Guidelines for Programmable Reports

Prerequisites

Planning

1. Determine if the report is meant to be run for a whole fabric or device(s).
2. Find out which **show** command(s) should be run on switch(es) to collect the required data.
   - Find out if the CLI output supports xml, json, or neither.
   - If neither, plain CLI output is returned by the switch.
   - Determine if the CLI response, including the command executed, needs to be stored in elasticsearch. You should be cautious as storing responses may increase storage tremendously.
3. Determine if you need to pre-validate the report creation input such as Recurrence, Period, etc. For example, does the report job support a periodic report, and how frequent should the job run?

Report Presentation

1. If you want **Summary**, choose the format to display the data:
   - Key-value pairs
   - Table
   - Chart (column, pie, line)
2. For **Section** (Detailed View), choose which format to display the data:
   - Key-value pairs
   - Array of JSON objects

- Chart (column, pie, line)

3. For **Formatters**, the following are applicable:
   - Add additional formatting to values displayed in UI
   - Markers supported: ERROR, SUCCESS, WARNING, INFO

Data Comparison between Reports

1. Determine if the report needs to compare data between the current report and an older report.
2. If yes, report infra APIs allow you to retrieve previous report(s) like:
   - One or more previously generated reports
   - The oldest report in the report job
   - Summary of a particular report
   - A particular section of a reportCLI Output Processing

CLI Output Processing

XML Format

If the CLI output returns data in XML format, you can use the report infra provided XML utilities to read the XML data out:

From reportlib.preport import *:

- getxmltree(xml_string, tag)
- getxmlrows(xml_tree, tag_xpath)
- getnodevalue(xml_tree, node_xpath)
- has_tag(xml_tree, tag)

For an example, see the report template **switch_inventory**.

JSON Format

If the CLI output returns data in JSON format, import Python's json module and use the **json.loads()** method to parse a JSON string.

import json

json_string = <CLI response>

json_obj = json.loads(json_string)

For an example, see the report template **fabric_nve_vni_counter**.

Plain CLI Output

If the CLI output returns data in the same format as seen on the CLI UI, you need to write your own parsing method to read the data out in the CLI response.

Logger

Logger allows you to log messages from a report template. Logged messages are written to /usr/local/cisco/dcm/fm/logs/preport_jython.log.

Report Template

Template Properties

Specify the following mandatory template properties:

```
name = <template-name>;
tags = fabric or device;
userDefined = true or false;
templateType = REPORT;
templateSubType = GENERIC;
contentType = PYTHON;
```

Note

- Set **tags = fabric** if report is run for a fabric; **tags = device** if report is run for a device
- Set **userDefined = true** if template is created by a customer; **userDefined = false** if template is created by a DCNM developer.

Template Variables

```
Specify the following template variables:
##template variables
@(IsInternal=true)
string fabric_name or serial_number;
string user_input;
```

Note

- Configure variable *fabric_name* if **tags = fabric**
- Configure variable *serial_number* if **tags = device**
- User variables are optional. All data types and annotations supported by DCNM template infra can be used.

Template Content

Imported Libraries

The following 2 python libraries are required. Note that **reportlib.preport** contains all reporting infrastructure APIs.

```
##template content
from com.cisco.dcbu.vinci.rest.services.jython import WrappersResp
from reportlib.preport import *
```

Template Functions

generateReport()

generateReport() is the entry function and is invoked while generating a report. All the report implementation logic should be provided here. This function takes a context object. The 'context' parameter is created by the report infrastructure when a report job is created.

```
def generateReport(context):

    report = Report("Report title")   ## Create a report object
     ## Gather data and fill in content for the report

    respObj = WrappersResp.getRespObj()
    respObj.setSuccessRetCode()
    respObj.setValue(report)
    return respObj
```

Note

- This function must return a WrappersResp object.
- If there is no error in generating the report, the report object created within this function must be set in **WrappersResp.setValue()** before the **WrappersResp** object is returned.

Run CLI and Process CLI Response

Below is a sample code on how to send **show** command(s) to one or multiple devices, and on how to process the responses from the device(s).

```
show_cmd1 = 'show xxx'
show_cmd2 = 'show yyy'
device_list = [device1,device2]
## run the command(s) on each device in the device_list


cli_responses = show(device_list, show_cmd1, show_cmd2)
## run the command(s) on each device in the device_list and store the CLI response(s)

cli_responses = show_and_store(device_list, show_cmd1, show_cmd2)
```

For resp in cli_responses:

```
command = resp['command'].strip()

    if show_cmd1 in command:
        cmd1_response = resp['response'].strip()
        ## process show_cmd1 response
```

```
    elif show_cmd2 in command:
        cmd2_response = resp['response'].strip()
        ## process show_cmd1 response
```

validate()

The **validate()** function is an optional function and is used to perform pre-validation of report creation input such as Recurrence, Period, etc. This function, if defined, is called while creating a report job. The report job is only created if this function returns a WrappersResp with SuccessRetCode. If validation fails, a WrappersResp with FailureRetCode with errors should be returned.

```
def validate(context):
    respObj = WrappersResp.getRespObj()
    ## Validation content

    if validation_failed:
        respObj.addErrorReport(…)
        respObj.setFailureRetCode()
    else:
        respObj.setSuccessRetCode()
    return respObj
```

report.add_summary

Each report can have one summary and the content is a python dictionary.

```
summary = report.add_summary()
summary[key] = value
summary.add_message(msg)
## Present the summary in a table format

table = summary.add_table(title,_id)   ## _id must be a unique id for the table
table.append(value,_id)   ## adding rows to table
## Present the summary in a chart format

chart = summary.add_chart(ChartType,_id)
## ChartTypes: ChartTypes.COLUMN_CHART, ChartTypes.PIE_CHART, ChartTypes.LINE_CHART
```

report.add_section

Section is a logical grouping of report content. Section details are displayed in **View Details**.

```
section = report.add_section(title,_id)   ## _id must be a unique id for the section
section[key] = value
section.append(key, json_obj, _id)   ## adding rows of json objects to section
## Present the section details in a chart format

chart = section.add_chart(ChartType,_id)
## ChartTypes: ChartTypes.COLUMN_CHART, ChartTypes.PIE_CHART, ChartTypes.LINE_CHART
```