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
  ```