



## Cisco Connect Denver 2018

# DEVNET ACI Programmability Hands On

Powered by



dCloud

dCloud: The Cisco Demo Cloud

11/2018, Version 1.0

## ACI Programmability Introduction:

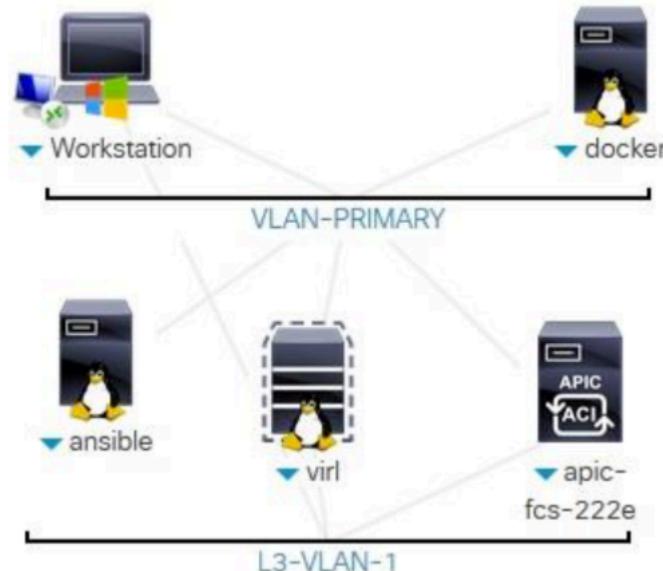
Stuff

### Lab Overview

This lab introduces 3 foundational concepts in performing basic programmability interaction with ACI. This lab will provide all attendees first-hand experience on how to use the API Inspector while performing tasks within ACI to glean specific URL path and payload data to accomplish programmability tasks within the fabric. Lab participants will then be able to take this data over to postman for analysis in a simplified “single rest call” review with various methods which can later be leveraged in code. Finally, the participants will use the combination of items learned with the API inspector and context derived with postman to review and build ansible tasks both with the included ACI modules as well as the rest calls. Upon completion of this lab, the student will be able to:

- Use API Inspector & Identify Valuable data
- Understand the Auth Cookie & Perform Basic Rest Calls in Postman
- Setup a Tenant, VRF, BridgeDomain, Application Profile, EPG and Contracts with Ansible

Lab Topology:



## Components

**Windows Workstation:** A Windows VM which has connectivity to the rest of the component in the lab. All the components in the lab are in a private network environment and can be accessed via VPN or through this workstation via RDP. This workstation will also be the principle for the ACI GUI and Postman interface for the lab.

**Ansible VM:** A Centos VM where the users will edit and execute the ansible and python lab exercises. This VM has connectivity to the ACI simulator. Ansible, Python and all required packages have been installed on the VM. This station can be accessed with ssh from the windows VM or directly if connecting through VPN.

**ACI Simulator:** The ACI simulator is used for providing a virtual ACI environment that can be leveraged for the lab exercises.

**VIRL:** Present in the Lab but not leveraged for this set of exercises.

The table below contains details on divides available for your virtual lab environment

Component	Access options	Credentials
Windows VM	198.18.133.252	administrator/C1sco12345
Ansible Centos VM	198.18.134.50	ansible/ansible
ACI Simulator	198.18.133.200	admin/C1sco12345

## Pre-requisites for this lab:

- A Laptop computer with access to the Internet
- AnyConnect, VPN software used to connect your desktop into the Virtual Lab
- Microsoft Remote Desktop, used to connect into the virtual Lab Desktop

## Lab Access - VPN

Each participant will have an individual POD to work on. Pod Credentials will be at each station/seat location. Each POD is dedicated to an individual user, so there is no impact to other PODs when making changes. You will receive a PoD already started for you Follow step by step instructions to login into your PoD and complete the lab

**Step 1:** Go into dCloud to access the Lab session being assigned to you. Click on View Session

## Step 2: Look into Session details and scroll down to find out your credentials for VPN AnyConnect

Cisco Datacenter Automation with Ansible v1

The screenshot shows the Cisco Datacenter Automation with Ansible v1 interface. At the top, there are tabs for Details, Servers, and Resources, with the time 12:39:31 displayed. Below the tabs is a network diagram illustrating a topology with a Workstation connected to a docker host via VLAN-PRIMARY. This host is connected to a virtual center (2) which includes hosts ansible, virf, and apic-fcs-222e. These hosts are also connected to L3-VLAN-1. To the right, a 'Session Details' window is open, showing the following information:

VPN Available:	true
Virtual Center:	2
AnyConnect Credentials	
Connect up to 16 devices to the session via Cisco AnyConnect.	
Host	dcloud-sjc-anyconnect.cisco.com
User	v293user1
Password	101058

## Step 3: Connect to AnyConnect VPN ***dcloud-sjc-anyconnect.cisco.com*** with the username and password provided on session details information window



Below is an example of user logging into POD1



Cisco AnyConnect | dcloud-sjc-anyconnect.cisco.com

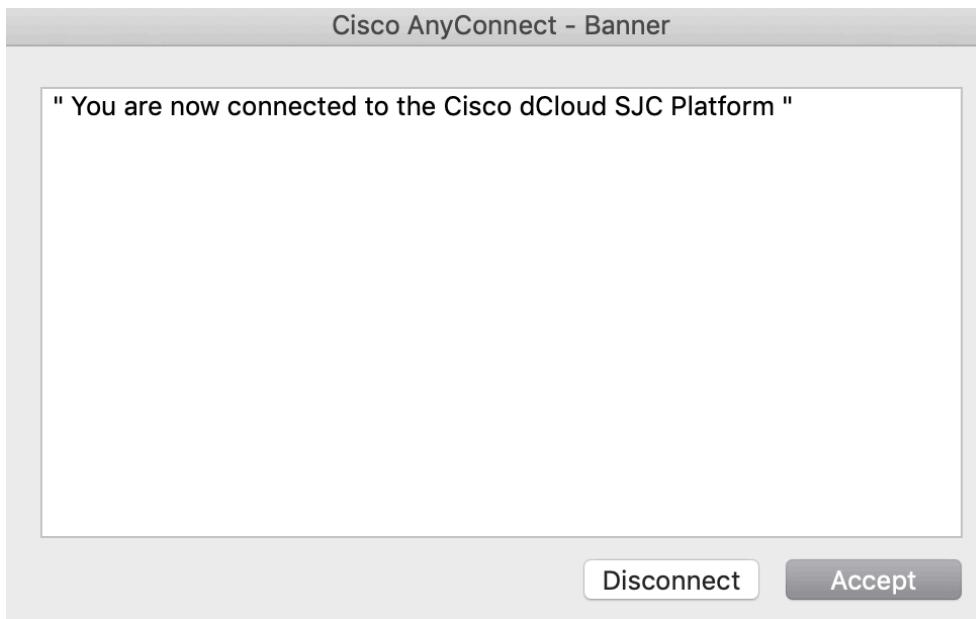
Group: Anyconnect-to-dCloud

Username: v293user1

Password: •••••

Cancel OK

Hit accept when the prompt appears to accept the VPN connection login

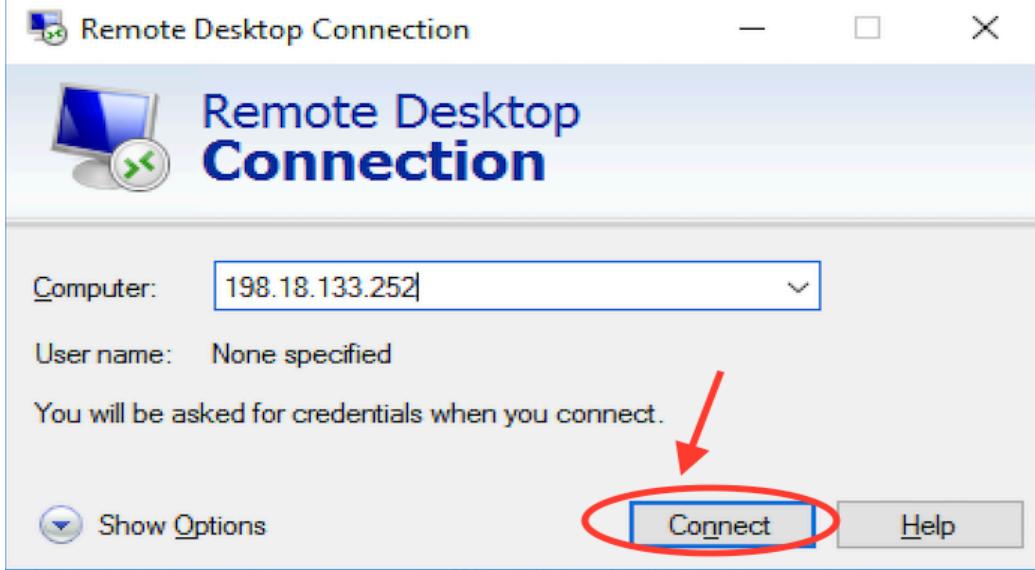


#### Step 4:

- Once, the vpn is connected. Use Remote Desktop to connect to the respective POD.
- If the RDP icon is not present on the desktop, for windows users go to *RUN* → *type mstsc* to get the remote desktop screen, for mac users the Microsoft remote desktop APP can be used  

- Enter the ip address provided for the POD to connect to the remote client

Windows:



Remote Desktop Connection

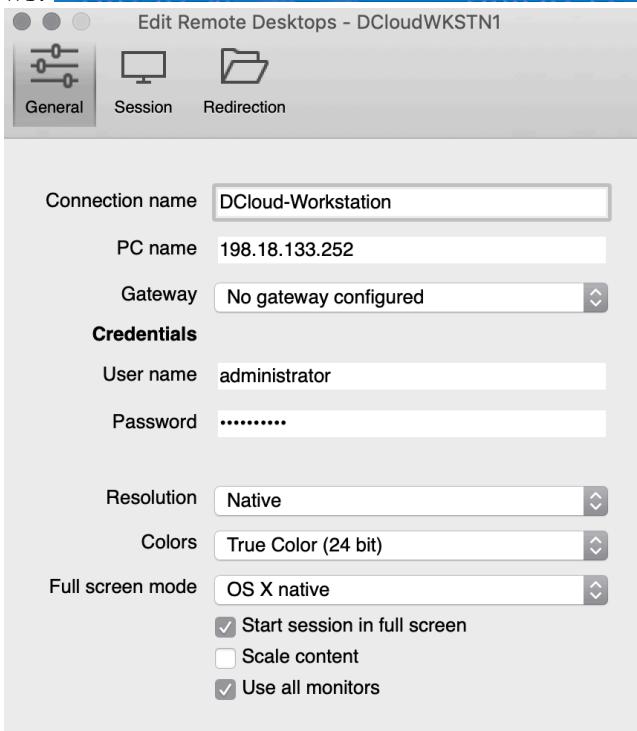
Computer: 198.18.133.252

User name: None specified

You will be asked for credentials when you connect.

Show Options Connect Help

MAC:



Edit Remote Desktops - DCloudWKSTN1

General Session Redirection

Connection name: DCloud-Workstation

PC name: 198.18.133.252

Gateway: No gateway configured

**Credentials**

User name: administrator

Password: .....

