Managing REST API

This chapter contains the following sections:

The REST API

Once you develop your own Cisco UCS Director features as modules using the Cisco UCS Director Open Automation tool, you can develop and expose REST API support for the modules in the Cisco UCS Director GUI.

The following are terms used when using the Cisco UCS Director REST API:

- MO—The entities are instantiated as managed objects (MOs). The create or update operation must target a specific MO. MOs are exposed through the API.
- Class—Templates that define the properties and states of objects in the management information tree.
- Attribute—An attribute is a persistent piece of information that characterizes all objects in the same class.
- MoReference—An annotation providing the path reference of the MO.

To expose the REST API, create a new MO for the REST API support and register it with the connector. This enable developers to view the registered REST APIs with the respective MO entities in the Cisco UCS Director GUI. Developers can perform CRUD operations on connectors using the REST APIs.

You can execute each API by using the **REST API Browser** or the **REST Client**. See Cisco UCS Director REST API Getting Started Guide for detailed steps.

Identifying Entities

Identify MOs for REST API support from features and the objects' reports.

For example, you can retrieve all the cloud accounts in a particular physical datacenter or all the VDCs created in a particular cloud. Thus datacenter, cloud Account, and VDC are some of the entities modelled as objects that can be retrieved.

Configuring a POJO Class for REST API Support

Step 1 Implement a `TaskConfigIf` interface for each POJO class to indicate that the POJO class is exposed through REST.

Step 2 Annotate each POJO with the `XmlRootElement` tag.

Note Ensure that the POJO name is the same as the one defined for the POJO in the MO resource path. For example:

```
@XmlRootElement(name=”foo”)
```

Step 3 Annotate the identifier field of each POJO with the `@MoReference` tag.

For example,

```
(@MoReference(path=”foo.ID.account.ID.sample)
```

Step 4 Specify the POJO path in the MO definitions file (`<featurename>-api.mo` file) and in the properties file named as `<featurename>-url-mapping.properties`For example:

```
foo=foo.*.account.*.sample.*
```

This path specifies the location of the POJO in the tree hierarchy (for example: `datacenter.ID`, `cloud.ID`, `vdc.ID`).

Note You can use camel-casing in the keywords. Do not use special characters in the keywords.

Step 5 Register a `MoPointer` for each POJO that includes the API resource path, resource class, and adaptor of the MO. For example:

```
FooAdaptor fooAdaptor = new fooAdaptor ();
MoPointer p = new MoPointer("foo", "foo.ID.account.ID.sample",fooAdaptor, Foo.class);

parser.addMoPointer(p);
```

Input Controllers

Every Config file has a *Controller* that can be used to validate specific fields and modify the appearance and behavior of form inputs.

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 UCS Manager from a drop-down list, you can prompt for user credentials for UCS Manager or change the list of values (LOVs) in the drop-down list to shown only available ports on a server.
- **Validate form fields:** You can validate the data entered by a user. 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 the Cisco UCS Director database or data sources and use them to populate GUI form objects.

Marshalling and Unmarshalling UI Objects

A Controller object is always associated with a Config object. Controllers work in two stages, *marshalling* and *unmarshalling*. Both stages have two substages, *before* and *after*. These four substages correspond to four Action methods in the Controller Object. To use a controller, you marshall (control UI form fields) and unmarshall (validate user inputs) the related GUI form objects using the controller's methods.

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.

To validate fields with a controller:

1. In the Config file, use an annotation to set validate equal to `true`.
2. Implement one or more Action calls in the Controller.

Following are examples of a Config and its corresponding Controller.

Config:

Following is the start page of an example UI. The file ends with the `Config` method and implements the `TaskConfigIf` interface. `@FormField` annotations are used to design the front end.

```
@FormController(value="com.cloupia.feature.compute.api.config.controller.ComputeAccountSpecificConfigController")
@XmlRootElement(name = "ComputeAccount")
@PersistenceCapable(detachable = "true", table = "compute_account_device")
public class ComputeAccountSpecificConfig implements TaskConfigIf{
public static final String HANDLER_NAME = "Compute Account Controller";
public static final String HANDLER_LABEL = "Compute Account Controller";

public static final String TYPE_STANDARD = "0";
public static final String TYPE_ADVANCED = "1";
public static final String TYPE_CUSTOM = "2";

@Persistent
private long actionId;
@Persistent
private long configEntryId;

private int modelID;
}

// Create a controller by extending the AbstractObjectUIController.
public class ComputeAccountSpecificConfigController extends AbstractObjectUIController {

private static Logger logger = Logger.getLogger(ComputeAccountSpecificConfigController.class);
@Override
public void beforeMarshall(Page page, String id, ReportContext context, Object pojo) throws
 Exception
{
}

}
```

Controller:

Name the controller file `<ConfigName>Controller`, where *<ConfigName>* is the base name of the configuration file. In this way the framework can fetch only this particular controller.

