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.