cisco Live!
Amsterdam | February 6–10, 2023

LTROPS-1964

Test Automation for everyone using CXTA

Sofia Athanasiou – Customer Success Specialist
Maciej Godlewski – Software Consulting Engineer

Overview

Nowadays with software-based services all over the network infrastructure, there is the need for a fast and precise way to ensure the functionality of the network. Using CXTA to automate the process of testing among the network, the user is saving a lot of time.

CXTA (Customer Experience Test Automation) is the new era of test automation in Cisco. It uses the opensource Robot Framework and provides numerous custom keywords to test the functionality of several software-oriented Cisco Products such as NSO, ACI and allows also testing in devices directly.

Main advantage is that it is written using keywords, so it is easy to understand and be developed by non-programmers. So now with CXTA everyone can write their own test cases!

Learning Objectives

Upon completion of this lab, you will be able to:

- Create a testbed that contains the information of the devices,
- Create and run test cases towards ACI, NSO and devices,
- Construct test case scenarios and push configuration using CXTA.

The lab has 3 main exercises:

- **Exercise 1**: Get familiar with the environment and validate that CXTA is working properly
- **Exercise 2**: Test cases for NX-OS and IOS-XR devices
- **Exercise 3**: Test cases towards ACI using REST APIs and configuration changes
- **Exercise 4**: Test cases towards NSO, including using CXTA to provision a service as well as negative test case scenarios

Disclaimer

This training document is to familiarize with Customer Experience Test Automation (CXTA). Although the lab design and configuration examples could be used as a reference, it's not a real design, thus not all recommended features are used, or enabled optimally. For design-related questions please contact your representative at Cisco, or a Cisco partner.

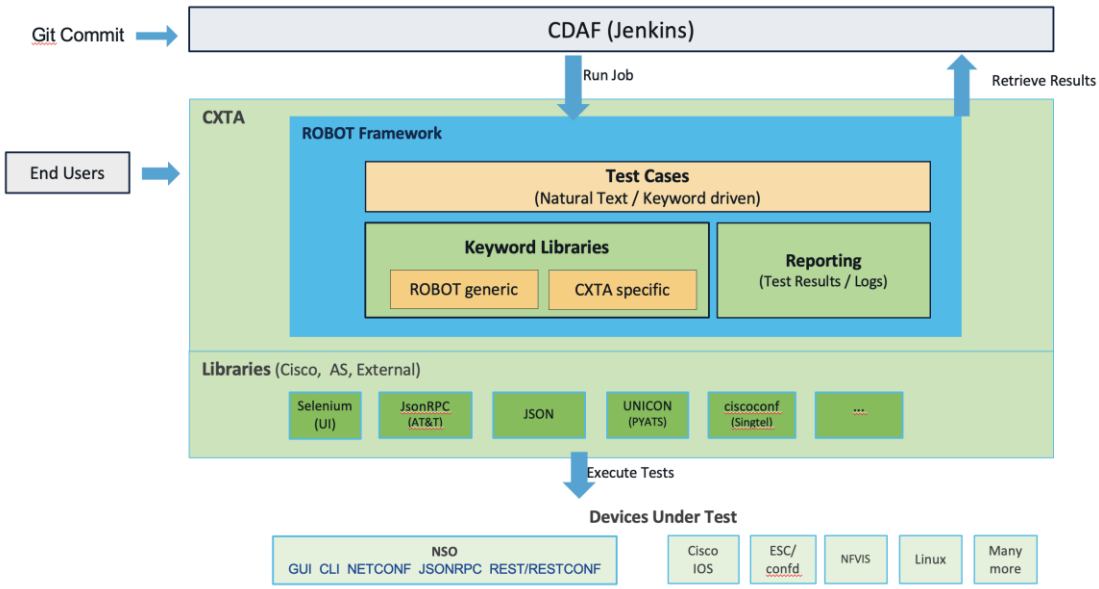
CXTA Overview

Cisco® CXTA (**CX T**est **A**utomation, pronounced *Siesta*) is a set of Cisco network test automation libraries built around Robot Framework. It was created originally for internal purposes, in order to support faster delivery of Network Service Orchestrator (NSO) projects. Fast enough, after observing the advantages, there was the realization that this could be a great ally to customers as well.

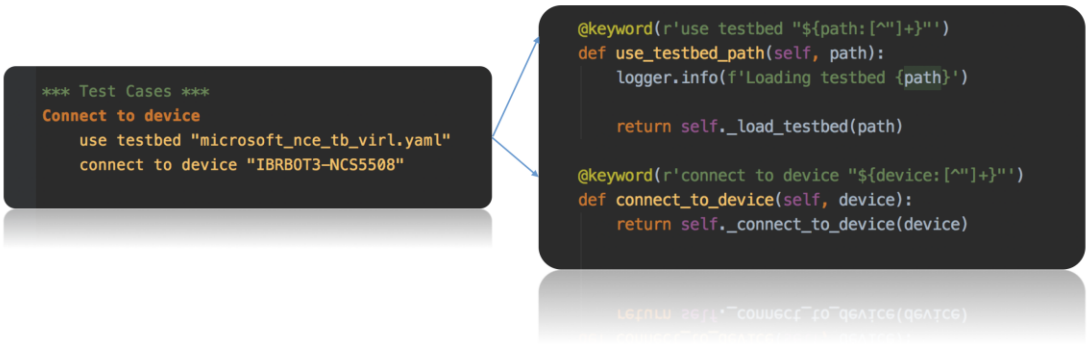
CXTA provides a large variety of keywords which enables the interaction and test with a broad number of devices and systems, including, but not limited to

- Cisco network devices (IOS, IOS-XR, NXOS and others)
- REST/RESTCONF APIs of solutions components like NSO, BPA, UCSD etc.
- Generic SSH CLI interaction with any system
- Keywords for Cisco solutions like VTS or SDA or ACI
- User Interface testing (via Selenium), including pre-built keywords for NSO, BPA, Prime Service Catalog

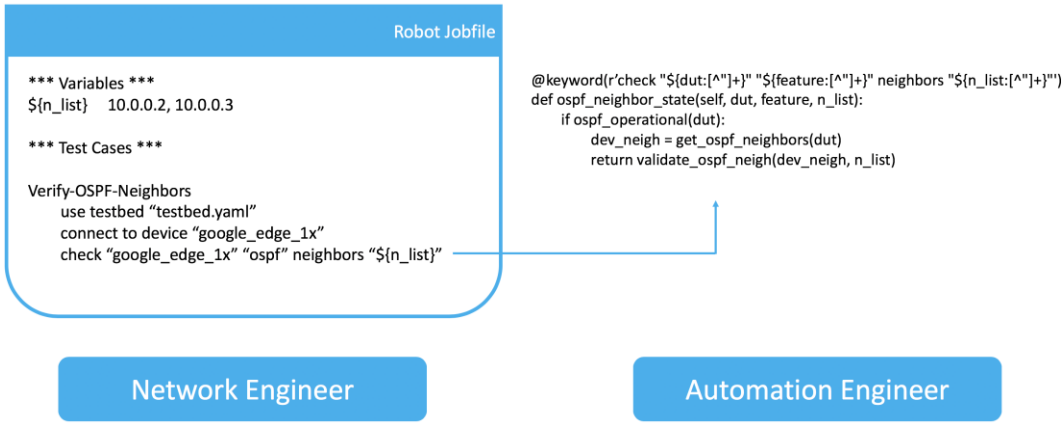
Below there is the architecture of CXTA. One can use CXTA as part of a Continuous Integration/Continuous Delivery (CI/CD) part or as a standalone runner.



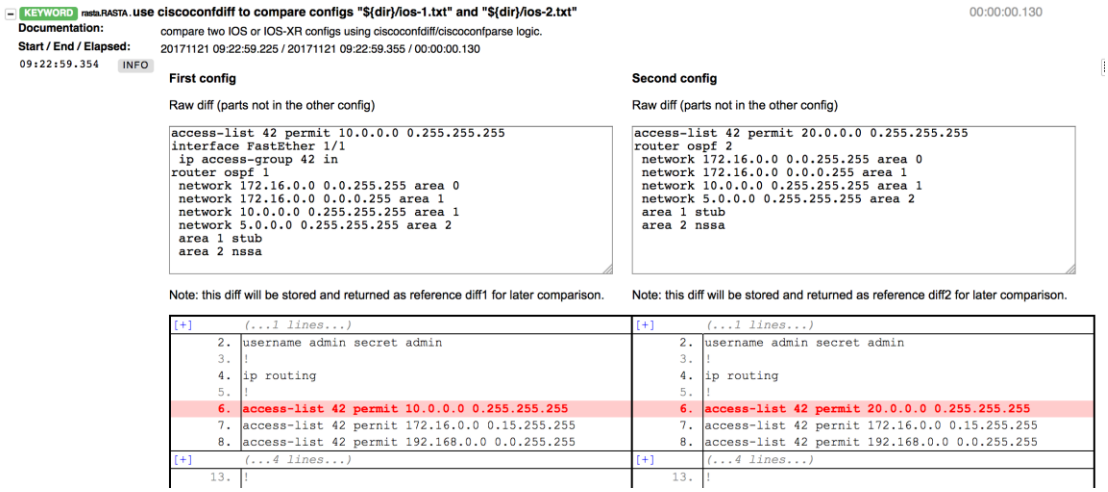
The keywords that are used are actually annotations of Python and in the background after all Python will contribute in order for the test cases to be executed.



There are two main advantages of using CXTA. The first advantage is the fact that it is written using keywords. This provides the flexibility and the ability of test automation to everyone, without being in the need of learning a programming language such as Python, a network engineers can create their own test automation. In the photo below, there is the same use case, on the left as it is described by CXTA scenario and on the right, as it would have been if we would like to create the same keyword in Python.



The second advantage of CXTA comes in the reporting. After running the Test Cases, several files will be automatically created, among them, the log.html file. There is for example someone is running a Test Case so that they confirm if their configuration is compliant or not based on ISO standards, then it is clearly stated out from the test results, where the expected configuration does not match with the actual devices' configuration.



For example, here, the expected configuration for the Access List (ACL) would be:

```
access-list 42 permit 10.0.0.0 255.255.255.255
```

but instead in the device the configuration is:

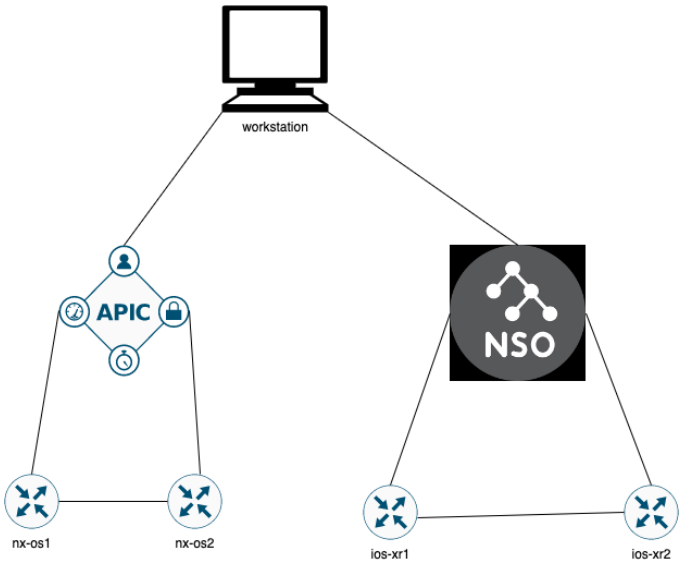
```
access-list 42 permit 20.0.0.0 255.255.255.255
```

So, without having to log in to every different device, the user could have the differences of the actual devices' configuration and the expected configuration in a nice reporting way.

Lab Introduction

Topology

CXTA and CXTA documentation are installed and running as a Docker containers in an Ubuntu workstation. CXTA container is able to communicate with APIC that has two NX-OS devices attached and with those devices as well. Also has communication with NSO and its corresponding onboarded IOS-XR devices.



Lab environment

The lab runs inside dCloud in an Ubuntu machine.

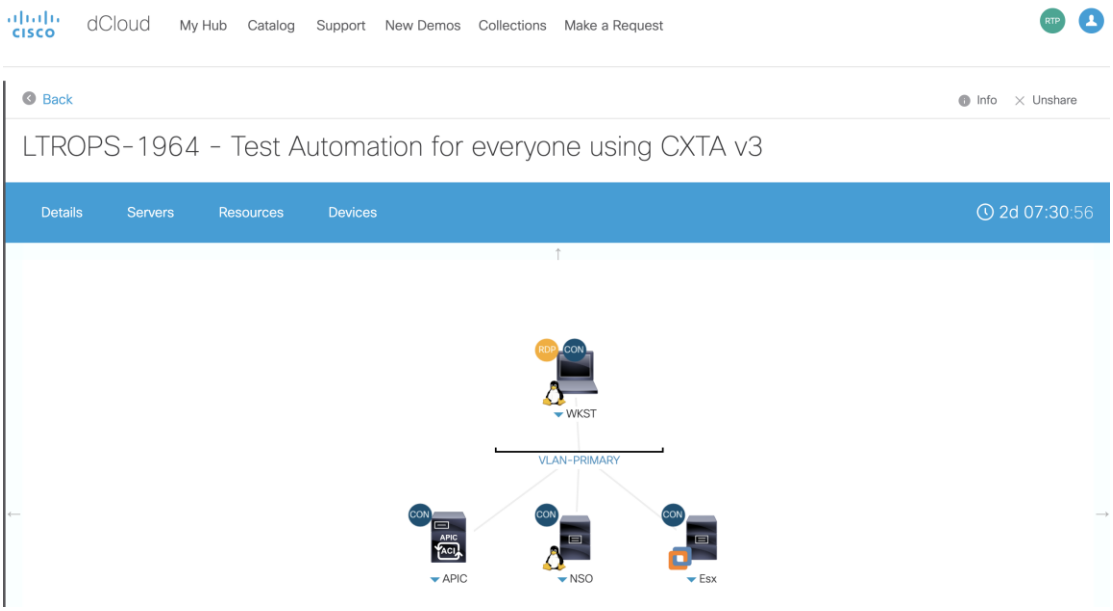
Ways of development possible for this lab.

