Cisco UCS Director Custom Task Getting Started Guide, Release 6.6

First Published: 2018-04-27

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

CONTENTS

Preface

Audience

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

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

Conventions

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

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

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

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

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

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

Warning IMPORTANT SAFETY INSTRUCTIONS

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

SAVE THESE INSTRUCTIONS

Related Documentation

Cisco UCS Director Documentation Roadmap

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

Cisco UCS Documentation Roadmaps

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

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

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

Documentation Feedback

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

Obtaining Documentation and Submitting a Service Request

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

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

CHAPTER 1

New and Changed Information for This Release

New and Changed Information for This Release

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

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

Feature	Description	Where Documented
Write/Execute CloupiaScript	User must have permission to write and execute CloupiaScript to write and execute a custom task using CloupiaScript.	Prerequisites, on page 15
Custom Input Validation	Provides an option to validate any input at runtime using a customer-provided script. The script can flag errors in the input and can require valid input before running a service request.	Custom Input Validation, on page 16

CHAPTER 2

Overview of Custom Tasks

Why Use Custom Tasks

Custom tasks extend the capabilities of Cisco UCS Director Orchestrator. Custom tasks enable you to create functionality that is not available in the predefined tasks and workflows that are supplied with Cisco UCS Director. You can generate reports, configure physical or virtual resources, and call other tasks from within a custom task.

How Custom Tasks Work

Once created and imported into Cisco UCS Director, custom tasks function like any other tasks in Cisco UCS Director Orchestrator. You can modify, import, and export a custom task and you can add it to any workflow.

How to Use Custom Tasks

You write, edit, and test custom tasks from within Cisco UCS Director. You must have administrator privileges to write custom tasks.

You write custom tasks using CloupiaScript, a version of JavaScript with Cisco UCS Director Java libraries that enable orchestration operations. You then use your custom tasks like any other task, including them in workflows to orchestrate work on your components.

CloupiaScript supports all JavaScript syntax. CloupiaScript also supports access to a subset of the Cisco UCS Director Java libraries, enabling custom tasks access to Cisco UCS Director components. Because CloupiaScript runs only on the server, client-side objects are not supported.

CloupiaScript uses the Nashorn script engine. For more details about Nashorn, see the technical notes on Oracle's website at https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/api.htm.

Implicit Variables in Custom Tasks

Three predefined top-level variables are included automatically in any custom task:

Variable	Description
ctxt	The workflow execution context. This context object contains information about the current workflow, the current task, and available inputs and outputs. It also has access to the Cisco UCS Director Java APIs, with which you can perform create, read, update, and delete (CRUD) operations, invoke other tasks, and call other API methods. The *ctxt* variable is an instance of the platform API class `com.cloupia.service.cIM.inframgr.customactions.` `CustomActionTriggerContext`.
logger	The workflow `logger` object. The workflow logger writes to the service request (SR) log. The *logger* variable is an instance of the platform API class `com.cloupia.service.cIM.inframgr.customactions.CustomActionLogger`.
util	An object that provides access to utility methods. The *util* variable is an instance of the platform API class `com.cloupia.lib.util.managedreports.APIFunctions`.

For more information about the API classes of the implicit variables, see the CloupiaScript Javadoc included in the Cisco UCS Director script bundle.

Changes to CloupiaScript due to JDK Upgrade

From Cisco UCS Director Release 5.4, the JDK version has been upgraded from 1.6 to 1.8. While the JDK 1.6 version was based on the Rhino JavaScript engine, the JDK 1.8 version ships with a new Nashorn Javascript engine. The Nashorn JavaScript engine has changes in syntax and usage of certain functions and classes in the script.

Following are changes to be aware of when you script custom tasks for Cisco UCS Director, Release 5.5:

- **Converting an object to a map for retrieving the values of the object property**

  Up through Cisco UCS Director Release 5.3, use the following code snippet to get values of each property of an object (for example, vminfo) using the `for` loop:

```
importPackage(com.cloupia.service.cIM.inframgr);
importPackage(com.cloupia.model.cIM);
importPackage(com.cloupia.lib.util);
importPackage(java.util);
importPackage(java.lang);
var vmSummary ="";
var vminfo = ctxt.getAPI().getVMwareVMInfo(306);//306 is vmId
for(var x in vminfo){
//escaping getter and setter methods
if(x.match(/get*/) == null && x.match(/set*/) == null && x.match(/jdo*/) == null &&
x.match(/is*/)
 == null && x.match(/hashCode/) == null && x.match(/equals/) == null)
{
vmSummary += x  +":"+ vminfo[x] + '#';
};
};
logger.addInfo("VMSUMMARY="+vmSummary);
```

Beginning with Cisco UCS Director Release 5.4, convert the object (for example, vminfo) into a map using the convertObjectToMap () method of the ObjectToMap class and then use the object in the `for` loop to retrieve the object values. The following code snippet shows how to get the value of each property of an object:

```
importPackage(com.cloupia.service.cIM.inframgr);
importPackage(com.cloupia.model.cIM);
importPackage(com.cloupia.lib.util);
importPackage(java.lang);
importPackage(java.util);

var vmSummary = "";
var vminfo = ctxt.getAPI().getVMwareVMInfo(4);//4 is vmId
var vminfo = ObjectToMap.convertObjectToMap(vminfo);
for (var x in vminfo) {
vmSummary += x  +":"+ vminfo[x] + '#';
}
logger.addInfo("VMSUMMARY="+vmSummary);
```

Note The ObjectToMap.convertObjectToMap(vminfo) class can be used only when the object (for example, vminfo) contains properties of primitive or string type. The best practice is to use the standard getter methods such as getVmId() and getVmName() to retrieve the attributes of an object.

- **Using the print( ) function**

  Use `print( )` instead of `println( )`.

  Note JDK 1.8 still supports `println( )` for backward compatibility.

- **Change in syntax for passing a class<T> parameter to a method or constructor**

  The syntax for passing a Class<T> parameter to a method or constructor has changed. In JDK1.6, the following syntax was valid:

  ```
  var fml = new FormManagedList(PrivateCloudNetworkPolicyNICPortGroup);
  ```

  However, in JDK1.8, you must append `.class` to pass the `PrivateCloudNetworkPolicyNICPortGroup` Java class as an argument, like this:

  ```
  var fml = new FormManagedList(PrivateCloudNetworkPolicyNICPortGroup.class);
  ```

- **Change in syntax to import classes and packages**

  The syntax to import classes and packages has changed. The newer import statement improves localizing the usage of the class or package. The earlier import statement made the class or package available in the global space of the javascript execution, which was not always required.

