Cisco UCS Director Custom Task Getting Started Guide, Release 6.0

First Published: 2016-09-16

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`.

Text Type	Indication
System output	Terminal sessions and information that the system displays appear in `this font`.
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.

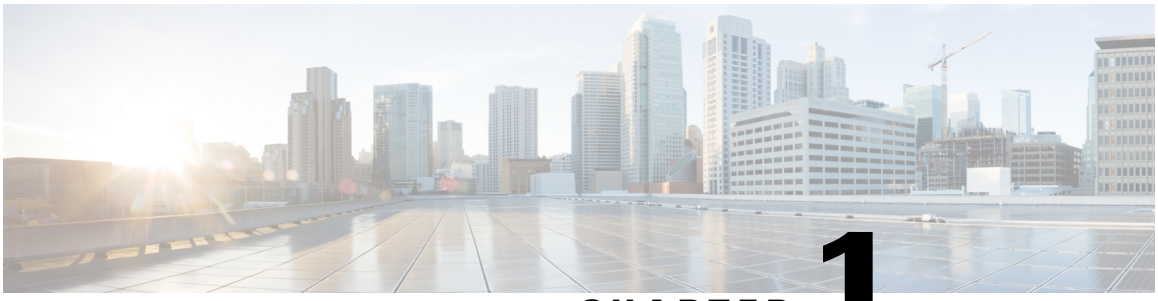
Documentation Feedback

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 What's New in Cisco Product Documentation RSS feed. RSS feeds are a free service.



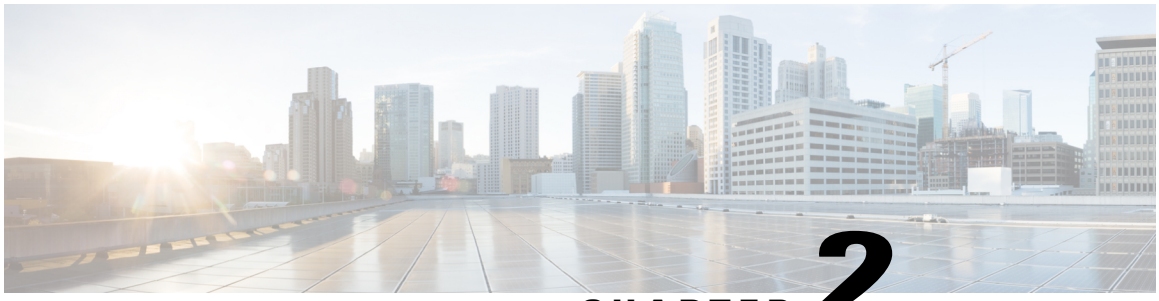
CHAPTER 1

New and Changed Information for this Release

New and Changed Information for this Release

No significant changes were made to this guide for the current release.

New and Changed Information for this Release



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. Additionally, CloupiaScript 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**

  Till Cisco UCS Director Release 5.3, you have to 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);
```

From Cisco UCS Director Release 5.4, you have to 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 values 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 property of primitive or string types. The best practice is to use the standard getter methods such as, getVmId() and getVmName() to retrieve the values 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 class<T> parameter to method or constructor**

  The syntax for passing the Class<T> parameter to 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 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, while the earlier import statement made the class or package available in the global space of the javascript execution, which was not always required.

  Import statements in the Rhino JavaScript Engine:

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

  Import statements 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) would be loaded in its memory. For example, sometimes it is useful to import many Java packages at a time. We can use 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, the static method could be accessed not only through the class name (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();// No 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`.

  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 UCS Director and native JSON object, UCS Director stores the native class using the name NativeJSON. So for example the following are static method calls of the native object:

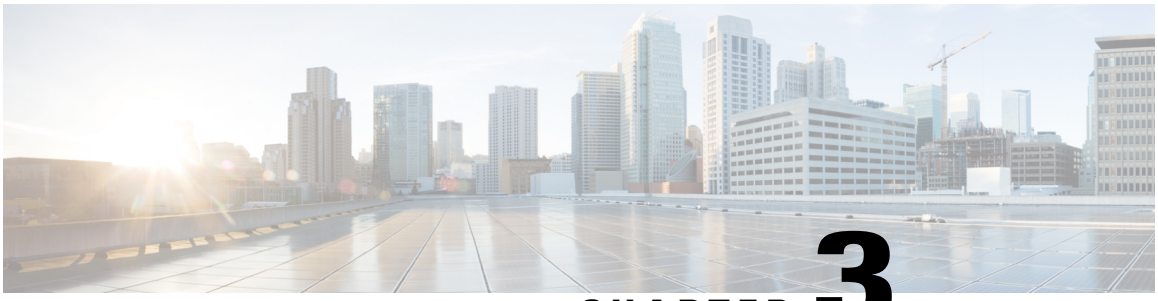
  ```
  NativeJSON.stringify(object myObj); OR
  NativeJSON.parse(String mystr);
  ```

- **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
  ```



