Collecting Account Inventory

This chapter contains the following sections:

About the Inventory Collector

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

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

Guidelines for Developing a Module

When you develop a new module to support new devices, ensure that:

- You develop a module for a device family so that you have only one module to support all these devices.
- You do not develop a module that supports both a network switch and a storage controller; instead, split them into two modules. Ideally, a module must support only devices within the same category, so that a module can handle only compute devices, network devices, or storage devices.
- The devices supported by the same module must be similar.
- The same device may come in different models that are meant for distinct purposes, and it may be appropriate to use different modules to support them.

Creating an Account Type Entry

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

// This is mandatory. Register Inventory Collectors for this account type.
// Refer to section “Creating Inventory Collectors” for more detail.
```

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

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

Creating an Inventory Collector

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

Inventory Collection Handlers

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

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

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

where

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

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

Inventory Listener

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

Registering Collectors

You must register the collectors as follows:

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

Registering a Report Context

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

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

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

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

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

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

Converged Stack Builder

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

Before You Begin

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

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

Procedure

Step 1 Provide an implementation of ConvergedStackComponentBuilderIf.
Extend the abstract implementation:
com.cloupia.service.cIM.inframgr.reports.contextresolve.AbstractConvergedStackComponentBuilder.

Step 2 Supply a device icon mapping file.
This XML file is used to map the data supplied by your ConvergedStackComponentBuilderIf to the actual images to be used in the UI. This XML file must be named as device_icon_mapping.xml and it must be packaged inside your jar.

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

Step 3 Package the images in a module.zip file and place the zip file in the **resources** folder.
The framework copies all your images in the **resources** folder and places them in an uploads folder.