  Here is an example of an import statement in the Rhino JavaScript Engine:

  ```
  importPackage(com.cloupia.model.cIM);
  importClass(java.util.ArrayList);
  ```

  Here is an example of import statement in the Nashorn JavaScript Engine:

```
var CollectionsAndFiles = new JavaImporter( java.util, java.io, java.nio);
with (CollectionsAndFiles) {
 var files = new LinkedHashSet();
 files.add(new File("Filename1"));
 files.add(new File("Filename2"));
}
```

The `with` statement defines the scope of the variable given as its argument with respect to the duration of time the object(s) are loaded in its memory. For example, sometimes it is useful to import many Java packages at a time. Using the JavaImporter class along with the `with` statement, all class files from the imported packages are accessible within the local scope of the `with` statement.

Importing Java packages:

```
var imports = new JavaImporter(java.io, java.lang);
with (imports) {
    var file = new File(__FILE__);
    System.out.println(file.getAbsolutePath());
    // /path/to/my/script.js
}
```

Note The older `importPackage()` and `importClass()` statements are still supported in Cisco UCS Director 5.5 for backward compatibility. The engine at the back-end calls load('nashorn:Mozilla_compat.js') before executing a custom task script.

- **Accessing Static Methods**

  The flexibility of accessing static methods is reduced in the Nashorn engine. In Rhino's version of the engine, a static method can be accessed not only through the class name (using the same syntax as in Java), but also from any instance of that class (unlike Java).

  Accessing Static Methods in Rhino:

```
var myRBUtil = new com.cloupia.service.cIM.inframgr.i18n.RBUtil();
myRBUtil.getString();// No error
com.cloupia.service.cIM.inframgr.i18n.RBUtil.getString();// No error
```

  Accessing static methods in Nashorn:

```
var myRBUtil = new com.cloupia.service.cIM.inframgr.i18n.RBUtil();
myRBUtil.getString();// Error
com.cloupia.service.cIM.inframgr.i18n.RBUtil.getString();// No error
```

- **Comparison of the native JSON object with com.cloupia.lib.util.JSON**

  The Nashorn environment consists of a native JSON object which has built-in functions to convert objects to JSON format and vice versa. Cisco UCS Director has its own version of the JSON object called `com.cloupia.lib.util.JSON`.

  A library of JSON payloads is available in Cisco UCS Director. Load the JSON payload library by running the following command in CloupiaScript:

```
loadLibrary("JSON-JS-Module/JSON-JS-ModuleLibrary");
```

  The following methods are available:

  - JSON2.parse—The JSON2.parse method converts a JSON string to a JavaScript object.

- JSON2.stringify—The JSON2.stringify method converts a JavaScript object to a JSON string.

If the Cisco UCS Director class is imported, access to the native JSON object is lost because the same object name is in use. To enable use of both the Cisco UCS Director and native JSON objects, Cisco UCS Director stores the native class using the name NativeJSON. So, for example, the following are static method calls of the native object:

```
NativeJSON.stringify(object myObj);
NativeJSON.parse(String mystr);
```

- **Using the new operator for strings**

  Explicitly add the keyword **new** when creating an object.

  For example:

```
var customName = new java.lang.String(input.name);
var ai = new CMDB.AdditionalInfo();// static class
```

Guidelines for Using API Operations from CloupiaScript

Executing the XML REST API

The following table provides a list of methods that are used to execute the XML REST API operations:

API Operation	Method
Get	getMoResourceAsJson(resourcePath);
Create	createMoResource(resourcePath, payload);
Update	updateMoResource(resourcePath, payload);
Delete	deleteMoResource(resourcePath, payload);

To execute a method, you must pass at least one of the following parameters:

- resourcePath—The resourcePath can be taken from the **Resource URL** field of the **REST API Browser**. Pass the resourcePath as a string for all API operations (get, create, update, and delete). For example, `/cloupia/api-v2/user`.
- payload—Construct the payload as a JSON string and pass it as the payload to the API operation.

Note By default, execution of these methods are controlled based on user role. If the task developer (system admin) wants to allow non-admin user to execute any of these methods with admin role, the admnistrator has to pass an additional argument as True as follows:

```
getMoResourceAsJson(resourcePath, true)
```

Note Read operations are shown in the following examples. All the JSON operations (Create, Read, Update, and Delete) are executed in a similar manner.

Example 1: Using the get method to retrieve a list of users

```
//retrieve users in JSON string format
var userRes = ctxt.getAPI().getMoResourceAsJson("/user");
```

Example 2: Using the get method to retrieve the details of a user

```
//retrieve a specific user (admin) in the JSON string format
var userRes = ctxt.getAPI().getMoResourceAsJson("/user/admin");
//convert a JSON string to a JavaScript object using the JSON2 library
var jsUserObj = JSON2.parse(userRes);
//get the access level and login name of the user from the JavaScript object and use those
 values in CloupiaScript
var accessLevel = jsUserObj.cuicOperationResponse.response.user.access.accessLevel;
var loginName = jsUserObj.cuicOperationResponse.response.user.access.loginName;
```

Example 3: Using the create method to create a user

```
var resourcePath = "/user";
//To create a payload, create a JavaScript object as shown below:
var cuicRequest ={};
var requestPayloadObj = {};
var addUserConfigObj = {};
addUserConfigObj.userType = "AdminAllPolicy";
addUserConfigObj.loginName = "apadmin";
addUserConfigObj.password = "cloupia123";
addUserConfigObj.confirmPassword = "cloupia123";
addUserConfigObj.userContactEmail = "apadmin@cisco.com";

var payloadObj = {};
payloadObj.AddUserConfig = addUserConfigObj;
requestPayloadObj.payload = payloadObj;
cuicRequest.cuicOperationRequest = requestPayloadObj;
//Convert the JavaScript object to a JSON string using the stringify JSON2 library.
var cuicRequestStr = JSON2.stringify(cuicRequest);
var apiResponse = ctxt.getAPI().createMoResource(resourcePath, cuicRequestStr);
```

Example 4: Using the update method to update the user details

```
var resourcePath = "/group";
//To create a payload, create JavaScript object as shown below:
var requestPayload = {};
var modifyGroupConfigObject = {};
modifyGroupConfigObject.groupId = "16";
modifyGroupConfigObject.groupDescription = "description updated";
modifyGroupConfigObject.groupContact = "sdk-group@cisco.com";
var payloadObj = {};
payloadObj.ModifyGroupConfig = modifyGroupConfigObject;
var cuicReq = {};
cuicReq.payload = payloadObj;
requestPayload.cuicOperationRequest = cuicReq;
//Convert the JavaScript object to a JSON string using the stringify JSON2 library.
var requestPayloadStr = JSON2.stringify(requestPayload);
var apiResponse = ctxt.getAPI().updateMoResource(resourcePath, requestPayloadStr);
```

Example 5: Using the delete method to delete a user

```
var resourcePath = "/datacenter/Default Pod/cloud/cloud_95/vmComputingPolicy/sdk_cp";

//Some delete APIs do not require payload, in such cases, pass the payload as empty string
```

```
 or null.
//If the delete API requires the payload data, form the JSON string payload as explained
in the create and update method examples.
var payload = "";
var apiResponse = ctxt.getAPI().deleteMoResource(resourcePath, payload);
```

Executing the JSON API

Use the **performOperationOnJSONPayload** method to execute the JSON API.