CHAPTER 3

CloupiaScript Interpreter

About 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.

The CloupiaScript interpreter offers the following built-in functions:

- `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 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Click the **Custom Workflow Tasks** tab.

Step 3 Click **Launch Interpreter**.

The **Cloupia Script Interpreter** dialog box appears.

Step 4 Click in the text input field at the bottom of the interpreter dialog.

Step 5 Enter a line of JavaScript code and 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 and launch the CloupiaScript Interpreter with all the context variables that are defined for executing that custom task.

When you launch the interpreter, the interpreter prompts you for values of the custom task's input fields and populates the input object of the task. All the variables that would be available if you were actually executing the custom task are made available.

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

Step 1 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Click the **Custom Workflow Tasks** tab.

Step 3 Choose a custom task for which you need to test the JavaScript.

Step 4 Click the **Launch Interpreter with Context** action.
The **Launch Interpreter** dialog box 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 form.

Step 6 Click **Submit**.

Step 7 Click **Submit**.
The **Cloupia Script Interpreter** dialog box appears.

Step 8 Click in the text input field at the bottom of the interpreter dialog.

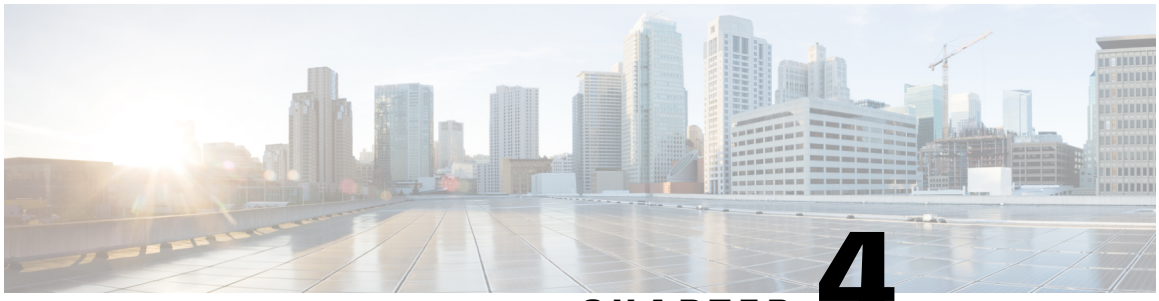
Step 9 Enter a line of JavaScript code and press **Enter**.
The code is executed and the result is displayed. If there is any syntax error in the code, the error is displayed.

Example: Using the CloupiaScript Interpreter

The printObj function prints all the properties and methods it contains. You can call the `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. You can use the input type list to define the input for custom workflow 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.

Creating a Custom Workflow Input

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

Step 1 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Choose the **Custom Workflow Inputs** tab.

Step 3 Click the **Add** icon.

Step 4 In the **Add Custom Workflow Input** dialog box, complete the following fields:

Name	Description
Custom Input Type Name field	A unique name for the custom input type.
Input Type button	Choose a type of input. Based on the selected input, the other fields appear. For example, when you choose the **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.

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 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Choose the **Custom Workflow Inputs** tab.