The `AbstractObjectUIController` has four methods you can override to control UI input. These methods are called before and after both marshalling and unmarshalling, and have different functions with respect to validation.

Before Marshall

Before page data is loaded, the `beforeMarshall` method is called to validate the loaded page data.

Example:

```
Public void beforeMarshall (Page page, String id, ReportContext context, Object pojo)
throws Exception
{
  logger.info ("In Controller before Marshall " + context.getId());
  ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
  logger.info(" before Marshall ");
  page.setEmbeddedLOVs(id + ".chargeDuration", getDurationLOV());
}
```

After Marshall

After page data is loaded, the `afterMarshall` method is called to validate the fields and hide on any fields.

Example:

```
Public void afterMarshall (Page page, String id, ReportContext context, Object pojo)
throws Exception
{
  ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
  page.setEditable (id + ".accountName", false);
  String accountName = context.getId();
}
```

Before Unmarshall

Before the UI is loaded, the `beforeUnmarshall` method is called.

```
Public void beforeUnmarshall (Page page, String id, ReportContext context, Object pojo)
 throws Exception
{
}
```

After Unmarshall

After form submission, the `afterUnmarshall` method is called. This method does page validation.

Example:

```
Public void afterUnmarshall(Page page, String id, ReportContext context, Object pojo)
throws Exception
{
 ComputeAccountSpecificConfig config = (ComputeAccountSpecificConfig) pojo;
 if (page.isPageSubmitted()) {
  String accountName = config.getAccountName();
  String gateWayAddress = ipAddressBlock.getDefGw();
  String Size = ipAddressBlock.getSize();
  String toAddress = calculateIPRange(fromAddress, Size);
  if (!validateAccountName(accountName)) {
   page.setPageMessage("Invalid account name. Please use only characters. Special
characters are not allowed.");
   throw new Exception ("Invalid account name. Please use only characters. Special
characters are not allowed.");
  }
 }
}
```

Implementing a Workflow Task

Each workflow task has a configuration class and a respective task handler for the task. The configuration class and the task handler implement the `TaskConfigIf` and `AbstractTask` classes, respectively.

To implement the workflow tasks, the framework must accommodate two scenarios:

- The existence of a workflow configuration class and its handler in the presence of a front-end POJO that is used to get the database information. For example: `DummyAccount`, `DummyAccountCreateConfig.class`, and `DummyAccountHandler.java`.
- The existence of a workflow configuration class and its handler in the absence of a front-end POJO. For example: `DummyAccountCreateConfig`.

Log Files

The API logs are stored in a `logfile.txt` file located in the `/op/infra/inframgr` directory.

The log file includes the following information:

- `INFO` (the severity keyword)

- The date-timestamp in the UTC format: `yyyy/MM/dd-HH:mm:ss,SSS`.
- The Java class where instrumentation is implemented.
- The name of the instrumented action.

See Cisco UCS Director Open Automation Troubleshooting Guide for scenarios that you may encounter and the recommended ways to handle them.

Examples

Example 1

The following code snippet provides registration for REST API support using an adaptor-based handler:

```
/* A REST adaptor is used to handle the CRUD operations for resources.
* You can extend the adaptor functionality by inheriting the WFTaskRestAdaptor.
* You can override the executeCustomAction, getTaskConfigImplementation, getTaskName and
getTaskOutputDefinitions methods according to need.
*/
WFTaskRestAdaptor restAdaptor = new WFTaskRestAdaptor();
/* MoPointer is a placeholder to register the REST APIs.
* @param0 is a mandatory field, and is used to define a resource name.
* @param1 is a mandatory field, and is used to define a ResourceURL.
* @param2 is a mandatory field, and is used to define a restAdaptor.
* @param3 is a mandatory field, and is used to define the resource config class.
* If you don't want to allow the read operation for an API, use the following constructor:
* MoPointer(String name, String path, MoResourceListener moListener, Class moModel, boolean
 isMoPersistent, boolean isReadAPISupported)
* mopointer must be registered in order for the API to display in REST API browser.
*/
MoPointer p = new MoPointer("ComputeAccount", "ComputeAccount", restAdaptor,
HelloWorldConfig.class);
/*
* The createOARestOperation method is used to register REST API operations through the Open
 Automation connector.
* @param0 is a mandatory field, and is used to define an operation name.
* @Param1 is a mandatory field, and is used to define a resource handler name. The handler
 name is used for handling the REST API operations.
* @param2 is a mandatory field, and is used to define a resource config class. The resource
 config class is required for executing a handler operation. .
*/
p.createOARestOperation("CREATE_OA", ComputeAccountCreateConfig.HANDLER_NAME,
ComputeAccountCreateConfig.class);
p.createOARestOperation("DELETE_OA", ComputeAccountDeleteConfig.HANDLER_NAME,
ComputeAccountDeleteConfig.class);
p.createOARestOperation("UPDATE_OA", ComputeAccountUpdateConfig.MODIFY_HANDLER_NAME,
ComputeAccountUpdateConfig.class);
/*
* Category is used for the REST API browser folder structure.
*/
p.setCategory(ComputeConstants.REST_API_FOLDER_NAME);
/*
* The registered REST APIs are communicated to the framework through a MoParser. Loading
REST APIs in the framework is mandatory.
*/
parser.addMoPointer(p);
```