To execute the method, you must pass one of the following parameters:

- OperationName—Name of the JSON REST API operation, which starts with **userAPI**.
- OperationData—Data of the JSON REST API, which is used as a request parameter to fetch the response.

Example 1: Retrieving data without passing any variable in the OpData parameter

```
REST API URL is :
/app/api/rest?formatType=json&opName=userAPIGetMyLoginProfile&opData={}

var payload = {};
var payloadString = JSON2.stringify(payload);
var response = ctxt.getAPI().performOperationOnJSONPayload('userAPIGetMyLoginProfile',
payloadString);
```

Example 2: Retrieving data by passing a variable in the OpData parameter

```
/app/api/rest?formatType=json&opName=userAPIGetGroupByName&opData={param0:"Default Group"}


var payload = {};
payload.param0 = 'Default Group';
var payloadString = JSON2.stringify(payload);
var response = ctxt.getAPI().performOperationOnJSONPayload('userAPIGetGroupByName',
payloadString);
```

Note The other JSON operations (Create, Update, and Delete) are executed in a manner similar to that shown for the preceding Read examples.

CHAPTER 3

CloupiaScript Interpreter

About the CloupiaScript Interpreter

The CloupiaScript interpreter is a JavaScript interpreter populated with built-in libraries and APIs. You can use the CloupiaScript interpreter to test CloupiaScript code without having to create and run a workflow task.

Built-in Functions of the CloupiaScript Interpreter

- `PrintObj()`—Takes an object as an argument and prints out all the properties and methods in the object. The printed result provides the names and values for variables in the object and the names of all the object's functions. You can then call `toString()` on any of the method names to examine the method signature.

- `Upload()`—Takes a filename as an argument and uploads the file's contents to the CloupiaScript interpreter.

Starting the CloupiaScript Interpreter

To open the CloupiaScript interpreter, do the following:

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click **Launch Interpreter**.
The **Cloupia Script Interpreter** screen appears.

Step 4 Enter a line of JavaScript code in the text input field at the bottom of the **Cloupia Script Interpreter** field.

Step 5 Press **Enter**.
The code is executed and the result is displayed. If there is a syntax error in the code, the error is displayed.

Starting the CloupiaScript Interpreter with a Context

You can evaluate JavaScript in the context of a particular a custom task. To do so, you select a custom task, launch the CloupiaScript Interpreter, and supply the context variables that are defined for executing that custom task.

When you launch the interpreter, it prompts you for values of the custom task input fields and populates the input object of the task. All the variables that are available when you actually execute the custom task are made available.

To open the CloupiaScript interpreter with a context available, do the following:

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click the row with the custom task for which you need to test the JavaScript.

Step 4 Click **Launch Interpreter with Context**.
The **Launch Interpreter** screen appears with input fields to collect input values for the custom task. The input fields are those defined for the custom task you have selected.

Step 5 Enter input values in the screen.

Step 6 Click **Submit**.

Step 7 Click **Submit**.
The **Cloupia Script Interpreter** screen appears.

Step 8 Enter a line of JavaScript code in the text input field at the bottom of the **Cloupia Script Interpreter** field.

Step 9 Press **Enter**.
The code is executed and the result is displayed. If there is any syntax error in the code, the error is displayed.

Example: Using the CloupiaScript Interpreter

The printObj( ) function prints all the properties and methods it contains. Call `functiontoString()` to find more details about a function. The following example shows how to examine the ReportContext class and get details about `ReportContext.setCloudName()`.

```
session started
> importPackage(com.cloupia.model.cIM);
> var ctx = new ReportContext();
> printObj(ctx);
properties =
cloudName:null
class:class com.cloupia.model.cIM.ReportContext
filterId:null
id:null
targetCuicId:null
type:0
ids:[Ljava.lang.String;@4de27bc5
methods =
setIds
jdoReplaceField
```

```
jdoReplaceFields
toString
getCloudName
wait
getClass
jdoReplaceFlags
hashCode
jdoNewInstance
jdoReplaceStateManager
jdoIsDetached
notify
jdoGetVersion
jdoProvideField
jdoCopyFields
jdoGetObjectId
jdoGetPersistenceManager
jdoCopyKeyFieldsToObjectId
jdoGetTransactionalObjectId
getType
getFilterId
setType
jdoIsPersistent
equals
setCloudName
jdoNewObjectIdInstance
jdoIsDeleted
getTargetCuicId
setId
setFilterId
jdoProvideFields
jdoMakeDirty
jdoIsNew
requiresCloudName
getIds
notifyAll
jdoIsTransactional
getId
jdoReplaceDetachedState
jdoIsDirty
setTargetCuicId
jdoCopyKeyFieldsFromObjectId

> var func = ctx.setCloudName;
> func
void setCloudName(java.lang.String)
> func.toString();
function setCloudName() {/*
void setCloudName(java.lang.String)
*/}
```

CHAPTER 4

Creating Custom Workflow Tasks

About Custom Workflow Inputs

Cisco UCS Director Orchestrator offers a list of well-defined input types for custom tasks. Cisco UCS Director also enables you to create a customized workflow input for a custom workflow task. You can create a new input type by cloning and modifying an existing input type.

Prerequisites

Before writing custom tasks, you must meet the following prerequisites:

- Cisco UCS Director is installed and running on your system. For more information about how to install Cisco UCS Director, refer to the Cisco UCS Director Installation and Configuration Guide.
- You have a login with administrator privileges. You must use this login when you create and modify custom tasks.
- You must have the write CloupiaScript permission to write a custom task using CloupiaScript.
- You must have the execute CloupiaScript permission to execute a custom task created using CloupiaScript.

Creating a Custom Workflow Input

You can create a custom input for a custom workflow task. The input is displayed in the list of input types that you can map to custom task inputs when you create a custom workflow task.

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Inputs**.

Step 3 Click **Add**.

Step 4 On the **Add Custom Workflow Input** screen, complete the following fields:

- **Custom Input Type Name**—A unique name for the custom input type.
- **Input Type**—Check a type of input and click **Select**. Based on the selected input, other fields appear. For example, when you choose **Email Address** as the input type, a list of values (LOV) appears. Use the new fields to limit the values of the custom input.

Step 5 Click **Submit**.
The custom workflow input is added to Cisco UCS Director and is available in the list of input types.

Custom Input Validation

Customers may need to validate workflow inputs using external resources. Out of the box, Cisco UCS Director cannot meet every customer's validation needs. To fill this gap, Cisco UCS Director provides an option to validate any input at runtime using a customer-provided script. The script can flag errors in the input and can require valid input before running a service request. The script can be written in any language, can access any external resource, and has access to all the workflow input values.