1. (preferred) Visual Studio Code. You can find a shortcut in the desktop to this application. When you start it, you will be connected to Ubuntu Terminal, and you will be able to connect and run the Docker Container as well as edit your code.
2. Using vim inside the cxta container.

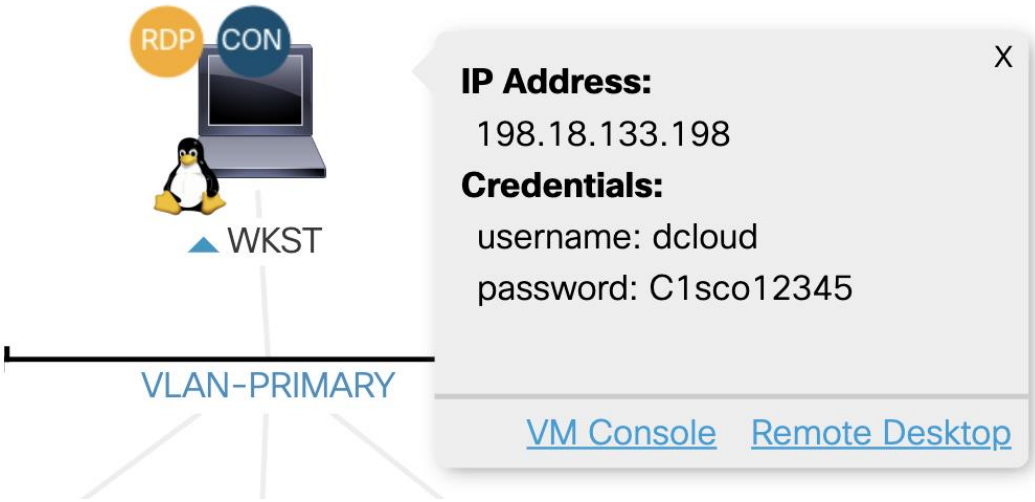
Lab Introduction and Verification

To prepare your lab a few steps need to be executed.

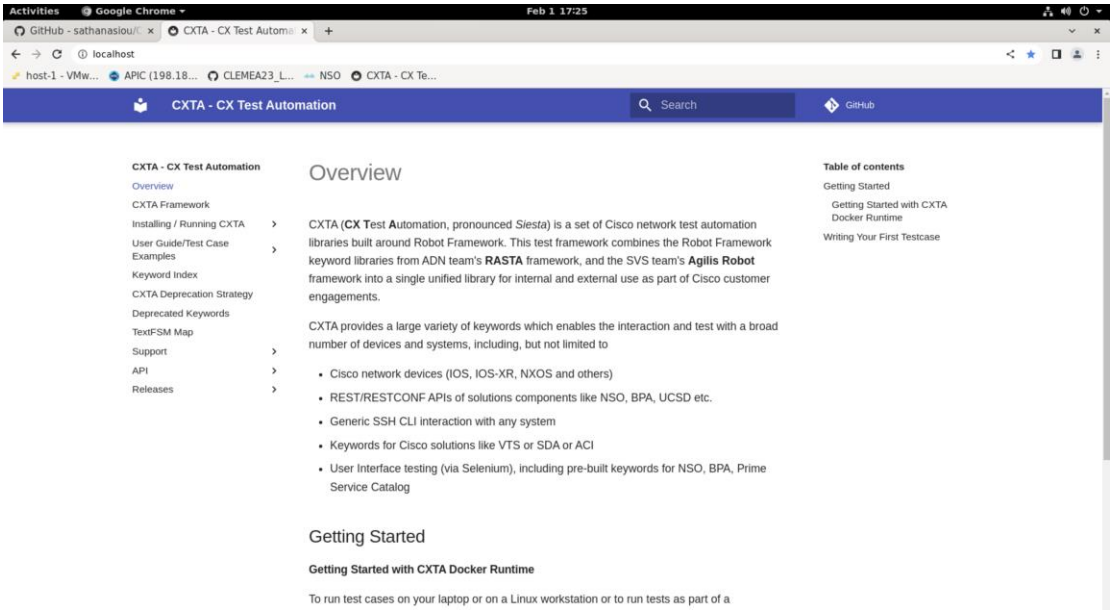
1. Use Dcloud interface to see the topology



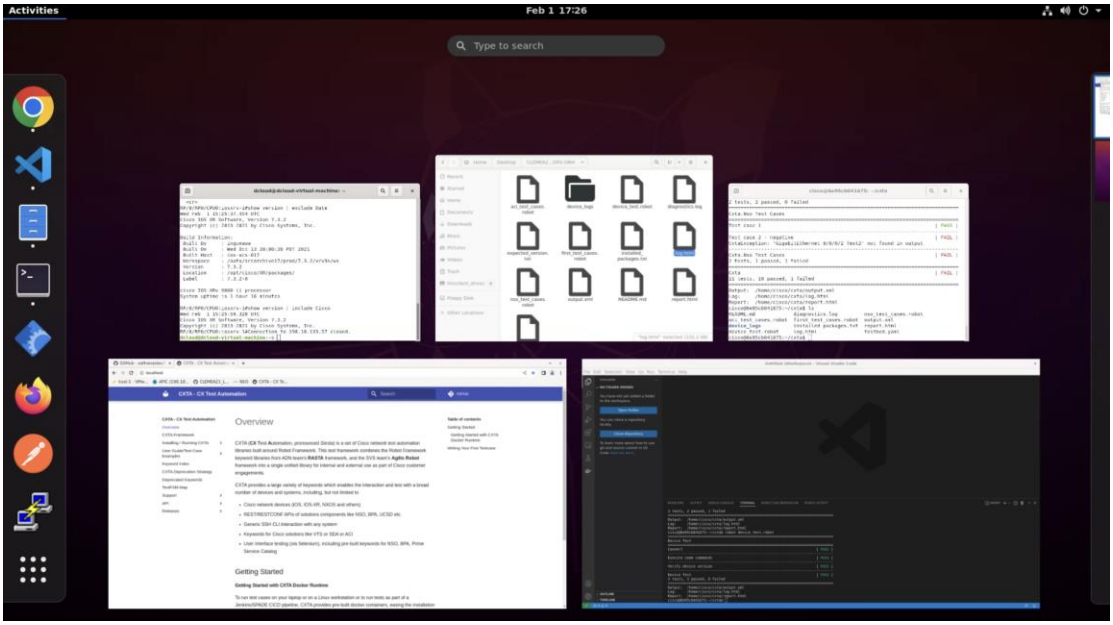
2. Open WKST object



3. Select Remote Desktop to access Workstation interface

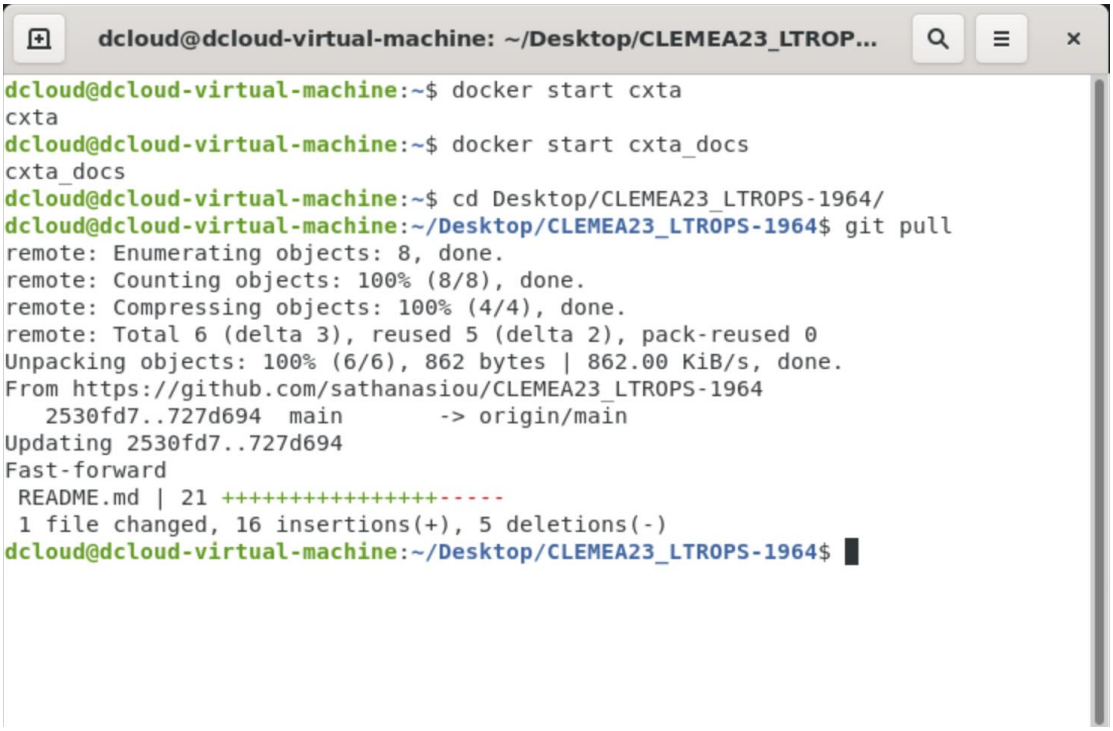


4. Go to 'Activities' in top right corner and open Console or Visual Studio Code to use built-in terminal:

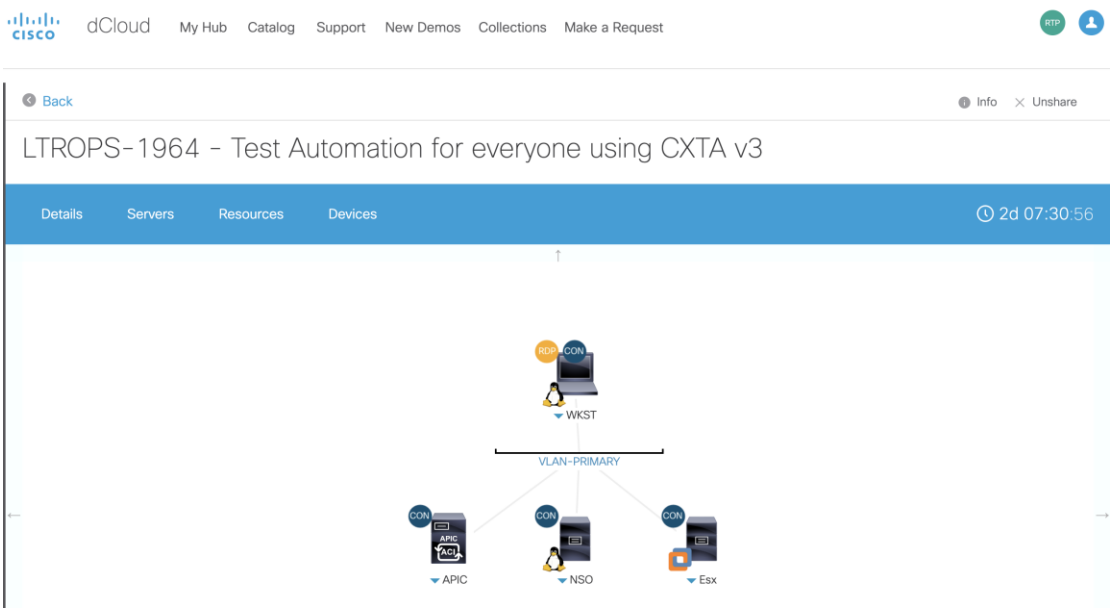


5. With terminal open run following commands to start CXTA and CXTA documentation containers and pull latest Lab repository

```
dcloud@dcloud-virtual-machine:~$ docker start cxta
cxta
dcloud@dcloud-virtual-machine:~$ docker start cxta_docs
cxta_docs
dcloud@dcloud-virtual-machine:~$ cd Desktop/CLEMEA23 LTROPS-1964/
dcloud@dcloud-virtual-machine:~/Desktop/CLEMEA23_LTROPS-1964$ git pull
```