Resolution: Native

Colors: True Color (24 bit)

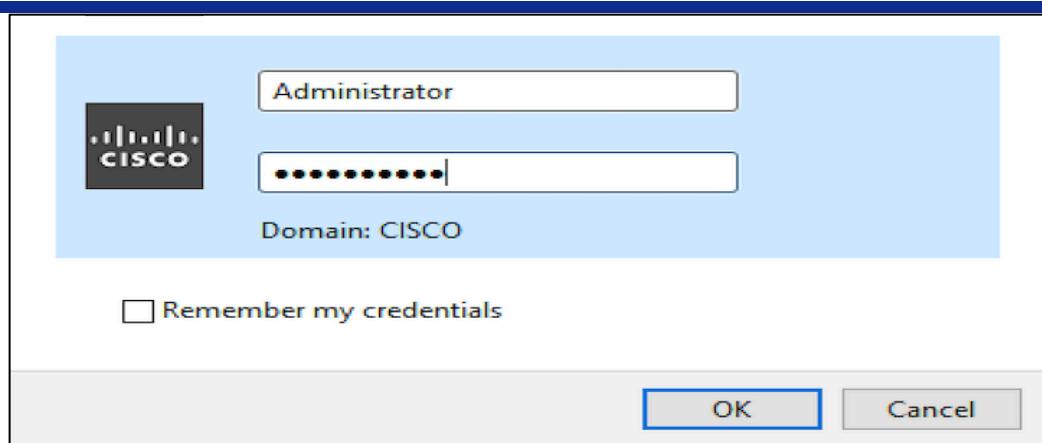
Full screen mode: OS X native

Start session in full screen

Scale content

Use all monitors

- Windows - Enter the credentials **Administrator / C1sco12345** to login to your POD if prompted

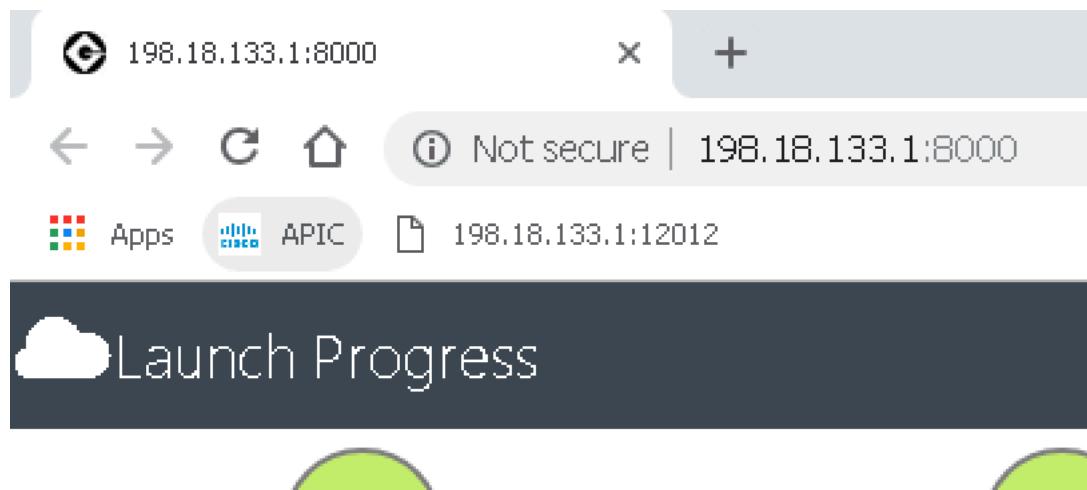


- Accept the connection certificate to login to the Virtual Lab Windows Desktop.

## Lab verification

### Verifying Lab Elements on the Windows VM Desktop

**Step 1:** Once logged into the Windows desktop, you will see a browser with the Lab Launch Progress. The APIC Icon can be clicked and the APIC GUI should show a login window. If not you can browse directly to <https://198.18.133.200/>





Log into the APIC GUI using the admin/C1sco12345 credentials.

**Step 2:** Check the Postman application and the collection as follows:



The postman application should load and there should be a default collection available in the postman application called Devnet-ACI-Lab



The screenshot shows the Postman application interface. On the left, there's a sidebar with tabs for 'History' and 'Collections'. The 'Collections' tab is selected, showing a folder named 'DevNet-ACI-Lab' containing 11 requests. The main area is the 'Builder' tab, which has a 'GET' dropdown and a 'Enter request URL' input field. Below these are tabs for 'Authorization', 'Headers', 'Body', 'Pre-request Script', and 'Tests'. The 'Authorization' tab is selected, showing 'No Auth' selected in a dropdown. A note says 'This request does not use any auth'. At the bottom right of the builder area, it says 'Hit the Send button to get a response'. The bottom of the interface has icons for 'New', 'Import', 'Runner', and a plus sign.

## Verifying Lab Elements on the Ansible VM

**Step 1:** From the windows VM click the ansible putty shortcut on the desktop



This Putty link will auto log into the system using the credentials.



NOTE: If connecting via VPN from your local machine use SSH to 198.18.134.50 and use credentials ansible/ansible.

**Step 2:** Verify the ansible version running on the VM using the “ansible –version” command. You should see ansible 2.7 is installed.

```
[ansible@ansible ~]$ ansible --version
ansible 2.7.2
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/ansible/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Aug  4 2017, 00:39:18) [GCC 4.8.5 20150623 (Red Hat 4.8.5-16)]
```

If ansible 2.7 is not installed run the “sudo pip install ansible –upgrade” command and provide the credentials for the ansible user (PWD: ansible)

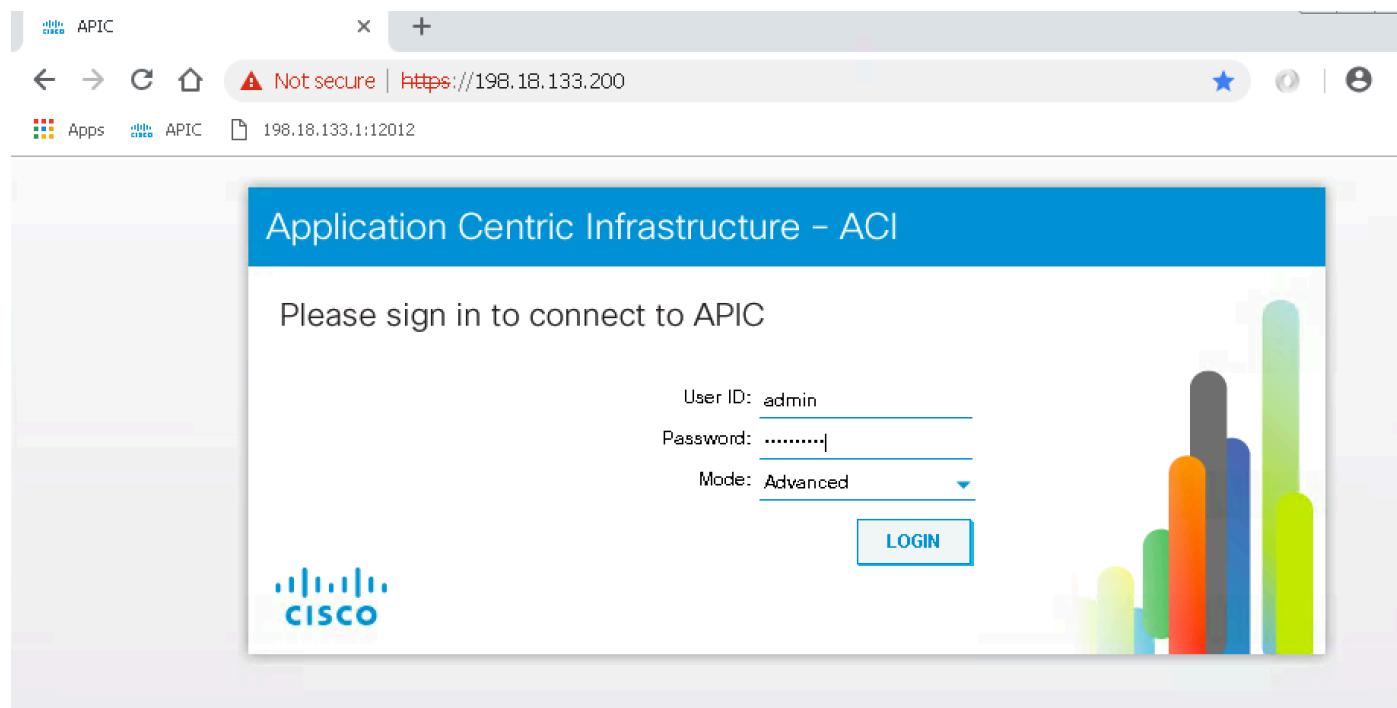
**Step 3:** Verify in the ansible home directory there should also be several YML files for the lab exercises and the missions.

```
[ansible@ansible ~]$ ls
3.Get-All-Tenants.yml      4b.Tenant-Get-subtree.yml      4c.Tenant-Get-All-Items.yml      6.Create-APandEPG.yml
4a.Tenant-Get-1Layer.yml   4c.Tenant-Get-All-Items.retry  5.Create-Tenant-Networking.yml  7.CreateContractAssignEPGs.yml
```

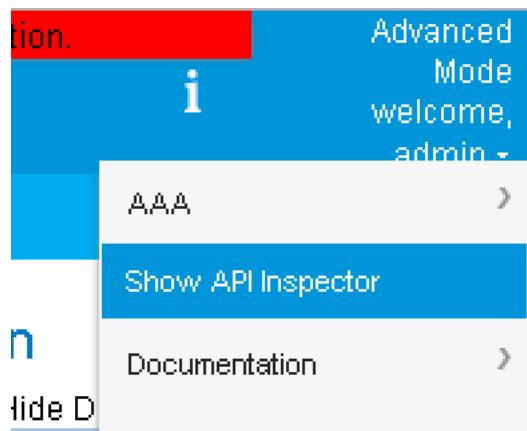
## ACI API Inspector

In this section, we will perform some exercises using the ACI inspector as functions are performed in the GUI to extrapolate rest calls.

**Step 1:** In the Windows VM log into the ACI simulator

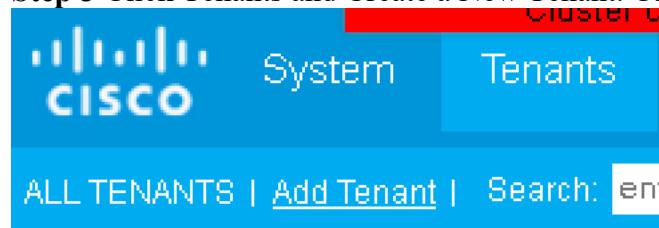


**Step 2** In the upper left-hand side of the screen find where it has out login name click the down arrow and find the Show API Inspector and click. This will open a separate window with the API inspector. It may be necessary to resize the parent and API windows, so they can be easily toggled.





**Step 3** Click Tenants and Create a New Tenant. Call the Tenant “GUITenant”.



## Create Tenant

Specify tenant details

Name: GUITenant

Alias:

**SUBMIT**

**CANCEL**

**Step 4** Once completed switch to the API inspector window and locate the POST command that was used during the execution of the create tenant function. A couple of things to note in this process. There will be several GET's that will constantly be coming up on the screen. This is a normal function of the GUI as it routinely gets up to date data. This can be slightly challenging to search through an analyze specific data. One way to make the experience better is to disable the scroll to latest feature.



Just like other CRUD functions. When elements in ACI are created a POST method will be used. So, the thing to search on when looking or elements as they are created (or that submit button is clicked throughout the UI) is a POST function.

To quickly identify POST functions, they can be searched for in the search pane and if there were multiple actions taken then the use of the next and previous buttons would be helpful in searching through the logs.