You can write custom validation scripts using JavaScript, Python, a bash shell script, or any other scripting language.

The following example validation scripts can be found in Cisco UCS Director in **Orchestration** > **Custom Workflow Inputs**:

- `example-bash-script-validator`
- `example-javascript-validator`
- `example-python-validator`

You can copy or clone the example scripted workflow inputs to create a new validated input. You can also use the example scripted workflow inputs as a guide for developing your own scripts.

Regardless of the scripting language, the following features and rules apply to scripted custom input validation:

- All scripted validation is run in a separate process, so that a failing validation process does not affect the Cisco UCS Director process.
- Only generic text inputs can be validated using scripts.

- Validation scripts are run one at a time, in sequence, in the same order in which the inputs appear in the workflow inputs page. A separate process is launched for each validated input.
- A nonzero return value from the script indicates a failed validation. Optionally, you can pass an error message back to the workflow input form.
- All workflow inputs are passed to the validation script in two ways:
  - As arguments to the script in the form `"key"="value"`.
  - As environment variables to the script process. The variable names are the input labels.

    For example, if the workflow has an input labeled as Product-Code and the input value is AbC123, the variable is passed to the validator script as `"Product-Code"="AbC123"`.

  These input variables can be used by the script if necessary to implement the validation. Exception: Table values contain only the row number of the table selection, and are therefore probably useless.
- The Edit Custom Workflow Input page makes the script available in the Custom Task editor. Syntax is highlighted for all languages. Syntax errors are checked only for JavaScript validators.

Cloning a Custom Workflow Input

You can use an existing custom workflow input in Cisco UCS Director to create a custom workflow input.

Before you begin

A custom workflow input must be available in Cisco UCS Director.

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Inputs**.

Step 3 Click the row with the custom workflow input to be cloned.
The **Clone** icon appears at the top of the custom workflow inputs table.

Step 4 Click **Clone**.

Step 5 Enter the custom input type name.

Step 6 Use the other controls in the **Clone Custom Workflow Input** screen to customize the new input.

Step 7 Click **Submit**.
The custom workflow task input is cloned after confirmation and is available for use in the custom workflow task.

Creating a Custom Task

To create a custom task, do the following:

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click **Add**.

Step 4 On the **Add Custom Workflow Task** screen, complete the following fields:

- **Task Name** field—A unique name for the custom workflow task.
- **Task Label** field—A label to identify the custom workflow task.
- **Register Under Category** field—The workflow category under which the custom workflow task has to be registered.
- **Activate Task** check box—If checked, the custom workflow task is registered with Orchestrator and is immediately usable in workflows.
- **Brief Description** field—A description of the custom workflow task.
- **Detailed Description** field—A detailed description of the custom workflow task.

Step 5 Click **Next**.
The **Custom Task Inputs** screen appears.

Step 6 Click **Add**.

Step 7 On the **Add Entry to Inputs** screen, complete the following fields:

- **Input Field Name** field—A unique name for the field. The name must start with analphabetic character and must not contain spaces or special characters.
- **Input Field Label** field—A label to identify the input field.
- **Input Field Type** drop-down list—Choose the data type of the input parameter.
- **Map to Input Type (No Mapping)** field—Choose a type of input to which this field can be mapped, if this field that can be mapped from another task output or global workflow input.
- **Mandatory** check box— If checked, user must provide a value for this field.
- **RBID** field—Enter the RBID string for the field.
- **Input Field Size** drop-down list—Choose the field size for text and tabular inputs.
- **Input Field Help** field—(Optional) A description that is shown on when you hover the mouse over the field.
- **Input Field Annotation** field—(Optional) Hint text for the input field.
- **Field Group Name** field—If specified, all the fields with matching group names are put into the field group.
- **TEXT FIELD ATTRIBUTES** area—Complete the following fields when the input field type is text.
    - **Multiple Input** check box—If checked, the input field accepts multiple values based on the input field type:
        - For an LOV—The input field accepts multiple input values.
        - For a text field—The input field becomes multi-line text field.
    - **Maximum Length of Input** field—Specify the maximum number of characters that you can enter in the input field.
- **LOV ATTRIBUTES** area—Complete the following fields when the input type is List of Values (LOV) or LOV with Radio buttons.
    - **List of Values** field—A comma-separated list of values for embedded LOVs.

    - **LOV Provider Name** field—The name of the LOV provider for non-embedded LOVs.

- **TABLE ATTRIBUTES** area—Complete the following fields when the input field type is Table, Popup Table, or Table with selection check box.
    - **Table Name** field—A name of the tabular report for the table field types.
- **FIELD INPUT VALIDATION** area—One or more of the following fields is displayed depending on your selected data type. Complete the fields to specify how the input fields are validated.
    - **Input Validator** drop-down list—Choose a validator for the user input.
    - **Regular Expression** field—A regular expression pattern to match the input value against.
    - **Regular Expression Message** field—A message that displays when the regular expression validation fails.
    - **Minimum Value** field—A minimum numeric value.
    - **Maximum Value** field—A maximum numeric value.
- **HIDE ON FIELD CONDITION** area—Complete the following fields to set the condition to hide the field in a form.
    - **Hide On Field Name** field—An internal name to the field so the program that handles the form can identify the field.
    - **Hide On Field Value** field—The value that has to be sent once the form is submitted.
    - **Hide On Field Condition** drop-down list—Choose a condition at which the field has to be hidden.
    - **HTML Help** field—The help instructions for the hidden field.

Step 8 Click **Submit**.
The input entry is added to the table.

Step 9 Click **Add** to add more entry to inputs.

Step 10 When you are done adding inputs, click **Next**.
The **Custom Workflow Tasks Outputs** screen appears.

Step 11 Click **Add**.

Step 12 On the **Add Entry to Outputs** screen, complete the following fields:

- **Output Field Name** field —A unique name for the output field. It must start with an alphabetic character and must not contain spaces or special characters.
- **Output Field Description** field —A description of the output field.
- **Output Field Type** field—Check a type of output. This type determines how the output can be mapped to other task inputs.

Step 13 Click **Submit**.
The output entry is added to the table.

Step 14 Click **Add** to add more entry to outputs.

Step 15 Click **Next**
The **Controller** screen appears.

Step 16 (Optional) Click **Add** to add a controller.

Step 17 On the **Add Entry to Controller** screen, complete the following fields:

- **Method** drop-down list—Choose either a marshalling or unmarshalling method to customize the inputs and/or outputs for the custom workflow task. The method can be one of the following:
  - **beforeMarshall**—Use this method to add or set an input field and dynamically create and set the LOV on a page (form).
  - **afterMarshall**—Use this method to hide or unhide an input field.
  - **beforeUnmarshall**—Use this method to convert an input value from one form to another form—for example, when you want to encrypt a password before sending it to the database.
  - **afterUnmarshall**—Use this method to validate a user input and set the error message on the page.