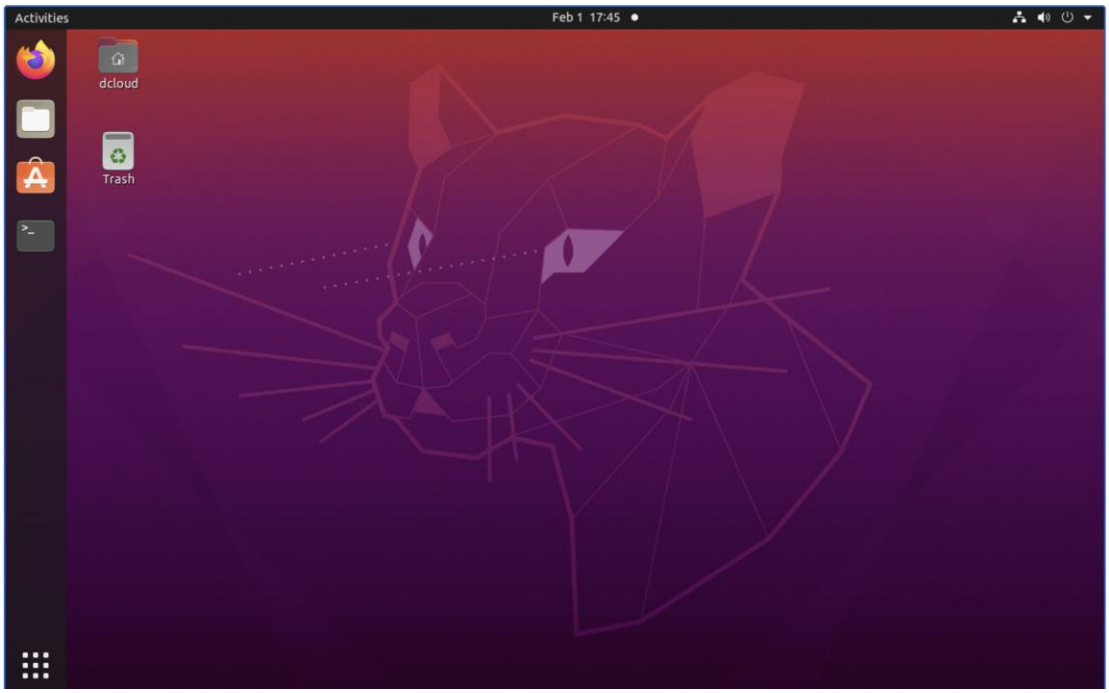


6. Go back to Dcloud interface

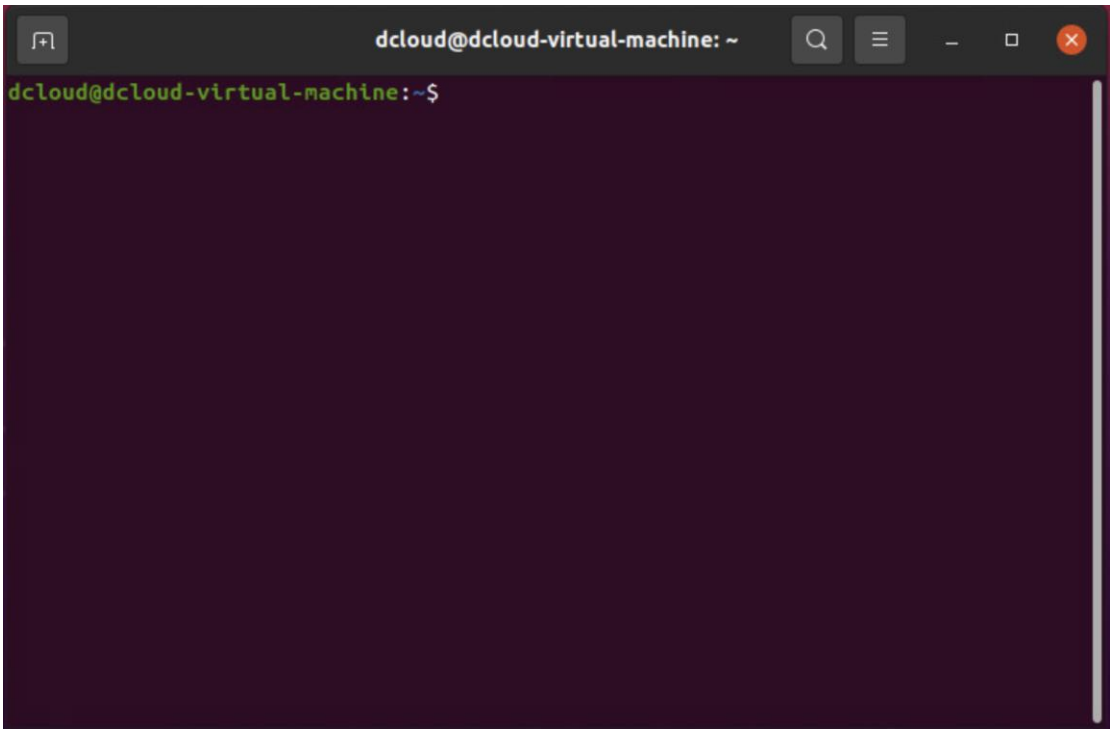


7. Select NSO machine

8. Access NSO machine using VM Console



9. Open terminal 



10. Execute following commands to start NSO server

```
dcloud@dcloud-virtual-machine:~$ cd ncs-run/
dcloud@dcloud-virtual-machine:~/ncs-run$ ncs --status
```

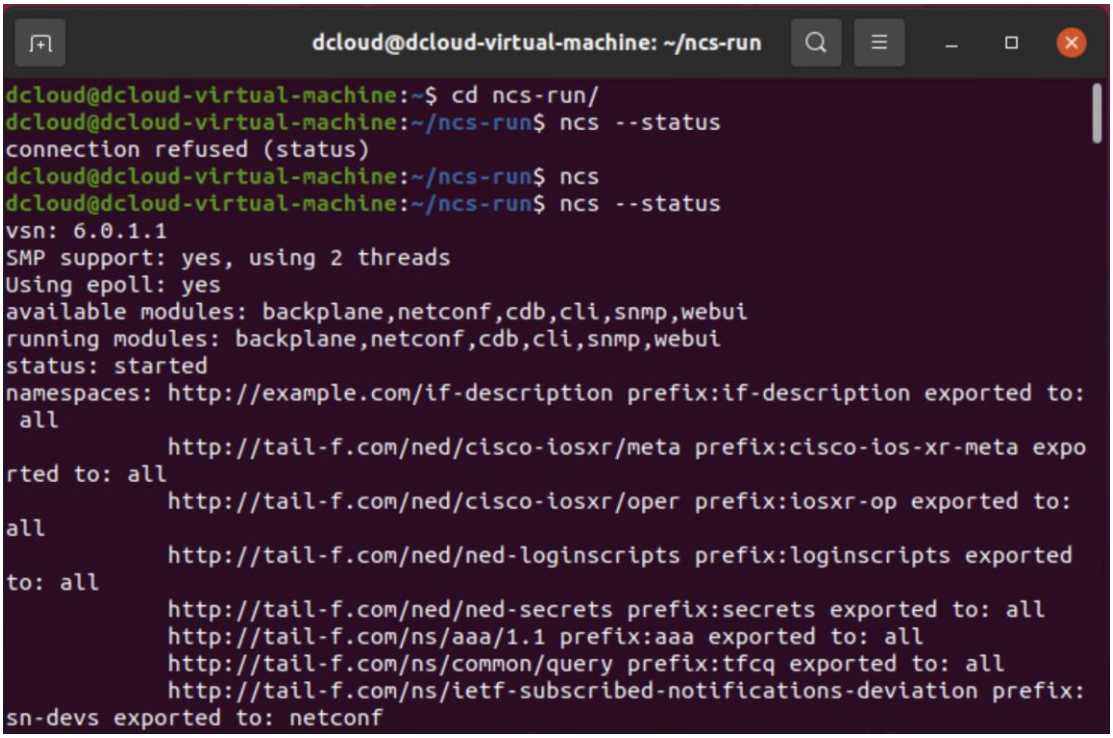
If result is 'connection refused (status)' run:

```
dcloud@dcloud-virtual-machine:~/ncs-run$ ncs
```

Run again:

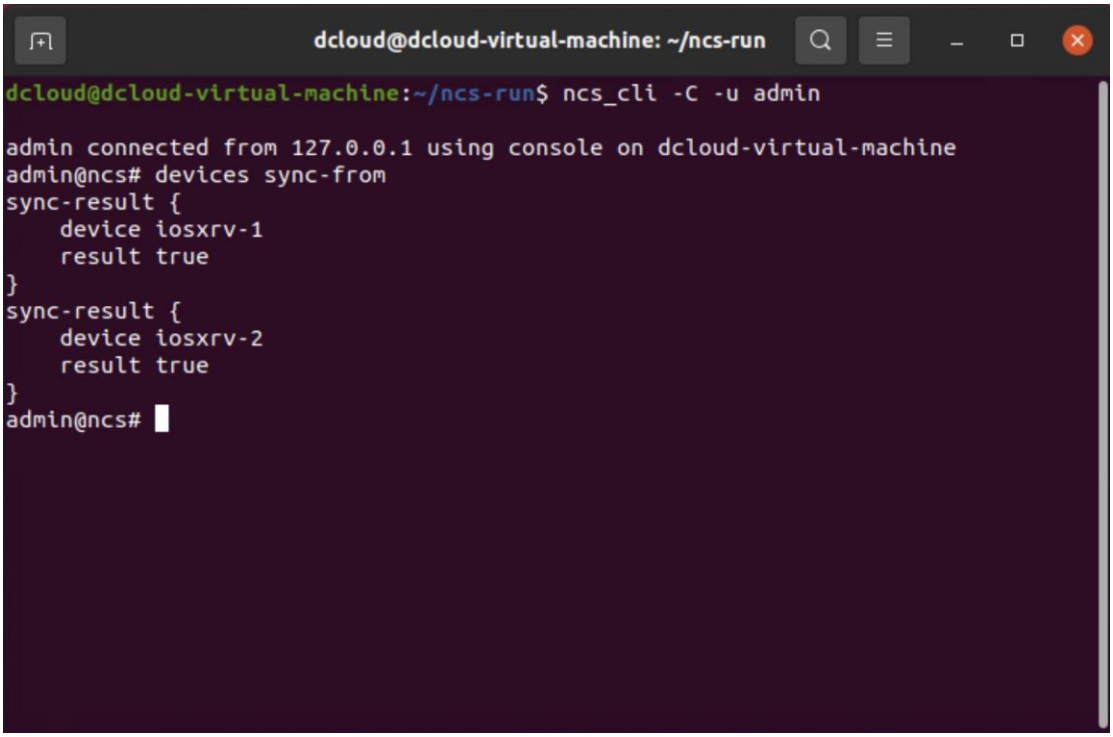
```
dcloud@dcloud-virtual-machine:~/ncs-run$ ncs --status
```

NOTE: It can take 2 minutes. Notify the proctor if it takes more than 5min.



11. Enter NSO CLI and execute ‘devices sync-from’ action

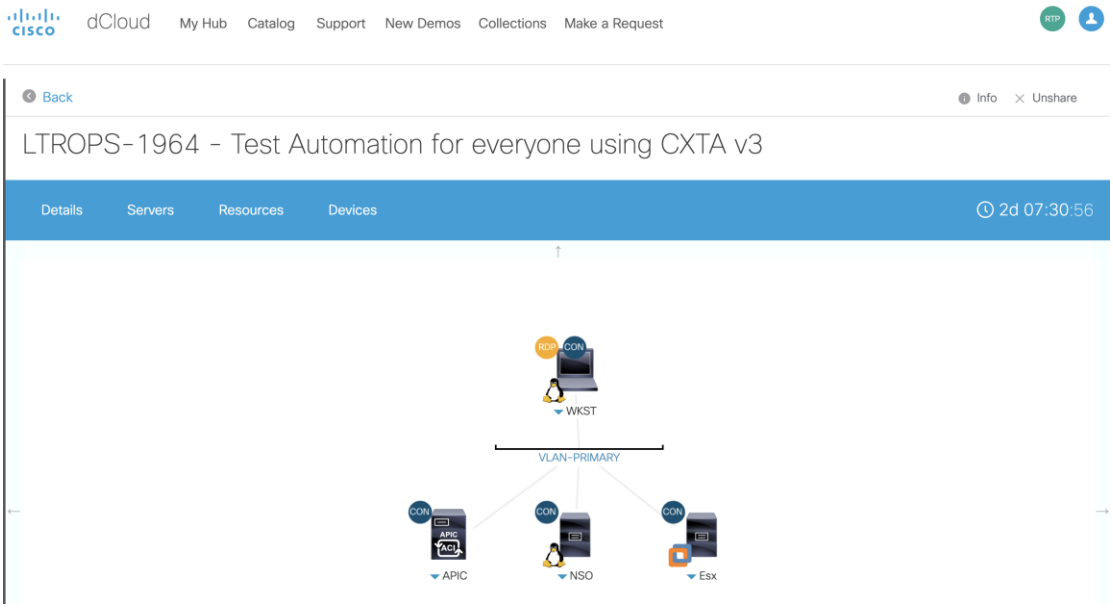
```
dcloud@dcloud-virtual-machine:~/ncs-run$ ncs_cli -C -u admin
...
admin@ncs# devices sync-from
```