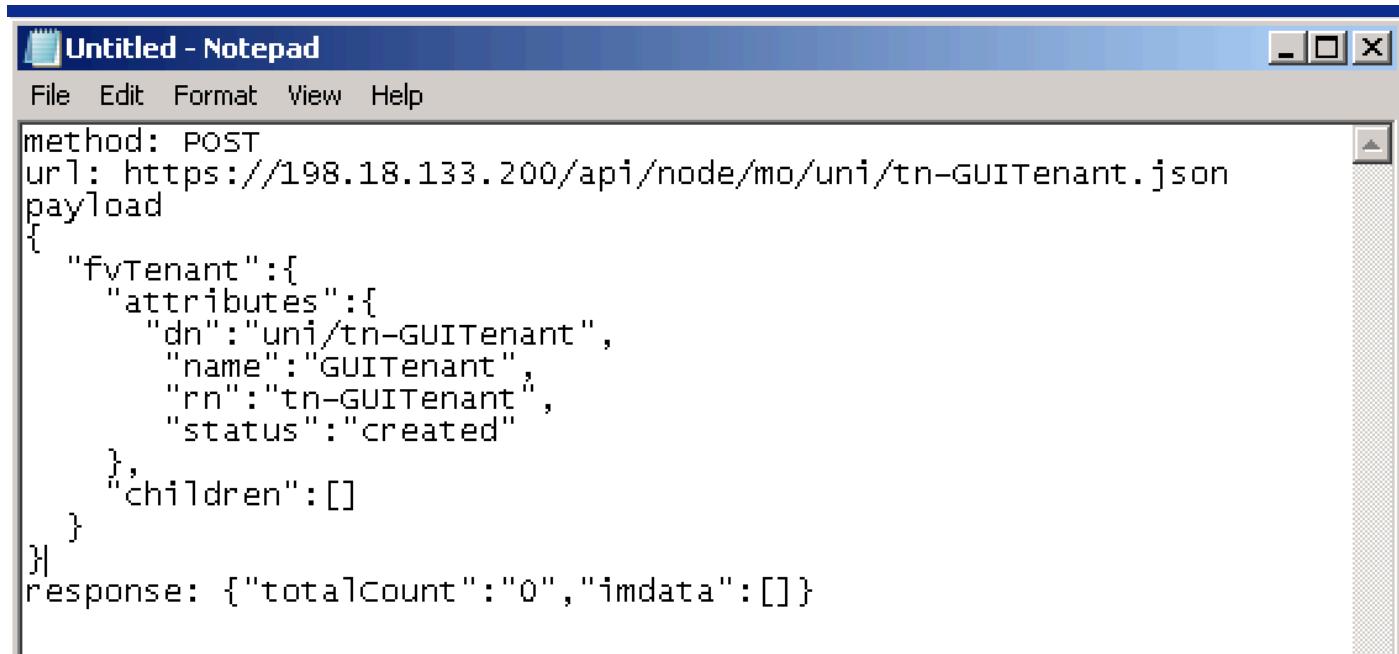
Search: <input type="text" value="POST"/>	<input type="button" value="Reset"/>	<input type="checkbox"/> Regex	<input type="checkbox"/> Match case	<input type="checkbox"/> Disable
<input type="button" value="Next"/>	<input type="button" value="Previous"/>	<input type="checkbox"/> Filter	<input type="checkbox"/> Highlight all	



The screenshot shows a browser window titled "API Inspector - Google Chrome" with the URL "about:blank". The window contains a log viewer interface. At the top, there are filters for "trace", "debug" (which is checked), "info", "warn", "error", "fatal", and "all". Below the filters is a search bar containing "POST" with a "Reset" button. There are also buttons for "Next", "Previous", "Filter", and "Highlight all". Underneath these are "Options" for "Log" (checked), "Wrap", "Newest at the top" (unchecked), "Scroll to latest" (checked), "Clear", and "Close". The main area displays log entries in green text. The entries show various API requests and responses, including POST requests to URLs like `https://198.18.133.200/api/node/class/fvTenant.json?query-target-filter=eq(fvTe...`, `https://198.18.133.200/api/node/class/aaaDomain.json?query-target-filter=and(ne...`, and `https://198.18.133.200/api/node/mo/uni/tn-GUITenant.json`. The log also includes event channel messages and timestamped DEBUG logs.

```
timestamp: 23:00:21 DEBUG
method: GET
url: https://198.18.133.200/api/node/class/fvTenant.json?query-target-filter=eq(fvTe...
response: {"totalCount": "1", "subscriptionId": "72057611234705434", "imdata": [{"fvTenan...
timestamp: 23:00:21 DEBUG
method: GET
url: https://198.18.133.200/api/node/class/aaaDomain.json?query-target-filter=and(ne...
response: {"totalCount": "0", "subscriptionId": "72057611234705435", "imdata": []}
timestamp: 23:01:01 DEBUG
method: POST
url: https://198.18.133.200/api/node/mo/uni/tn-GUITenant.json
payload("fvTenant": {"attributes": {"dn": "uni/tn-GUITenant", "name": "GUITenant", "rn": "...
response: {"totalCount": "0", "imdata": []}
timestamp: 23:01:01 DEBUG
method: Event Channel Message
response: {"subscriptionId": ["72057611234705430", "72057611234705431"], "imdata": [{"fv...
timestamp: 23:01:01 DEBUG
method: Event Channel Message
```

**Step 5** Copy and paste the method, URL, and payload to the notepad or text editor for further inspection.



```

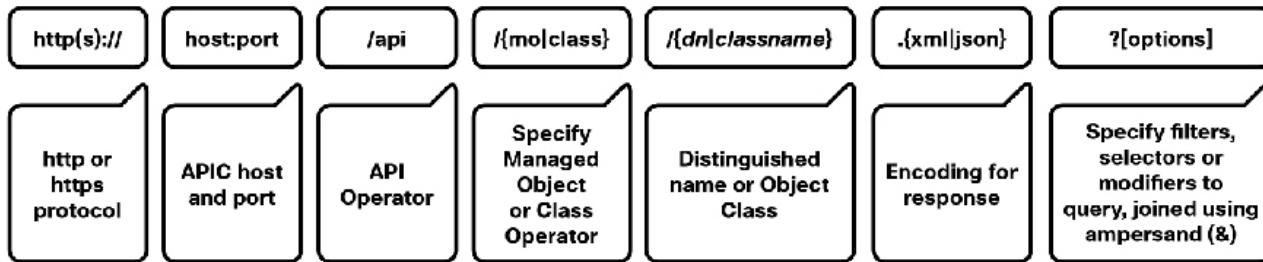
Untitled - Notepad
File Edit Format View Help

method: POST
url: https://198.18.133.200/api/node/mo/uni/tn-GUITenant.json
payload
{
  "fvTenant": {
    "attributes": {
      "dn": "uni/tn-GUITenant",
      "name": "GUITenant",
      "rn": "tn-GUITenant",
      "status": "created"
    },
    "children": []
  }
}
response: {"totalCount":"0","imdata":[]}

```

#### Step 6 Let's break apart the two specific pieces of the REST call:

1. The URI, the REST API is HTTP-based, defining the URI to access a certain resource type is important. The first two sections of the request URI simply define the protocol and access details of the APIC (<https://198.18.133.200/>). Next in the request URI is the literal string `/api`, indicating that the API will be invoked. The next part of the URI specifies whether the operation will be for an MO (Managed Object) or class. The next component defines either the fully qualified domain name (DN) being queried for object-based queries, or the package and class name for class-based queries. The final mandatory part of the request URI is the encoding format: either `.xml` or `.json`. This is the only method by which the payload format is defined, vs a content-type header as the APIC ignores Content-Type and other headers. Read operations using the GET method generally contain specific URI information and potential filter options using the `?`  then followed by various filters. Both create and update operations in the REST API are implemented using the POST method, so that if an object does not already exist, it will be created, and if it does already exist, it will be updated to reflect any changes between its existing state and desired state. Both create and update operations can contain complex object hierarchies, so that a complete tree can be defined in a single command so long as all objects are within the same context root and are under the 1MB limit for data payloads for the REST API.



Read properties for an EPG by Distinguished Name

`http://apic/api/mo/uni/tn-Cisco/ap-Software/epg-Download.xml`

Find all 10G ports on Fabric

`http://apic/api/class/11PhysIf.xml?query-target-filter=eq(11PhysIf.speed"10G")`

349945

For the POST URI operation, we can derive our relative path in the API is the root node of the APIC (/node), managed object (/mo), policy definition/resolution of universe (/uni) and the FV:Tenant class (tn-GUITenant) and we are sending a JSON payload (.json). How was this derived? The APIC Management Information Model Reference can be a tool to assist:

<https://developer.cisco.com/site/apic-mim-ref-api/>

Select API Version ▾



## APIC Management Information Model Reference, Release 4.0(1)

?

All Packages
<code>Classes</code>
<code>fv-StretchedCont</code> (Container for StretchedBD)
Gipo Allocation
<code>fv-Subnet</code> (Subnet)
<code>fv-SubnetBDDefCont</code> (Implicit Bridge Domain Container Mo Used to Update BD with EPg Subnet Info)
<code>fv-SubnetBDDefContTask</code>
<code>fv-SvcBD</code> (Bridge Domain)
<code>fv-SvcDep</code> (Svc Deployment)
<code>fv-SvcEpP</code> (Service Endpoint Profile)
<code>fv-SyntheticIclp</code> (Synthetic IP (fvCEp + fvlp))
<code>fv-SystemGIPoDef</code> (System GIPo)
<code>fv-TabooCtxDefCont</code> (Internal Mo Used to Update Context Shard with Taboo Info)
<code>fv-TabooCtxDefContTask</code>
<code>fv-Tenant</code> (Tenant)
<code>fv-TenantTask</code>
<code>fv-TepPoolDef</code> (TEP Pool Definition)
<code>fv-TnICl</code> (Tunnel VRF)
<code>fv-TnIEP</code> (dot1q-tunnel)
<code>fv-TnIEP</code> (dot1q-tunnel EpP)
<code>fv-To</code>
<code>fv-TunDef</code> (Tunnel)
<code>fv-TunDefRef</code> (Tunnel)
<code>fv-UnkMacUcastActMod</code> (BD Policy)
<code>fv-Up</code>
<code>fv-UpdateContract</code> (Update Contract)
<code>fv-UseEpPRequestor</code> (Generic EPg Requestor)
<code>fv-UsegSrc</code> (Source Reporting Mbr)

Overview	Naming	Diagram	Containers	Contained	Inheritance	Stat Counters	Stats	Events	Faults	FSMs	Properties	Summary	Properties	Detail
<b>Class fv:Tenant (CONCRETE)</b>														
Class ID:1975 Class Label: Tenant Encrypted: false - Exportable: true - Persistent: true - Configurable: true - Subject to Quota: Disabled - Abstraction Layer: Logical Model Write Access: [admin] Read Access: [aaa, access-connectivity-l1, access-connectivity-l2, access-connectivity-l3, access-connectivity-mgmt, access-connectivity-util, access-equipment, access-protocol-access-protocol-l2, access-protocol-l3, access-protocol-mgmt, access-protocol-ops, access-protocol-util, access-qos, admin, fabric-connectivity-l1, fabric-connectivity-l2, fabric-connectivity-l3, fabric-connectivity-mgmt, fabric-connectivity-util, fabric-equipment, fabric-protocol-l1, fabric-protocol-l2, fabric-protocol-l3, fabric-protocol-mgmt, fabric-protocol-op-fabric-protocol-util, nw-svc-device, nw-svc-devshare, nw-svc-params, nw-svc-policy, ops, tenant-connectivity-l1, tenant-connectivity-l2, tenant-connectivity-l3, tenant-connectivity-mgmt, tenant-connectivity-util, tenant-epg, tenant-ext-connectivity-l1, tenant-ext-connectivity-l2, tenant-ext-connectivity-l3, tenant-ext-connectivity-mgmt, tenant-ext-connectivity-util, tenant-ext-protocol-l1, tenant-ext-protocol-l2, tenant-ext-protocol-l3, tenant-ext-protocol-mgmt, tenant-ext-protocol-util, tenant-network-profile, tenant-protocol-l1, tenant-protocol-l2 tenant-protocol-l3, tenant-protocol-mgmt, tenant-protocol-ops, tenant-qos, tenant-security, vmm-connectivity, vmm-ep, vmm-policy, vmm-protocol-ops, vmm-security] Createable/Deletable: yes (see Container Mos for details) Semantic Scope: EPG Semantic Scope Evaluation Rule: Explicit Monitoring Policy Source: Explicit Monitoring Flags: [ IsObservable: true, HasStats: true, HasFaults: true, HasHealth: true, HasEventRules: false ]														
A policy owner in the virtual fabric. A tenant can be either a private or a shared entity. For example, you can create a tenant with contexts and bridge domains shared by other tenants. A shared type of tenant is typically named common, default, or infra.														
<b>Naming Rules</b>														

The MIM can provide very specific detail of various aspects of elements. Scroll to the Classes fv:Tenant class and let's look at what the MIM can provide from a details perspective. Notice from a basic navigation perspective in the classes pane there is a (Tenant) notation from an overview, to where the element sits in the container hierarchy (this helps with understanding the path information possibly needed in the URI), to the contained hierarchy (what children elements are contained

below), and properties detail (this is helpful in determining the payload and some of the attributes sent along during the rest call

2. Now let's look at the payload:

```
payload
{
  "fvTenant": {
    "attributes": {
      "dn": "uni/tn-GUITenant",
      "name": "GUITenant",
      "rn": "tn-GUITenant",
      "status": "created"
    },
    "children": []
  }
}
```

The payload consists of a DN, Name, RN, and Status. From the MIM we can unpack this to see the specifics and definition of each of the attributes, for almost every one of these items we can look in the property's summary section of the tenant class:

- The Distinguished Name (DN) format (uni/tn-{name}) is defined in the Naming rules at the top and is the fully qualified name of element.

