Cisco Connect Denver 2018

DEVNET ACI Programmability Hands On

Powered by

dCloud: The Cisco Demo Cloud

11/2018, Version 1.0

ACI Programmability Introduction:

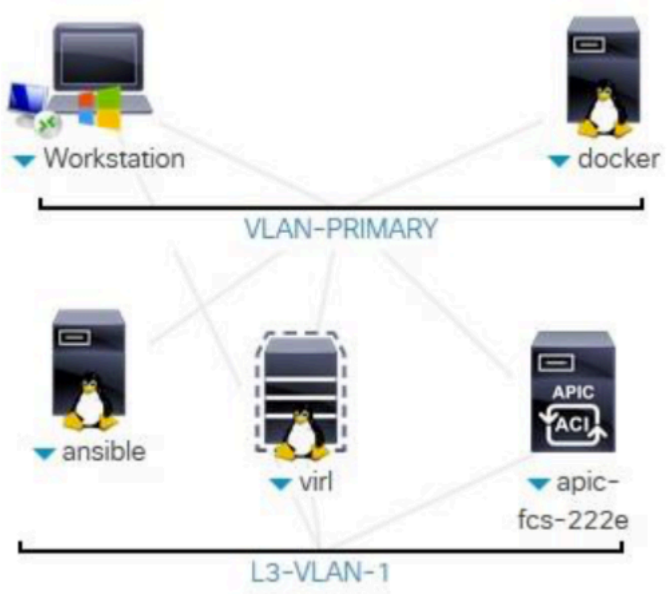
Stuff

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:

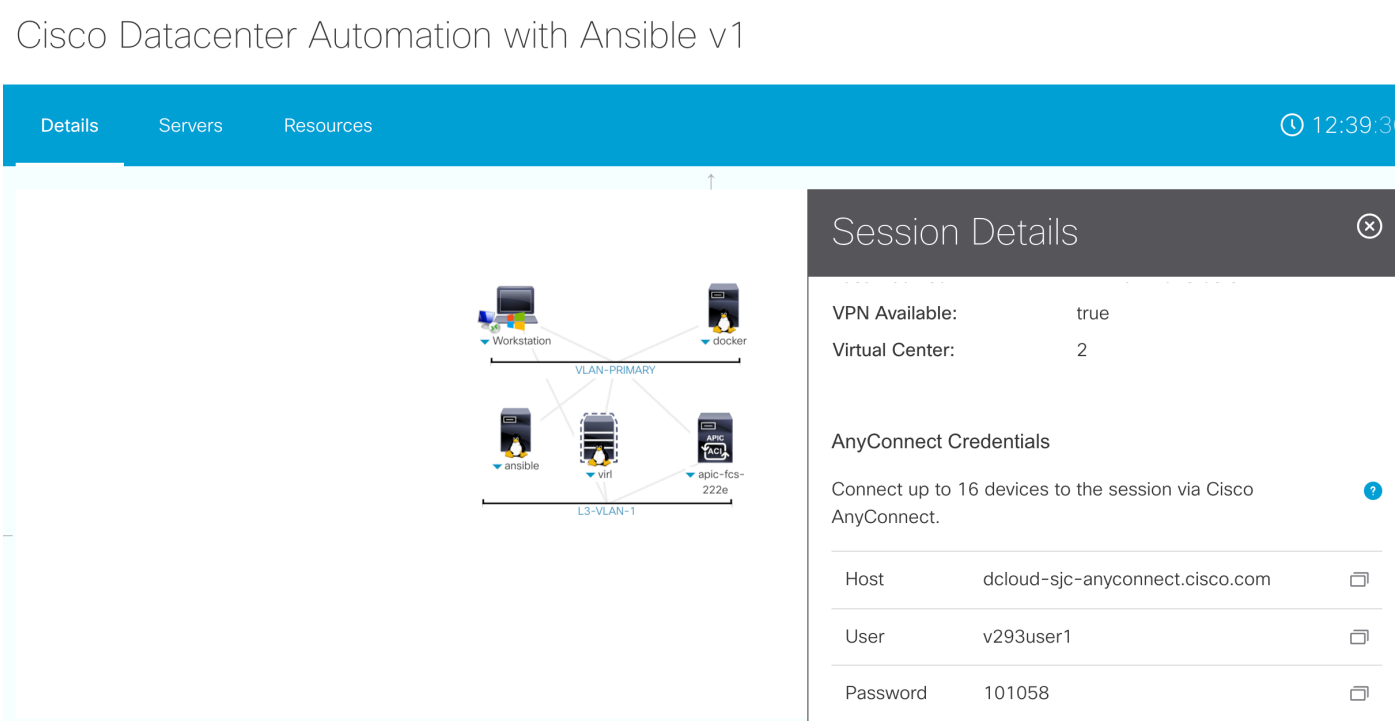
- A Laptop computer with access to the Internet
- AnyConnect, VPN software used to connect your desktop into the Virtual Lab
- Microsoft Remote Desktop, used to connect into the virtual Lab Desktop

Lab Access - VPN

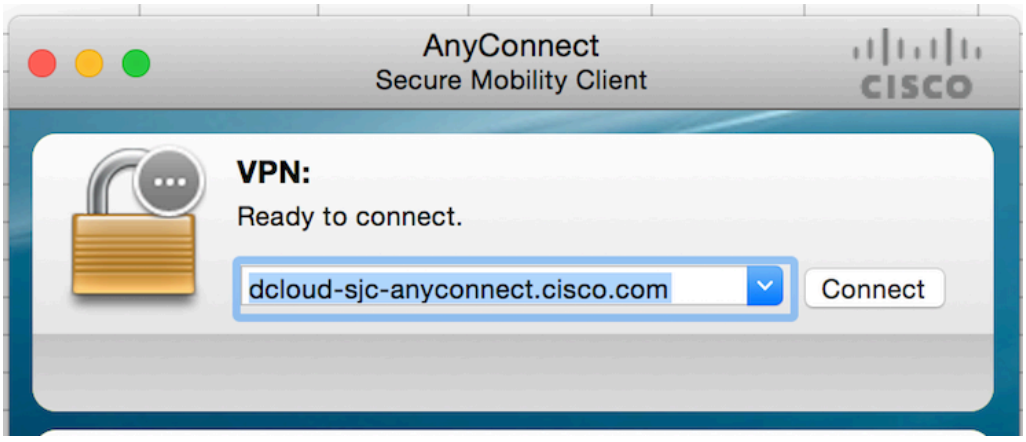
Each participant will have an individual POD to work on. Pod Credentials will be at each station/seat location. Each POD is dedicated to an individual user, so there is no impact to other PODs when making changes. You will receive a PoD already started for you Follow step by step instructions to login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

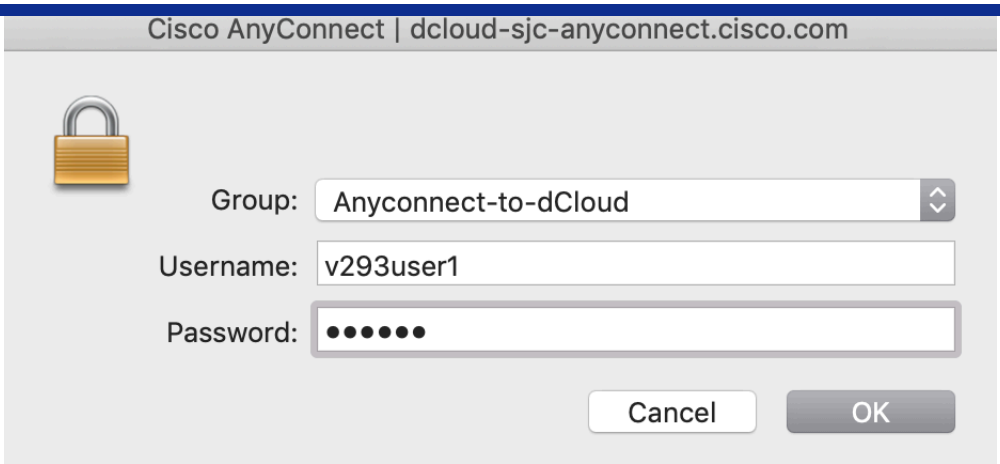
Step 2: Look into Session details and scroll down to find out your credentials for VPN AnyConnect



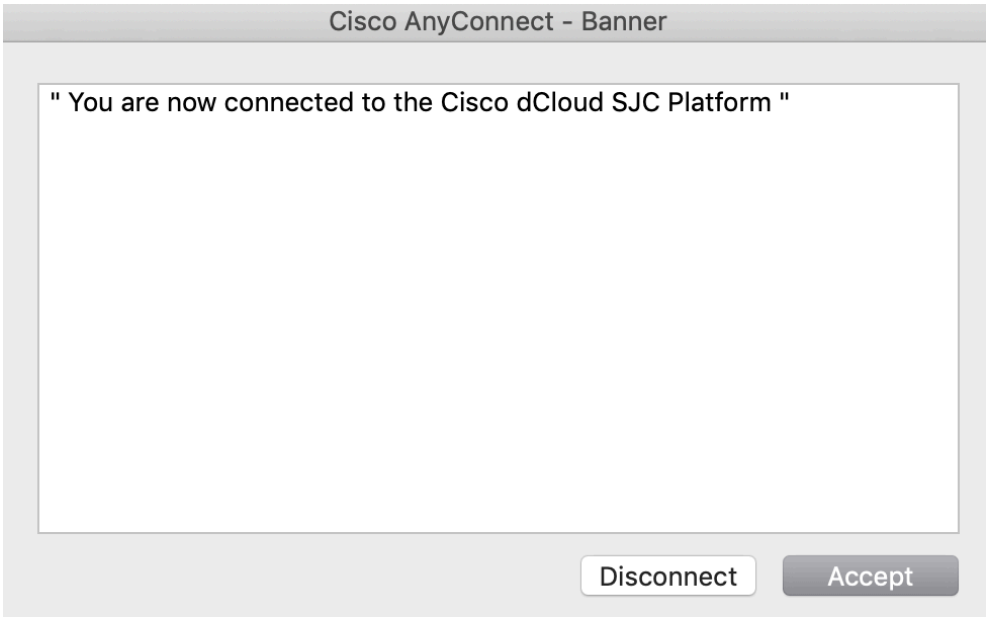
Step 3: Connect to AnyConnect VPN ***dcloud-sjc-anyconnect.cisco.com*** with the username and password provided on session details information window



Below is an example of user logging into POD1

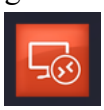


Hit accept when the prompt appears to accept the VPN connection login



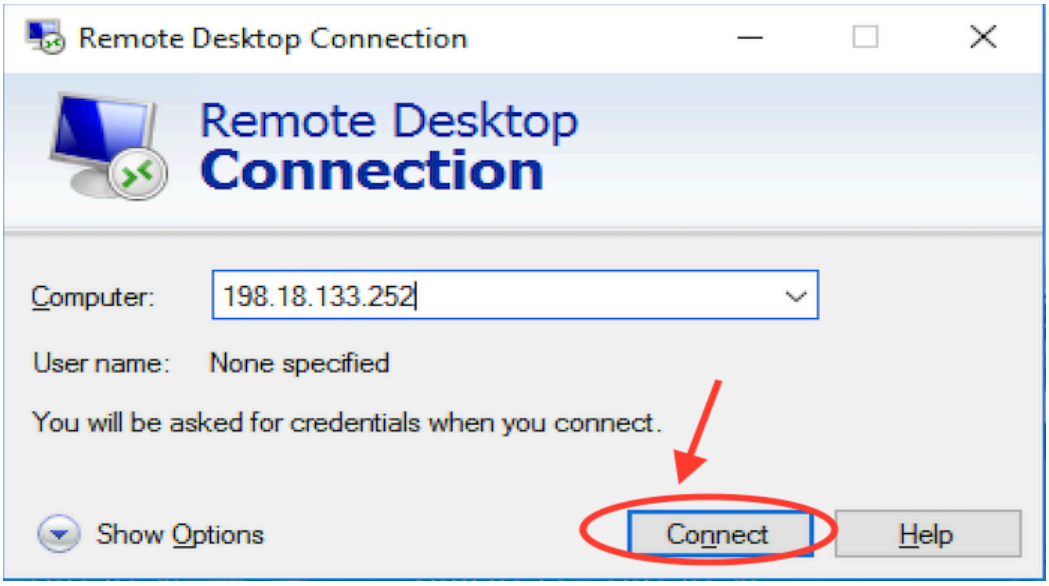
Step 4:

- 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

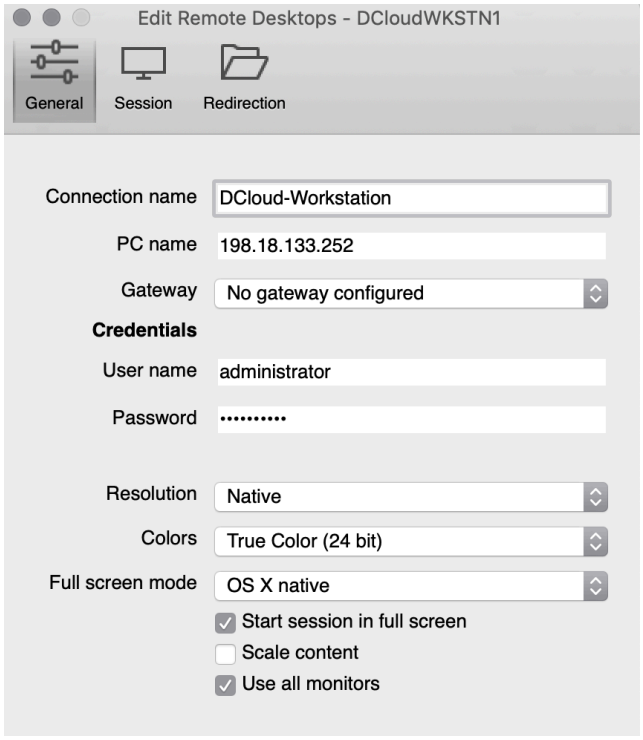


- Enter the ip address provided for the POD to connect to the remote client

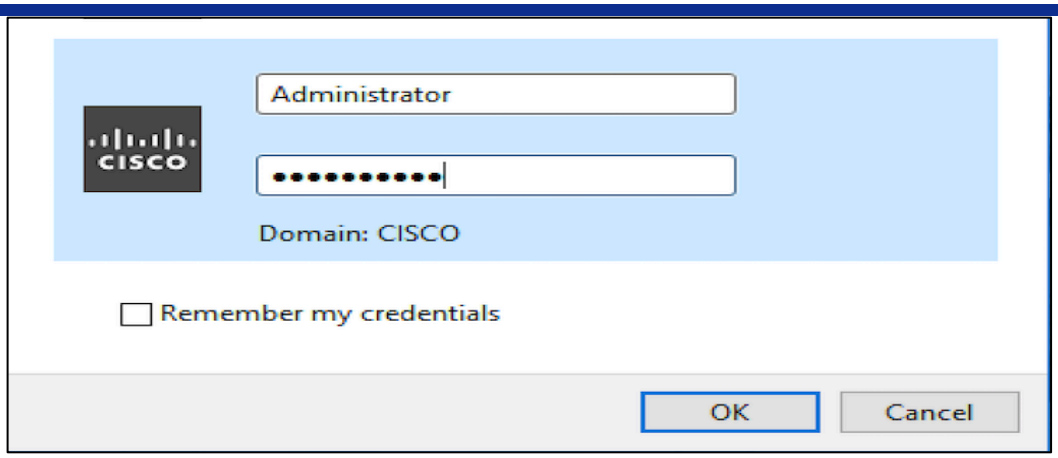
Windows:



MAC:



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

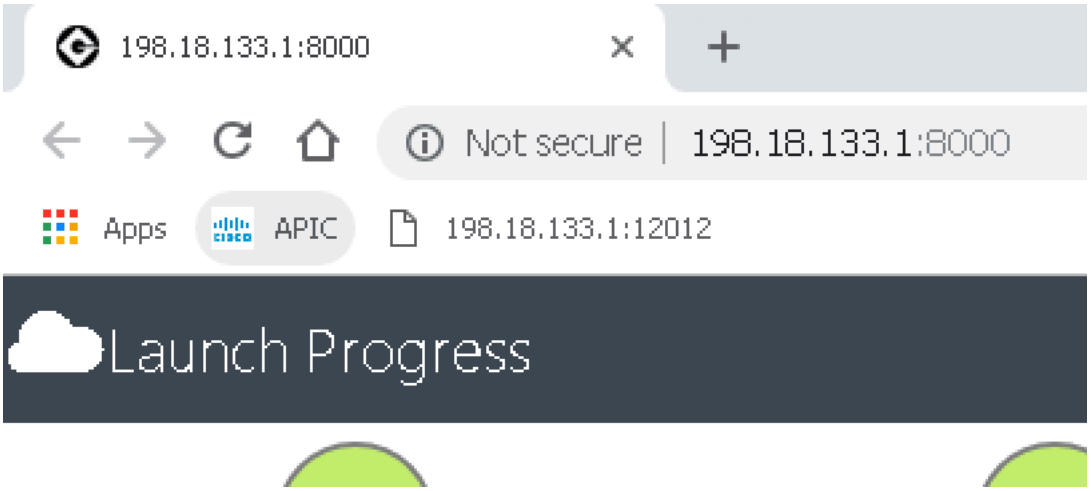


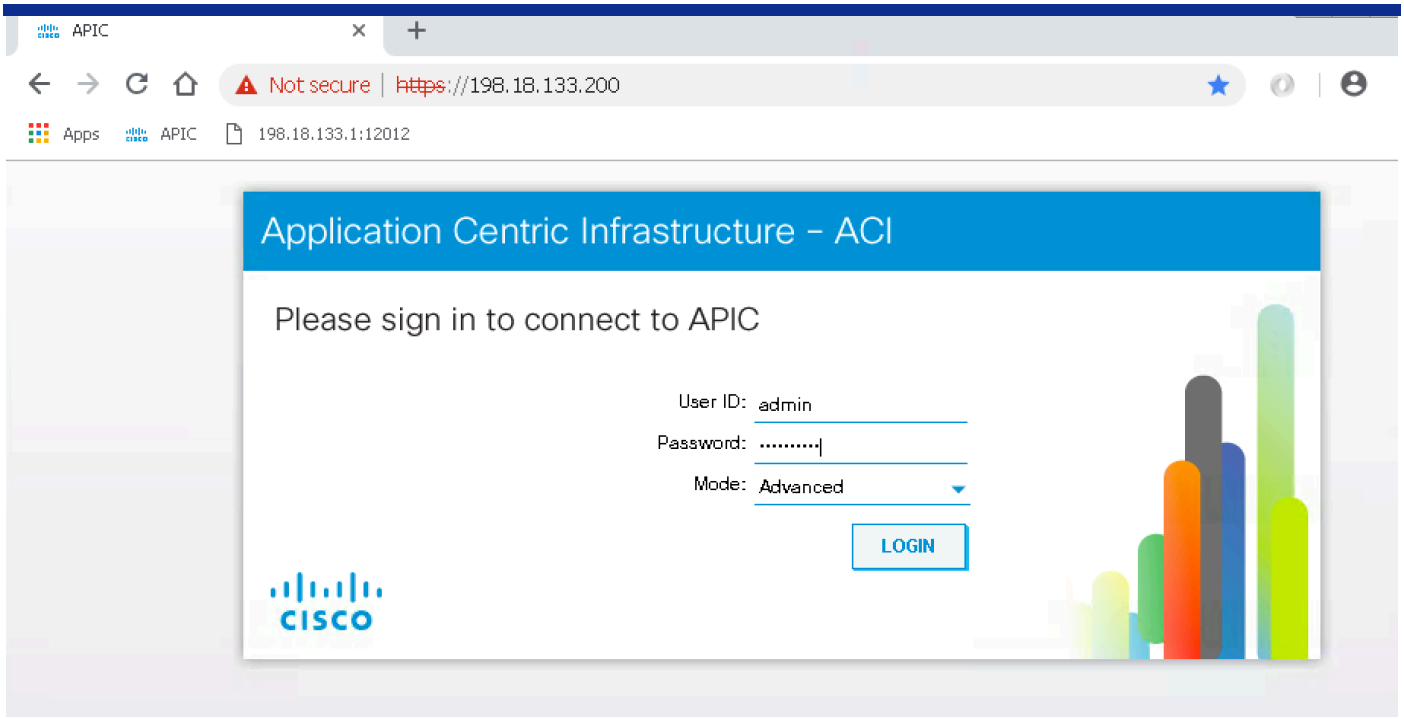
- Accept the connection certificate to login to the Virtual Lab Windows Desktop.

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/

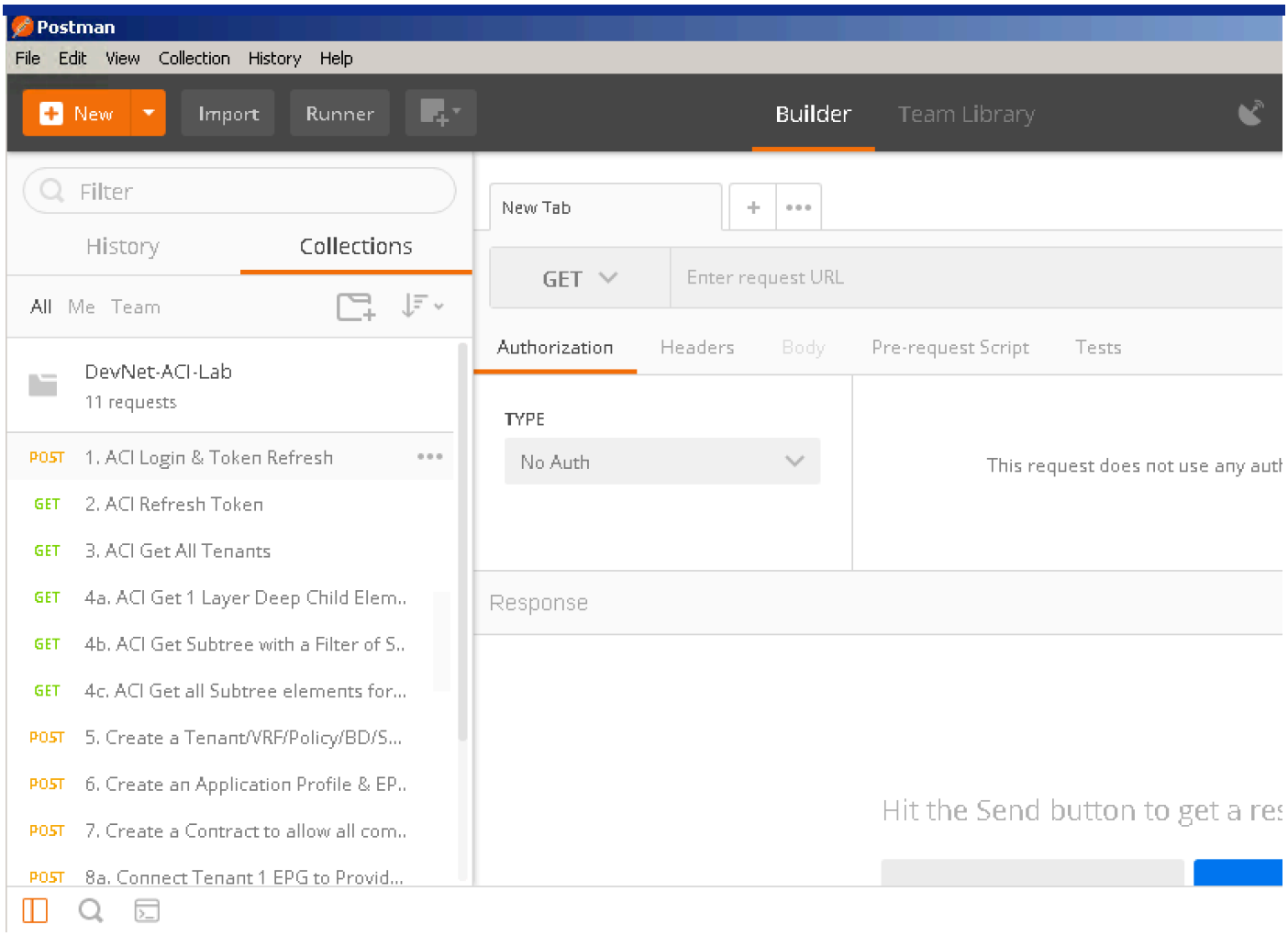




Log into the APIC GUI using the admin/C1sco12345 credentials.