  See Example: Using Controllers, on page 28.
- **Script** text area—For the method you chose from the **Method** drop-down list, add the code for the GUI customization script.

  Note Click **Add** if you want to add code for more methods.

Step 18 Click **Submit**.
The controller is added to the table.

Step 19 Click **Next**.
The **Script** screen appears.

Step 20 From the **Execution Language** drop-down list, choose a language.

Step 21 In the **Script** field, enter the CloupiaScript code for the custom workflow task.

Note The CloupiaScript code is validated when you enter the code. If there is any error in the code, an error icon (red cross) is displayed next to the line number. Hover the mouse over the error icon to view the error message and the solution.

Step 22 Click **Save Script**.

Step 23 Click **Submit**.
The custom workflow task is created and is available for use in the workflow.

Custom Tasks and GitHub Repositories

When you create a custom task, rather than typing in the custom task code into the script window or cutting and pasting code from a text editor, you can import the code from a file stored in a GitHub repository. To do this, you:

1. Create one or more text files in a GitHub repository, either in github.com or a private enterprise GitHub repository.

Note Cisco UCS Director supports only GitHub (github.com or an enterprise GitHub instance). It does not support other Git hosting services including BitBucket, GitLab, Perforce, or Codebase.

2. Register the repository in Cisco UCS Director. See Adding a GitHub Repository in Cisco UCS Director, on page 21.
3. Select the repository and specify the text file that contains the custom task script. See Downloading Custom Task Script Code from a GitHub Repository, on page 21.

Adding a GitHub Repository in Cisco UCS Director

To register a GitHub repository in Cisco UCS Director, do the following:

Before you begin

Create a GitHub repository. The repository can be on any GitHub server, public or private that is accessible from your Cisco UCS Director.

Check in one or more files containing JavaScript code for your custom tasks into your repository.

Step 1 Choose **Administration** > **Integration**.

Step 2 On the **Integration** page, click **Manage Repositories**.

Step 3 Click **Add**.

Step 4 On the **Add Repository** page, complete the required fields, including the following:

a) In the **Repository Nickname** field, enter a name to identify the repository within Cisco UCS Director.
b) In the **Repository URL** field, enter the URL of the GitHub repository.
c) In the **Branch Name** field, enter the name of the repository branch you want to use. The default name is **main** branch.
d) In the **Repository User** field, enter the username for your GitHub account.
e) In the **Repository Password** field, enter the password for your GitHub account. (The password characters are not shown.)
f) To default to this repository when you create a new custom task, check **Make this my default repository**.
g) To test whether Cisco UCS Director can access the repository, click **Test Connectivity**.
The state of connectivity with the repository is displayed in a banner at the top of the page.

Note If you are unable to connect and communicate with the GitHub repository from Cisco UCS Director, update Cisco UCS Director to access the Internet through a proxy server. See the Cisco UCS Director Administration Guide.

Step 5 When you are satisfied that the repository information is correct, click **Submit**.

Downloading Custom Task Script Code from a GitHub Repository

To create a new custom task by importing text from a GitHub repository, do the following:

Before you begin

Create a GitHub repository and check in one or more text files containing the JavaScript code for your custom tasks into your repository.

Add the GitHub repository to Cisco UCS Director. See Adding a GitHub Repository in Cisco UCS Director, on page 21.

Step 1 On the **Orchestration** page, click **Custom Workflow Tasks**.

Step 2 Click **Add**.

Step 3 Complete the required fields on the Custom Task Information page. See Creating a Custom Task, on page 17.

Step 4 Complete the required fields on the Custom Task Inputs page. See Creating a Custom Task, on page 17.

Step 5 Complete the required fields on the Custom Task Outputs page. See Creating a Custom Task, on page 17.

Step 6 Complete the required fields on the Controller page. See Creating a Custom Task, on page 17.

Step 7 On the **Script** page, complete the required fields:

a) From the **Execution Language** drop-down list, select JavaScript.
b) Check **Use Repository for Scripts** to enable the custom task to use a script file from a repository. This enables you to select the repository and specify the script file to use.
c) From the **Select Repository** drop-down list, select the GitHub repository containing the script files. For details on how to add repositories, see Adding a GitHub Repository in Cisco UCS Director, on page 21.
d) Enter the full path to the script file in the **Script filename** text field.
e) To download the script, click **Load Script**.
The text from the file is copied in the **Script** text edit area.
f) Optionally, make changes to the downloaded script in the **Script** text edit area.
g) To save the script as it appears in the **Script** text edit area, click **Save Script**.

Note When you press **Save Script**, the script is saved to your current work session. You must click **Submit** to save the script to the custom task that you are editing.

Step 8 To save the custom task, click **Submit**.

Note If you made changes to the downloaded script in the **Script** text edit area, the changes are saved to the custom task. No changes are saved to the GitHub repository. If you would like to discard the loaded script and enter your own script, click **Discard Script** to clear the script window.

What to do next

You can use the new custom task in a workflow.

Importing Workflows, Custom Tasks, Script Modules, and Activities

To import artifacts into Cisco UCS Director, do the following:

Step 1 Choose **Orchestration**.

Step 2 On the **Orchestration** page, click **Workflows**.

Step 3 Click **Import**.

Step 4 On the **Import** screen, click **Select a File**.

Step 5 On the **Choose File to Upload** screen, choose the file to be imported. Cisco UCS Director import and export files have a `.wfdx` file extension.

Step 6 Click **Open**.
When the file is uploaded, the **File Upload** screen displays `File ready for use`.

Step 7 Click **Next**.
The **Import** screen displays a list of Cisco UCS Director objects contained in the uploaded file.

Step 8 (Optional) Specify how objects are handled if they duplicate names already in the workflow folder. On the **Import** screen, complete the following fields:

Name	Description
Workflows	Choose from the following options to specify how identically named workflows are handled: • **Replace**—Replace the existing workflow with the imported workflow. • **Keep Both**—Import the workflow as a new version. • **Skip**—Do not import the workflow.
Custom Tasks	Choose from the following options to specify how identically named custom tasks are handled: • **Replace** • **Keep Both** • **Skip**
Script Modules	Choose from the following options to specify how identically named script modules are handled: • **Replace** • **Keep Both** • **Skip**
Activities	Choose from the following options to specify how identically named activities are handled: • **Replace** • **Keep Both** • **Skip**

Name	Description
Import Workflows to Folder	Check **Import Workflows to Folder** to import the workflows. If you do not check **Import Workflows to Folder** and if no existing version of a workflow exists, that workflow is not imported.
Select Folder	Choose a folder into which to import the workflows. If you chose **[New Folder..]** in the drop-down list, the **New Folder** field appears.
New Folder	Enter the name of the new folder to create as your import folder.

Step 9 Click **Import**.

Exporting Workflows, Custom Tasks, Script Modules, and Activities

To export artifacts from Cisco UCS Director, do the following:

Step 1 Click **Export**.