#### **DN FORMAT:**

[ 1 ] [uni/tn-{name}](#)

- The Relative Name (RN) Identifies an object from its siblings within the context of its parent object and its format is tn-{name}.

Defined in: <a href="#">mo:TopProps</a>	
<a href="#">mo:ModificationChildAction</a>	<a href="#">childAction</a> ( <a href="#">mo:TopProps:childAction</a> ) scalar:Bitmask32 Delete or ignore. For internal use only.
<a href="#">reference:BinRef</a>	<a href="#">dn</a> ( <a href="#">mo:TopProps:dn</a> ) A tag or metadata is a non-hierarchical keyword or term assigned to the fabric module.
<a href="#">reference:BinRN</a>	<a href="#">rn</a> ( <a href="#">mo:TopProps:rN</a> ) Identifies an object from its siblings within the context of its parent object. The distinguished <a href="#">name</a> contains a sequence of relative <a href="#">names</a> .

- The Name is the string of the tenant that we entered in the GUI

# Properties Summary

Defined in: **fv:Tenant**

<a href="#"><u>mo:Annotation</u></a> <a href="#"><u>string:Basic</u></a>	<b>annotation</b> ( <a href="#">fv:Tenant:annotation</a> ) NO COMMENTS
<a href="#"><u>naming:Descr</u></a> <a href="#"><u>string:Basic</u></a>	<b>descr</b> ( <a href="#">fv:Tenant:descr</a> ) The description of the tenant.
<a href="#"><u>mo:ExtMngdByType</u></a> <a href="#"><u>scalar:Bitmask32</u></a>	<b>extMngdBy</b> ( <a href="#">fv:Tenant:extMngdBy</a> ) NO COMMENTS
<a href="#"><u>reference:BinRef</u></a>	<b>monPolDn</b> ( <a href="#">fv:Tenant:monPolDn</a> ) Monitoring policy attached to this observable object
<a href="#"><u>naming:Name</u></a> <a href="#"><u>string:Basic</u></a>	<b>name</b> ( <a href="#">fv:Tenant:name</a> ) Overrides: <a href="#">pol:Obj:name</a>   <a href="#">naming:NamedObject:name</a> The <b>name</b> of the tenant.

- The Status is a method that can provide a specific result depending on the operation being performed (i.e. created for the first-time vs modified or deleted). The GUI by using the status created is asking for the tenant to be created, and if the tenant already exists the API would return an error.

## status

```
Type: mo:ModificationStatus
Primitive Type: scalar:Bitmask32

Units: null
Encrypted: false
Access: implicit
Category: TopLevelStatus
```

Comments:

The upgrade status. This property is for internal use only.

## Constants

created	<b>2u</b>	created	In a setter method: specifies that an object should be created. An error is returned if the object already exists. In the return value of a setter method: indicates that an object has been created.
modified	<b>4u</b>	modified	In a setter method: specifies that an object should be modified In the return value of a setter method: indicates that an object has been modified.
deleted	<b>8u</b>	deleted	In a setter method: specifies that an object should be deleted. In the return value of a setter method: indicates that an object has been deleted.

**Step 7** Back in the GUI, the Tenant would have been entered by default. Click on the VRF and create a new VRF named prod and BD named prod as well.

## Create VRF

### STEP 1 > VRF

1. VRF

Specify Tenant VRF

Name:	Prod
Alias:	
Description:	optional
Policy Control Enforcement Preference:	Enforced    Unenforced
Policy Control Enforcement Direction:	Egress    Ingress
End Point Retention Policy:	select a value
This policy only applies to remote L3 entries	
Monitoring Policy:	select a value
DNS Labels:	
enter names separated by comma	
Route Tag Policy:	select a value
Create A Bridge Domain: <input checked="" type="checkbox"/>	

[PREVIOUS](#)
[NEXT](#)
[CANCEL](#)

## STEP 2 > Bridge Domain

Specify Bridge Domain for the VRF

Name: **prod**

Alias:

Description: optional

Type: **fc** regular

Forwarding: Optimize ▾

Endpoint Dataplane Learning:

Limit IP Learning To Subnet:

Config BD MAC Address:

MAC Address: 00:22:BD:F8:19:FF

**PREVIOUS** **FINISH** **CANCEL**

This is an example of a couple operations happening at the same time within a rest call. Locate the post and copy it to a notepad as we did in the previous example. Space out the payload of the rest response to examine the details of the rest call

```

method: POST
url: https://198.18.133.200/api/node/mo/uni/tn-GUITenant.json
payload
{
  "fvTenant": {
    "attributes": {
      "dn": "uni/tn-GUITenant",
      "status": "modified"
    },
    "children": [
      "fvBD": {
        "attributes": {
          "dn": "uni/tn-GUITenant/BD-prod",
          "name": "prod",
          "rn": "BD-prod",
          "status": "created"
        },
        "children": [
          "fvRsCtx": {
            "attributes": {
              "tnFvCtxName": "Prod",
              "status": "created,modified"
            },
            "children": []
          }
        ]
      },
      "fvCtx": {
        "attributes": {
          "dn": "uni/tn-GUITenant/ctx-Prod",
          "name": "Prod",
          "rn": "ctx-Prod",
          "status": "created"
        },
        "children": []
      }
    ]
  }
}

```

Perform an Create/Update at the Tenant Level, Status indicates a modification.  
VRF's and BD's are hierarchically  
Children elements from a tenant level.

Children elements are lists of dictionaries so multiple elements can be created in a single call.

Create a BD Named Prod  
& Associate the BD to the VRF (Also named Prod)

Create a VRF Named Prod

From the REST call we can see that a single call was made under the GUI tenant to create both the VRF as well as the BD. One thing that can also be seen in the rest call is that the GUI also assigned the VRF to the bridge domain.

**Step 8** Lets now take a look at a read operation. NOTE: prior to proceeding it may be a good idea to clear the API inspector so that there are fewer results to sift through. Click in the Tenant, expand the Networking->Bridge Domains section and select the prod BD that was created in the previous step.

The screenshot shows the Cisco ACI GUI interface. At the top, there are tabs: System, Tenants (which is selected), Fabric, VM Networking, and L4-L7 Services. Below the tabs, there's a search bar with placeholder text 'Search: enter name, alias, descr'. Under the search bar, there are links for 'ALL TENANTS' and 'Add Tenant'. The main content area is titled 'Tenant GUITenant'. On the left, there's a navigation tree with 'Quick Start', 'Tenant GUITenant' (selected), 'Application Profiles', 'Networking' (selected), 'Bridge Domains' (selected), 'prod' (selected), and 'VRFs'. The 'prod' node is highlighted with a grey background.

The Policy will auto populate. Click on the API inspector and locate a GET request. A search for BD-prod can help locate the correct API call.

```

timestamp: 02:49:06 DEBUG
method: GET
url: https://198.18.133.200/api/node/mo/uni/tn-GUITenant/BD-prod.json?subscription=yes
response: {"totalCount":"1","subscriptionId":"72057606939738179","imdata":[{"fvBD":{"attributes":{"arpFlood":"no"
timestamp: 02:49:06 DEBUG
method: GET

```

The URI of the API call is `https://198.18.133.200/api/node/mo/uni/tn-GUITenant/BD-prod.json?subscription=yes`. The URI's path in the object tree has been amended one level below the tenant structure to query the bridge domain object. The GUI is also requesting a subscription to push-based events by using the `subscription=yes`. This is generally not required for basic rest calls. The resulting data from the rest call is:

```

Untitled - Notepad
File Edit Format View Help
method: GET
url: https://198.18.133.200/api/node/mo/uni/tn-GUITenant/BD-prod.json?subscription=yes
response:
{
  "totalCount": "1",
  "subscriptionId": "72057606939738179",
  "imdata": [
    {
      "fvBD": {
        "attributes": {
          "arpFlood": "no",
          "bcastP": "225.0.242.96",
          "childAction": "",
          "configIssues": "^",
          "descr": "",
          "dn": "uni/tn-GUITenant/BD-prod",
          "epcClear": "no",
          "epMoveDetectMode": "",
          "ipLearning": "yes",
          "l1own": "local",
          "limitIpLearnToSubnets": "no",
          "llAddr": "::",
          "mac": "00:22:BD:F8:19:FF",
          "mcastAllow": "no",
          "modTs": "2018-11-26T23:57:08.439+00:00",
          "monPolDn": "uni/tn-common/monepg-default",
          "mtu": "inherit",
          "multiDstPktAct": "bd-flood",
          "name": "prod",
          "nameAlias": "",
          "ownerKey": "",
          "ownerTag": "",
          "pcTag": "49154",
          "scope": "2621440",
          "seg": "15990734",
          "status": "",
          "type": "regular",
          "uid": "15374",
          "unicastRoute": "yes",
          "unkMacUcastAct": "proxy",
          "unkMcastAct": "flood",
          "vmac": "not-applicable"
        }
      }
    }
  ]
}

```

Looking at the GET response payload information is also valuable so that the resulting data can be parsed and variablized. The JSON response can make it simpler when working in code parsing a response.

Example: To retrieve the MTU variable would be similar to `imdata[0]["fvBD"]["attributes"]["mtu"]`. Each programming language may parse data slightly differently. Will look at this in subsequent sections.

#### Summary:

We covered the following in this section of the lab.

- API Inspector Basics
- Management Information Model Basics
- Rest Call Identification & Teardown
- Understanding some of the filter elements & responses in POST and GET methods

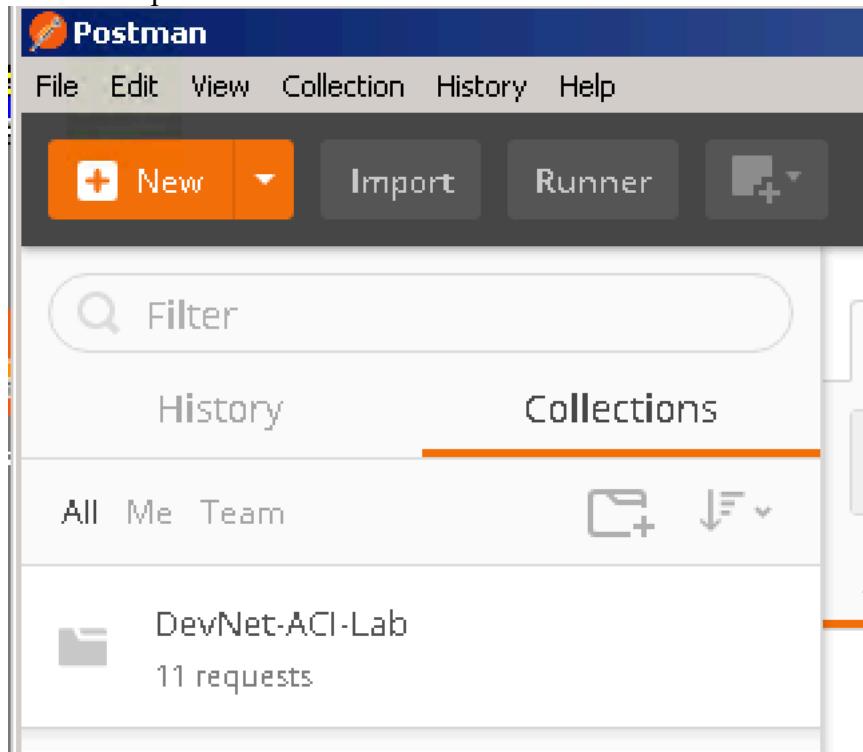
## Postman API Interaction

In the previous demo we learned the basics of identifying REST calls within API, and how to break apart the individual elements of the payload to understand their purpose. This section will build on that by using postman to perform basic REST calls to the ACI environment using both GET and POST methods.