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



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)

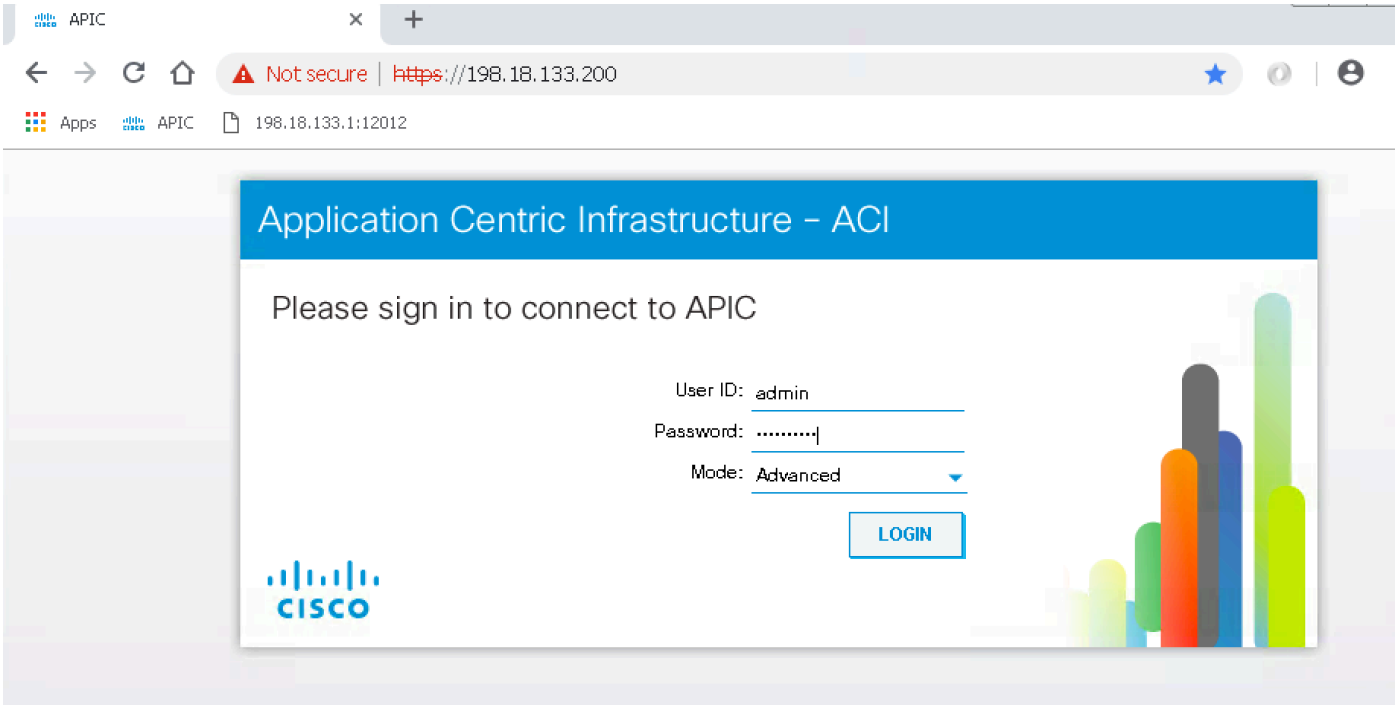
Step 3: Verify in the ansible home directory there should also be several YML files for the lab exercises and the missions.

```
[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
```

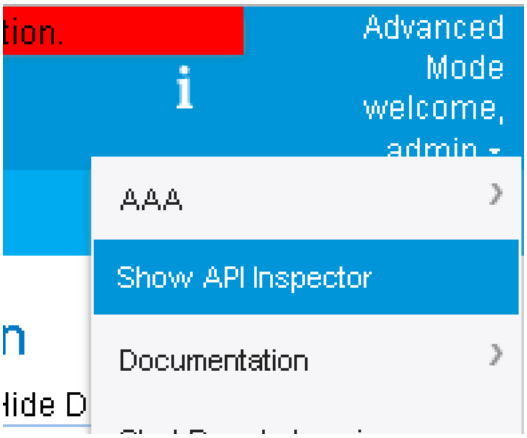
ACI API Inspector

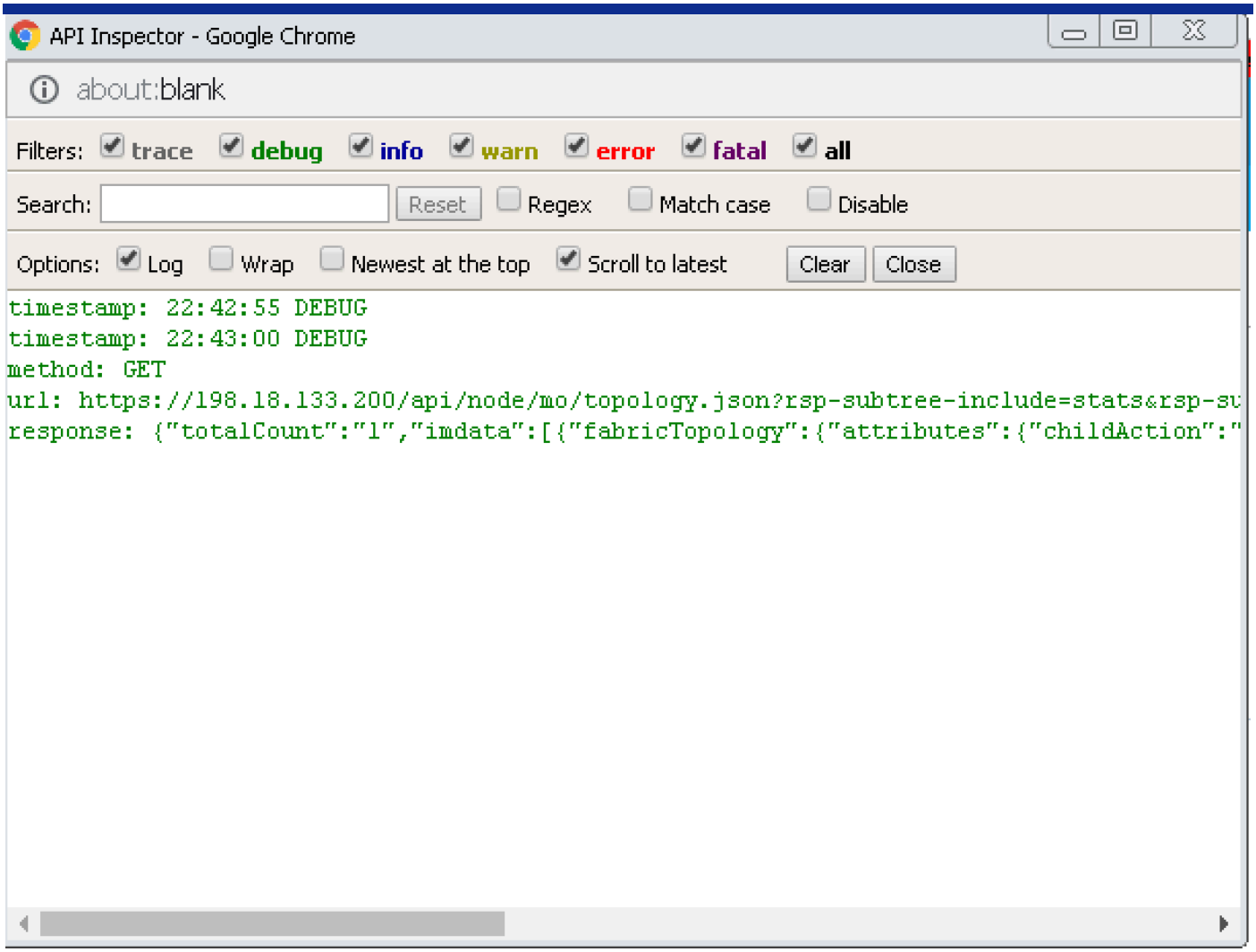
In this section, we will perform some exercises using the ACI inspector as functions are performed in the GUI to extrapolate rest calls.

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

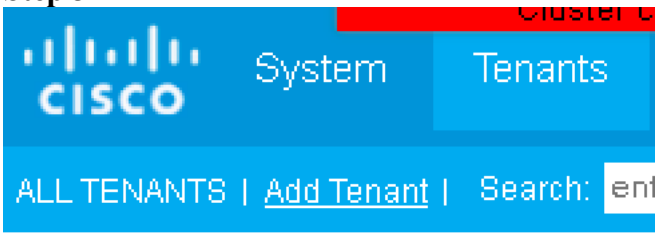


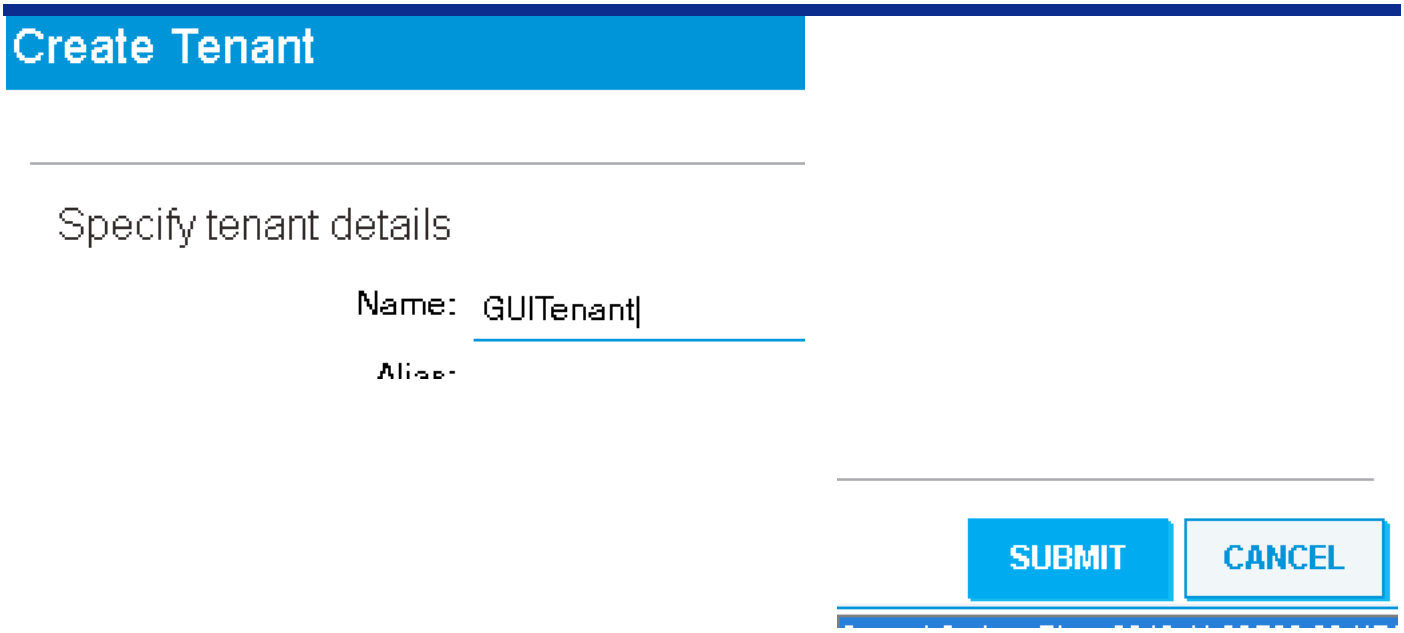
Step 2 In the upper left-hand side of the screen find where it has out login name click the down arrow and find the Show API Inspector and click. This will open a separate window with the API inspector. It may be necessary to resize the parent and API windows, so they can be easily toggled.





Step 3 Click Tenants and Create a New Tenant. Call the Tenant "GUITenant".