Example 2

The following code snippet provides registration for REST API support using a Listener-based handler.

```
/** REST Listener is used to handle the CRUD operations for the Resource.
* You can extend the Listener functionality by inheriting the AbstractResourceHandler.
* You can override the methods createResource, updateResource, deleteResource and query
according to need.
*/
ComputeResourceAPIListner nPolcyList = new ComputeResourceAPIListner();
/* MoPointer is a placeholder to register the REST APIs.
* @param0 is a mandatory field, and is used to define a resource name.
* @param1 is a mandatory field, and is used to define a ResourceURL.
* @param2 is a mandatory field, and is used to define a Listener implementation class.
* @param3 is a mandatory field, and is used to define the resource config class. mandatory
 field.
* If you don't want to allow the read operation for an API, use the following constructor:
* MoPointer(String name, String path, MoResourceListener moListener, Class moModel, boolean
 isMoPersistent, boolean isReadAPISupported)
* mopointer must be registered in order for the API to display in REST API browser.
*/
MoPointer p = new MoPointer("ComputeResource", "ComputeResource", new
ComputeResourceAPIListner, ComputeAccountListnerConfig.class);
p.setSupportedOps(MoOpType.CREATE, MoOpType.UPDATE, MoOpType.DELETE);
p.setCategory(ComputeConstants.REST_API_FOLDER_NAME);
parser.addMoPointer(p);
```

Example 3

The following code snippet provides the configuration classes for workflow task actions without the front-end POJO for the read operation.

```
WFTaskRestAdaptor adaptor = new WFTaskRestAdaptor();
boolean isMoPersistent = false;
p = new MoPointer("ComputeAccountConfig" ,"ComputeAccount", adaptor, null, isMoPersistent);

p.createOARestOperation("CREATE_OA", ComputeAccountCreateConfig.class);
parser.addMoPointer(p);
```

Note If you pass the parameter as null, the REST API will not support the READ operation.

The following list suggests URIs for the MO defined in all the provided examples.

- URI for GET—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and `<name>` is the MO pointer name specified during the registration.

  Exception: The GET URI for a configuration class that does not have a front end POJO is handled as a default operation by Cisco UCS Director.

- URI for POST—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and `<name>` is the MO pointer name specified during the registration (`ComputeAccountCreateConfig` in the example). Provide the configuration POJO (`ComputeAccountCreateConfig`) as an XML payload in the HTTP request body.

- URI for UPDATE—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and where `<name>` is the MO pointer name specified during the registration. Specify the XML form of the configuration POJO in the HTTP request body.

- URI for DELETE—`http://<address>/cloupia/api-v2/<name>`, where `<address>` is the IP address of the server and where `<name>` is the MO pointer name specified during the registration. Specify the XML form of the configuration POJO in the HTTP request body.

Invoking the REST API Using a Python Script

The Cisco UCS Director REST APIs can be invoked from an external system. This section provides an example of how to invoke the REST API from a Python script.

The example showcases only the set of APIs available in Cisco UCS Director after uploading and enabling the Open Automation (OA) compute module zip file. Other Cisco UCS Director REST APIs can be invoked in similar fashion. Cisco UCS Director partners can develop their own OA modules and expose REST APIs for their custom OA module features. The REST API for the custom OA module feature is available after uploading and enabling it on the Cisco UCS Director server.

Finally, REST APIs exposed through such a custom OA module can be executed from a Python script like the one that follows.

Prerequisites

Before running the following script, you must meet the following prerequisites:

On the Cisco UCS Director server, upload and enable the Open Automation compute module `zip` file. See The REST API, on page 1.