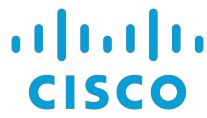
**Step 1:** If the postman application isn't launched, open it on the Windows VM desktop



Locate the pre-loaded collection called DevNet-ACI-Lab



**Step 2:** Similar to logging into the GUI, to make REST calls the postman application has to log into the APIC environment. Using the collection task 1 and clicking send a Token is received in the body of the response as well as in form of a cookie. See below for details:



## ACI Programmability with Postman & Ansible

The screenshot shows the Postman interface with a single collection named "1. ACI Login & Token". A POST request is selected with the URL <https://198.18.133.200/api/aaaLogin.json>. The request body contains the following JSON payload:

```
1 [ {  
2   "aaaUser": {  
3     "attributes": {  
4       "name" : "admin",  
5       "pwd" : "C1sco12345"  
6     }  
7   }  
8 }
```

Note the payload of the POST where we are sending the usernames and credentials as the initial authentication. This would not be how we would want to store & use a production level credential but certainly works in a lab/demo environment.

The screenshot shows the Postman interface with the "Body" tab selected. The response status is 200 OK, Time: 66 ms, and Size: 1.96 kB. The response body is displayed in JSON format:

```
1 {  
2   "totalCount": "1",  
3   "imdata": [  
4     {  
5       "aaaLogin": {  
6         "attributes": {  
7           "token": "VjokI4eyW/8uqmXb1pKYPUIHDNui2YaXrcrZ99yvw8Pf1IAr0+LLF1ToJ0sfDLH3IRLqK6E1b3jfKYeGOGJZ  
8             /47IB3Z89QRaa5jusF  
+49PUqIngC4dELM1IFjsUbcuCuuy17U4oNALnNV1SZhOieGAFhb4y1XhfUAJZM0V2Wh6P3gBJT1oOyKdLSBzfuYetH",  
"siteFingerprint": "tkEhtJidoPimwnHF".  
}
```

After the send button is clicked the response body is shown the token information is received. In programming this value can be stored for further action.

The screenshot shows the Postman interface with the "Body" tab selected. The response status is 200 OK, Time: 66 ms, and Size: 1.96 kB. The response body is displayed in JSON format:

```
1 {  
2   "totalCount": "1",  
3   "imdata": [  
4     {  
5       "aaaLogin": {  
6         "attributes": {  
7           "token": "VjokI4eyW/8uqmXb1pKYPUIHDNui2YaXrcrZ99yvw8Pf1IAr0+LLF1ToJ0sfDLH3IRLqK6E1b3jfKYeGOGJZ  
8             /47IB3Z89QRaa5jusF  
+49PUqIngC4dELM1IFjsUbcuCuuy17U4oNALnNV1SZhOieGAFhb4y1XhfUAJZM0V2Wh6P3gBJT1oOyKdLSBzfuYetH",  
"siteFingerprint": "tkEhtJidoPimwnHF".  
}
```

In addition to the response payload a cookie is also received that contains the same value as the Token in the payload.

Body	Cookies (1)	Headers (13)	Test Results	Status: 200 OK
Name	Value	Domain	Path	Expires
APIC-cookie	Vjokl4eyV/8uqmX b1pKYPUiHDNuiz YaXrcrZ99yvw8Pf 1IAr0+LLFfToj0sf DLM3IRLqK6E1b3 jfKYeGOGJZ/47IB 32890RaA5jusft4 9PUqlngC4dELMII FjsUbcuCuuyI7U4 oNALnNVISZhOie GAFhb4ylXhfUAJZ M0V2Vh6P3gBJTl oOyKdLSBzfuYet H	198.18.133.200	/	true

The nice thing about using postman is that postman stores this cookie in the program that can be used in subsequent rest calls. Aside from performing the initial call in step 1 there is no further action needed when using postman to perform subsequent calls with the stored cookie.

Examples (0) ▾
  
 
Save ▾
  
Cookies Code



The screenshot shows the 'MANAGE COOKIES' interface. At the top, there's a search bar with placeholder text 'Type a domain name' and an orange 'Add' button. Below this, a list shows a single cookie entry for the IP 198.18.133.200. The cookie details are as follows:

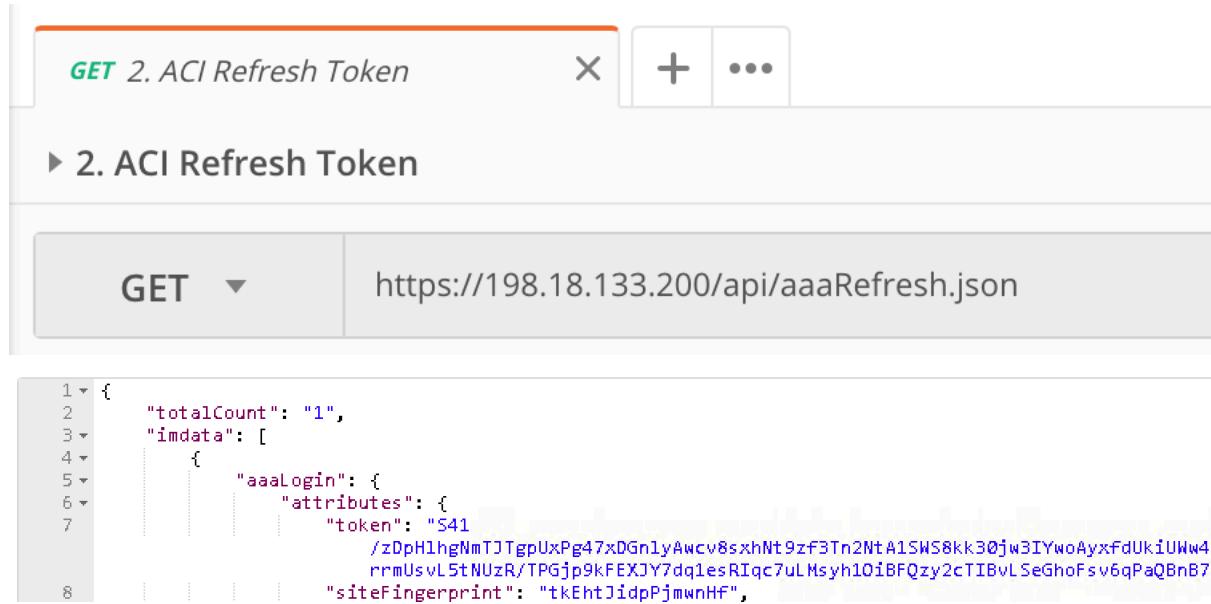
```
APIC-
cookie=VjokI4eyV/8uqmXb1pKYPUiHDNui2YaXrcrZ99yvw8Pf1IAr0+LLFIToJ0sfDLM3IRLqK6E1b3jfKYeGOG
JZ/47IB3Z890RaA5jusf+49PUqIngC4dELM1FJsUbcuCuuyyl7U4oNALnNV1SZhOieGAFhb4ylXhfUAJZM0V2Vh6P
3gBJTloOyKdLSBzfuYetH; path=/; domain=198.18.133.200; Secure; HttpOnly; Expires=Tue, 19
Jan 2038 03:14:07 GMT;
```

At the bottom right of the cookie entry are 'Cancel' and 'Save' buttons.

**Step 3:** Similar to a GUI login the credentials will expire after a period of time so will a REST token. That time data is contained in the response body of the initial login request. In order to keep the session going the token must be refreshed within the interval. If that doesn't happen the login step would need to be repeated to generate a new token. When refreshing the existing token, the values will change. See the image below that outlines the refresh timeout value.

```
"imdata": [
  {
    "aaaLogin": {
      "attributes": {
        "token": "VjokI4eyV/8uqmXb1pKYPUiHDNui2YaXrcrZ99yvw8Pf1
        /47IB3Z890RaA5jusf
        +49PUqIngC4dELM1FJsUbcuCuuyyl7U4oNALnNV1SZhOieGAFht
        "siteFingerprint": "tkEhtJidpPjmwnHF",
        "refreshTimeoutSeconds": "600",
        "maximumLifetimeSeconds": "86400",
        "guiIdleTimeoutSeconds": "1200",
        "nextT4timeoutSeconds": "000"
      }
    }
  }
]
```

Click step 2 which is titled “ACI Refresh Token”, a couple times and look at the token value after each click. Note that the value will change as the token is refreshed. Also note the different URI to refresh a token than to log in for the first time.

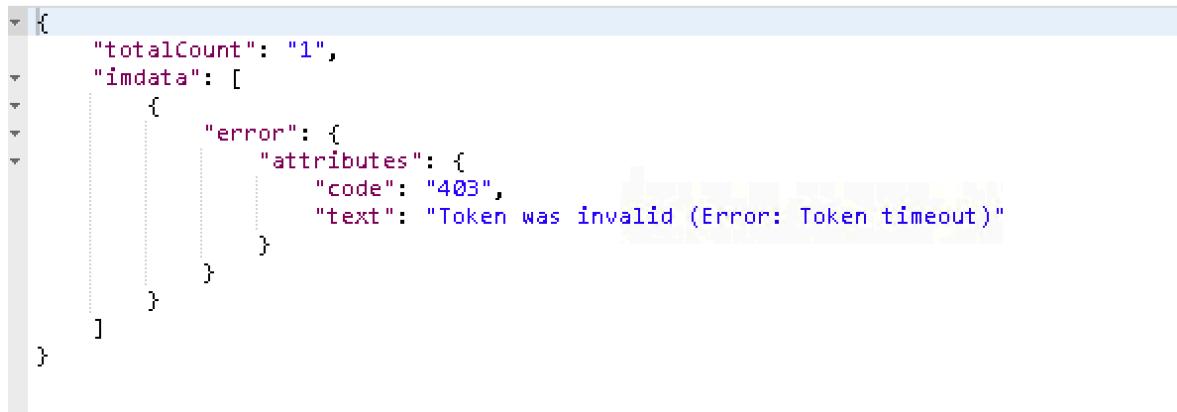


```

1  {
2    "totalCount": "1",
3    "imdata": [
4      {
5        "aaaLogin": {
6          "attributes": {
7            "token": "S41
     /zDpH1hgNmTJTpUxPg47xDGnlyAwcv8sxhNt9zf3Tn2NtA1SWs8KK30jw3IYwoAyxFdUkiUWw48Bi4Qd9mxz246GZeS
     rrmUsvl5tNUzR/TPGjp9kFXJY7dq1esRIqc7uLMsyh10iBFQzy2cTIBvLSeGhoFsv6qPaQBnB7K30T2PwfEyI+3I",
8            "siteFingerprint": "tkEhtJidpPjmwnHF",

```

If the token doesn't get refreshed before the timeout the following will occur. Don't forget throughout the lab to occasionally come back to step 2 and refresh the credential. If you do get the Token invalid error, click on the step 1 login option to regenerate a new token.



```

{
  "totalCount": "1",
  "imdata": [
    {
      "error": {
        "attributes": {
          "code": "403",
          "text": "Token was invalid (Error: Token timeout)"
        }
      }
    }
  ]
}

```

**Step 4:** Now that we have an active token let's explore some of the same actions we did with the GUI and the API explorer. Click and run step 3 which is the ACI Get All Tenants. A couple of things to note on the API call itself. The path is looking at a type class under the root node called fvTenant. This is the class that will return the Tenants (derived from the MIM). We are also asking for the response to be in XML format, which is accomplished by using the .xml at the end of the path.