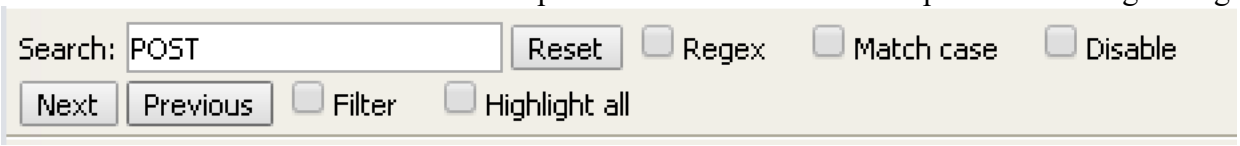


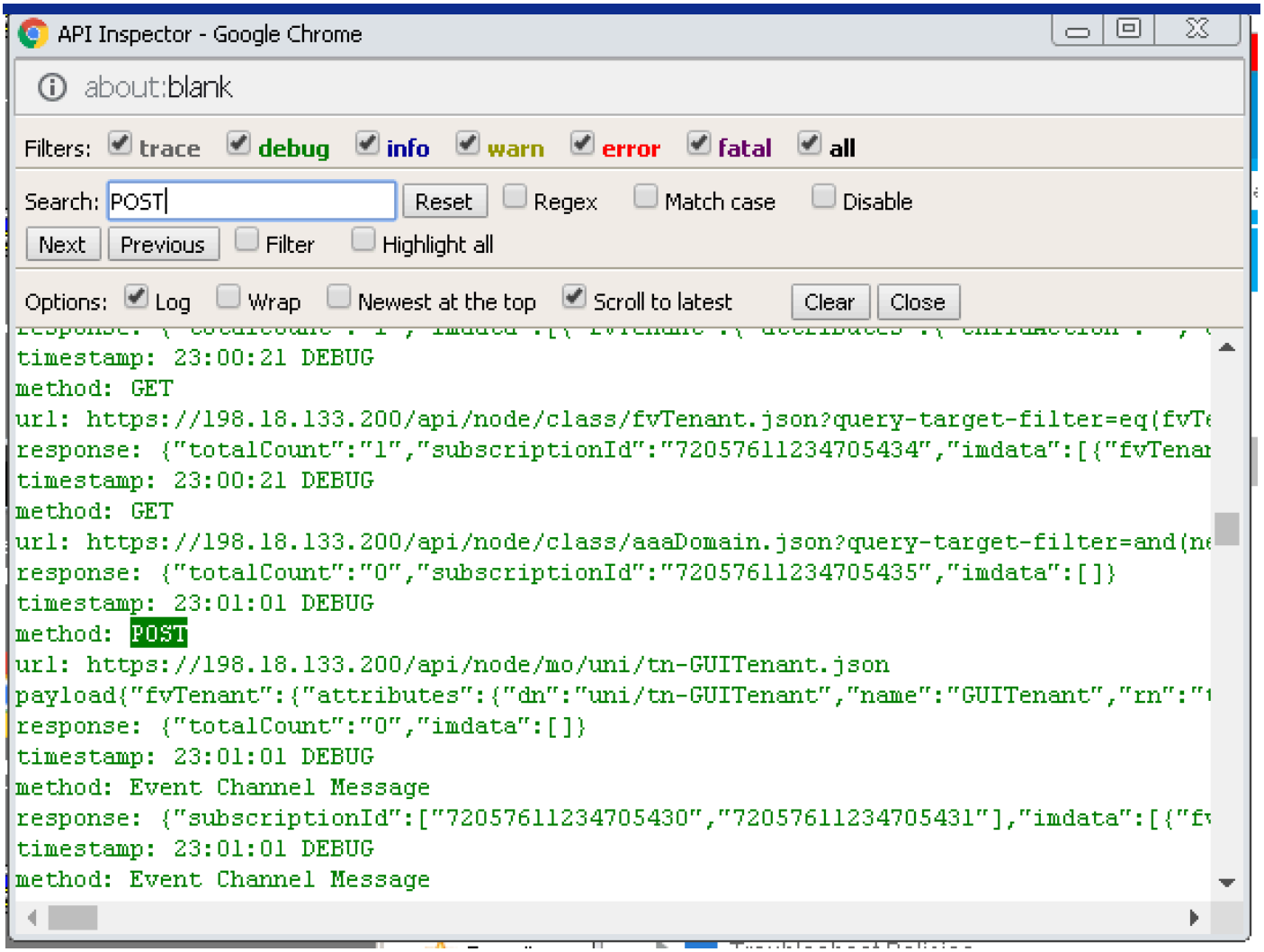


Step 4 Once completed switch to the API inspector window and locate the POST command that was used during the execution of the create tenant function. A couple of things to note in this process. There will be several GET's that will constantly be coming up on the screen. This is a normal function of 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.

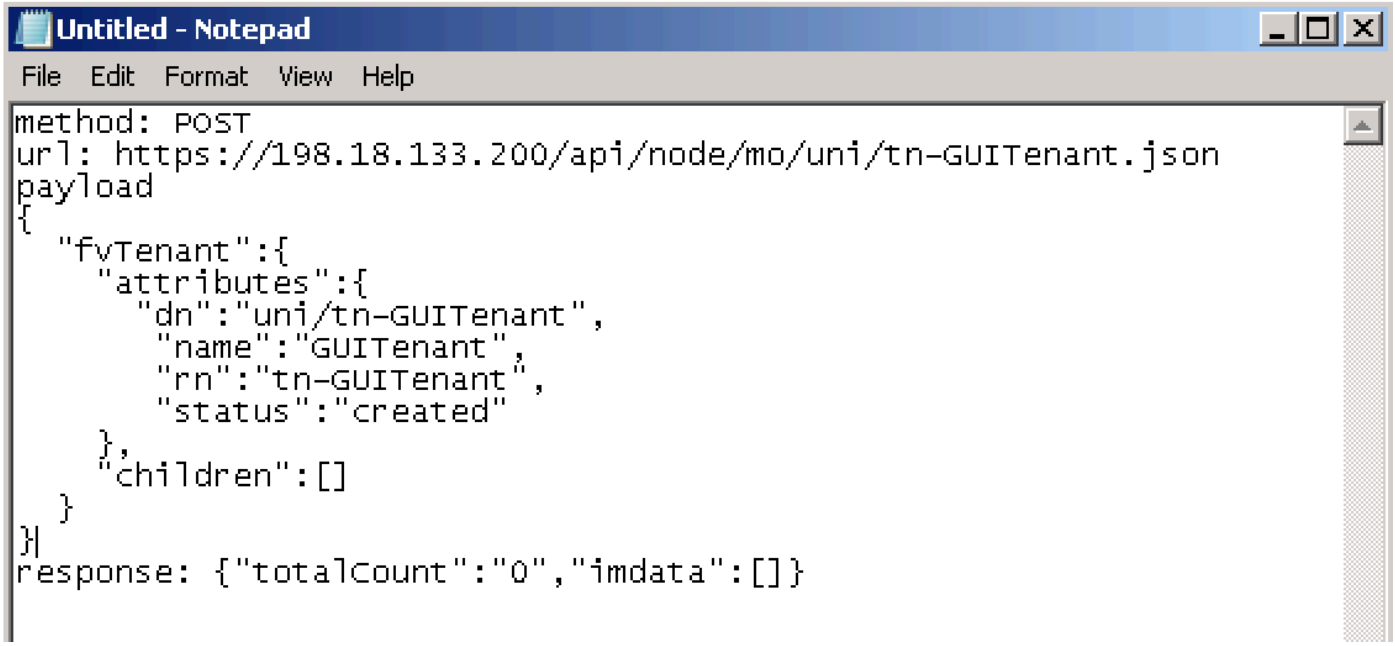
Just like other CRUD functions. When elements in ACI are created a POST method will be used. So, the thing to search on when looking or elements as they are created (or that submit button is clicked throughout the UI) is a POST function.

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.



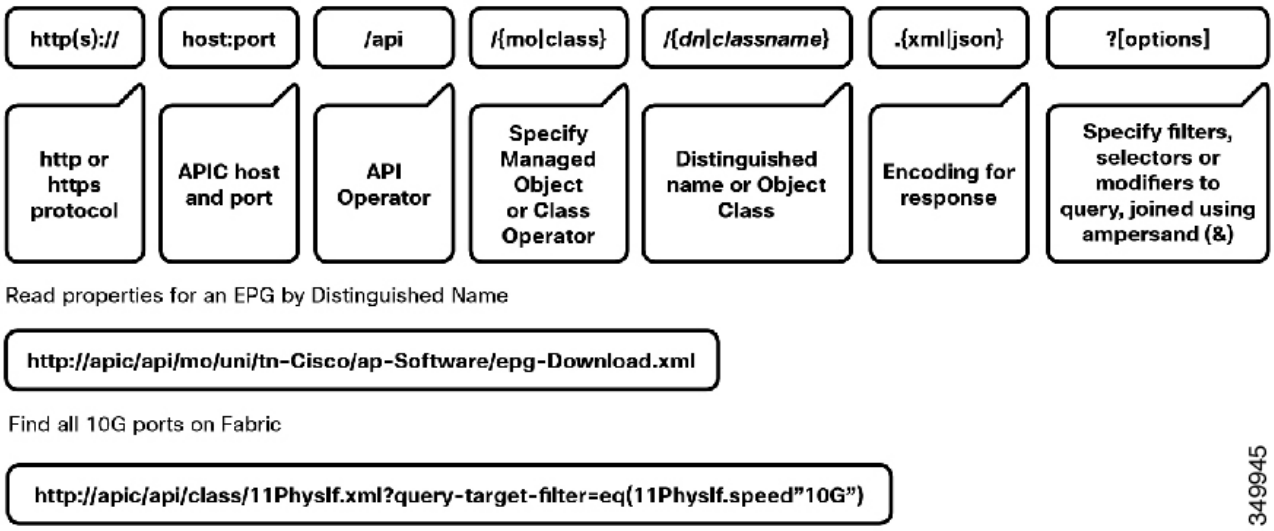


Step 5 Copy and paste the method, URL, and payload to the notepad or text editor for further inspection.

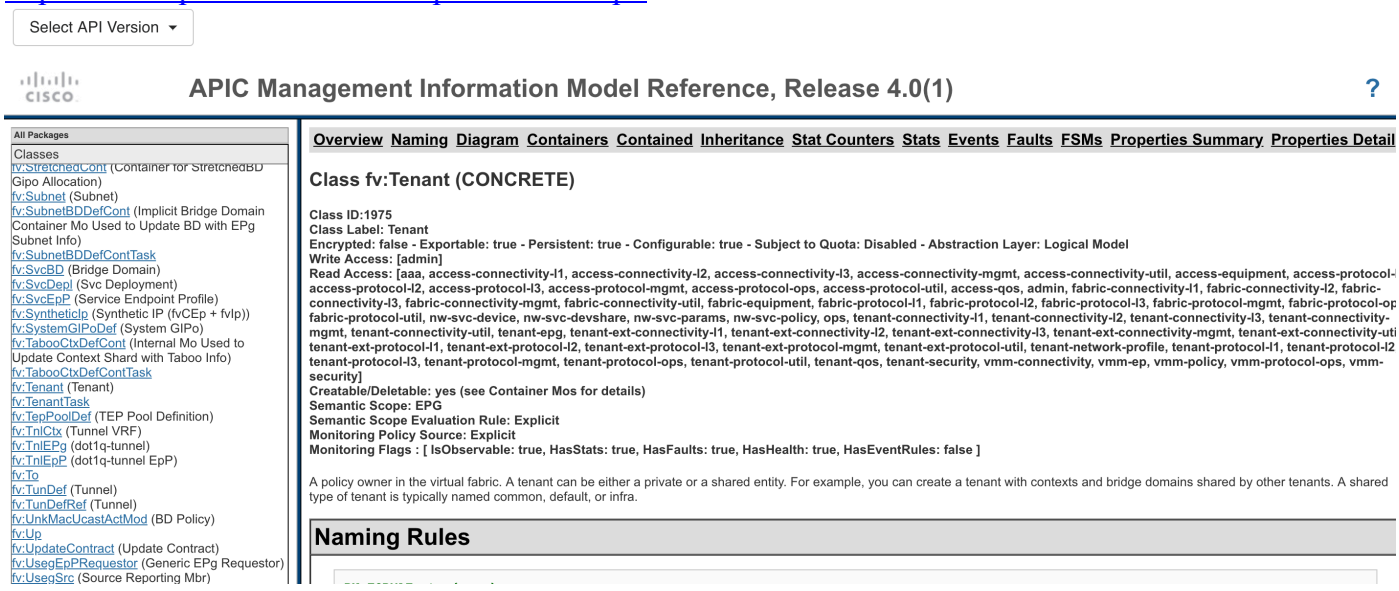


Step 6 Let's break apart the two specific pieces of the REST call:

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.