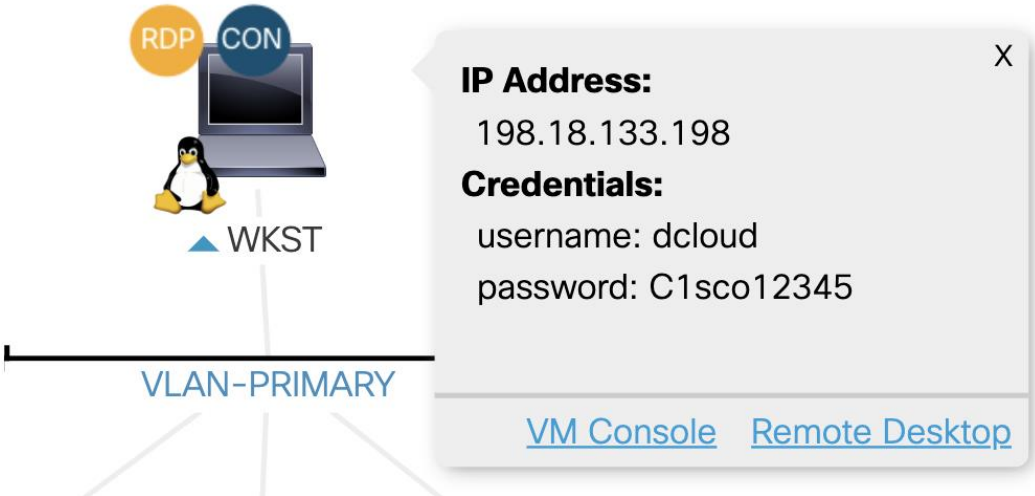
Exercise 1: Get familiar and run basic Test Cases

Task 1: Add devices to testbed.yaml

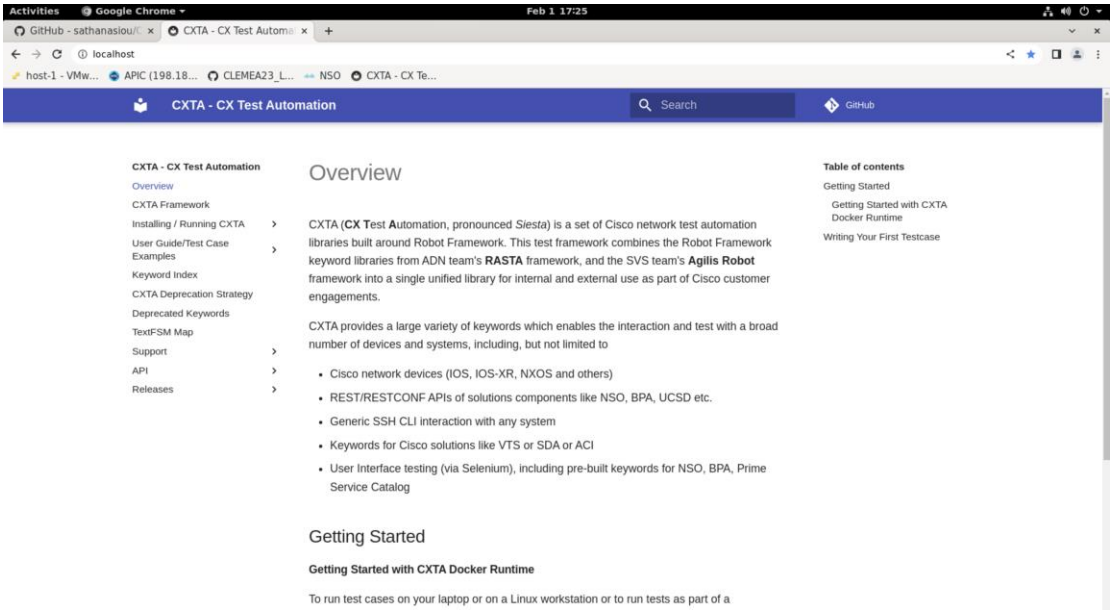
1. Use Dcloud interface to see the topology



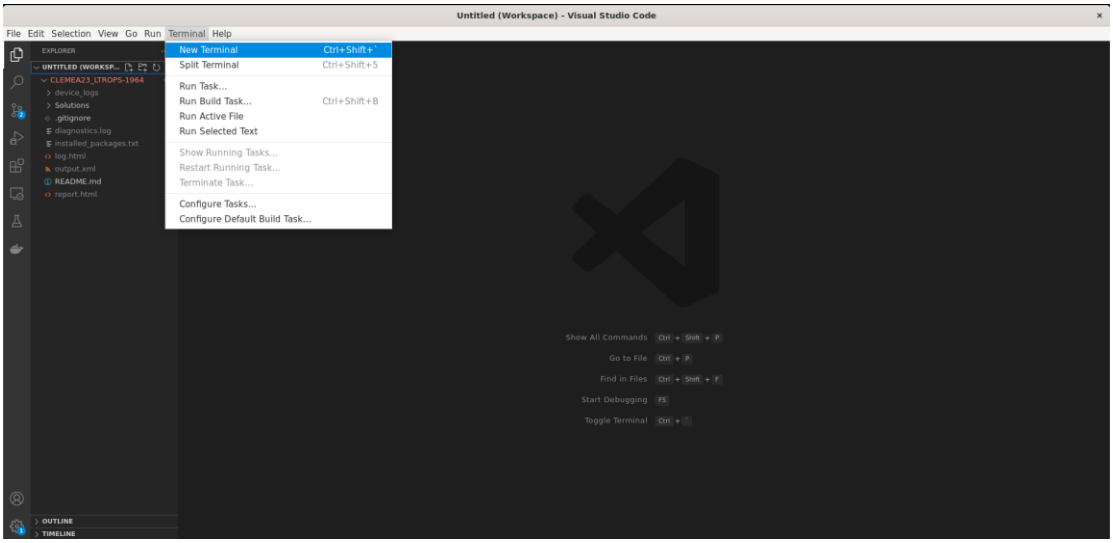
2. Open WKST object



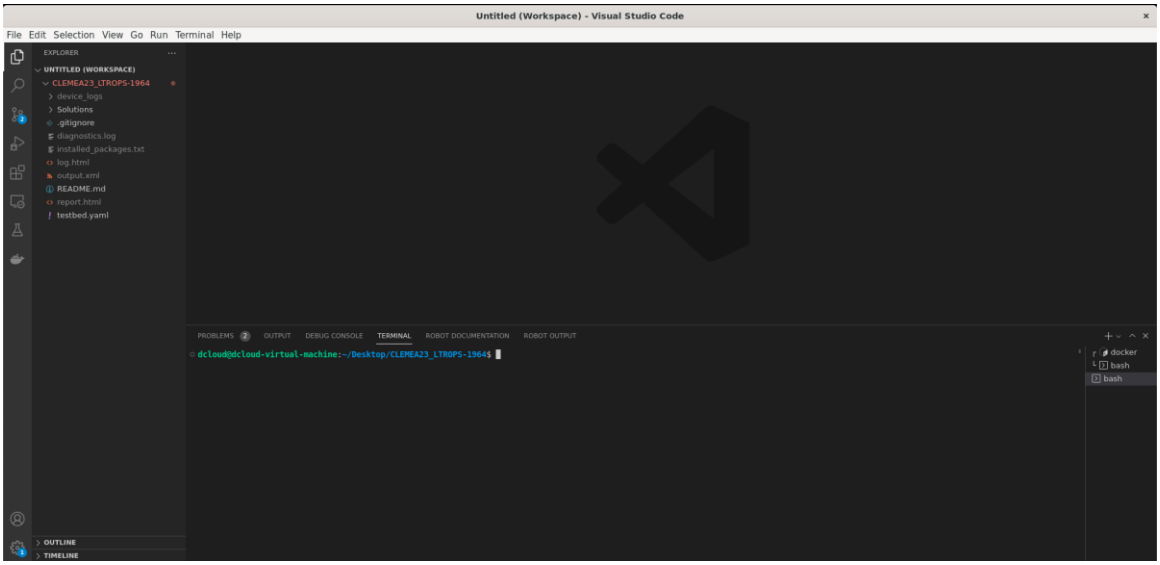
Select Remote Desktop to access Workstation interface



3. Go to 'Activities' in top right corner and open Visual Studio Code to use built-in terminal. After Visual Studio is open use Terminal Tab to create a New Terminal:



4. Inside there should be open a Workspace that includes a folder with name: CLEMEA23_LTROPS-1964



CXTA is retrieving the necessary connection information from a yaml file that is called testbed.yaml. There user should provide information such as the IP address of the target machines, the username and password, operating system of the target machine.

Testbed being a yaml file, means that it is space sensitive. The hierarchy is defined by using new lines and spaces:

```
devices:
  NX-OS-1:
    type: switch
    os: nxos
    credentials:
      default:
        username: admin
        password: C1sco12345
    connections:
      defaults:
        class: unicon.Unicon
        via: cli
      cli:
        protocol: ssh
        ip:  198.18.133.55
```

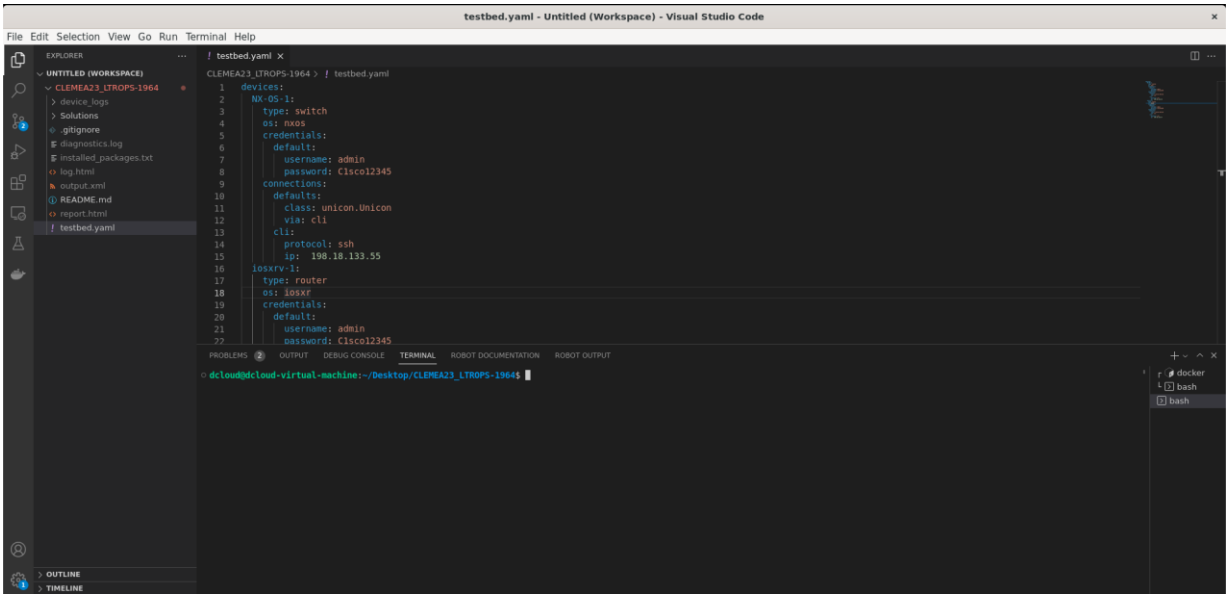
The table above describes the information needed in order to connect to a Nexus device using ssh. Using the testbed.yaml file, add also the rest of the devices to be connected.

Information regarding the devices to be added:

Host Name	Device Type	Operating System	IP	Username	Password	Connection Protocol
NX-OS-2	switch	nxos	198.18.33.56	admin	C1sco12345	ssh
iosxrv-2	router	iosxr	198.18.33.58	admin	C1sco12345	ssh
nso	nso	nso	198.18.33.100	dcloud	C1sco12345	ssh & http

apic1	apic	apic	198.18.33.200	admin	C1sco12345	Ssh & https

5. Open file testbed.yaml from Visual Studio Code and include the devices above:

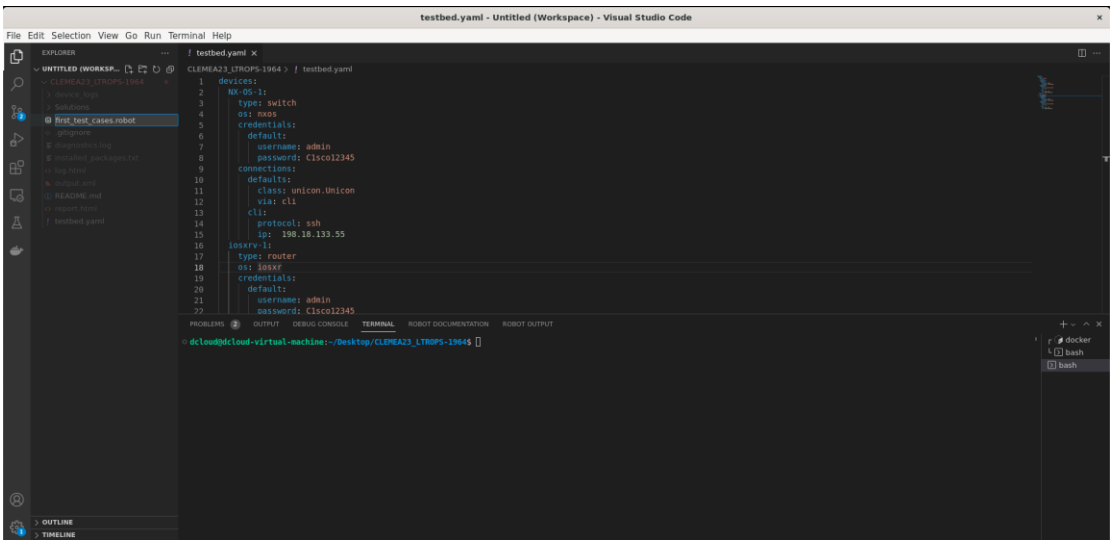


NOTE: Remember that spaces and lines are important in order to describe hierarchy in YAML.