Step 3 Choose the custom workflow input that needs to be cloned.

The **Clone** icon appears at the top of the custom workflow inputs table.

Step 4 Click the **Clone** icon.

Step 5 In the **Custom Input Type Name** field, type a name for the new input.

Step 6 Use the other controls in the **Clone Custom Workflow Input** dialog box 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 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Choose the **Custom Workflow Tasks** tab.

Step 3 Click the **Add** icon.

Step 4 In the **Add Custom Workflow Task** dialog box, complete the following fields:

Name	Description
Task Name field	A unique name for the custom workflow task.
Task Label field	A label to identify the custom workflow task.
Activate Task check box	If checked, the custom workflow task is registered with Orchestrator and is immediately usable in workflow.
Register Under Category field	The category under which the custom workflow task is registered.
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 Workflow Tasks Inputs** window appears.

Step 6 Click the **Add** icon.

Step 7 In the **Add Entry to Inputs** dialog box, complete the following fields:

Name	Description
Input Field Name field	A unique name for the field. The name must start with an alphabetic character and must not contain spaces or special characters.

Name	Description
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 button	Choose a type of input 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.
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.
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.
Text Field Attributes area—Complete the following fields when the input field type is text.	
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.	

Name	Description
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.

Step 8 Click **Submit**.
A successful entry addition message appears.

Step 9 Click **OK**.

Step 10 Click the **Add** icon to add more entry to inputs.

Step 11 Click **Next**.
The **Custom Workflow Tasks Outputs** window appears.

Step 12 Click the **Add** icon.

Step 13 In the **Add Entry to Outputs** dialog box, complete the following fields:

Name	Description
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 button	Choose a type of output. This type determines how the output can be mapped to other task inputs.

Step 14 Click **Submit**.
A successful entry addition message appears.

Step 15 Click **OK**.

Step 16 Click the **Add** icon to add more entry to outputs.

Step 17 Click **Next**
The **Controller** window appears.

Step 18 (Optional) Click the **Add** icon to add a controller.

Step 19 In the **Add Entry to Controller** dialog box, complete the following fields:

Name	Description
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.
Script text area	For the method you chose from the **Method** drop-down list, add the code for the GUI customization script here. **Note** Click the **Add** icon if you want to add code for more methods.

Step 20 Click **Submit**.
A successful entry addition message appears.

Step 21 Click **Next**.
The **Script** window appears.

Step 22 From the **Execution Language** drop-down list, choose the language.

Step 23 In the **Script** field, enter the CloupiaScript code for the custom workflow task.

Step 24 Click **Save Script**.

Step 25 Click **Submit**.
The custom workflow task is created and is available for use in the workflow.

Importing Workflows, Custom Tasks, Script Modules, and Activities

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

Step 1 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 In the **Orchestration** pane, click the **Workflows** tab.

Step 3 Click the **Import** action.

Step 4 In the **Import** dialog box, click **Upload**.

Step 5 In the **File Upload** dialog, click **Click and select a file from your computer**.

Step 6 Select the import file. Cisco UCS Director import and export files have a `.wfdx` file extension.
When the file is uploaded, the **File Upload** dialog displays `File ready for use`.

Step 7 Dismiss the **File Upload** dialog.

Step 8 Click **Next**.
The **Import** dialog displays a list of Cisco UCS Director objects contained in the uploaded file.

Step 9 (Optional) Specify how objects are handled if they duplicate names already in the workflow folder. In the **Import** dialog box, complete the following fields:

Name	Description
Workflows drop-down list	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 drop-down list	Choose from the following options to specify how identically named custom tasks are handled: • **Replace** • **Keep Both** • **Skip**
Script Modules drop-down list	Choose from the following options to specify how identically named script modules are handled: • **Replace** • **Keep Both** • **Skip**

Name	Description
Activities drop-down list	Choose from the following options to specify how identically named activities are handled: • **Replace** • **Keep Both** • **Skip**
Import Workflows to Folder check box	Check this check box to import the workflows. If you do not check the box and if no existing version of a workflow exists, that workflow is not imported.
Select Folder drop-down list	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 field	Enter the name of the new folder to create as your import folder.