## ACI Programmability with Postman & Ansible

The resulting payload is indeed in an XML format and should contain the three default tenants that are in ACI (common, infra, and mgmt). Take a moment to review the data and the format. Being able to request XML or JSON is as simple as changing the extension at the end of the URI path, selecting between the two is important depending on how your parsing data in code.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <imdata totalCount="3">
3   <fvTenant childAction="" descr="" dn="uni/tn-common" lcOwn="local" modTs="2018-11-30T03:51:17.730+00:00" monPolDn="uni/tn-common
   /monepg-default" name="common" nameAlias="" ownerKey="" ownerTag="" status="" uid="0"/>
4   <fvTenant childAction="" descr="" dn="uni/tn-infra" lcOwn="local" modTs="2018-11-30T03:51:24.823+00:00" monPolDn="uni/tn-common
   /monepg-default" name="infra" nameAlias="" ownerKey="" ownerTag="" status="" uid="0"/>
5   <fvTenant childAction="" descr="" dn="uni/tn-mgmt" lcOwn="local" modTs="2018-11-30T03:51:24.807+00:00" monPolDn="uni/tn-common
   /monepg-default" name="mgmt" nameAlias="" ownerKey="" ownerTag="" status="" uid="0"/>
6 </imdata>
```

**Step 5:** Lets dig a bit deeper into one of the tenants and get all the child elements of that tenant. Load the step 4a from the collection. Note the path has changed from looking at a specific class to moving into the managed objects and looking into a distinguished name of the infra tenant (from the previous step “dn=uni/tn-infra”). In addition to the path changing a filter parameter query-target is used to specify what information about the tenant we would like to get back.

Look at the returned result. (ALSO now in JSON since the path URL in this call was .JSON vs .XML).



## ACI Programmability with Postman & Ansible

The screenshot shows the Postman interface with a successful API call. The status bar indicates "Status: 200 OK Time: 55 ms Size: 3.65 KB". The "Body" tab is selected, displaying a JSON response with 9 items. The response structure includes "totalCount": "9", "imdata": [ ], and a single item object with attributes like "aaaDomainRef", "attributes", "childAction", "descr", "dn", "lcOwn", "modTs", "monPolDn", "name", "nameAlias", and "ownerKey".

```
1 {  
2   "totalCount": "9",  
3   "imdata": [  
4     {  
5       "aaaDomainRef": {  
6         "attributes": {  
7           "childAction": "",  
8           "descr": "",  
9           "dn": "uni/tn-infra/domain-infra",  
10          "lcOwn": "local",  
11          "modTs": "2018-11-30T03:51:24.823+00:00",  
12          "monPolDn": "uni/tn-common/monepg-default",  
13          "name": "infra",  
14          "nameAlias": "",  
15          "ownerKey": ""  
16        }  
17      }  
18    ]  
19  }  
20 }
```

The result shows all direct hierarchical children to the fvTenant class.

**Step 6:** In addition to pulling children objects we can dig into the entire tree of child elements, however that amount of information may be too much for what a program would need, and to be efficient we would want to limit the result in the search to only the specific information that we are looking to pull. Run the step 4b, while the path is the same the query parameters have changed. From Children in the previous step it is now subtree a second parameter called target-subtree-class has also been introduced to limit the results to only the specific classes that are desired for the results in this case the Application Profile and the EPGS

The screenshot shows a POST request in Postman. The URL is https://198.18.133.200/api/node/mo/uni/tn-infra.json?query-target=subtree&target-subtree-class=fvAp,fvAEPg. The "Params" tab is selected. The response is not yet visible.

4b. ACI Get Subtree with a Filter of Specific Items for a Tenant

Send

Params ● Authorization Headers Body Pre-request Script Tests

The results returned are again in JSON as that was the format requested in the path URL. In addition, only the two respective classes are present in the response. Again, limiting the results can cut down the overall processing/iteration time of an application if only the relevant data is returned.

```
"imdata": [  
  {  
    "fvAp": {  
      "attributes": {  
        "childAction": "",  
        "descr": "",  
        "dn": "uni/tn-infra/an-access"  
      }  
    }  
  }  
]
```

**Step 7:** In the last GET method example we will switched back to an XML output and keep the query as subtree in the infra tenant. The Subtree query is handy in both JSON and XML formats as it provides a format and framework as to how a REST payload could be put together to configure multiple objects within a single call. Like the API Explorer we can extrapolate from existing tenants how API calls and payloads can be leveraged in both GET methods to retrieve information as well as in POST methods on how multiple objects can be created in a single call.

▶ 4c. ACI Get all Subtree elements for a Tenant

GET ▾
https://198.18.133.200/api/node/mo/uni/tn-infra.xml?query-target=subtree
Send ▾

Note the resulting data, where we have BD information, QoS policies, and far more than we did when we filtered in the previous step to just the application profiles and EPGs.

Body Cookies (1) Headers (12) Test Results Status: 200 OK Time: 114 ms Size: 16.13 KB Save Download

Pretty Raw Preview XML ▾  

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <imdata totalCount="60">
3    <aaaDomainRef childAction="" descr="" dn="uni/tn-infra/domain-infra" lcOwn="local" modTs="2018-11-30T03:51:24.823+00:00"
4      monPolDn="uni/tn-common/monepg-default" name="infra" nameAlias="" ownerKey="" ownerTag="" status="" uid="0"/>
5    <fvSubnetBDDefCont bddefDn="uni/bd-[uni/tn-infra/BD-default]-isSvc-no" childAction="" dn="uni/tn-infra/ap-access/epg-default
6      /rsbd/subnetBddefDn-[uni/bd-[uni/tn-infra/BD-default]-isSvc-no]" lcOwn="local" modTs="2018-11-30T03:51:16.534+00:00"
      monPolDn="" name="" nameAlias="" status="">
7      <fvRsBd childAction="" dn="uni/tn-infra/ap-access/epg-default/rsbd" forceResolve="yes" lcOwn="local" modTs="2018-11-30T03:51:24
        .823+00:00" monPolDn="uni/tn-common/monepg-default" rType="mo" state="formed" stateQual="none" status="" tCl="fvBD"
        tContextDn="" tDn="uni/tn-infra/BD-default" tRn="BD-default" tType="name" tnFvBDName="default" uid="0"/>
8      <fvRsCustQosPol childAction="" dn="uni/tn-infra/ap-access/epg-default/rscustQosPol" forceResolve="yes" lcOwn="local" modTs="2018
        -11-30T03:51:24.823+00:00" monPolDn="uni/tn-common/monepg-default" rType="mo" state="formed" stateQual="default-target"
        targetDn="">
9    </fvSubnetBDDefCont>
10   <fvRsCustQosPol childAction="" dn="uni/tn-infra/ap-access/epg-default/rscustQosPol" forceResolve="yes" lcOwn="local" modTs="2018
        -11-30T03:51:24.823+00:00" monPolDn="uni/tn-common/monepg-default" rType="mo" state="formed" stateQual="default-target"
        targetDn="">
11   </fvRsCustQosPol>
12   </aaaDomainRef>
13 </imdata>

```

**Step 8:** Now let's begin making some changes in the ACI environment by using the POST methods. Using some of the last step's information as a baseline an XML payload can be made to construct a tenant, some policies, a VRF, as well as a bridge domain. An excerpt of the payload in task 5 of the postman collection:



## ACI Programmability with Postman & Ansible

```
<fvTenant descr="Postman API Networking Environemnt" dn="uni/tn-PostmanAPI" name="PostmanAPI" ownerKey="" ownerTag=""> #
  Tenant
    <fvCtx descr="" knwMcastAct="permit" name="Prod-VRF" ownerKey="" ownerTag="" pcEnfPref="enforced"> #VRF
    </fvCtx>
    <ndIfPol name="PostmanAPI-IF-Policy" ctrl="" descr="" hopLimit="64" mtu="9000" nsIntvl="1000" nsRetries="3" ownerKey=""
      ownerTag="" raIntvl="600" raLifetime="1800" reachableTime="0" retransTimer="0"/> # Jumbo MTU IF Policy
    <ndPfxPol ctrl="on-link,router-address" descr="" lifetime="1000" name="PostmanAPI-NDRAv6" ownerKey="" ownerTag=""
      prefLifetime="1000"/> # IPv6 RA Policy
    <fvBD arpFlood="no" descr="" mac="00:63:69:73:63:6F" multiDstPktAct="encap-flood" name="Prod-BD" ownerKey="" ownerTag=""
      unicastRoute="yes" unkMacUcastAct="proxy" unkMcastAct="flood"/> # Bridge Domain
    <fvRsBDToNdP tnNdIfPolName="PostmanAPI-IF-Policy"/> # Associating IF Policy to BD
    <fvRsCtx tnFvCtxName="Prod-VRF"/> # Associating BD to VRF
    <fvSubnet ctrl="nd" descr="" ip="fc00:10:100:1::1/64" name="" preferred="no" scope="private"> # New IPv6 Addr for BD
      <fvRsNdPfxPol tnNdPfxPolName="PostmanAPI-NDRAv6"/>
    </fvSubnet>
    <fvSubnet ctrl="nd" descr="" ip="fc00:10:100:2::1/64" name="" preferred="no" scope="private"> # New IPv6 Addr for BD
      <fvRsNdPfxPol tnNdPfxPolName="PostmanAPI-NDRAv6"/>
  </fvSubnet>
```

Since tenants are under the managed object “/mo” and universe objects “/uni” this will be the base URI for the call. See the Path for task 5 as well as the payload and execute it.

▶ 5. Create a Tenant/VRF/Policy/BD/Subnet in 1 Call

Ex

POST https://198.18.133.200/api/mo/uni.xml

Send

Note the response to the post indicating that the tenant was created successfully.

Status: 200 OK Time: 138 ms Size: 562 B

If you log into the APIC GUI you can see that the Tenant now exists, and if the tenant is drilled into that there is a VRF, BD, as well as some of the network policies. The BD also has some networking associated with it.



## ACI Programmability with Postman & Ansible

Tenant PostmanAPI

Bridge Domain - Prod-BD

Properties

Subnet	Gateway Address	Scope	Primary IP Address	Virtual IP
10.100.1.1/24		Private to VRF	False	False
10.100.2.1/24		Private to VRF	False	False
fc00:10:100:1::1/64		Private to VRF	False	False

**Step 9:** The next REST example will again use the same path however the content in the payload will we switched from XML to JSON, so instead of using <path>/uni.xml we will use <path>/uni.json in the POST method. If we examine the body of the JSON we are creating a new Application profile and under the new AP a couple of EPGs are being created as well as connected to the Prod-BD that was created in the previous step. This example shows how each class when nested in the JSON body can continue to have its respective children elements.

```
{  
  "fvAp": {  
    "attributes": {  
      "dn": "uni/tn-PostmanAPI/ap-PostmanAPI-AP",  
      "name": "PostmanAPI-AP",  
      "rn": "ap-PostmanAPI-AP"  
    },  
    "children": [{  
      "fvAEPg": {  
        "attributes": {  
          "dn": "uni/tn-PostmanAPI/ap-PostmanAPI-AP/epg-Tenant-1",  
          "name": "Tenant-1",  
          "rn": "epg-Tenant-1"  
        },  
        "children": [{  
          "fvRsBd": {  
            "attributes": {  
              "tnFvBDName": "Prod-BD"  
            },  
            "children": []  
          }  
        }]  
      }]  
    }  
  }  
}
```



When executing the step as in the previous POST step there should be a 200 OK returned from the server with an empty imdata information in the payload. This is an indication that the operation has completed successfully.

**Step 10:** Postman tasks 7, 8a, and 8b all have to do with contract creation and provider/consumer association to one of the EPGs. Task 7 is a similar in code to the previous two examples. The contract in question called AllowCommonCommunication has a single subject with the default filter.. effectively no blocking. Execute task 7 in postman to create the contract.

- ▶ 7. Create a Contract to allow all communications.

A screenshot of the Postman interface. The request method is set to POST, the URL is https://198.18.133.200/api/mo/uni.json, and the response status is 200 OK. The response body is empty, indicating the operation was successful.

Task 8a applies the contract to the EPG called Tenant-1 as a provider. The unique aspects of this piece of code is the path of the URI, now zeroing in on the specific tenant that we want to modify which reduces the amount of information that needs to go into the body/payload of the transaction, as well as the format of the message in JSON format.