Task 2: Create robot file and basic structure

In this step, you will create your first test case.

1. Using again Visual Studio code, in the same folder as the testbed.yaml create a new file with name: first_test_cases.robot. Using the ending .robot, it will indicate that the file we created is a robot file:



After the creation open the file. Now it is time to start writing your first test case. Firstly, some information regarding the structure of the .robot file and its content.

Each .robot file has at least two sections:

```
*** Settings ***

*** Test Cases ***
```

Inside the **Settings**, there are all the necessary Libraries and Resources in order your test cases to run. You need to include at least CXTA core libraries, and additional libraries you may need such as CXTA.robot.platforms.iosxr.config. You can also load variables from a yaml file.

The **Test Cases** section contains the actual test cases that will be executed.

Many times, we can see also two more sections:

```
*** Variables ***

*** Keywords ***
```

In **Variables**, you can define variables that will be used during this test suite and the syntax to describe them is:

```
*** Variables ***
${variable_name}      actual_value
```

Keywords part is where you can create your own Keywords, using pre-build Keywords. Why do so? For example, when there is a sequence of Keywords repeated in multiple test cases then it would be more useful to create a new Keyword using them and after all call only one Keyword at the time. Think of it as a function in programming. A Keywords section will look like:

```
*** Keywords ***

Perform a repeated or complex task
      [Arguments]     ${arg1}       ${arg2}
      Keyword1    ${arg1}
      Keyword2    ${arg2}
      Keyword3
      …
```

2. Create the *** Settings *** part of the test case inside the first_test_cases.robot file. Include as a Library the CXTA library and as a Resource the cxta.robot

```
*** Settings ***
```

3. Create a *** Test Cases *** section:

```
*** Test Cases ***
```

Here you will create two test cases:

a. Verify CXTA version

b. Specify the testbed.yaml and connect towards all devices

Task 3: Write Test Case to Verify CXTA version

Now you will create your first test case. After Task 2 your first_test_cases.robot file should look like that:

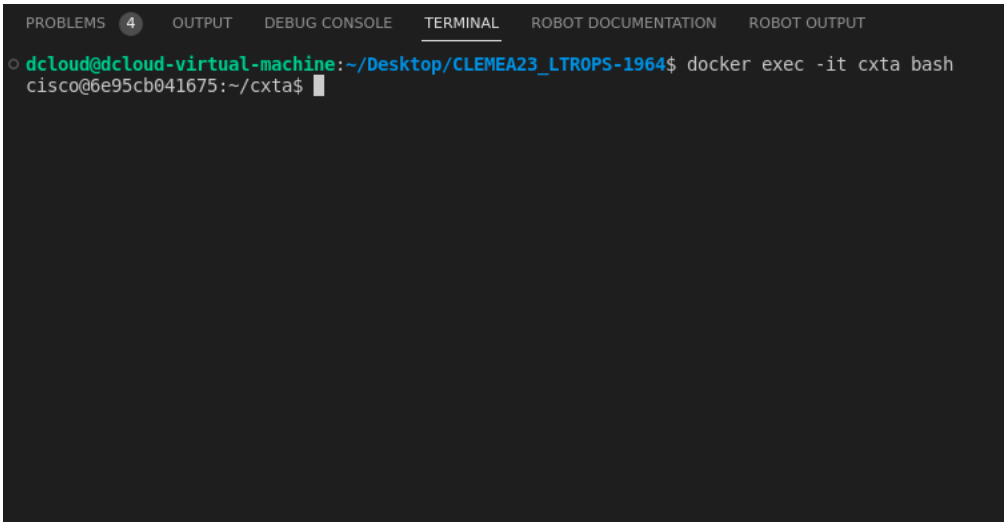
```
*** Settings ***
Library     CXTA
Resource    cxta.robot

*** Test Cases ***
```

The first task is to create a Test Case that will verify the CXTA Version that is installed inside the cxta container. In order to run all test cases, you will use a cxta container that is volumed in the folder CLEMEA23_LTROPS-1964. Every change that you are making in that folder will be visible in /home/cisco/cxta path of the container.

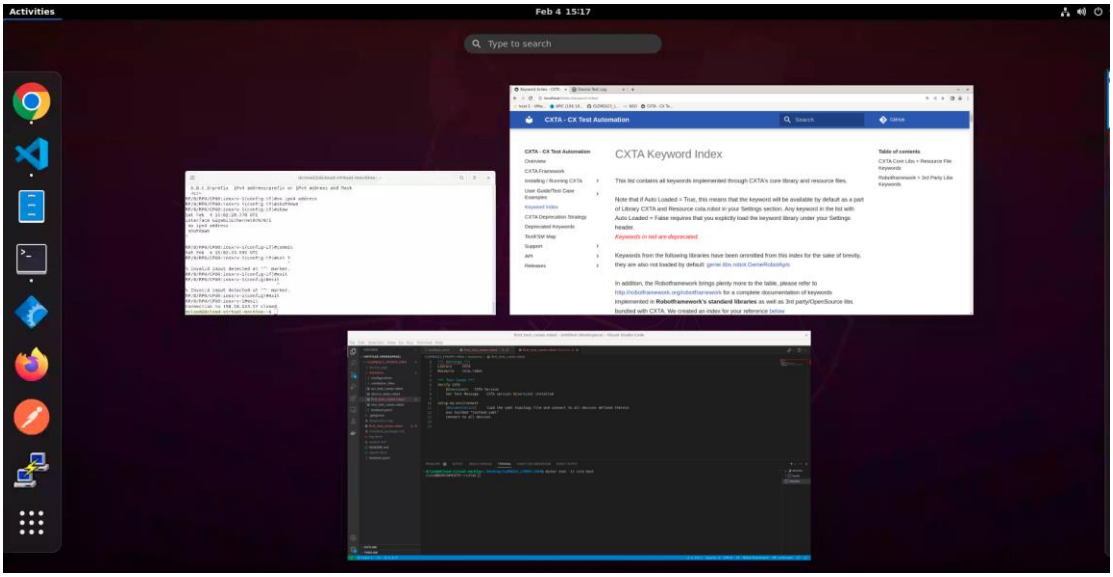
1. Connect to the container using:

```
docker exec -it cxta bash
```

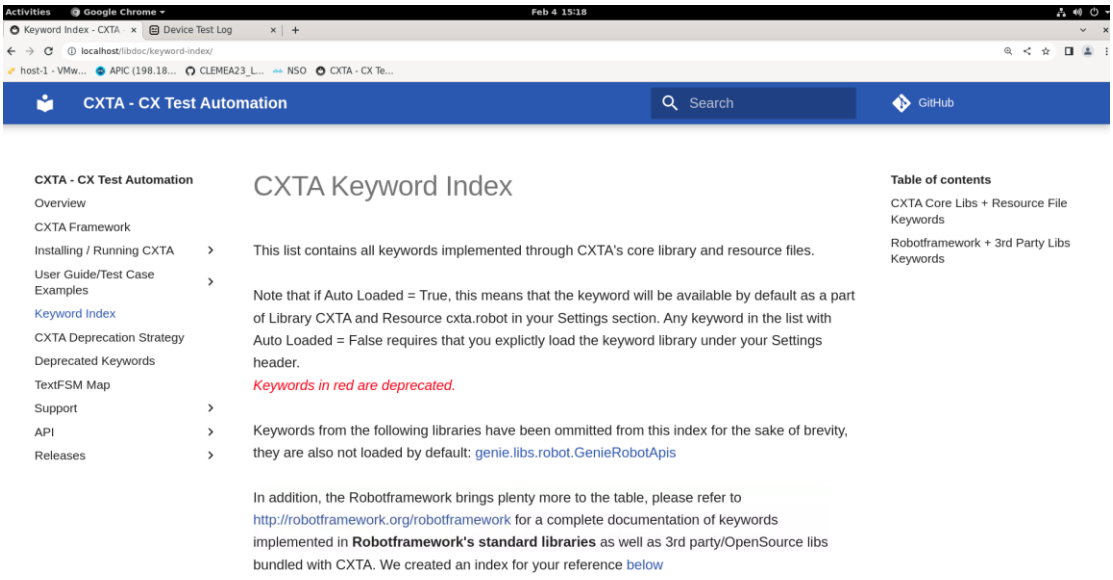
Run that command in the Visual Studio Terminal. This is how it will look like. From there the test cases will be executed.



2. In the Remote Desktop push Activities, that is located on the upper left side:



Open Google Chrome in order to get to the CXTA Keyword Index. In that section of the documentation, you can navigate and see all available Keywords and their documentation.

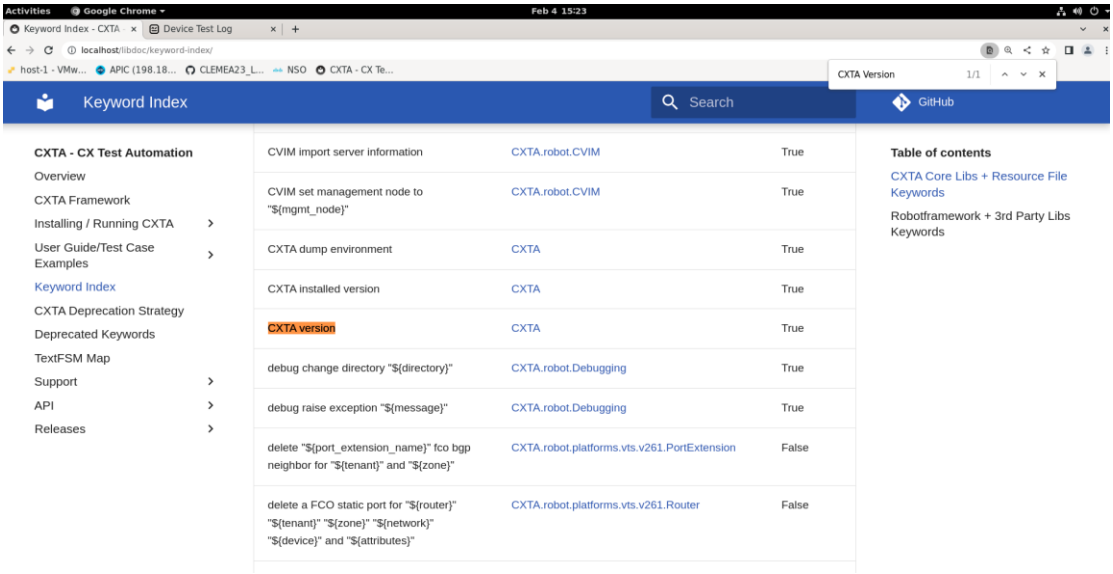


As you can navigate over the page, there are plenty of Keywords. Using control+F buttons you are able to find the Keyword that you need.

3. Under the *** Test Cases *** section create a Test case with name: Verify CXTA:

```
*** Settings ***
Library     CXTA
Resources   cxta.robot

*** Test Cases ***
Verify CXTA
```

4. Goal is to find the CXTA Version that the container is running. Using the Keyword Index and the searching we can find the keyword that we need:

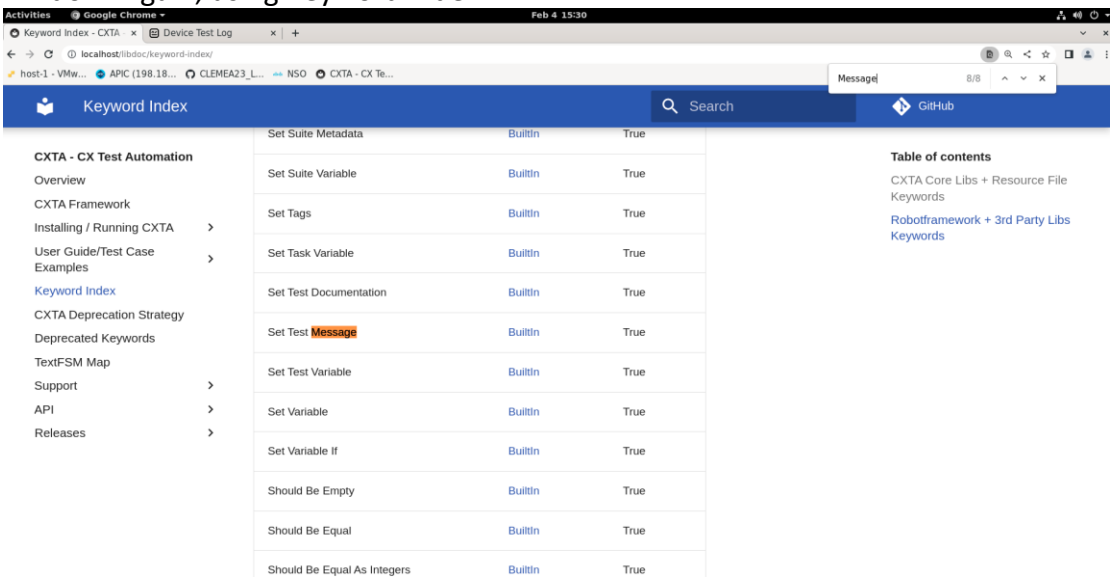


5. Create a variable named “version” that will store the output of the command CXTA Version:

```
*** Settings ***
Library     CXTA
Resources   cxta.robot

*** Test Cases ***
Verify CXTA
      ${version}=       CXTA Version
```

NOTE: Spaces and tabulations are playing also an important role for CXTA and robot in general.

