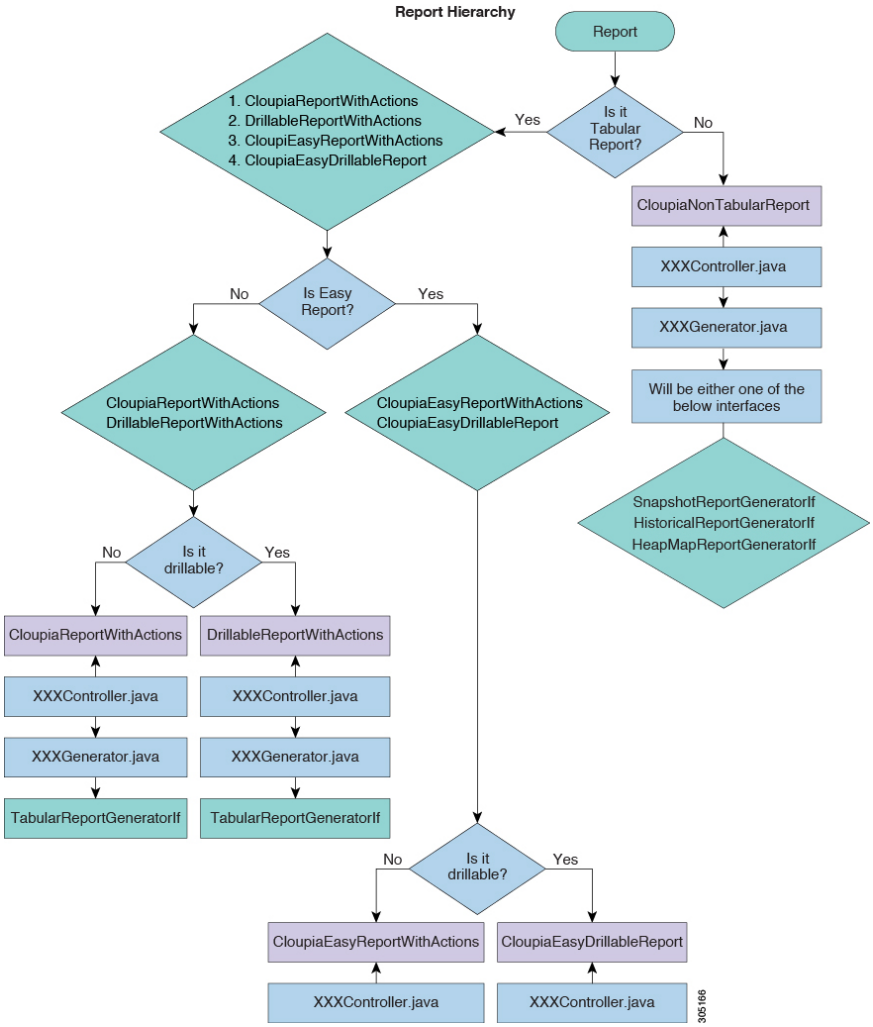
You can develop programmatic reports with the following classes:

- 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.

---

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.

   **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.

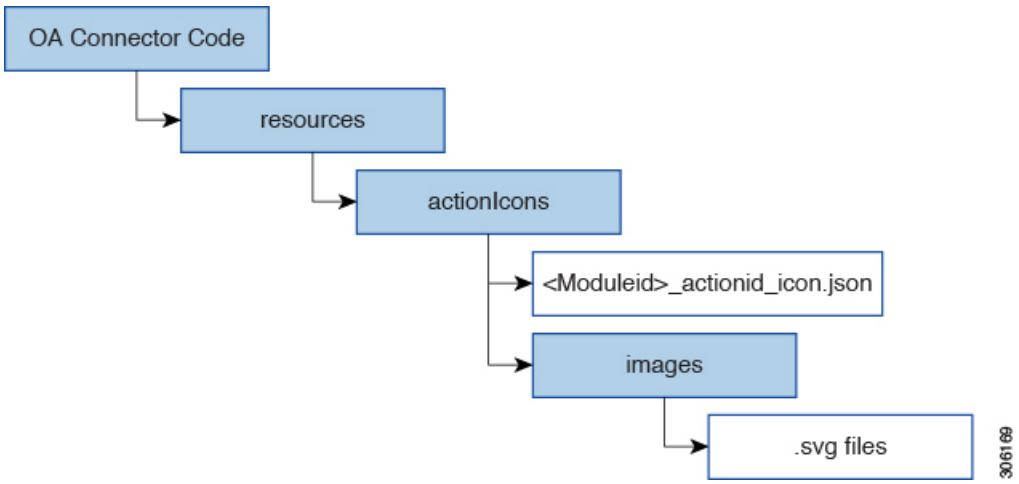
**Example:**

```
@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 17 and Developing Reports Using POJO and Annotations, on page 3.

### 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 6.

# 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.

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*.

```
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)
```