Step 2 On the **Select Workflows** screen, choose the workflows that you want to export.

Note Custom workflows, tasks, and scripts created in Cisco UCS Director before version 6.6 can fail to import if they contain XML data.

Step 3 Click **Next**.

Step 4 On the **Select Custom Tasks** screen, choose the custom tasks that you want to export.

Note The exported custom task contains all custom inputs that are used by that custom task.

Step 5 Click **Next**.

Step 6 On the **Export: Select Script Modules** screen, choose the script modules that you want to export.

Step 7 Click **Next**.

Step 8 On the **Export: Select Activities** screen, choose the activities that you want to export.

Step 9 Click **Next**.

Step 10 On the **Export: Confirmation** screen, complete the following fields:

Name	Description
Exported By	Your name or a note on who is responsible for the export.
Comments	Comments about this export.
Exported File Name	The name of the file on your local system. Type only the base filename; the file type extension (`.wfdx`) is appended automatically.

Step 11 Click **Export**.

You are prompted to save the file.

Cloning a Custom Workflow Task from the Task Library

You can clone tasks in the task library to use in creating custom tasks. You can also clone a custom task to create a custom task.

The cloned task is a framework with the same task inputs and outputs as the original task. However, the cloned task is a framework only. This means that you must write all the functionality for the new task in CloupiaScript.

Note also that selection values for list inputs, such as dropdown lists and lists of values, are carried over to the cloned task only if the list values are not system-dependent. Such things as names and IP addresses of existing systems are system-dependent; such things as configuration options supported by Cisco UCS Director are not. For example, user groups, cloud names, and port groups are system-dependent; user roles, cloud types, and port group types are not.

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click **Clone From Task Library**.

Step 4 On the **Clone from Task Library** screen, check the row with the task that you want to clone.

Step 5 Click **Select**.
A custom workflow task is created from the task library. The new custom task is the last custom task in the Custom Workflow Tasks report. The new custom task is named after the cloned task, with the date appended.

Step 6 Click **Submit**.

What to do next

Edit the custom workflow task to ensure that the proper name and description are in place for the cloned task.

Cloning a Custom Workflow Task

You can use an existing custom workflow task in Cisco UCS Director to create a custom workflow task.

Before you begin

A custom workflow task must be available in Cisco UCS Director.

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click the row with the custom workflow task that you want to clone.
The **Clone** icon appears at the top of the custom workflow tasks table.

Step 4 Click **Clone**.

Step 5 On the **Clone Custom Workflow Task** screen, update the required fields.

Step 6 Click **Next**.
The inputs defined for the custom workflow tasks appear.

Step 7 Click the row with the task input that you want to edit and click **Edit** to edit the task inputs.

Step 8 Click **Add** to add a task input entry.

Step 9 Click **Next**.
Edit the task outputs.

Step 10 Click **Add** to add a new output entry.

Step 11 Click **Next**.

Step 12 Edit the controller scripts. See Controlling Custom Workflow Task Inputs, on page 26.

Step 13 Click **Next**.

Step 14 To customize the custom task, edit the task script.

Step 15 Click **Submit**.

Controlling Custom Workflow Task Inputs

Using Controllers

You can modify the appearance and behavior of custom task inputs using the controller interface available in Cisco UCS Director.

When to Use Controllers

Use controllers in the following scenarios:

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

With input controllers you can do the following:

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

Marshalling and Unmarshalling GUI Form Objects

Controllers are always associated with a form in the **Workflow Designer's** task inputs interface. There is a one-to-one mapping between a form and a controller. Controllers work in two stages, *marshalling* and *unmarshalling*. Both stages have two substages, before and after. To use a controller, you marshall (control UI form fields) and/or unmarshall (validate user inputs) the related GUI form objects using the controller's scripts.

The following table summarizes these stages.

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

Building Controller Scripts

Controllers do not require any additional packages to be imported.

You do not pass parameters to the controller methods. Instead, the Cisco UCS Director framework makes the following parameters available for use in marshalling and unmarshalling:

Parameter	Description	Example

Page	The page or form that contains all the task inputs. You can use this parameter to do the following:
	• Get or set the input values in a GUI form.
	• Show or hide the inputs in a GUI form.
		`page.setHidden(id + ".portList", true);`
		`page.setValue(id + ".status", "No Port is up. Port List is Hidden");`

id	The unique identifier of the form input field. An id is generated by the framework and can be used with the form input field name.	`page.setValue(id + ".status", "No Port is up. Port List is Hidden");// here 'status' is the name of the input field.`

Parameter	Description	Example
Pojo	POJO (plain old Java object) is a Java bean representing an input form. Every GUI page must have a corresponding POJO holding the values from the form. The POJO is used to persist the values to the database or to send the values to an external device.	`pojo.setLunSize(asciiValue); //set the value of the input field 'lunSize'`

See Example: Using Controllers, on page 28 for a working code sample that demonstrates the controller functionality.

Example: Using Controllers

The following code example demonstrates how to implement the controller functionality in custom workflow tasks using the various methods — beforeMarshall, afterMarshall, beforeUnmarshall and afterUnmarshall.

```
/*
Method Descriptions:

Before Marshall: Use this method to add or set an input field and dynamically create and
set the LOV on a page(form).
After Marshall: Use this method to hide or unhide an input field.
Before UnMarshall: Use this method to convert an input value from one form to another form,
 for example, when you want to encrypt the password before sending it to the database.
After UnMarshall: Use this method to validate a user input and set the error message on the
 page.

*/

//Before Marshall:

/*
Use the beforeMarshall method when there is a change in the input field or to dynamically
create LOVs and to set the new input field on the form before it gets loaded.
 In the example below, a new input field 'portList' is added on the page before the form
is displayed in a browser.
*/
importPackage(com.cloupia.model.cIM);
importPackage(java.util);
importPackage(java.lang);

var portList = new ArrayList();
var lovLabel = "eth0";
var lovValue = "eth0";

var portListLOV = new Array();
portListLOV[0] = new FormLOVPair(lovLabel, lovValue);//create the lov input field
//the parameter 'page' is used to set the input field on the form
page.setEmbeddedLOVs(id + ".portList",  portListLOV);// set the input field on the form
===========================================================================================

//After Marshall :
/*
Use this method to hide or unhide an input field.
*/
page.setHidden(id + ".portList",  true); //hide the input field 'portList'.
page.setValue(id + ".status", "No Port is up.  Port List is Hidden");
page.setEditable(id + ".status", false);

=========================================================================================

//Before Unmarshall :

/*
  Use the beforeUnMarshall method to read the user input and convert it to another form
before inserting into the database. For example, you can read the password and store the
password in the database after converting it into base64 encoding, or read the employee
name and convert to the employee Id when the employee name is sent to the database.

In the code example below the lun size is read and converted into an ASCII value.

*/
importPackage(org.apache.log4j);
importPackage(java.lang);
importPackage(java.util);

var size = page.getValue(id + ".lunSize");
var logger = Logger.getLogger("my logger");

if(size != null){
                logger.info("Size value "+size);
                if((new java.lang.String(size)).matches("\\d+")){
                                var byteValue = size.getBytes("US-ASCII"); //convert the
lun size and get the  ASCII character array
                                var asciiValueBuilder = new StringBuilder();
                                for (var i = 0; i < byteValue.length; i++) {
                                                asciiValueBuilder.append(byteValue[i]);
                                }
                                var asciiValue = asciiValueBuilder.toString()+" - Ascii
value"
                                //id + ".lunSize" is the identifier of the input field
                                page.setValue(id + ".lunSize",asciiValue); //the parameter
 'page' is used to set the value on the input field .
                                pojo.setLunSize(asciiValue); //set the value on the pojo.
This pojo will be send to DB or external device.
                }
}

=========================================================================================
// After unMarshall :

/*
Use this method to validate and set an error message.
*/
importPackage(org.apache.log4j);
importPackage(java.lang);
importPackage(java.util);

//var size = pojo.getLunSize();
var size = page.getValue(id + ".lunSize");
var logger = Logger.getLogger("my logger");
logger.info("Size value "+size);
if (size > 50) { //validate the size
                page.setError(id+".lunSize", "LUN Size can not be more than 50MB "); //set
 the error message on the page
                page.setPageMessage("LUN Size can not be more than 50MB");
                //page.setPageStatus(2);
}
```