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/



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

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

Properties Summary

Defined in: fv:Tenant

mo:Annotation string:Basic	**annotation (fv:Tenant:annotation)** NO COMMENTS
naming:Descr string:Basic	**descr (fv:Tenant:descr)** The description of the tenant.
mo:ExtMngdByType scalar:Bitmask32	**extMngdBy (fv:Tenant:extMngdBy)** NO COMMENTS
reference:BinRef	**monPolDn (fv:Tenant:monPolDn)** Monitoring policy attached to this observable object
naming:Name string:Basic	**name (fv:Tenant:name)** Overrides:pol:Obj:name | naming:NamedObject:name The name 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
```

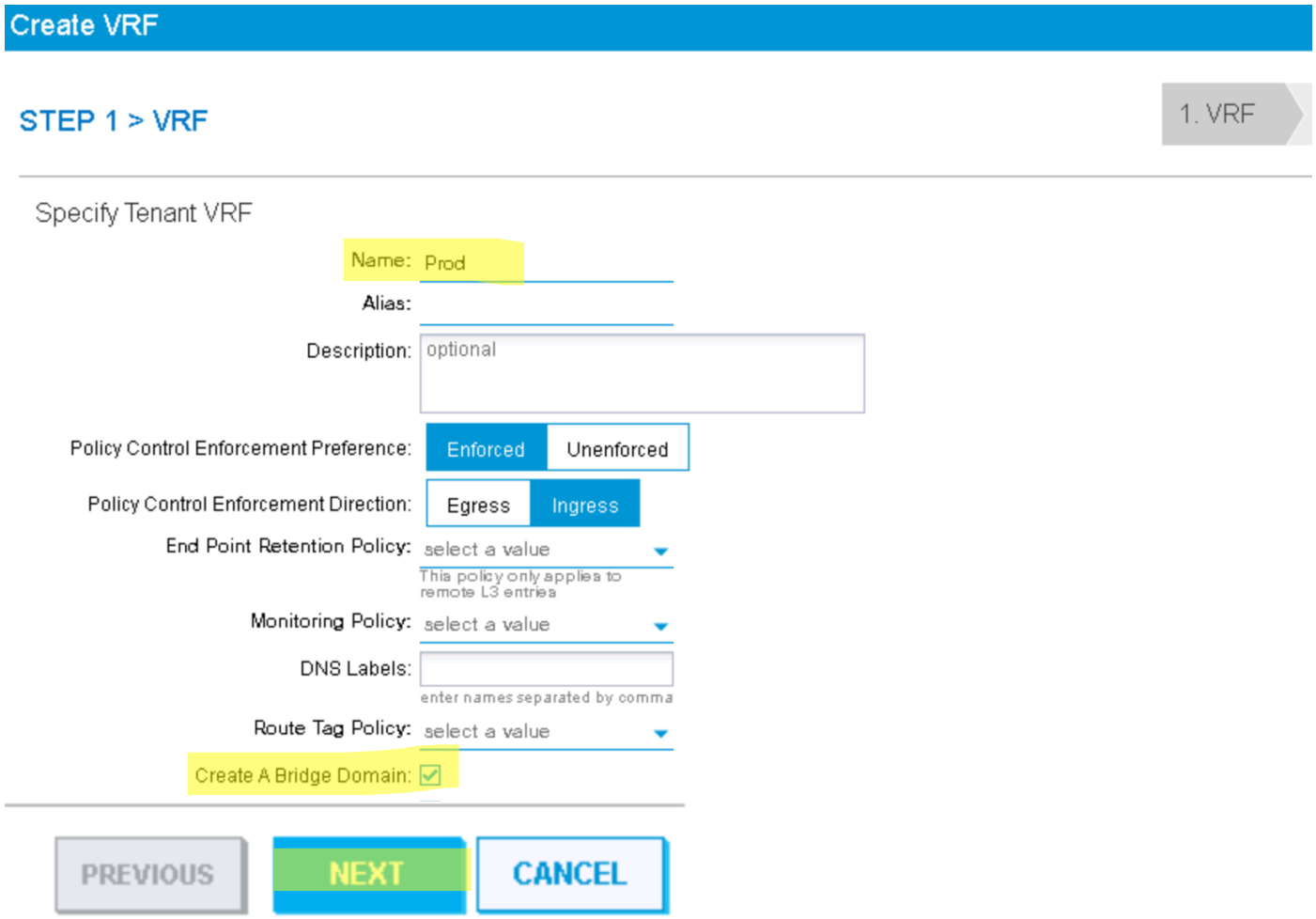
Comments:

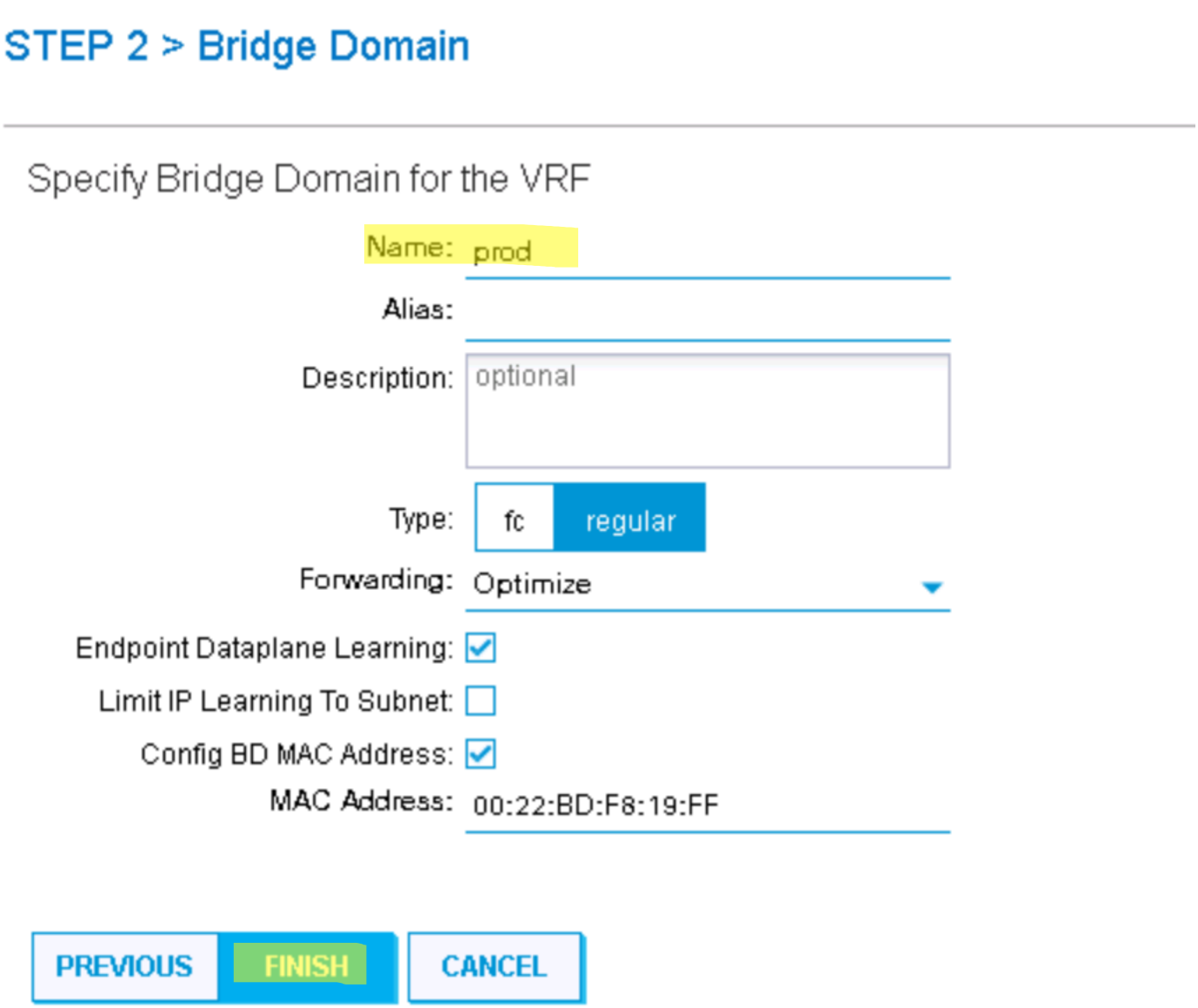
The upgrade status. This property is for internal use only.

Constants

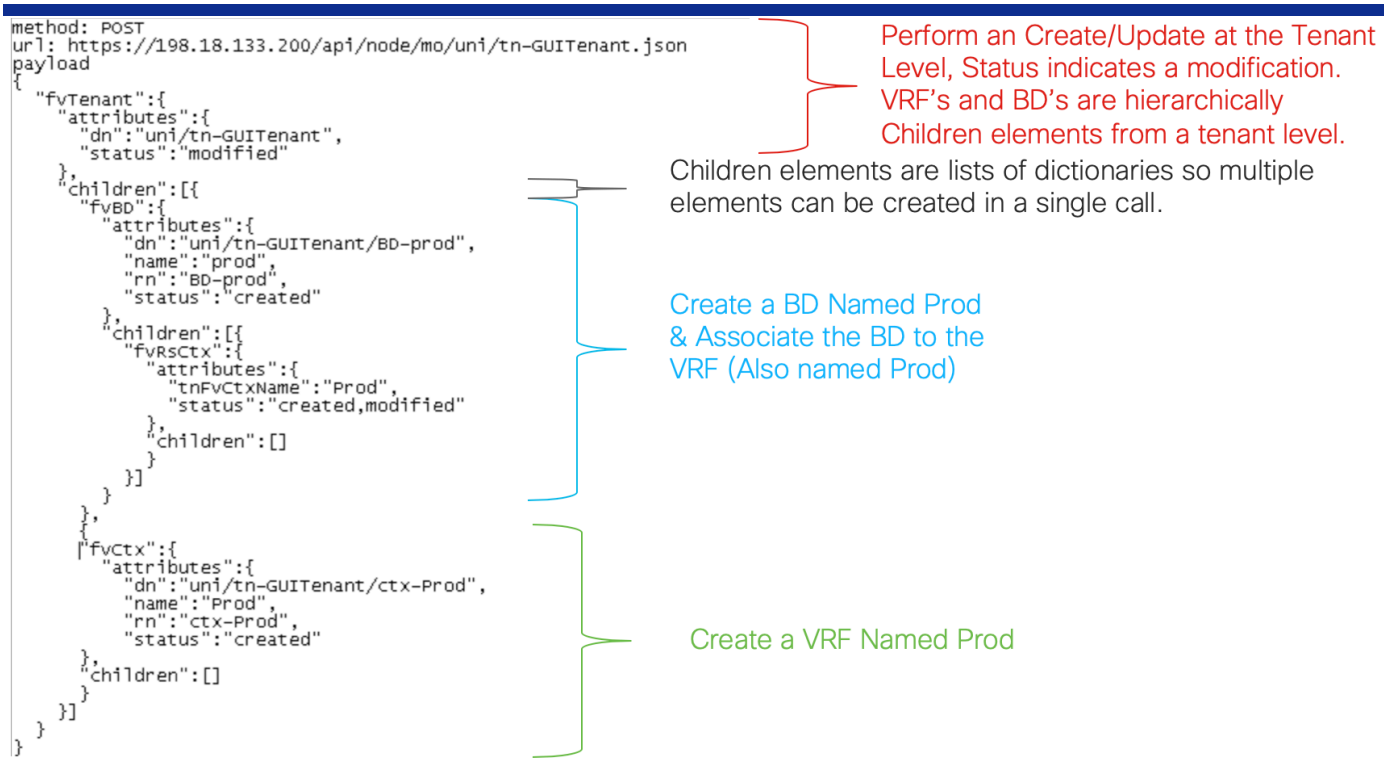
created	2u	created	In a setter method: specifies that an object should be created. An error is returned if the object already exists. In the return value of a setter method: indicates that an object has been created.
modified	4u	modified	In a setter method: specifies that an object should be modified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



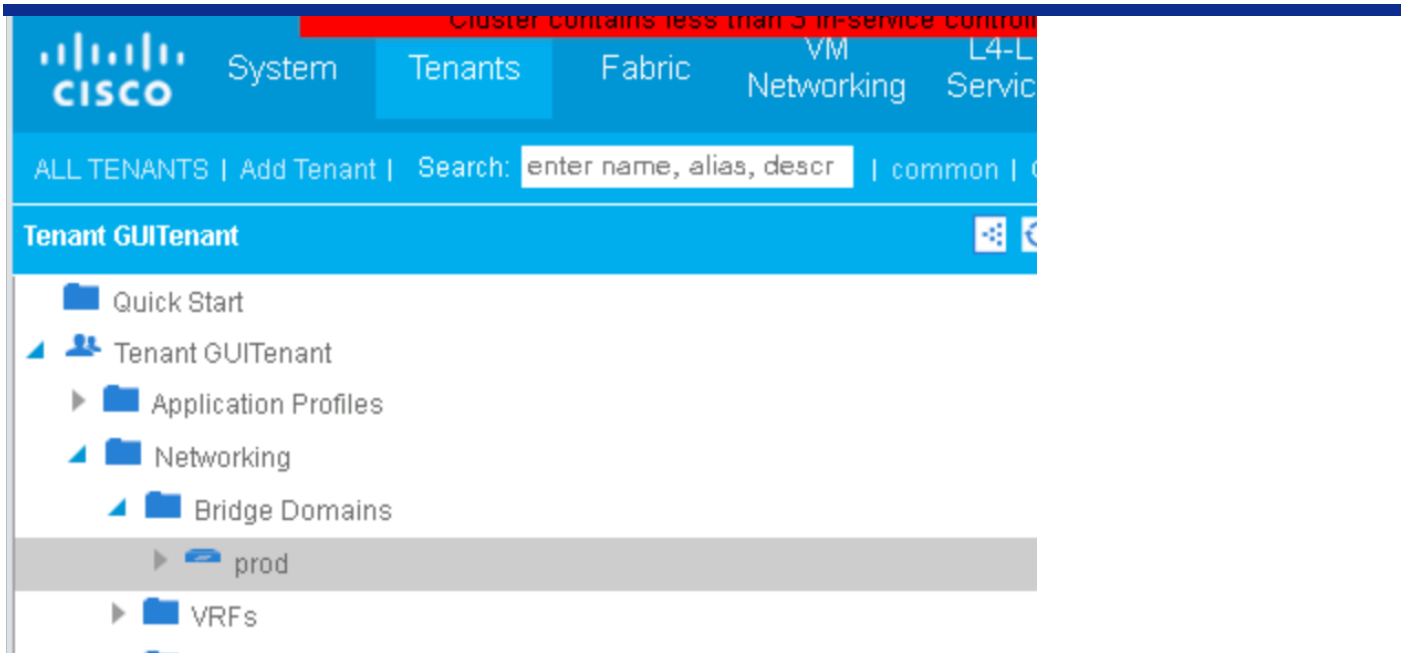


This is an example of a couple operations happening at the same time within a rest call. Locate the post and copy it to a notepad as we did in the previous example. Space out the payload of the rest response to examine the details of the rest call



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 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",
        "epClear":"no",
        "epMoveDetectMode":"",
        "ipLearning":"yes",
        "lcOwn":"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.

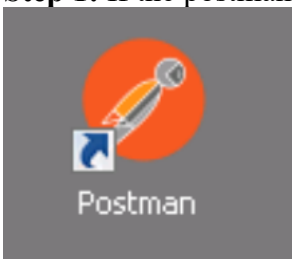
Summary:
We covered the following in this section of the lab.

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