Step 10 Click **Import**.

Exporting Workflows, Custom Tasks, Script Modules, and Activities

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

Step 1 On the menu bar, choose **Policies** > **Orchestration**.
Step 2 In the **Orchestration** pane, click the **Workflows** tab.
Step 3 On the **Workflows** tab, click **Export**.
Step 4 In the **Select Workflows** screen, select the workflows that you want to export.
Step 5 Click **Next**.
Step 6 In the **Select Custom Tasks** screen, select the custom tasks that you want to export.
Step 7 Click **Next**.
Step 8 In the **Export: Select Script Modules** screen, select the script modules that you want to export.
Step 9 Click **Next**.
Step 10 In the **Export: Select Activities** screen, select the activities that you want to export.
Step 11 In the **Export: Confirmation** screen, complete the following fields:

Name	Description
Exported By text field	Your name or a note on who is responsible for the export.

Name	Description
Comments text area	Comments about this export.
Exported File Name text field	The name of the file on your local system. Type only the base filename; the file type extension (`.wfdx`) is appended automatically.

Step 12 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.

The cloned task is a framework with the same task inputs and outputs as the original task. However, note that 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 On the menu bar, choose **Policies** > **Orchestration**.

Step 2 Choose the **Custom Workflow Tasks** tab.

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

Step 4 In the **Clone from Task Library** dialog box, click **Select**.

Step 5 Choose a task from the task list.

Step 6 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.

What to Do Next

Edit the custom workflow task.