Using Output of a Previous Task in a Workflow

You can use the output of a previous task as an input for an another task in a workflow directly from the script of a custom task and an Execute Cloupia Script task of the task library.

To access this output, you can use one of the following ways:

- Retrieve the variable from the workflow context using the **getInput()** method.
- Refer to the output using system variable notation.

To retrieve an output using the context getInput() method, use:

```
var name = ctxt.getInput("PreviousTaskName.outputFieldName");
```

For example:

```
var name = ctxt.getInput("custom_task1_1684.NAME"); // NAME is the name of the task1 output
 field that you want to access
```

To retrieve an output using system variable notation, use:

```
var name = "${PreviousTaskName.outputFieldName}";
```

For example:

```
var name = "${custom_task1_1684.NAME}"; // NAME is the name of the task1 output field that
 you want to access
```

Example: Creating and Running a Custom Task

To create a custom task, do the following:

Step 1 Choose **Orchestration**.

Step 2 Click **Custom Workflow Tasks**.

Step 3 Click **Add** and key in the custom task information.

Step 4 Click **Next**.

Step 5 Click + and add the input details.

Step 6 Click **Submit**.

Step 7 Click **Next**.
The **Custom Task Outputs** screen is displayed.

Step 8 Click + and add the output details for the custom task.

Step 9 Click **Next**.
The **Controller** screen is displayed.

Step 10 Click + and add the controller details for the custom task.

Step 11 Click **Next**.
The **Script** screen is displayed.

Step 12 Select JavaScript as the execution language and enter the following script to execute.

```
logger.addInfo("Hello World!");
logger.addInfo("Message "+input.message);
```

where **message** is the input field name.

Step 13 Click **Save Script**.

Step 14 Click **Submit**.
The custom task is defined and added to the custom tasks list.

Step 15 On the **Orchestration** page, click **Workflows**.

Step 16 Click **Add** to define a workflow, and define the workflow inputs and outputs.
Once the workflow inputs and outputs are defined, use the Workflow Designer to add a workflow task to the workflow.

Step 17 Double-click a workflow to open the workflow in the **Workflow Designer** screen.

Step 18 On the left side of the Workflow Designer, expand the folders and choose a custom task (for example, 'Hello world custom task').

Step 19 Drag and drop the chosen task to the workflow designer.

Step 20 Complete the fields in the **Add Task (<Task Name>)** screen.

Step 21 Connect the task to the workflow. See *Cisco UCS Director Orchstration Guide*.

Step 22 Click **Validate workflow**.

Step 23 Click **Execute Now** and click **Submit**.

Step 24 See the log messages in the **Service Request** log window.

CHAPTER 5

Managing Reports

Accessing Reports

You can access reports using CloupiaScript. You can use the report data to make dynamic decisions for subsequent tasks.

For example, to allocate an unassociated Cisco UCS B-Series Blade Server that is greater than 32GB, use the following script to query the list of all Cisco UCS servers that are managed by a specific Cisco UCS Manager. The script shows how to filter a subset of values selectively. The `getReportView(reportContext, reportName)` function takes reportContext and reportName as arguments and returns the TableView object which displays the content in a table format.

```
importPackage(java.lang);
importPackage(java.util);
importPackage(com.cloupia.lib.util.managedreports);

function getReport(reportContext, reportName)
{
     var report = null;
      try
      {

             report = ctxt.getAPI().getConfigTableReport(reportContext, reportName);
      } catch(e)
      {
      }

      if (report == null)
      {
             return ctxt.getAPI().getTabularReport(reportName, reportContext);
      } else
     {
           var source = report.getSourceReport();
           return ctxt.getAPI().getTabularReport(source, reportContext);
     }
}

function getReportView(reportContext, reportName)
{
      var report = getReport(reportContext, reportName);
```

```
        if (report == null)
        {
              logger.addError("No such report exists for the specified context "+reportName);

              return null;
        }

        return new TableView(report);
   }

   // following are only sample values and need to be modified based on actual UCSM account
   name
   var ucsmAccountName = "ucs-account-1";

   // report name is obtained from Report Meta. No need to change unless you need to access a
    different report
   var reportName = "UcsController.allservers.table_config";

   var repContext = util.createContext("ucsm", null, ucsmAccountName);
   // Enable Developer Menu in UCSD and find reportName in the Report Metadata for the specific
    report
   // Creating a ReportContext
   // @param contextName
   // Refer to UCSD API Guide for the available contexts
   // @param cloud
   // should be null unless contextName is "cloud" or "host node"

   // @param value
   // identifier of the object that is going to be referenced

   Report var report = getReportView(repContext, reportName);

   // Get only the rows for which Server Type column value is B-Series
   report = report.filterRowsByColumn("Server Type", "B-Series", false);

   // now look for unassociated servers only
   report = report.filterRowsByColumn("Operation State", "unassociated", false);

   // Make sure servers are actually in available state
   report = report.filterRowsByColumn("Availability", "available", false);

   var matchingIds = [];
   var count = 0;


   // Now look for Servers with memory of 32 GB or more
   for (var i=0; i<report.rowCount(); i++)
   {

         var memory = Integer.parseInt(report.getColumnValue(i, "Total Memory (MB)"));

         logger.addDebug("Possible Server "+report.getColumnValue(i, "ID")+", mem="+memory);

         if (memory >= 32*1024)
         {
                  matchingIds[count++] = report.getColumnValue(i, "ID");
         }
   }

   if (count == 0)
   {
        ctxt.setFailed("No servers matched the criteria");
        ctxt.exit();
   }
```

```
// Now randomly pick one of the item from the filtered list
var id = matchingIds[Math.round(Math.random()*count)];
logger.addInfo("Allocated server "+id);

// Save the Server-ID to the global inputs
ctxt.updateInput("SELECTED_UCS_SERVER_ID", id);
```