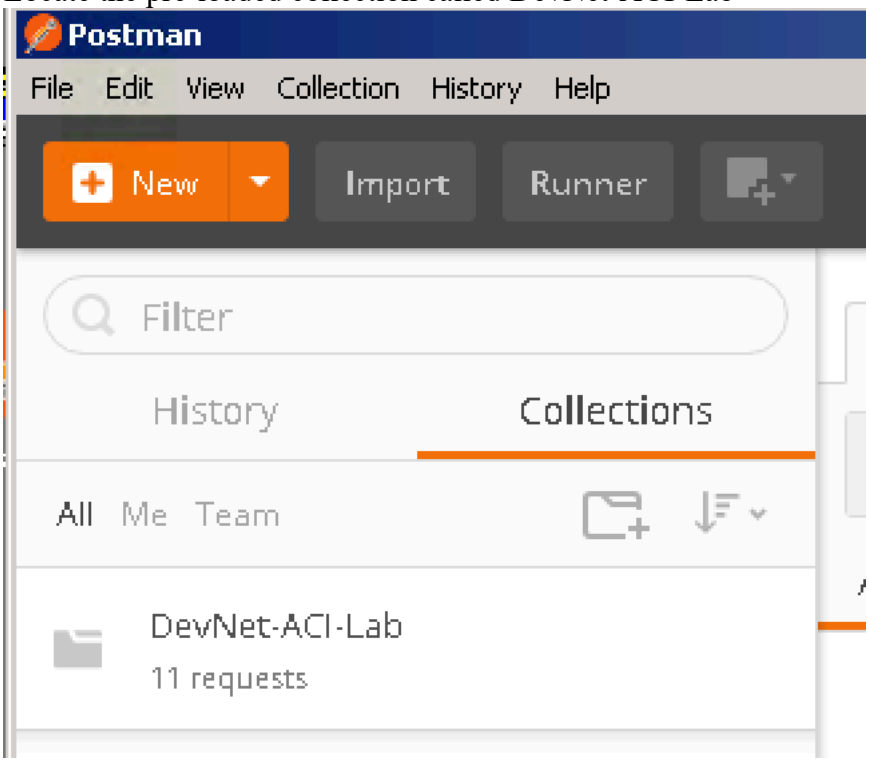
Postman API Interaction

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.

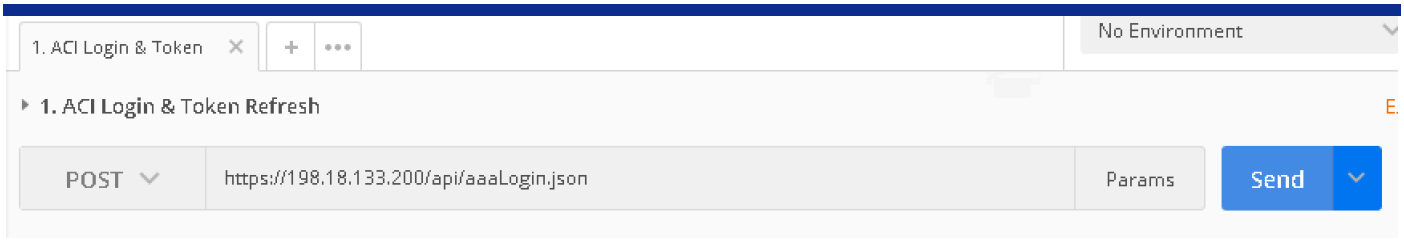
Step 1: If the postman application isn't launched, open it on the Windows VM desktop



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

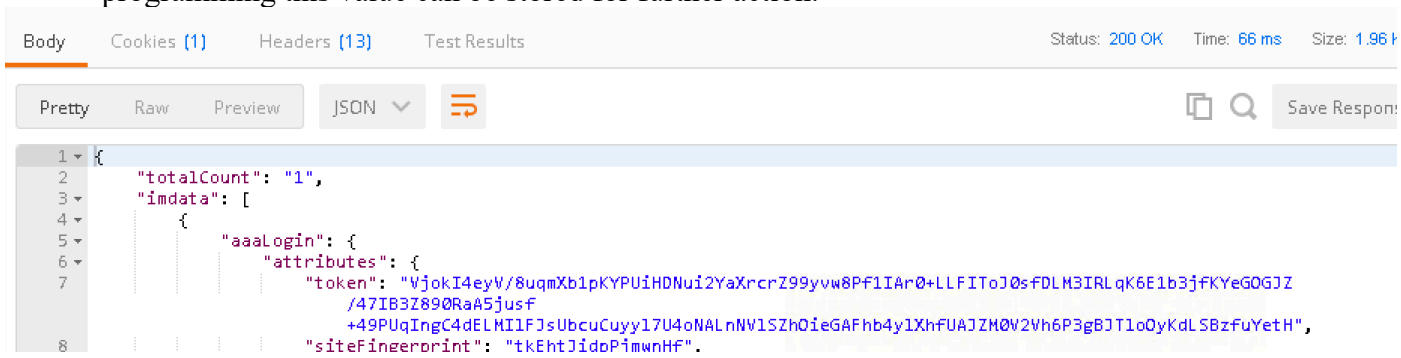


Step 2: Similar to logging into the GUI, to make REST calls the postman application has to log into the APIC environment. Using the collection task 1 and clicking send a Token is received in the body of the response as well as in form of a cookie. See below for details:



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.

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.

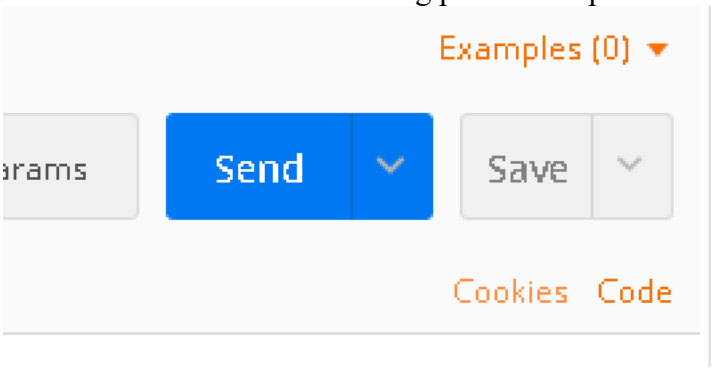


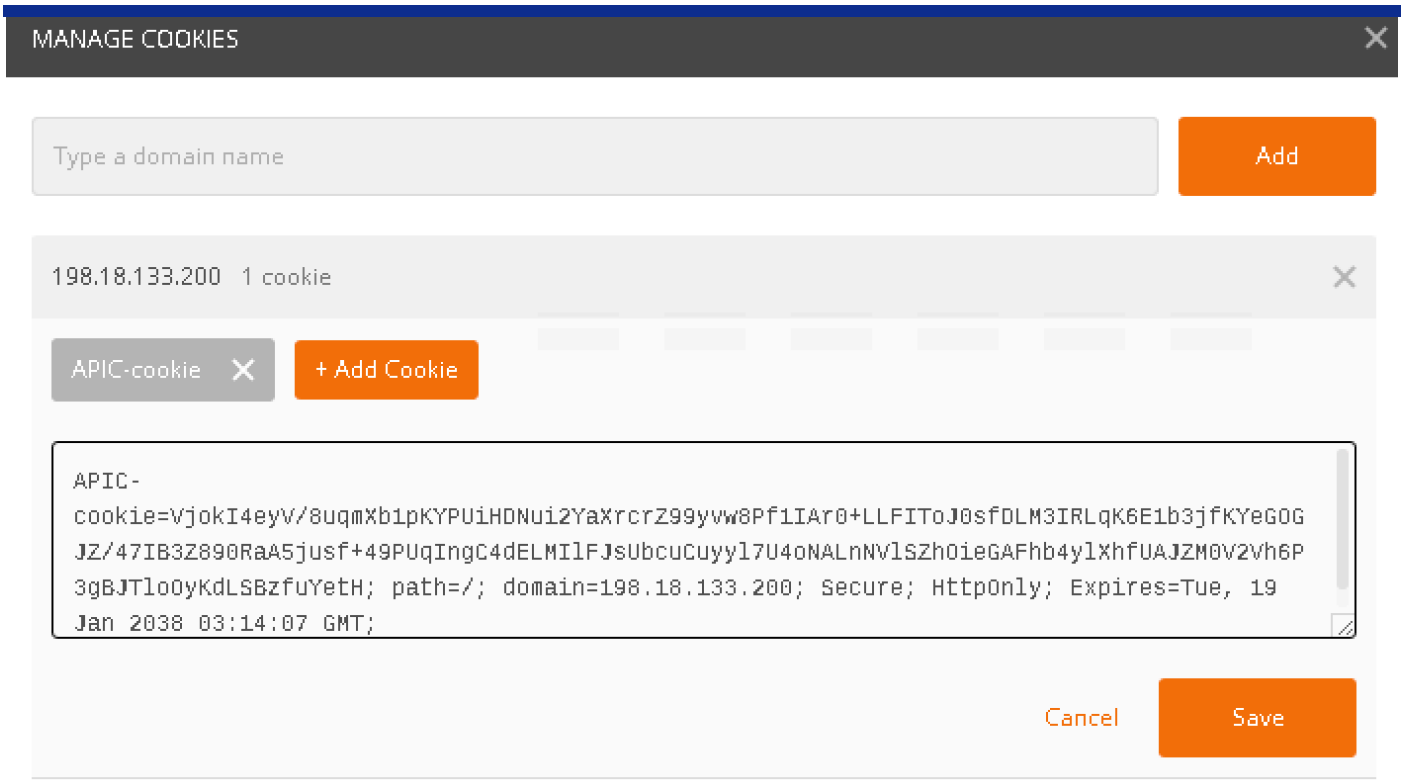
In addition to the response payload a cookie is also received that contains the same value as the Token in the payload.

Body Cookies (1) Headers (13) Test Results Status: 200 OK

Name	Value	Domain	Path	Expires	HTTP
APIC-cookie	VjokI4eyV/8uqmXb1pKYPUiHDNui2YaXrcrZ99yvw8Pf1IAr0+LLFITojOsfDLM3IRLqK6E1b3jfKYeGOGJZ/47IB3Z890RaA5jusf+49PUqIngC4dELMIIFJsUbcuCuyyl7U4oNALnNVISZhOieGAFhb4ylXhfUAJZM0V2Vh6P3gBJTIoOyKdLSBzfuYetH	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.

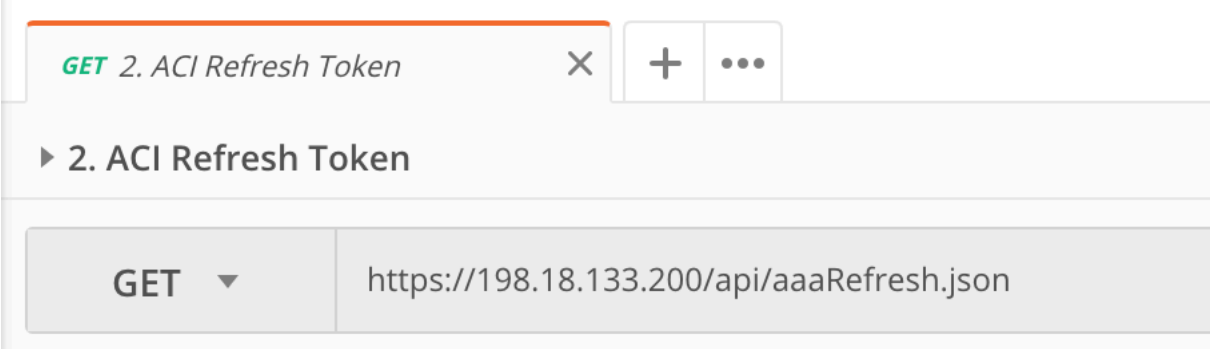




Step 3: Similar to a GUI login the credentials will expire after a period of time so will a REST token. That time data is contained in the response body of the initial login request. In order to keep the session going the token must be refreshed within the interval. If that doesn't happen the login step 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
                    +49PUqIngC4dELMIlFJsUbcuCuyyl7U4oNALnNVlSZhOieGAFhb
                "siteFingerprint": "tkEhtJidpPjmwnHf",
                "refreshTimeoutSeconds": "600",
                "maximumLifetimeSeconds": "86400",
                "guiIdleTimeoutSeconds": "1200",
```