Cloning a Custom Workflow Task

You can use the existing custom workflow tasks 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 On the menu bar, choose **Policies** > **Orchestration**.
Step 2 Choose the **Custom Workflow Tasks** tab.
Step 3 Choose the custom workflow task to clone.
The **Clone** icon appears at the top of the custom workflow tasks table.
Step 4 Click the **Clone** icon.
Step 5 In the **Clone Custom Workflow Task** dialog box, update the required fields.
Step 6 Click **Next**.
The inputs defined for the custom workflow tasks appear.
Step 7 Edit the task inputs.
Step 8 Click **Next**.
Edit the task outputs.
Step 9 Click the **Add** icon to add a new output entry.
Step 10 Click **Next**.
Step 11 Edit the controller scripts. See the following topic, Controlling Custom Workflow Task Inputs.
Step 12 Click **Next**.
Step 13 To customize the custom task, edit the task script.
Step 14 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 **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 the 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 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 on 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.`
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 22 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);
}
```

Example: Creating and Running a Custom Task

To create a custom task, do the following:

Step 1 Go to **Policies** > **Orchestration** > **Custom Workflow Tasks**.

Step 2 Click **Add** and key in the custom task information.

Step 3 Click **Next**.
The **Cloupia Script Interpreter** dialog box appears.

Step 4 Click+ and add the input details.

Step 5 Click **Submit**.

Step 6 Click **Next**.
The custom task output window is displayed.

Step 7 Click **Next**.
The custom task controller window is displayed.

Step 8 Click **Next**.
The script window is displayed.

Step 9 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 10 Click **Submit**.
The custom task is defined and added to the custom tasks list.

Step 11 Go to **Workflows** tab.

Step 12 Click **Add** and add a workflow.

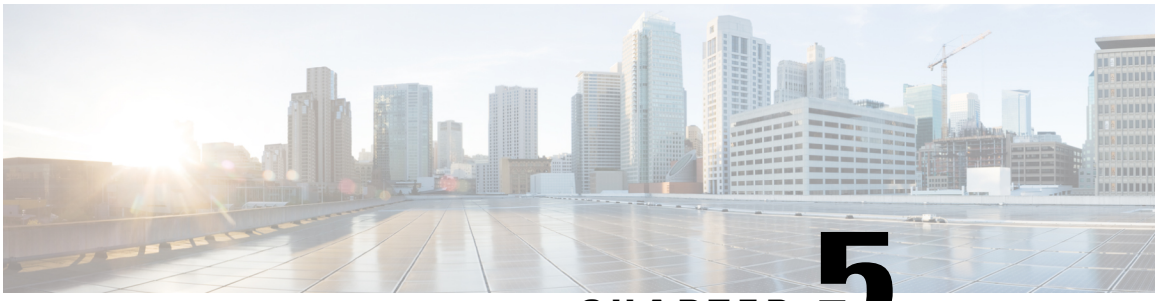
Step 13 Drag and drop the 'Hello world custom task' to the workflow designer once the workflow is created.

Step 14 Add 'Hello World custom task' to the designer.

Step 15 Click **Validate workflow**.

Step 16 Click **Execute Now** and click **Submit**.

Step 17 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 will take reportContext and reportName as arguments and will return 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 tab—To access the reports customization tab, choose **Administration** > **User Interface Settings** and choose **Reports Customization**. The customization tab displays the report details such as menu, context, report type, and so on. To customize the table columns, click the **Customize Table Columns** icon and check the checkbox of the column item to be shown. For example, to display the report ID, check the **ID** checkbox.
- 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 that is used in the REST URL when you are using 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 are using the getTabularReport API in the Cloupia script. This parameter is also applicable for the getConfigTableReport API.
- context—To construct the ReportContext, you need the two input parameters: contextName and contextValue. For regular contexts, use the 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 the 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

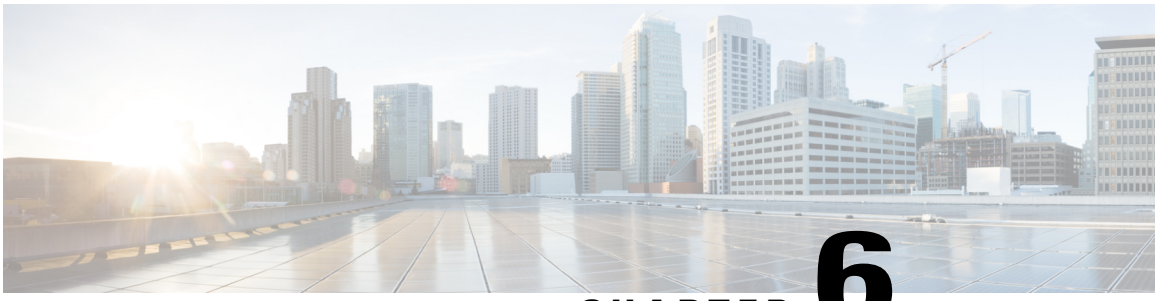
You can use the Cloupia script to email a report to a user. If you need this report to be emailed on a periodic basis, you can set up a workflow schedule for this workflow at the desired frequency.

The following script is used to email a list of all powered on VMs to a user specified in the workflow input variable *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);

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);
}

// Assume the To Email Addres is in the input variable 'Email Address'
var toEmail = [ ctxt.getInput("Email Address") ];

var message = new EmailMessageRequest();
message.setToAddrs(toEmail);
message.setSubject("VM List Reportl");
message.setFromAddress("no-reply@cisco.com");

var buffer = new StringWriter();
var printer = new PrintWriter(buffer);

// Formatter exists in multiple packages, so it needs fully qualified name
var formatter = new com.cloupia.lib.util.managedreports.Formatter(new File("."), printer);

var reportName  = "GLOBAL_VM_LIST_REPORT";
var repContext  = util.createContext("global", null, null);
var report = getReportView(repContext, reportName);

// Filter Active State VMs
report = report.filterRowsByColumn("Power State", "ON", false);
formatter.printTable(report);

printer.close();

var body = "<head><style type='text/css'>";

// Specify CSS for the report
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>List of Powered ON VMs</h1><br>" + buffer.toString();

message.setMessageBody(body);
```

```
logger.addInfo("Sending email");

// Now, send the report via email. First parameter is just a label used in the  internal
logs
MailManager.sendEmail("VM List 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 for enabling rollback 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 DirectorCloupiaScript 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.