Accessing Tabular Reports

If you use the `getTabularReport(reportName, reportContext)` API to access a tabular report, you can view the report details in the user interface (UI) in one of the following ways:

- Reports Customization—To access the reports customization tab, choose **Administration** > **User Interface Settings** and click **Reports Customization**. The customization report displays the report details such as menu, context, report type, and so on. To customize the table columns, click the down arrow that appears on the column header when you place the cursor. From the drop-down menu, choose **Cloumns** and check the columns to be shown. For example, to display the report ID, check **ID**.
- Report Metadata—The report metadata link appears in the UI only when the developer menu is enabled.

Some of the important report details are:

- API report ID—You can use the API report ID column to get the value for the reportID parameter to use in the REST URL when you use the userAPIGetTabularReport API. This REST API is used to retrieve the tabular report from a web browser or other REST client application.
- ID—The ID column displays the report name. You can use the ID column to get the reportName parameter when you use the getTabularReport API in CloupiaScript. This parameter is also applicable for the getConfigTableReport API.
- context—To construct the ReportContext, you need two input parameters: contextName and contextValue. For regular contexts, use `util.createContext ("contextName", null, "instanceName")`. For example, `util.createContext("vm",null, vmId)`, where vmId is the integer VM ID value to uniquely identify a VM in UCS Director. For cloud contexts, use `util.createContext("contextName","cloudInstanceName", null)` , or `util.createContext("contextName",null,"cloudInstanceName")`. For example, `util.createContext("cloud","All Clouds",null)` , or `util.createContext("cloud", null, "All Clouds")`.

Emailing Reports

You can use CloupiaScript to email a report to a user. To email this report periodically, set up a workflow schedule for this workflow at the desired frequency.

The following script enables user to choose a report name from the report list and email the report to the specified email address.

```
importPackage(java.util);
importPackage(java.lang);
importPackage(java.io);
importPackage(com.cloupia.model.cEvent.notify);
importPackage(com.cloupia.model.cIM);
importPackage(com.cloupia.lib.util.mail);
importPackage(com.cloupia.fw.objstore);
importPackage(com.cloupia.lib.util.managedreports);
importPackage(com.cloupia.lib.util);
importPackage(com.cloupia.service.cIM.inframgr);
importPackage(org.apache.commons.httpclient);
importPackage(org.apache.commons.httpclient.cookie);
importPackage(org.apache.commons.httpclient.methods);
importPackage(org.apache.commons.httpclient.auth);
importPackage(com.cloupia.model.cEvent.notify);

function getMailSettings()
{
      return ObjStoreHelper.getStore((new MailSettings()).getClass()).getSingleton();
}
function getReport(reportContext, reportName)
{
     var report = null;
      try
      {
             report = ctxt.getAPI().getConfigTableReport(reportContext, reportName);
      } catch(e)

      {
      }
      if (report == null)
      {
             return ctxt.getAPI().getTabularReport(reportName, reportContext);
      } else
     {
           var source = report.getSourceReport();
           return ctxt.getAPI().getTabularReport(source, reportContext);
     }
}
function getReportView(reportContext, reportName)
{
      var report = getReport(reportContext, reportName);
     if (report == null)
     {
           logger.addError("No such report exists for the specified context "+reportName);
           return null;
     }
     return new TableView(report);
}
var ucsmAccountName = ctxt.getInput("Account_Name");
var reportName = ctxt.getInput("REPORT_NAME");

var reportContextType=ReportContext.getDynamicContextLevel("apic_controller");
var repContext = util.createContextByType(reportContextType, null, ucsmAccountName);

var report = getReportView(repContext, reportName);
logger.addInfo("Report Context Type is :::::::::: " + reportContextType);
//var repContext = util.createContext("apic_controller", null, ucsmAccountName);
//var report = getReportView(repContext, reportName);
var numRowsFound = report.rowCount();
logger.addInfo("Number of Rows found: " + numRowsFound);
var toEmail = [ ctxt.getInput("Email Address") ];
var message = new EmailMessageRequest();

message.setToAddrs(toEmail);
message.setSubject("APIC Report : "+ ucsmAccountName);
message.setFromAddress("no-reply@cisco.com");
var buffer = new StringWriter();
var printer = new PrintWriter(buffer);
var formatter = new com.cloupia.lib.util.managedreports.Formatter(new File("."), printer);

formatter.printTable(report);
```

```
printer.close();
var body = "<head><style type='text/css'>";
body = body + "table { font-family: Verdana, Geneva, sans-serif; font-size: 12px; border:
thin solid #039; border-spacing: 0; background: #ffffff; } ";
body = body + " th { background-color: #6699FF; color: white; font-family: Verdana, Geneva,
 sans-serif; font-size: 10px; font-weight: bold; border-color: #CCF; border-style: solid;
border-width: 1px 1px 0 0; margin: 0; padding: 5px; } ";
body = body + " td { font-family: Verdana, Geneva, sans-serif; font-size: 10px; border-color:
 #CCF; border-style: solid; border-width: 1px 1px 0 0; margin: 0; padding: 5px; background:
 #ffffff; }";
body = body + "</style></head>";
body = body+ "<body><h1>APIC Report</h1><br>" + buffer.toString();

message.setMessageBody(body);
logger.addInfo("Sending email");
MailManager.sendEmail("APIC Report", getMailSettings(), message);
```

CHAPTER 6

Best Practices

Creating a Rollback Script

When you create a custom task script, it is good practice to create a corresponding rollback script. The rollback script undoes whatever change was made in the custom task script. For example, if the custom task creates a resource, the rollback script should remove the resource.

Of course, many rollback scenarios require information about the state of the system before the custom task was executed. The CloupiaScript library contains a `ChangeTracker` API to enable you to reverse the effects of a custom task. Using the `ChangeTracker` API, you create an `UndoableResource` object that collects state information before creating a resource. During rollback, the `UndoableResource` uses this information to restore the resource to its previous state.

The `ChangeTracker` API contains two methods to enable rolling back of modification and deletion of a resource, respectively:

- `ChangeTracker.undoableResourceModified()`
- `ChangeTracker.undoableResourceDeleted()`

For an example of how to use the ChangeTracker API to create a rollback script, see the *Cisco UCS Director CloupiaScript Cookbook* available at the following URL: http://www.cisco.com/c/en/us/support/servers-unified-computing/ucs-director/products-programming-reference-guides-list.html.