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.

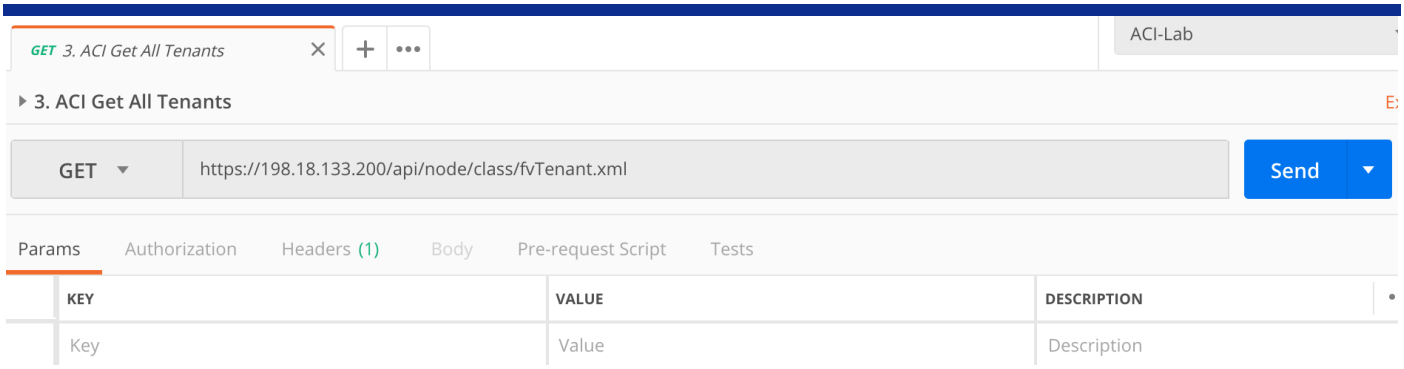



```
{
    "totalCount": "1",
    "imdata": [
        {
            "aaaLogin": {
                "attributes": {
                    "token": "S41
                        /zDpHlhgNmTJTgpUxPg47xDGnlyAwcv8sxhNt9zf3Tn2NtA1SWS8kk30jw3IYwoAyxfdUkiUWw48Bi4Qd9mxz246GZeS
                        rrmUsvL5tNUzR/TPGjp9kFEXJY7dq1esRIqc7uLMsyh1OiBFQzy2cTIBvLSeGhoFsv6qPaQBnB7K30T2PwfEyI+3I",
                    "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)"
                }
            }
        }
    ]
}
```

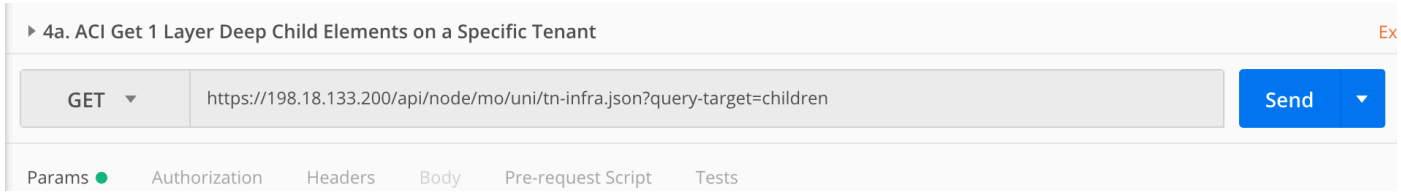

Step 4: Now that we have an active token let’s explore some the of the same actions we did with the GUI and the API explorer. Click and run step 3 which is the ACI Get All Tenants. A couple of thigs to note on the API call itself. The path is looking at a type class under the root node called fvTenant. 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.



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.

```
<?xml version="1.0" encoding="UTF-8"?>
<imdata totalCount="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"/>
    <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"/>
    <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"/>
</imdata>
```

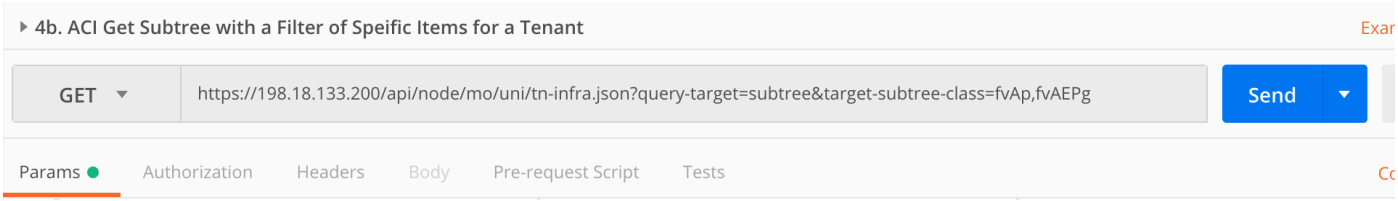
Step 5: Lets dig a bit deeper into one of the tenants and get all the child elements of that tenant. Load the step 4a from the collection. Note the path has changed from looking at a specific class to moving into the managed objects and looking into a distinguished name of the infra tenant (from the 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.

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 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.

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.

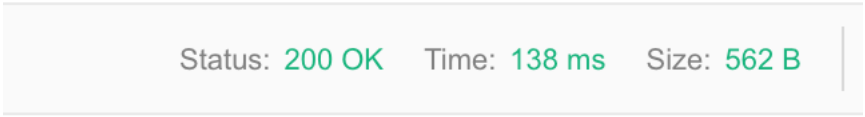
Note the resulting data, where we have BD information, QoS policies, and far more than we did when we filtered in the previous step to just the application profiles and EPGs.

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:

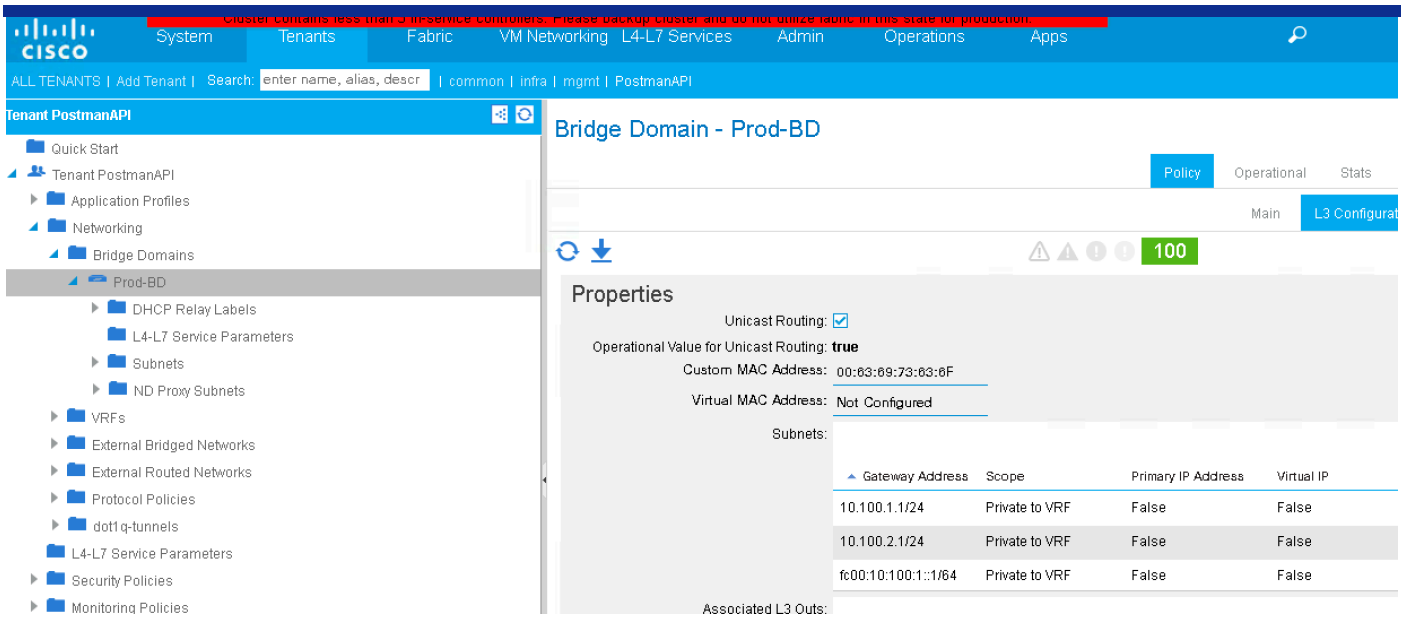
```
<fvTenant descr="Postman API Networking Environemnt" dn="uni/tn-PostmanAPI" name="PostmanAPI" ownerKey="" ownerTag=""> #
    Tenant
  <fvCtx descr="" knwMcastAct="permit" name="Prod-VRF" ownerKey="" ownerTag="" pcEnfPref="enforced"> #VRF
  </fvCtx>
  <ndIfPol name="PostmanAPI-IF-Policy" ctrl=""  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.

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



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.

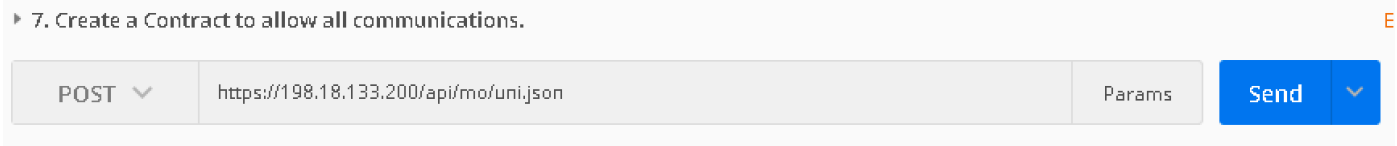


Step 9: The next REST example will again use the same path however the content in the payload will we switched from XML to JSON, so instead of using <path>/uni.xml we will use <path>/uni.json in the POST method. If we examine the body of the JSON we are creating a new Application profile and under the 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.

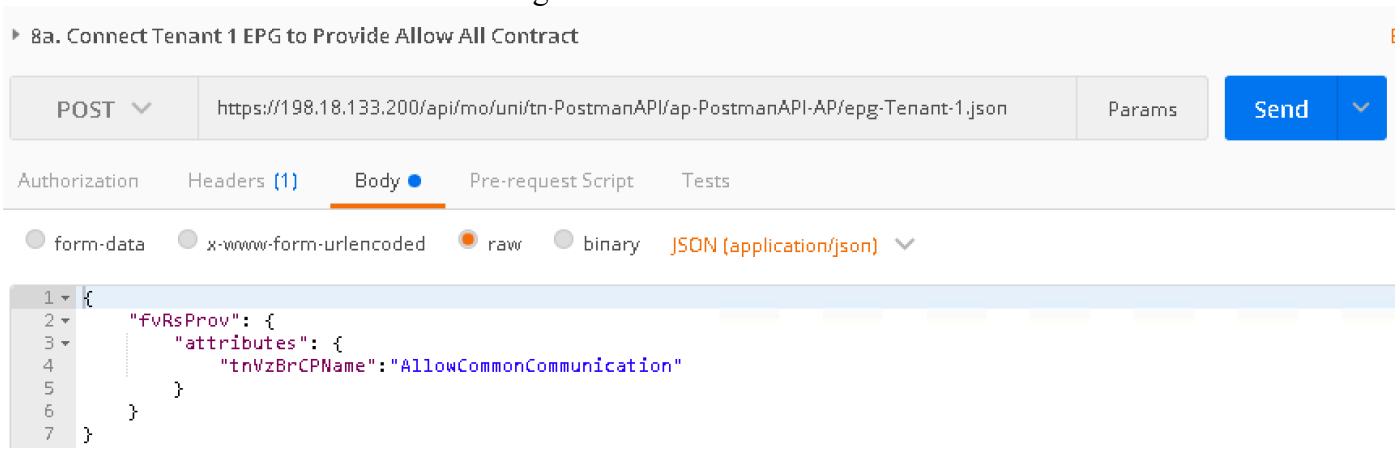
```
{
    "fvAp": {
            "attributes": {
              "dn":"uni/tn-PostmanAPI/ap-PostmanAPI-AP",
              "name":"PostmanAPI-AP",
              "rn":"ap-PostmanAPI-AP"
            },
            "children" : [{
                "fvAEPg": {
                    "attributes": {
                        "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 in indication that the operation has completed successfully.