6. Use a keyword that will allow you to display a message in the execution window. Again, using Keyword Index:



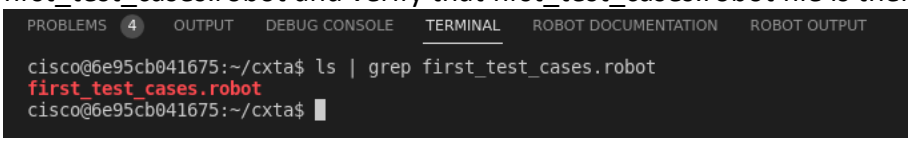
Now the Test Case will look like that:

```
*** Settings ***
Library     CXTA
Resources   cxta.robot
```

```
*** Test Cases ***
Verify CXTA
      ${version}=       CXTA Version
      Set Test Message        CXTA version ${version} installed
```

7. Time to run your first test case:

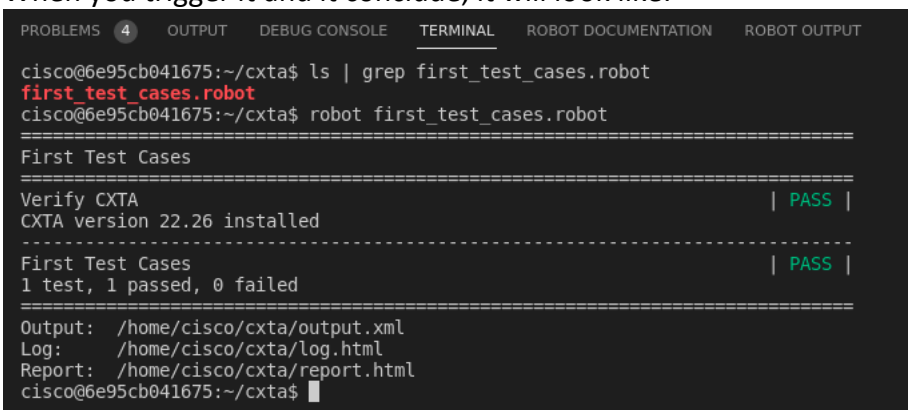
   Go back to Visual Studio Code and from inside the cxta container run ls | grep first_test_cases.robot and verify that first_test_cases.robot file is there:



   In order to run the test cases, you need to run the following inside the cxta docker container:

```
robot first_test_cases.robot
```

   When you trigger it and it conclude, it will look like:

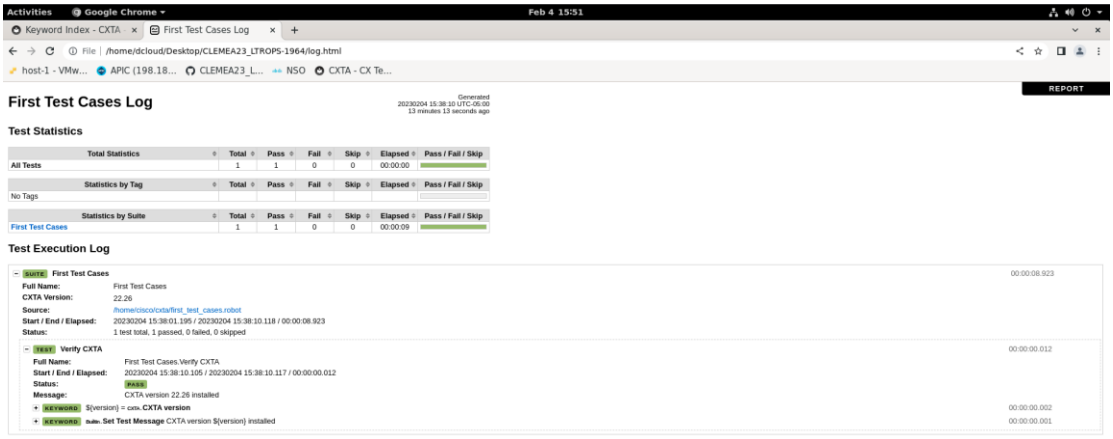


   Congratulations! You just wrote and ran your first Test Case.

   Observing the result, we can see that it provides the Test Result, in that case PASS, the Test Case name as well as the message that you instructed the Test Case to print.

Moreover, you can see some files created. The one that will provide us with the most information the test cases are passing they are failing is the log.html file. That file will be located in the same folder as the test case that you are running. For this lab, it is already open in Google Chrome, and you only need to refresh the page.

8. Get back to Google Chrome and go to the tab that is located the log.html file:



As you can see the log.html file is quite informative. Taking a closer look:

Test Execution Log

SUITE First Test Cases

Full Name:	First Test Cases
CXTA Version:	22.26
Source:	/home/cisco/cxta/first_test_cases.robot
Start / End / Elapsed:	20230204 15:38:01.195 / 20230204 15:38:10.118 / 00:00:08.923
Status:	1 test total, 1 passed, 0 failed, 0 skipped

TEST Verify CXTA

Full Name:	First Test Cases.Verify CXTA
Start / End / Elapsed:	20230204 15:38:10.105 / 20230204 15:38:10.117 / 00:00:00.012
Status:	PASS
Message:	CXTA version 22.26 installed

KEYWORD ${version} = CXTA.**CXTA version**

Documentation:	Returns the CXTA version
Start / End / Elapsed:	20230204 15:38:10.112 / 20230204 15:38:10.114 / 00:00:00.002

15:38:10.114 INFO CXTA version 22.26

15:38:10.114 INFO ${version} = 22.26

KEYWORD Builtin.**Set Test Message** CXTA version ${version} installed

Documentation:	Sets message for the current test case.
Start / End / Elapsed:	20230204 15:38:10.116 / 20230204 15:38:10.117 / 00:00:00.001

15:38:10.116 INFO Set test message to:
CXTA version 22.26 installed

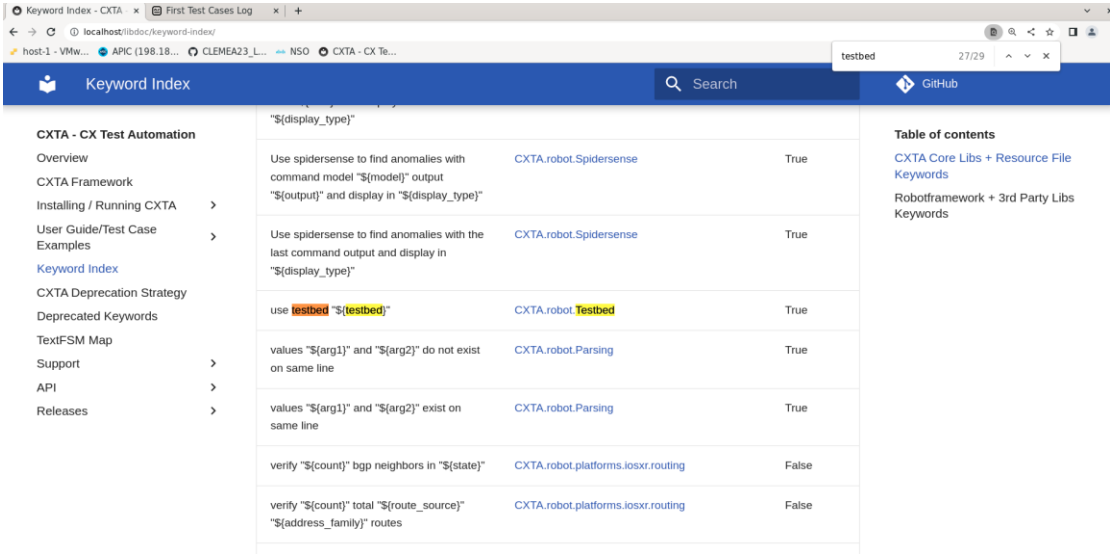
Take a few moments to see what is written in the log.html file since it will be your best allay while executing test cases.

Task 4: Specify testbed.yaml and verify connection with all devices

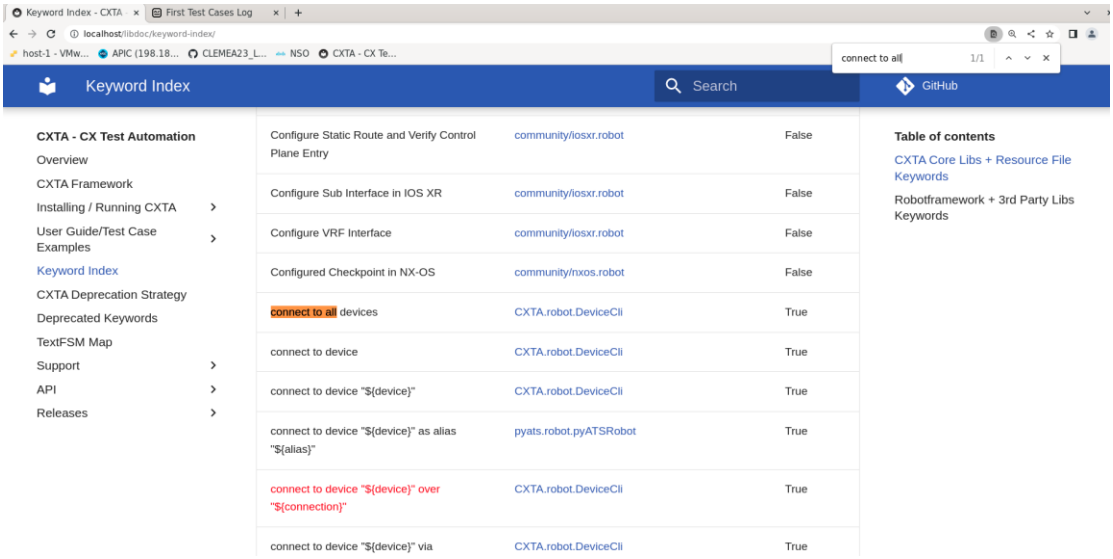
1. Get back to the first_test_cases.robot file you created in the previous Task. Create a new Test Case with name: setup-my-environment:

```
*** Settings ***
Library     CXTA
Resources   cxta.robot

*** Test Cases ***
Verify CXTA
      ${version}=       CXTA Version
      Set Test Message        CXTA version ${version} installed

setup-my-environment
```

2. Specify the testbed you will use. Get the right keyword using Keyword Index:



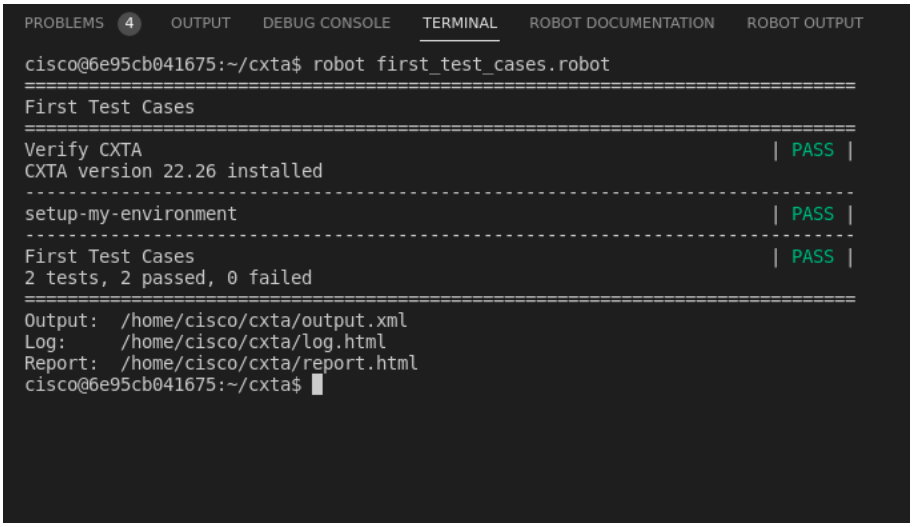
3. Connect to all devices, you know where to find the Keyword:



4. Now your test case should look like:

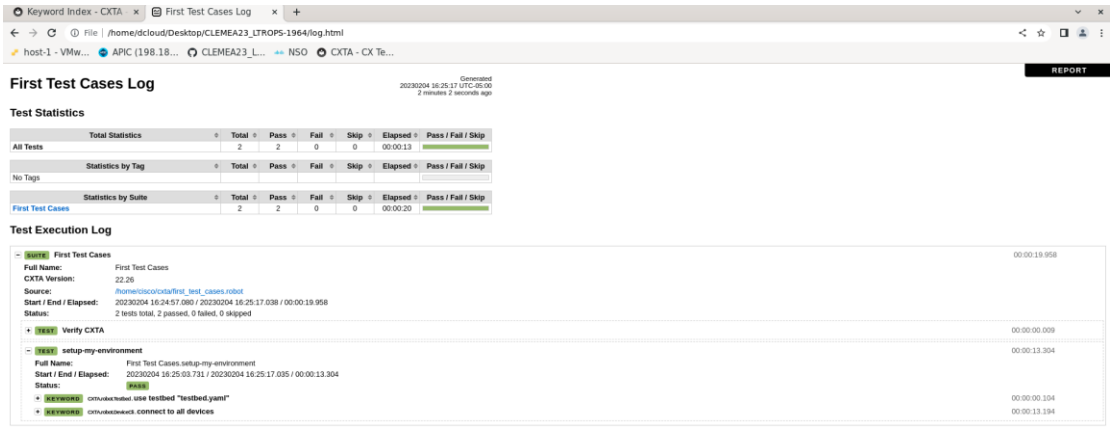
```
*** Settings ***
Library     CXTA
Resources   cxta.robot

*** Test Cases ***
Verify CXTA
      ${version}=       CXTA Version
      Set Test Message        CXTA version ${version} installed

setup-my-environment
      use testbed "testbed.yaml"
      connect to all devices
```

5. From inside the container run the test cases:

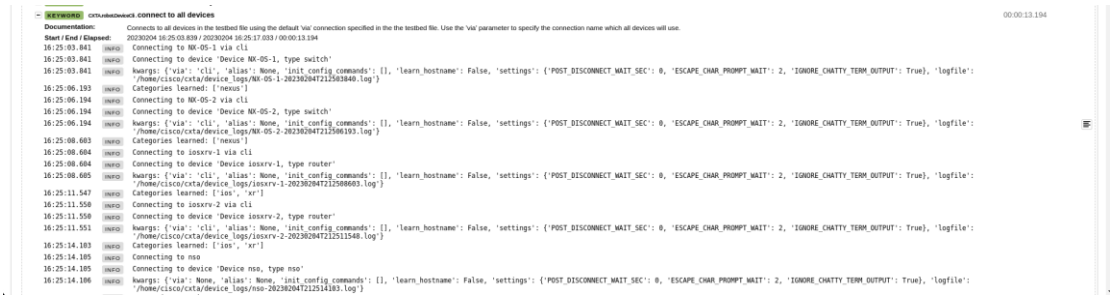


If everything is green means that you are able to use the testbed information and connect to all the devices.

6. Observe the log.html file that is in Google Chrome, after refreshing you will see the results of the new test cases:



7. Expand the connect to all devices section and observe the connections. It will help you to have a better view of how CXTA is executing the Keywords:



Well done! Exercise 1 is over. Now you know:

1. How the structure of a test case looks like,
2. How to create the testbed file,
3. How to write and execute test cases

4. How the log.html file looks like

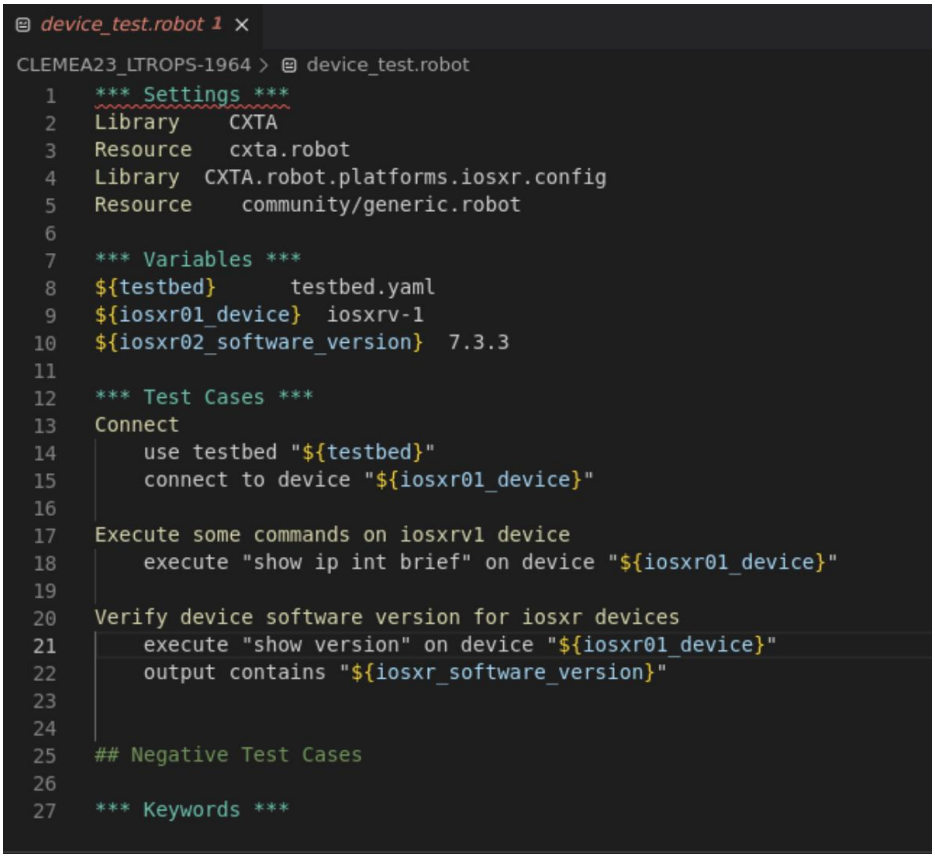
Time for Exercise 2.

Exercise 2: Test Cases Towards Devices

In this exercise, you will explore the functionalities of CXTA in running commands towards the devices. You will connect to the devices, execute commands, verify the configuration and create a new Keyword. Furthermore, in the last exercise, you will run test cases that are designed to Fail so that you observe how a log.html file looks like when the devices do not have the expected configuration.

Task 1: Connect to devices and run commands

1. Open devices_tests.robot file in Visual Studio Code



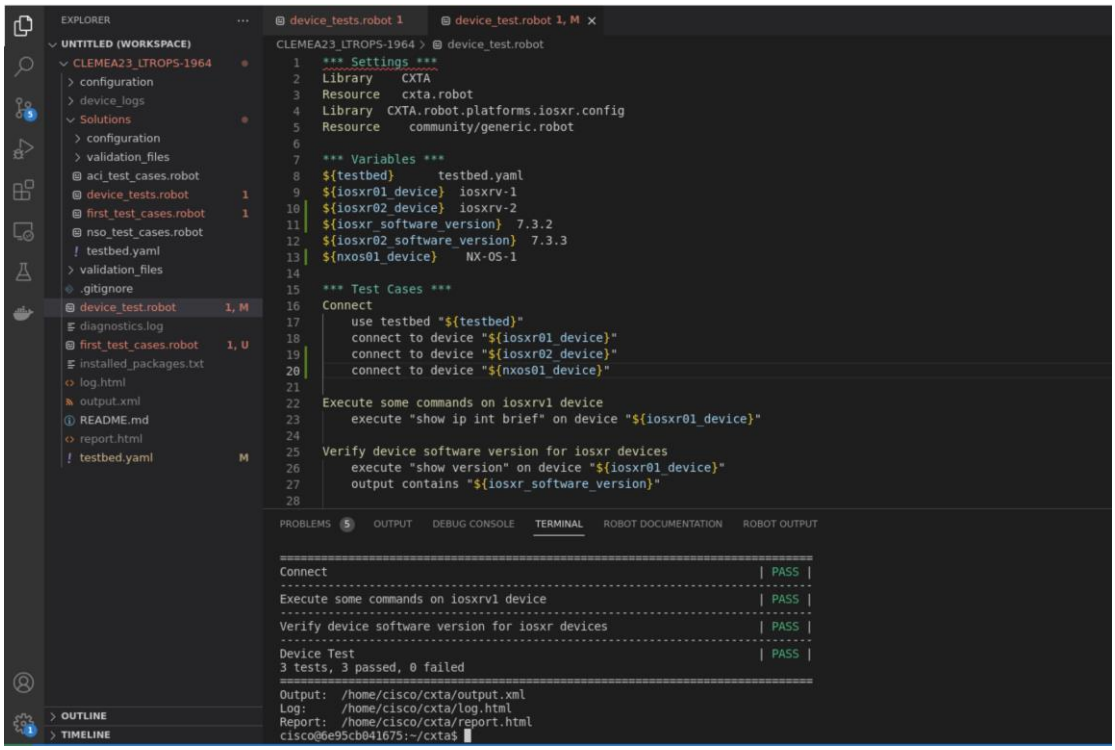
   Observe that some Test Cases and some Keywords are already provided.

2. Add three more variables:
   a. A variable for iosxrv-2 device with name iosxr02_device
   b. A variable with name iosx_software_version and value 7.3.2
   c. A variable with NX-OS-1 device
3. Add the necessary keywords under Connect so that you get connected to devices:
   a. iosxrv-2
   b. NX-OS-1

Following those 3 steps and with the existing Keywords, you can run the device_test.robot. Take a look at the test cases, what do they examine?

Do you expect your Test Cases to PASS or FAIL?

4. Run the current test scenarios:



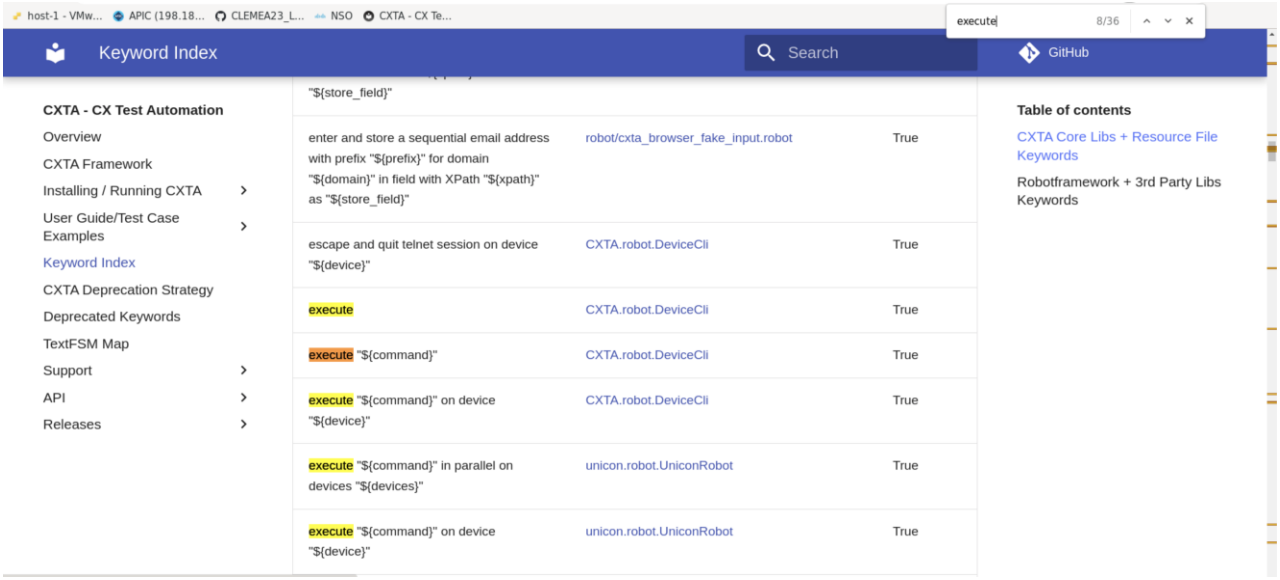
So now you know how to add variables and connect to the devices

5. The second Test Cases Suite is instructing CXTA to run some commands on iosxrv-1 device. Add some more commands:

    a. Run command: “show version”

    b. Run command: “show ip protocol”

    In order to find the correct Keywords use Keywords Index:

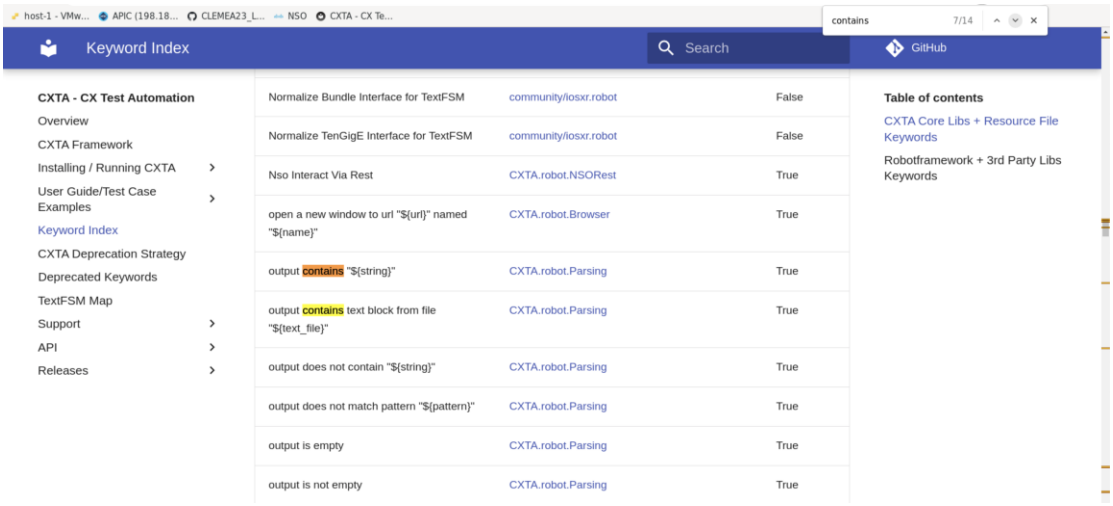


6. In next Test Cases Suite, we verify that the software version of iosxrv-1 is the one that we want (7.3.2). Verify the same for device iosxrv-2:

    a. Run command show version

    b. Verify that the output of the command contains the correct iosxr Software Version.