On the clent, do the following:

- Install Python version 2, release 2.6 or later
- Install the Python `requests` http library
- Install the Python `lxml` XML library
- Acquire the server IP address and REST API Access Key for your Cisco UCS Director installation

Note The `requests` and `lxml` libraries are not available in the default Python installation, and must be installed separately. The following script uses `requests` to make HTTP REST API calls and `lxml` to construct XML payloads. You can use other libraries instead to handle these tasks.

Python Script

```
# This script demonstrates how to invoke the Cisco UCS Director XML REST API
#
# This script requires Python version 2, release 2.6 or later
# The Python 'requests' HTTP library is used to to invoke the REST API
# Cisco UCSD XML REST API payload and response are in XML format
# The Python 'lxml' XML library is used to construct the XML payload and to parse the XML
response
#
# Python executable

import sys
import requests
```

```
from lxml import etree
import logging

# Use the logging module to log events
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Read the UCSD server IP and REST key from the command line
try:
 number_of_arguments = len(sys.argv)
 if number_of_arguments != 3:
  raise Exception("Invalid number of arguments !! Please run script as 'python
script_file_name.py ipAddress UCSD_REST_KEY'")
except Exception as e:
 logger.error("Exception occurred while executing this script : " +str(e))
 sys.exit(1)
# IP address of UCSD server on which the REST API is to be invoked
IP_address = sys.argv[1]

# Headers for HTTP requests
headers = {}

# Authenticate the user with the REST key
rest_key_value = sys.argv[2]
content_type = "application/xml"
headers["X-Cloupia-Request-Key"] = rest_key_value
headers["content-type"] = content_type

custom_operation_type = None
resource_URL = None
http_request_type = None
resource_complete_URL = None
response = None
payload_data = None

def main():
 # Invoke the HTTP POST REST API operation 'CREATE_COMPUTE_ACCOUNT' to create a ComputeAccount

 # custom_operation_type = CREATE_COMPUTE_ACCOUNT,
 # resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

 custom_operation_type = "CREATE_COMPUTE_ACCOUNT"
 resource_URL = "/cloupia/api-v2/ComputeAccount"
 http_request_type = "POST"
 resource_complete_URL = "https://" + IP_address + resource_URL

 # Construct XML Payload for CREATE_COMPUTE_ACCOUNT
 cuic_operation_request = etree.Element('cuicOperationRequest')
 operation_type = etree.SubElement(cuic_operation_request, 'operationType')
 operation_type.text = custom_operation_type
 payload = etree.SubElement(cuic_operation_request, 'payload')
 compute_account = etree.Element('ComputeAccount')
 account_name = etree.SubElement(compute_account, 'accountName')
 account_name.text = 'sdk_compute'
 status = etree.SubElement(compute_account, 'status')
 status.text = 'On'
 ip = etree.SubElement(compute_account, 'ip')
 ip.text = '1.1.1.1'
 inner_text = etree.tostring(compute_account)
 payload.text = etree.CDATA(inner_text)
 payload_data = etree.tostring(cuic_operation_request)

 logger.info("Executing HTTP POST CREATE_COMPUTE_ACCOUNT REST API...")
 logger.info("payload = %s",payload_data)
```

```
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP GET REST API 'Read' operation to read all ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None
logger.info("Executing HTTP GET REST API to read all ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET
resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None
logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'UPDATE_COMPUTE_ACCOUNT' to update a ComputeAccount

# custom_operation_type = UPDATE_COMPUTE_ACCOUNT,
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

custom_operation_type = "UPDATE_COMPUTE_ACCOUNT"
resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for UPDATE_COMPUTE_ACCOUNT
cuic_operation_request = etree.Element('cuicOperationRequest')
operation_type = etree.SubElement(cuic_operation_request, 'operationType')
operation_type.text = custom_operation_type
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_account = etree.Element('ComputeAccount')
account_name = etree.SubElement(compute_account, 'accountName')
account_name.text = 'sdk_compute'
status = etree.SubElement(compute_account, 'status')
status.text = 'Off'
ip = etree.SubElement(compute_account, 'ip')
ip.text = '2.2.2.2'
inner_text = etree.tostring(compute_account)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST UPDATE_COMPUTE_ACCOUNT REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the ComputeAccount updated with new ip address and status
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'DELETE_COMPUTE_ACCOUNT' to delete a ComputeAccount

# custom_operation_type = DELETE_COMPUTE_ACCOUNT,
# resource_URL = /cloupia/api-v2/ComputeAccount, HTTP request type = POST

custom_operation_type = "DELETE_COMPUTE_ACCOUNT"
resource_URL = "/cloupia/api-v2/ComputeAccount"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for DELETE_COMPUTE_ACCOUNT

cuic_operation_request = etree.Element('cuicOperationRequest')
operation_type = etree.SubElement(cuic_operation_request, 'operationType')
operation_type.text = custom_operation_type