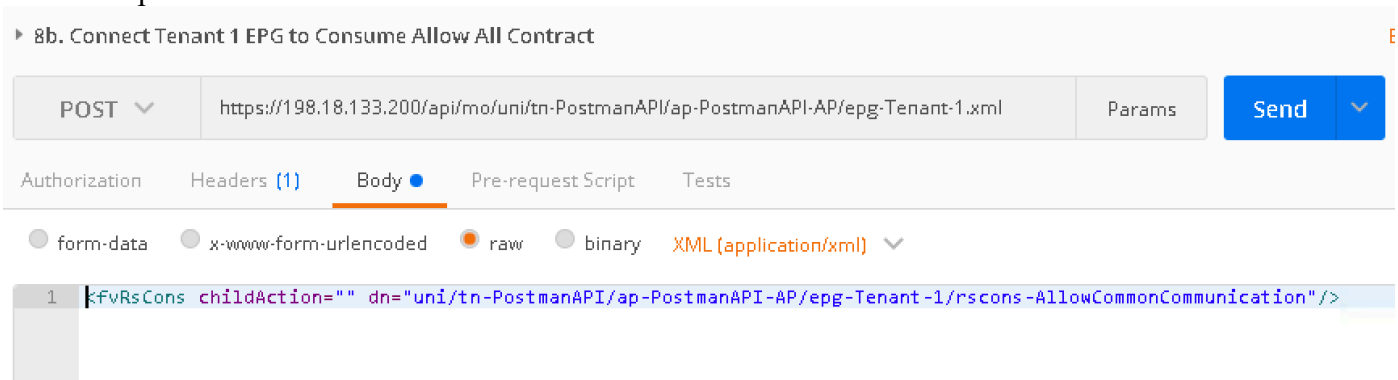
Step 10: Postman tasks 7, 8a, and 8b all have to do with contract creation and provider/consumer association to one of the EPGs. Task 7 is a similar in code to the previous two examples. The contract in question called AllowCommonCommunication has a single subject with the default filter.. effectively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



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 to information that needs to go into the body/payload of the transaction, as well as the format of the message in JSON format.



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



Inspect the GUI and look at the new tenant and various constructs that where 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.

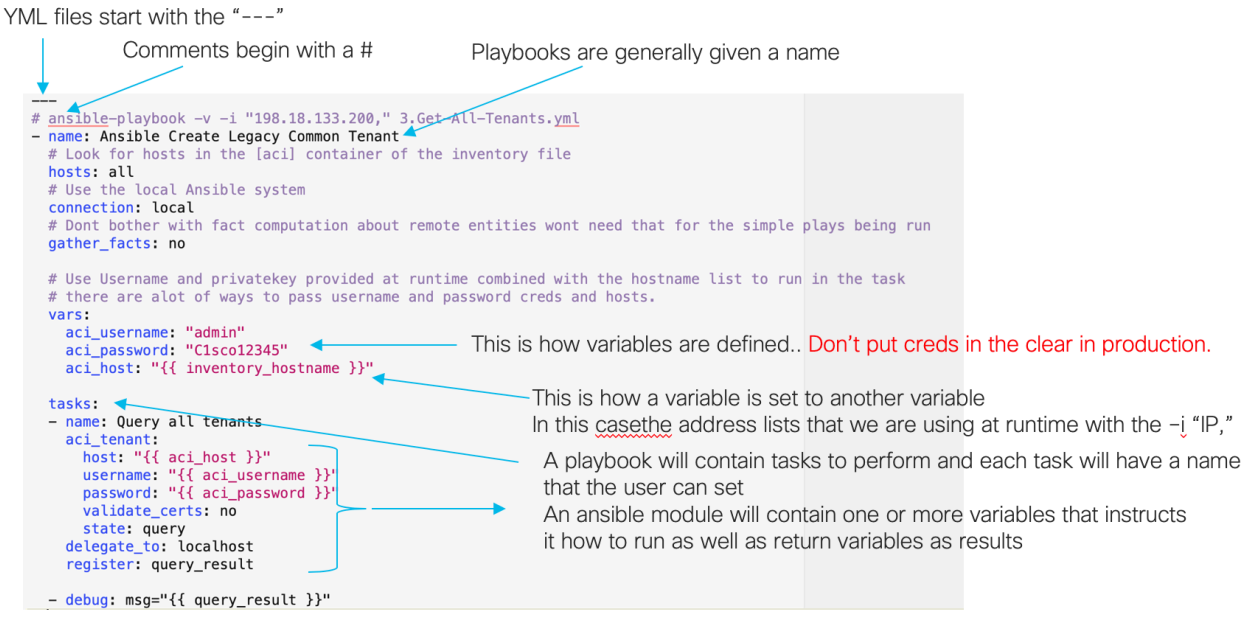
Ansible Programmability with ACI

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.



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

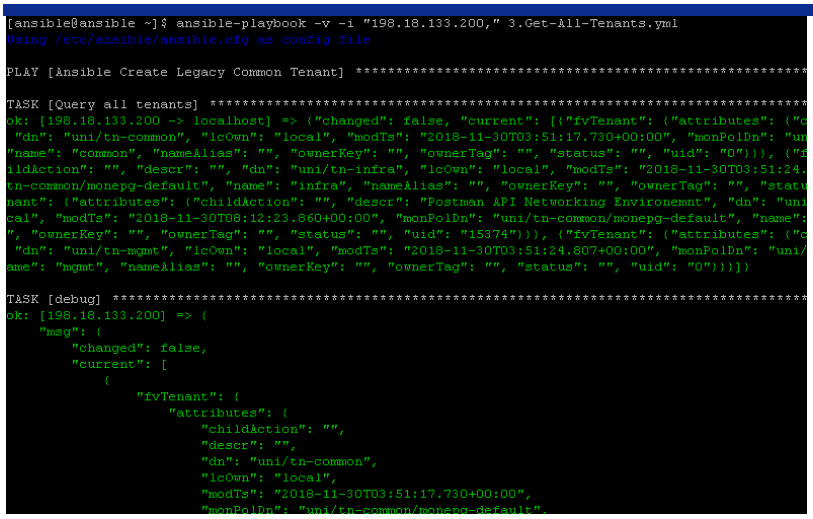
i along with a list of IPs can be provided at runtime to remove the need for an inventory file to be present. If we wanted to use the same playbook on more than one device at a time all that would need to happen is to amend the IP list. Ex. -i "198.18.133.200,198.18.200.199,hostfqdn.example.com,". Notice 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

```
    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.



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)

Step 4: Instead of using a predefined ansible module we can submit direct REST calls similar to what we did in postman. This allows is 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.

```
    use_proxy: 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
          </fvSubnet>
        </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

#     - name: Output Result
#       debug:
#         msg: "{{ bd_query_result.current[0].fvTenant.children[0].fvBD.attributes.name }}" # Can Parse Jason using .<a

    - 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-Prodt"
           },
           "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 }}"
```

Ansible Bridge Domain Module

Specify the tenant to get the information on

Save the result

A Debug I was using to play with the parsing now commented out

Using the name of the BD found in the query result as a variable in the JSON body of the creation request

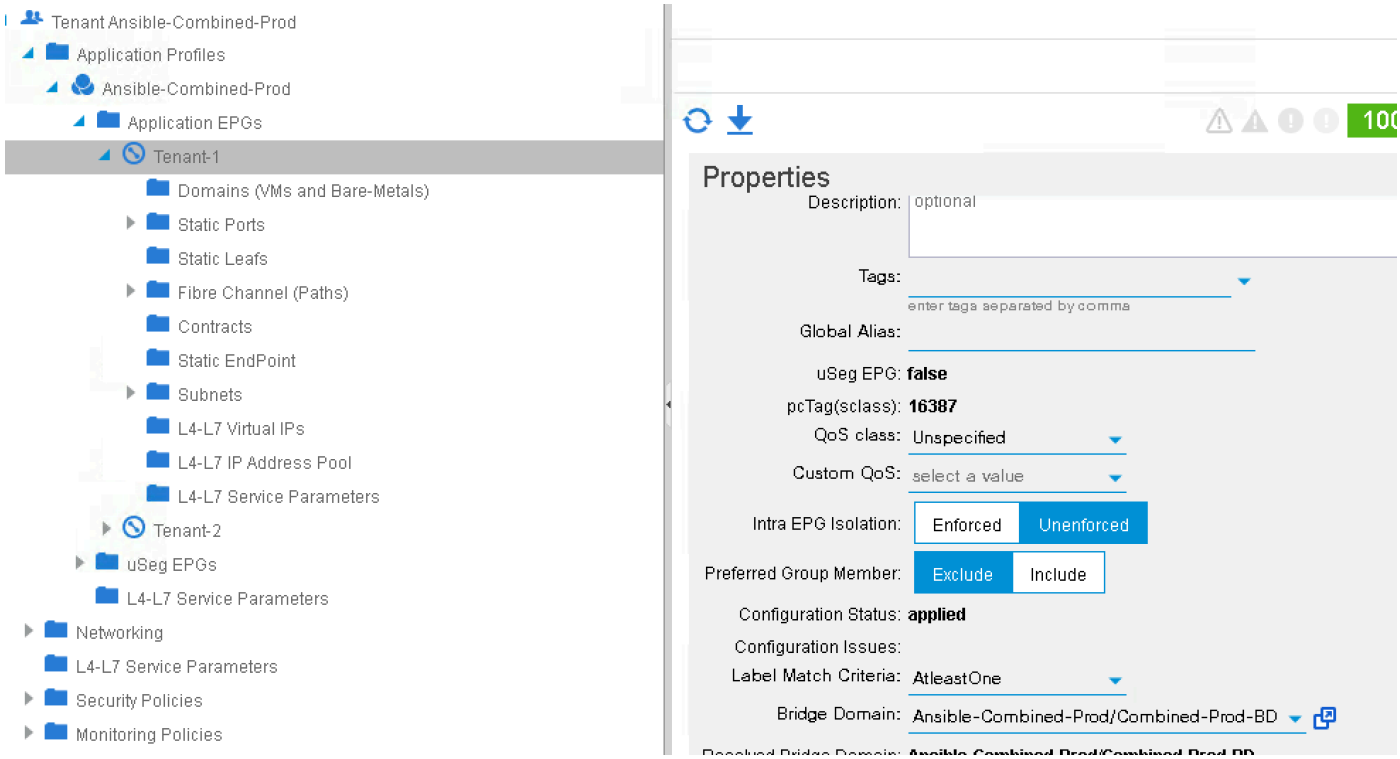
When this runs in ansible the task to query the BD information returns an OK (as nothing is modified in a GET) and the subsequent task where we are configuring can be used.

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



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

state	**Choices:** • absent • **present** ← • query	Use `present` or `absent` for adding or removing. Use `query` for listing an object or multiple objects.

```yaml
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.

```
[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: <u>LAB CLEANUP</u> – The 8th 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.**