- ▶ 8a. Connect Tenant 1 EPG to Provide Allow All Contract

A screenshot of the Postman interface for Task 8a. The request method is POST, the URL is https://198.18.133.200/api/mo/uni/tn-PostmanAPI/ap-PostmanAPI/AP/epg-Tenant-1.json, and the response status is 200 OK. The response body is empty. The Body tab shows a JSON payload:

```
1 {  
2   "fvRsProv": {  
3     "attributes": {  
4       "tnVzBrCPName": "AllowCommonCommunication"  
5     }  
6   }  
7 }
```

The Task 8b performs a similar contract association to the Tenant-1 but as a consumer and is doing so in XML format. The various examples are to show how the same or similar type of operations can be performed in the different formats

- ▶ 8b. Connect Tenant 1 EPG to Consume Allow All Contract

A screenshot of the Postman interface for Task 8b. The request method is POST, the URL is https://198.18.133.200/api/mo/uni/tn-PostmanAPI/ap-PostmanAPI/AP/epg-Tenant-1.xml, and the response status is 200 OK. The response body is empty. The Body tab shows an XML payload:

```
1 <fvRsCons childAction="" dn="uni/tn-PostmanAPI/ap-PostmanAPI/AP/epg-Tenant-1/rscons-AllowCommonCommunication"/>
```

Inspect the GUI and look at the new tenant and various constructs that were created during this exercise.



---

**Summary:**

We covered the following in this section of the lab.

- REST Call Basics using GET and POST methods
- Understanding how Cookies work in postman and the significance of the APIC Token
- How to select between XML and JSON and deal with various payload examples
- How to use parameters to filter or expand details on the data being requested
- How to create several items in a single REST call.

## Ansible Programmability with ACI

In the previous demos we learned the basics of identifying REST calls within API, and how to break apart the individual elements of the payload to understand their purpose. This section will build on that by using postman to perform basic REST calls to the ACI environment using both GET and POST methods. As well as using postman to perform many basic GET and POST methods. In this module we will continue to build on this practice by performing similar operations in ansible as well as using combinations of methods to extract information and perform tasks.

**Step 1:** From the windows VM click the ansible putty shortcut on the desktop



This Putty link will auto log into the system using the credentials.

NOTE: If connecting via VPN from your local machine use SSH to 198.18.134.50 and use credentials ansible/ansible.

**Step 2:** In the Ansible user's home folder there will be several YML files listed. Unlike the previous lab demo in postman where we had to login, and keep a token alive to perform other calls, the ansible application will do that for us. Let's break down one of the ansible files to get a lay of the land.

YML files start with the “---”

```
# ansible-playbook -v -i "198.18.133.200," 3.Get All-Tenants.yml
- name: Ansible Create Legacy Common Tenant
# Look for hosts in the [aci] container of the inventory file
hosts: all
# Use the local Ansible system
connection: local
# Dont bother with fact computation about remote entities wont need that for the simple plays being run
gather_facts: no

# Use Username and privatekey provided at runtime combined with the hostname list to run in the task
# there are alot of ways to pass username and password creds and hosts.
vars:
  aci_username: "admin"
  aci_password: "C1sc012345"
  aci_host: "{{ inventory_hostname }}"

tasks:
- name: Query all tenants
  aci_tenant:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    state: query
    delegate_to: localhost
    register: query_result

- debug: msg="{{ query_result }}"
```

Comments begin with a #

Playbooks are generally given a name

This is how variables are defined.. **Don't put creds in the clear in production.**

This is how a variable is set to another variable  
In this casethe address lists that we are using at runtime with the **-i** "IP,"

A playbook will contain tasks to perform and each task will have a name that the user can set

An ansible module will contain one or more variables that instructs it how to run as well as return variables as results

Ansible files are run by executing “ansible-playbook -v -i "198.18.133.200," <Ansible file>”. The -

i along with a list of IPs can be provided at runtime to remove the need for an inventory file to be present. If we wanted to use the same playbook on more than one device at a time all that would need to happen is to amend the IP list. Ex. -i

“198.18.133.200,198.18.200.199,hostfqn.example.com,”. Notice the “,” at the end of the ansible list. That is required .

**Step 3:** The ansible file that was just examined in the previous step will be the first file that will be run.

This file uses the ansible ACI\_Tenant module to query all the tenants in the ACI environment. We can execute the file by running: “ansible-playbook -v -i “198.18.133.200,” 3.Get-All-Tenants.yml” at the command line. For more information on the ACI tenant module for ansible follow the link below:

[https://docs.ansible.com/ansible/2.7/modules/aci\\_tenant\\_module.html#aci-tenant-module](https://docs.ansible.com/ansible/2.7/modules/aci_tenant_module.html#aci-tenant-module)

```
aci_tenant:  
  host: "{{ aci_host }}"  
  username: "{{ aci_username }}"  
  password: "{{ aci_password }}"  
  validate_certs: no  
  state: query  
  delegate_to: localhost  
  register: query_result
```

The host is populated but the inventory variable which is ultimately populated by what we have in the runtime execution. The username's and passwords are fixed variables in the files, again DON'T do it like this in production. You don't want you credentials in the clear. We are choosing not to validate certs (handy if a self-signed cert is what is running on the APICS), and the state:query command is instructing the module to do a query GET operation. We are storing the result of the GET operation with the register statement, so we can pretty print it in the next task. The resulting output of the ansible operation is two tasks should have run the query task and the debug task.



## ACI Programmability with Postman & Ansible

```
[ansible@ansible ~]$ ansible-playbook -v -i "198.18.133.200," 3.Get-All-Tenants.yml
Using /etc/ansible/ansible.cfg as config file

PLAY [Ansible Create Legacy Common Tenant] ****
TASK [Query all tenants] ****
ok: [198.18.133.200 => localhost] => {"changed": false, "current": [{"fvTenant": {"attributes": {"cName": "uni/tn-common", "l1Out": "local", "modTs": "2018-11-30T03:51:17.730+00:00", "monPolDn": "uni/tn-common", "nameAlias": "", "ownerKey": "", "ownerTag": "", "status": "", "uid": "0"}}, {"fvTenant": {"attributes": {"childAction": "", "descr": "Postman API Networking Environment", "dn": "uni/tn-common/monepg-default", "name": "infra", "nameAlias": "", "ownerKey": "", "ownerTag": "", "status": "", "uid": "15374"}, "fvTenant": {"attributes": {"cName": "uni/tn-mngmt", "l1Out": "local", "modTs": "2018-11-30T03:51:24.807+00:00", "monPolDn": "uni/tn-mngmt", "nameAlias": "", "ownerKey": "", "ownerTag": "", "status": "", "uid": "0"}}}], "msg": {"changed": false, "current": [{"fvTenant": {"attributes": {"childAction": "", "descr": "", "dn": "uni/tn-common", "l1Out": "local", "modTs": "2018-11-30T03:51:17.730+00:00", "monPolDn": "uni/tn-common/monepg-default"}}, {"fvTenant": {"attributes": {"childAction": "", "descr": "Postman API Networking Environment", "dn": "uni/tn-common/monepg-default", "name": "infra", "nameAlias": "", "ownerKey": "", "ownerTag": "", "status": "", "uid": "15374"}, "fvTenant": {"attributes": {"cName": "uni/tn-mngmt", "l1Out": "local", "modTs": "2018-11-30T03:51:24.807+00:00", "monPolDn": "uni/tn-mngmt", "nameAlias": "", "ownerKey": "", "ownerTag": "", "status": "", "uid": "0"}}}]}}

TASK [debug] ****
ok: [198.18.133.200] => (
```

From the output of the ansible operation if you scroll up and look at the output you can see that this was functionally the same to the get all tenants that we did in the previous lab in task 3 with one exception. In ansible we will generally receive JSON results, this doesn't mean we can't send XML. (this will be covered in a later step)

GET ▾

<https://198.18.133.200/api/node/class/fvTenant.json>

**Step 4:** Instead of using a predefined ansible module we can submit direct REST calls similar to what we did in postman. This allows us to specify a path, methods, and depending on method the specific content we want to exchange with the destinations. Let's look at the example in Task 4a. Hopefully host, username, password, and validate certs are understood at this point.

### tasks:

```
- name: Query All Tenants
  aci_rest:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    use_proxy: no
    method: get
    path: api/node/mo/uni/tn-infra.json?query-target=children
    output_level: debug
    register: result
```

Use proxy means to use a proxy server. Method is the REST method that ansible should use to

communicate to ACI (GET or POST). Path is the URI. This goes back to knowing the object or class of object we want to retrieve. You can reference the API explorer or the MIM to determine the path. In looking at the Ansible documentation the output level can affect what is displayed and/or retrieved during the task execution. Again, in this example we are saving the result, so we can effect changes later (or in this case print it out).

**Run the task 4a by `ansible-playbook -v -i "198.18.133.200," 4a.Tenant-Get-1Layer.yml` and review the output.**

Query Parameters at the end of the path are also valid . See task 4b the only significant delta is in the filter parameters where the query target was changed from Children to subtree and an additional filter to only obtain the application profiles and EPGs is now present.

```
path: api/node/mo/uni/tn-infra.json?query-target=subtree&target-subtree-class=fvAp,fvAEPg
```

**Run the task 4b by `ansible-playbook -v -i "198.18.133.200," 4b.Tenant-Get-subtree.yml` and review the output.**

The output of the program should look exactly the same as the 4b postman task (it's the same REST call). Since we are looking at the same rest calls the task 4c again will query the tenants and using only the subtree query target will get a lot of data. One thing to note here is that ansible can iterate through this data and retrieve values for use later. We will look at a rudimentary example of this in a subsequent step.

```
method: get
path: api/node/mo/uni/tn-infra.json?query-target=subtree
output_level: debug
```

**Run the task 4c by `ansible-playbook -v -i "198.18.133.200," 4c.Tenant-Get-All-Items.yml` and review the output.**

**NOTE:** Try modifying the URL on one of the get functions from .json to .xml and see what the output looks like. What did ansible do to the XML result?

**Step 5:** Leveraging the ACI\_REST module we can make changes to the ACI environment in much the same way that we did in the postman exercises. When using the aci\_rest module to create or update an object the method would change from get to post and a new field called “content:” is used to store the desired payload for that particular call. Let’s look at the changes needed to effectively send code that will create (in XML) to create a tenant, a VRF, and a bridge domain, along with some of the requisite networking policies.



```
    https://ACI-PROXY1.NO
    method: post
    path: /api/mo/uni.xml
    content: |
        <fvTenant descr="Legacy Combined Networking Environemnt" dn="uni/tn-Ansible-Combined-Prod" name="Ansible-Comb
            <fvCtx descr="" knwMcastAct="permit" name="Combined-Prod-VRF" ownerKey="" ownerTag="" pcEnfPref="enforced">
                </fvCtx>
                <ndIfPol name="Ansible-IF-Policy" ctrl="" descr="" hopLimit="64" mtu="9000" nsIntvl="1000" nsRetries="3" o
                <ndPfxPol ctrl="on-link,router-address" descr="" lifetime="1000" name="Ansible-NDRAv6" ownerKey="" ownerTag
                <fvBD arpFlood="no" descr="" mac="00:63:69:73:63:6F" multiDstPktAct="encap-flood" name="Combined-Prod-BD" o
                    <fvRsBDToNdP tnNdIfPolName="Ansible-IF-Policy"/> # Associating IF Policy to BD
                    <fvRsCtx tnFvCtxName="Combined-Prod-VRF"/> # Associating BD to VRF
                    <fvSubnet ctrl="nd" descr="" ip="fc00:10:100:1::1/64" name="" preferred="no" scope="private"> # New IPv6
                        <fvRsNdPfxPol tnNdPfxPolName="Ansible-NDRAv6"/>
                    </fvSubnet>
                    <fvSubnet ctrl="nd" descr="" ip="fc00:10:100:2::1/64" name="" preferred="no" scope="private"> # New IPv6
                        <fvRsNdPfxPol tnNdPfxPolName="Ansible-NDRAv6"/>
                    </fvSubnet>
                    <fvSubnet ctrl="nd" descr="" ip="10.100.1.1/24" name="" preferred="no" scope="private"> # New IPv4 Addr f
                </fvSubnet>
                <fvSubnet ctrl="nd" descr="" ip="10.100.2.1/24" name="" preferred="no" scope="private"> # New IPv4 Addr f
            </fvBD>
        </fvTenant>
    register: result
```

The method is now post, the path is /api/mo/uni.xml and the content has the XML data that we want to provide to create the objects.