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_account = etree.Element('ComputeAccount')
account_name = etree.SubElement(compute_account, 'accountName')
account_name.text = 'sdk_compute'
inner_text = etree.tostring(compute_account)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST DELETE_COMPUTE_ACCOUNT REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the deletion of ComputeAccount
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeAccount
# resource_URL = /cloupia/api-v2/ComputeAccount/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeAccount/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeAccount resource ...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP POST REST API operation 'CREATE' to create a ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource, HTTP request type = POST

resource_URL = "/cloupia/api-v2/ComputeResource"
http_request_type = "POST"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for ComputeResource@CREATE

cuic_operation_request = etree.Element('cuicOperationRequest')
payload = etree.SubElement(cuic_operation_request, 'payload')
compute_resource = etree.Element('ComputeResource')
account_Name = etree.SubElement(compute_resource, 'accountName')
account_Name.text = 'sdk_compute'
status = etree.SubElement(compute_resource, 'status')
status.text = 'On'
ip = etree.SubElement(compute_resource, 'ip')
ip.text = '1.1.1.1'
inner_text = etree.tostring(compute_resource)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP POST ComputeResource@CREATE REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the creation of ComputeResource
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP PUT REST API operation 'UPDATE' to update a ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = UPDATE

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "PUT"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

# Construct XML Payload for ComputeResource@UPDATE

cuic_operation_request = etree.Element('cuicOperationRequest')
payload = etree.SubElement(cuic_operation_request, 'payload')
```

```
compute_resource = etree.Element('ComputeResource')
account_name = etree.SubElement(compute_resource, 'accountName')
account_name.text = 'sdk_compute'
status = etree.SubElement(compute_resource, 'status')
status.text = 'Off'
ip = etree.SubElement(compute_resource, 'ip')
ip.text = '2.2.2.2'
inner_text = etree.tostring(compute_resource)
payload.text = etree.CDATA(inner_text)
payload_data = etree.tostring(cuic_operation_request)

logger.info("Executing HTTP UPDATE ComputeResource@UPDATE REST API...")
logger.info("payload = %s",payload_data)
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify ComputeResource account updated with new ip address and status
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET

resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Invoke the HTTP DELETE REST API Read operation - DELETE a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = DELETE


resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "DELETE"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP DELETE REST API to delete a specific ComputeResource resource
...")
response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
logger.info("API Response HTTP Status Code = %s", response.status_code);
logger.info("Response content type = %s", response.headers['content-type'])
logger.info("Response content = %s", response._content)

# Verify the deletion of ComputeResource 'sdk_compute'
# Invoke the HTTP GET REST API 'Read' operation to read a specific ComputeResource
# resource_URL = /cloupia/api-v2/ComputeResource/{accountName}, HTTP request type = GET
resource_URL = "/cloupia/api-v2/ComputeResource/"+"sdk_compute"
http_request_type = "GET"
resource_complete_URL = "https://" + IP_address + resource_URL
payload_data = None
response = None

logger.info("Executing HTTP GET REST API to read a specific ComputeResource resource ...")

response = invoke_rest_API(resource_complete_URL, payload_data, http_request_type)
```

```
 logger.info("API Response HTTP Status Code = %s", response.status_code);
 logger.info("Response content type = %s", response.headers['content-type'])
 logger.info("Response content = %s", response._content)

def invoke_rest_API(complete_URL, payload_data, http_request_type):
 requests.packages.urllib3.disable_warnings()
 response = None
 if http_request_type == "GET":
  response = requests.get(complete_URL, headers=headers, verify=False)
 elif http_request_type == "POST":
  response = requests.post(complete_URL, data=payload_data, headers=headers, verify=False)
 elif http_request_type == "PUT":
  response = requests.put(complete_URL, data=payload_data, headers=headers, verify=False)
 elif http_request_type == "DELETE":
  response = requests.delete(complete_URL, headers=headers, verify=False)
 else:
  raise Exception("Invalid HTTP request type")
 return response

if __name__ == "__main__":
 try:
  main ()
 except Exception as e:
  logger.error("Exception occurred while executing this script : " +str(e))
```

Executing the Script from the Command Line

Run the script from the command line as follows:

```
python scriptfile.py ip_address UCSD_REST_KEY
```

For example:

```
python execute_UCSD_REST_API.py 172.29.110.222 111F3D780A424C73A1C60BDD65BABB0B
```