#### Run task 5 by ansible-playbook -i "198.18.133.200," 5.Create-Tenant-Networking.yml and review the output.

One important change that should be observed is the addition now of a changed=1 result in the ansible operation. This output implies that 2 tasks have successfully run and one of the tasks resulted in a change in the environment. (the second task is still the debugging output)

```
PLAY RECAP ****
198.18.133.200 : ok=2    changed=1    unreachable=0    failed=0
```

This is an important distinction in ansible when tasks are re-run with the same data. Re-run task 5 and see what output we get in the play recap.

```
PLAY RECAP ****
198.18.133.200 : ok=2    changed=0    unreachable=0    failed=0
```

Despite the REST call in the task and the post function ansible determined that no changes where actually made on the second run.

**Step 6:** Multiple tasks can be combined in ansible as well as the use of variables and iteration on the fly for subsequent tasks to be successful. In task 6 we will configure the Application Profile, and two EPGs under the ansible tenant we created in the previous step. The one enhancement to this is prior to the creation step we are going to do a read operation on the tenant using an ansible module called bridge domain. This provides the information regarding the BD's in the tenant. A register will be used to save that resulting data, then on the configure step that register will be used to insert the bridge domain information on the fly. Let's break down the pieces of this file:

```

tasks:
- name: Get the BD for the Ansible Tenant # Shows how to work with vars in Ansible
  aci_bd:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    tenant: Ansible-Combined-Prod
    state: query
    delegate_to: localhost
    register: bd_query_result
      Specify the tenant to
      get the information on
      Save the result

# - name: Output Result
#   debug:
#     msg: "{{ bd_query_result.current[0].fvTenant.children[0].fvBD.attributes.name }}" # Can Parse Jason using .a
      A Debug I was using to play with
      the parsing now commented out

- name: Configure Ansible Common Tenant Application Configuration AP/EPG
  aci_rest:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    use_proxy: no
    validate_certs: no
    method: post
    path: /api/mo/uni.json
    content: |
      "fvAp": {
        "attributes": {
          "dn": "uni/tn-Ansible-Combined-Prod/ap-Ansible-Combined-Prod",
          "name": "Ansible-Combined-Prod",
          "rn": "ap-Ansible-Combined-Prod"
        },
        "children": [
          "fvAEPg": {
            "attributes": {
              "dn": "uni/tn-Ansible-Combined-Prod/ap-Ansible-Combined-Prod/epg-Tenant-1",
              "name": "Tenant-1",
              "rn": "epg-Tenant-1"
            },
            "children": [
              "fvRsBd": {
                "attributes": {
                  "tnFvBDName": "{{ bd_query_result.current[0].fvTenant.children[0].fvBD.attributes.name }}"
                }
              }
            ]
          }
        ]
      }
      Using the name of the BD found in the
      query result as a variable in the JSON
      body of the creation request

```

When this runs in ansible the task to query the BD information returns an OK (as nothing is modified in a GET) and the subsequent task where we are configuring can be used.

**Run task 6 by ansible-playbook -i "198.18.133.200," 6.Create-APandEPG.yml and review the**

## output.

```
[ansible@ansible ~]$ ansible-playbook -i "198.18.133.200," 6.Create-APandEPG.yml
PLAY [Ansible Create Legacy Common Tenant] ****
TASK [Get the BD for the Ansible Tenant] ****
ok: [198.18.133.200 -> localhost]

TASK [Configure Ansible Common Tenant Application Configuration AP/EPG] ****
changed: [198.18.133.200]

PLAY RECAP ****
198.18.133.200 : ok=2    changed=1    unreachable=0    failed=0
```

If we look back at the GUI in the Tenant->AP->EPG (Tenant-1)->Policy we can see that the associated bridge domain is in fact the Combined-Prod-BD

The screenshot shows the Cisco ACI GUI interface. On the left, there is a navigation sidebar with a tree structure. The 'Tenant-1' node is expanded, showing various sub-components like Domains, Static Ports, Fibre Channel, Contracts, Subnets, L4-L7 Virtual IPs, and so on. On the right, a detailed configuration pane is open for an EPG. The top part of the pane is labeled 'Properties' with fields for 'Description' (optional), 'Tags' (enter tags separated by comma), 'Global Alias' (empty), 'uSeg EPG' (false), 'pcTag(class)' (16387), 'QoS class' (Unspecified), 'Custom QoS' (select a value), 'Intra EPG Isolation' (Enforced), 'Preferred Group Member' (Exclude), 'Configuration Status' (applied), 'Configuration Issues' (empty), 'Label Match Criteria' (AtleastOne), and 'Bridge Domain' (Ansible-Combined-Prod/Combined-Prod-BD). The status bar at the bottom of the screen also displays the same bridge domain name.

**Step 7:** One of the benefits of the Ansible modules and how the functions are broken down into tasks is that each task can be evaluated for success when a playbook is executed. This allows for a prescriptive order of operations to be achieved. The other significant benefit is that if there isn't a predefined ansible module to

the task we want to achieve then the aci\_rest module can be used. The final example uses the following ansible modules to complete a contract creation and a tenant association to the contract:

- aci\_contract
- aci\_contract\_subject
- aci\_contract\_subject\_to\_filter
- aci\_epg\_to\_contract
- aci\_rest (just to level set functions that we have seen in the postman step)

The outcome of this task like in the postman section is to create a contract called AllowCommonCommunication with the AllowAll subject and the default filter binding the tasks are show below. Notice the state:present variable in the tasks will create. How did we know this? It's in the ansible module documentation: [https://docs.ansible.com/ansible/2.7/modules/aci\\_contract\\_module.html#aci-contract-module](https://docs.ansible.com/ansible/2.7/modules/aci_contract_module.html#aci-contract-module)

state	<b>Choices:</b> <ul style="list-style-type: none"> <li>• absent</li> <li>• <b>present</b> ←</li> <li>• query</li> </ul>	Use <code>present</code> or <code>absent</code> for adding or removing. Use <code>query</code> for listing an object or multiple objects.

```
tasks:
  - name: Add a new contract
    aci_contract:
      host: "{{ aci_host }}"
      username: "{{ aci_username }}"
      password: "{{ aci_password }}"
      validate_certs: no
      tenant: Ansible-Combined-Prod
      contract: AllowCommonCommunication
      description: Allow All
      scope: application-profile
      state: present
    delegate_to: localhost

  - name: Add the AllowAll Subject
    aci_contract_subject:
      host: "{{ aci_host }}"
      username: "{{ aci_username }}"
      password: "{{ aci_password }}"
      validate_certs: no
      tenant: Ansible-Combined-Prod
      contract: AllowCommonCommunication
      subject: AllowComs
      reverse_filter: yes
      state: present
    delegate_to: localhost

  - name: Add a new contract subject to filer binding
    aci_contract_subject_to_filter:
      host: "{{ aci_host }}"
      username: "{{ aci_username }}"
      password: "{{ aci_password }}"
      validate_certs: no
      tenant: Ansible-Combined-Prod
      contract: AllowCommonCommunication
      subject: AllowComs
      filter: default
      state: present
    delegate_to: localhost
```



One of the other benefits to using the pre-defined ansible modules is that the code is more human readable as to the intent with the Tenant: ; ap: ; epg:; contract info, etc. Vs the when using a rest call that has the specific class definitions. Take a look at this example where we are performing a similar task in joining the Tenant 1 EPG to the AllowCommonCommunication task, the more “ansiblish” step which is connect the EPG as the provider vs the rest step which does the same thing as the consumer. You decide.. which is easier to understand.

```
- name: Connect Tenant 1 EPG to Provide Allow All Contract
  aci_epg_to_contract:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    tenant: Ansible-Combined-Prod
    ap: Ansible-Combined-Prod
    epg: Tenant-1
    contract: AllowCommonCommunication
    contract_type: provider
    state: present
    delegate_to: localhost

- name: Connect Tenant 1 EPG to Consume Allow All Contract
  aci_rest:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    use_proxy: no
    method: post
    path: /api/mo/uni/tn-Ansible-Combined-Prod/ap-Ansible-Combined-Prod/epg-Tenant-1.json
    content: |
      "fvRsCons": {
        "attributes": {
          "tnVzBrCPName": "AllowCommonCommunication",
        },
        "children": []
      }
  delegate_to: localhost
```

**Run task 7 by ansible-playbook -i "198.18.133.200," 7.CreateContractAssignEPGs.yml and review the output.**

Note how each step provides a status update as to if there was data changed or not.



## ACI Programmability with Postman & Ansible

```
[ansible@ansible ~] $ ansible-playbook -i "198.18.133.200," 7.CreateContractAssignEPGs.yml  
PLAY [Ansible Create Legacy Common Tenant] *****  
  
TASK [Add a new contract] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Add the AllowAll Subject] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Add a new contract subject to filer binding] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Connect Tenant 1 EPG to Provide Allow All Contract] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Connect Tenant 1 EPG to Consume Allow All Contract] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Connect Tenant 2 EPG to Provide Allow All Contract] *****  
changed: [198.18.133.200 -> localhost]  
  
TASK [Connect Tenant 2 EPG to Consume Allow All Contract] *****  
changed: [198.18.133.200 -> localhost]  
  
PLAY RECAP *****  
198.18.133.200 : ok=7    changed=7    unreachable=0    failed=0
```

**Step 8: LAB CLEANUP** – The 8<sup>th</sup> step of the lab is an ansible script that goes and collects all tenants in the system, compares those tenants to the initial tenant list (infra, mgmt, common) that have been created through the lab exercises and or any off-script action and deletes them. This script is also very cool from an implementation perspective as it continues to build off of the concepts of using variables and routines in between steps.

**Run task 8 by ansible-playbook -i "198.18.133.200," 8.LabCleanup.yml**



```
register: query_result

# Parse the query result and grab only the name attribute into a list and save that to a new var c
- set_fact:
  tenant_list: "{{ query_result | json_query('current[*].fvTenant.attributes.name') }}"

# Use the difference function to derive a list from the tenants in the system - the protected tena
- set_fact:
  delete_tenant_list: "{{ tenant_list | difference(my_protected_tenants) }}"

# Use the ACI Tenant module and loop through the task with delete list substituted for the tenant
# The state:absent implies a delete function.
- name: Delete Lab Tenants
  aci_tenant:
    host: "{{ aci_host }}"
    username: "{{ aci_username }}"
    password: "{{ aci_password }}"
    validate_certs: no
    tenant: "{{ item }}"
    state: absent
  loop: "{{ delete_tenant_list }}"
```

Some pieces to call out specifically are the use of the saved variable `query_result` which is piped to a `json_query` function to iterate and extract the name values out of the result and save those to a list called `tenant_list`. Then a separate routine which uses a difference function will compare and extract the tenants NOT in the protected list and store those to a separate variable called `delete_tenant_list`. This final variable is used in the last step where the `aci_tenant` module is leveraged to loop through and delete the tenants in the delete list.

## Summary:

We covered the following in this section of the lab.

- Ansible Basics – Framework of YML and common aspects to executing ansible playbooks
- Using ansible modules and ansible rest modules
- Using variables in ansible and debugs
- Iterating through results and looping through tasks.



Americas Headquarters  
Cisco Systems, Inc.  
San Jose, CA

Asia Pacific Headquarters  
Cisco Systems (USA) Pte. Ltd.  
Singapore

Europe Headquarters  
Cisco Systems International BV Amsterdam,  
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at [www.cisco.com/go/offices](http://www.cisco.com/go/offices).

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: [www.cisco.com/go/trademarks](http://www.cisco.com/go/trademarks). Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)