To verify that an output contains a specific string:



Check using the Documentation how the Keyword output contains “${sting}” is functioning.

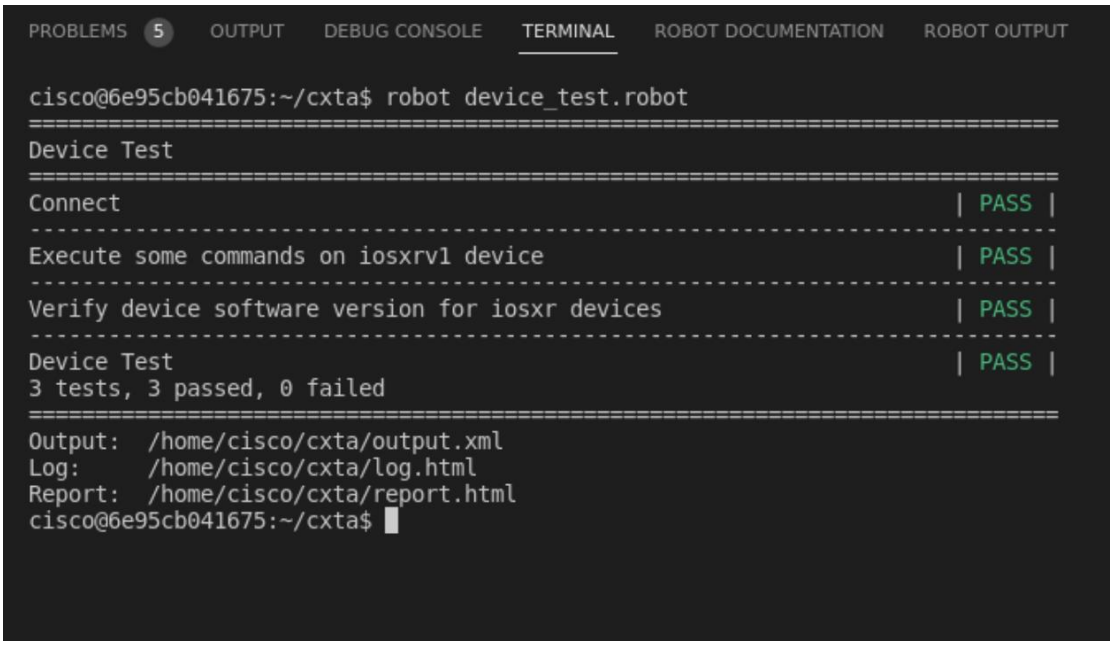
7. Run the Test Cases that you have created so far.

    This is how your code should look like, more or less. If you have created different variables’ names, it is ok as long as your have the same name everywhere that you are calling the variable.

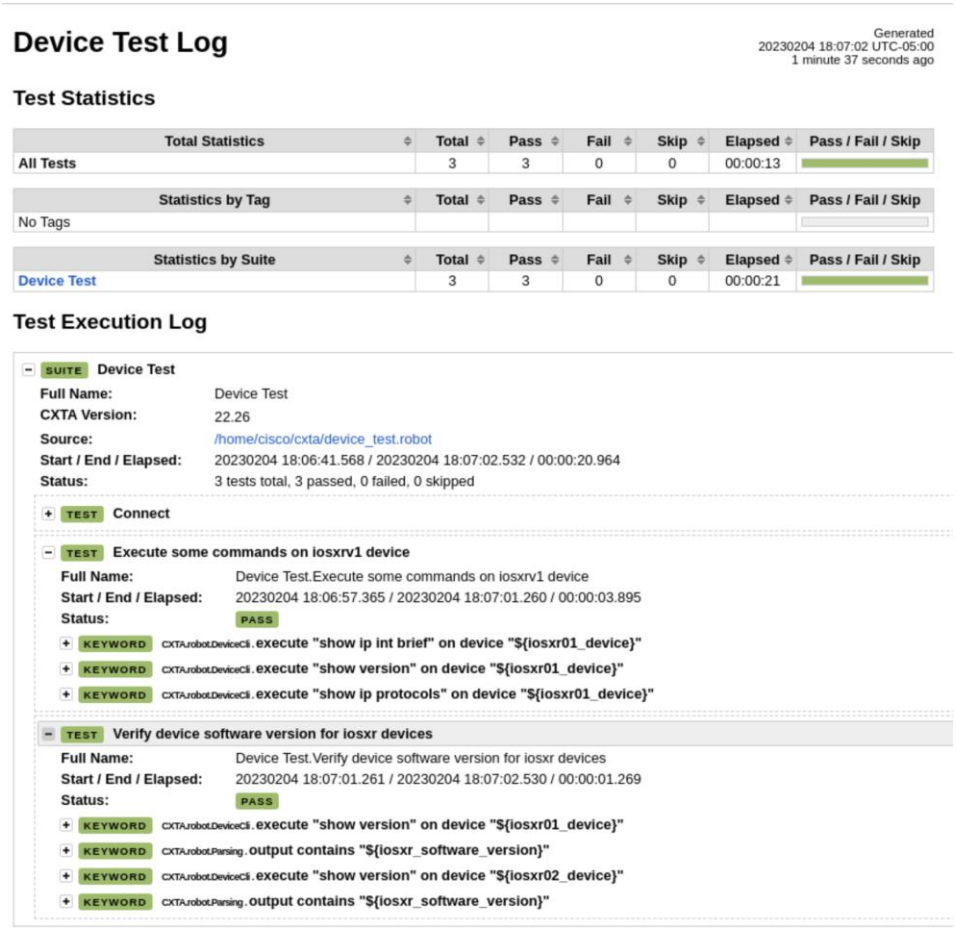
```
*** Settings ***
Library     CXTA
Resource    cxta.robot
```

```
Library   CXTA.robot.platforms.iosxr.config
Resource     community/generic.robot

*** Variables ***
${testbed}       testbed.yaml
${iosxr01_device}   iosxrv-1
${iosxr02_software_version}   7.3.3

*** Test Cases ***
Connect
    use testbed "${testbed}"
    connect to device "${iosxr01_device}"
    connect to device "${iosxr02_device}"
    connect to device "${nxos01_device}"

Execute some commands on iosxrv1 device
    execute "show ip int brief" on device "${iosxr01_device}"
    execute "show version" on device "${iosxr01_device}"
    execute "show ip protocols" on device "${iosxr01_device}"

Verify device software version for iosxr devices
    execute "show version" on device "${iosxr01_device}"
    output contains "${iosxr_software_version}"
    execute "show version" on device "${iosxr02_device}"
    output contains "${iosxr_software_version}"
```

Run Test Cases:



8. Study log.html file:



Device Test Log

Generated
20230204 18:07:02 UTC-05:00
1 minute 37 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	3	0	0	00:00:13	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Device Test	3	3	0	0	00:00:21	

Test Execution Log

SUITE **Device Test**

Full Name: Device Test
CXTA Version: 22.26
Source: /home/cisco/cxta/device_test.robot
Start / End / Elapsed: 20230204 18:06:41.568 / 20230204 18:07:02.532 / 00:00:20.964
Status: 3 tests total, 3 passed, 0 failed, 0 skipped

TEST **Connect**

TEST **Execute some commands on iosxrv1 device**

Full Name: Device Test.Execute some commands on iosxrv1 device
Start / End / Elapsed: 20230204 18:06:57.365 / 20230204 18:07:01.260 / 00:00:03.895
Status: PASS

KEYWORD CXTA.robot.DeviceCli . **execute "show ip int brief" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show ip protocols" on device "${iosxr01_device}"**

TEST **Verify device software version for iosxr devices**

Full Name: Device Test.Verify device software version for iosxr devices
Start / End / Elapsed: 20230204 18:07:01.261 / 20230204 18:07:02.530 / 00:00:01.269
Status: PASS

KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.Parsing . **output contains "${iosxr_software_version}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr02_device}"**
KEYWORD CXTA.robot.Parsing . **output contains "${iosxr_software_version}"**

When it comes to the Test Case of devices' software verification, do you see any improvement that could apply?

Task 2: Keyword creation

The answer to the previous question is to create a new Keyword. That keyword will have as an inputs:

a. Device Name

b. Software Version

1. Create a Keywords section after the *** Test Cases *** section

```
*** Keywords ***
```

2. Create the Keyword name. Remember that for this Keyword it is necessary to use two input parameters:

```
Connect to device "${device_name}" and verify is running
"${software_version}"
```

NOTE: It should be all in the same row.

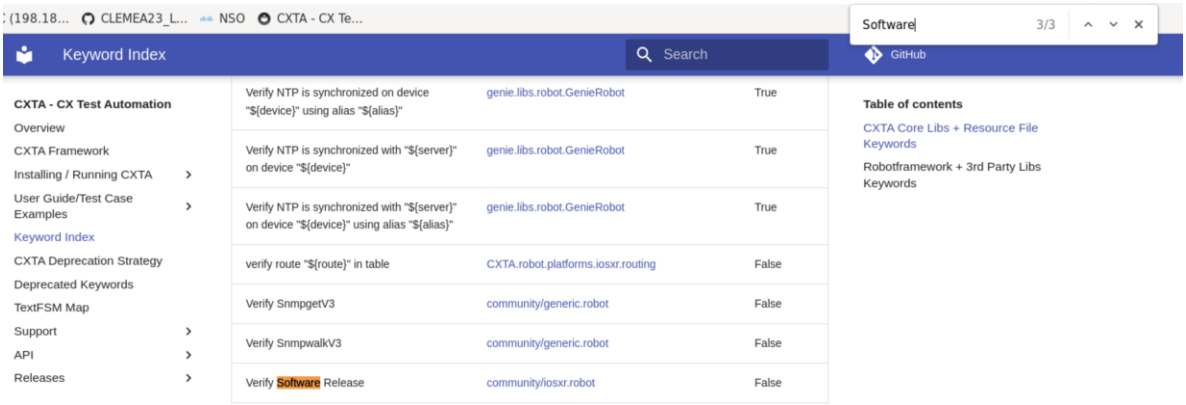
3. Provide the first action of the Keyword, which will be to connect tp device,
4. Execute the show version command to the device,
5. Verify if the output contains the desired software version,
6. Use the Keyword in a Test Case in order to verify that iosxrv-1 and iosxrv-2 are running the 7.3.2 IOS-XR Version,
7. You Keyword and Test Case should look like:

```
Verify device software version using Keyword
    Connect to device "${iosxr01_device}" and verify is running
"${iosxr_software_version}"
    Connect to device "${iosxr02_device}" and verify is running
"${iosxr_software_version}"


*** Keywords ***

Connect to device "${device_name}" and verify is running
"${software_version}"
    connect to device "${device_name}"
    execute "show version" on device "${device_name}"
    output contains "${software_version}"
```

8. Execute again the device_test.robot:

```
PROBLEMS 5   OUTPUT   DEBUG CONSOLE   TERMINAL   ROBOT DOCUMENTATION   ROBOT OUTPUT

cisco@6e95cb041675:~/cxta$ robot device_test.robot
==============================================================================
Device Test
==============================================================================
Connect                                                               | PASS |
------------------------------------------------------------------------------
Execute some commands on iosxrv1 device                               | PASS |
------------------------------------------------------------------------------
Verify device software version for iosxr devices                      | PASS |
------------------------------------------------------------------------------
Verify device software version using Keyword                          | PASS |
------------------------------------------------------------------------------
Device Test                                                           | PASS |
4 tests, 4 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

9. Navigating in Keywords Index you can find that there is a Keyword developed specifically for iosxr Software Version validation:



The third column in the Keyword Index indicated if a Keyword is by default part of CXTA. If the value is True then it is, if it is False on the other hand, it is not. That means, user needs either to add a Library either a Resource.

Enter the documentation and go at the beginning of the page. You will find:

Library scope: GLOBAL

Introduction

Resource file contains keyword created by the community, specifically for IOSXR platforms. These keywords are not available by default, to load the keywords add the resource to your Settings section after the cxta library and resource, e.g.

Library	CXTA
Resource	cxta.robot
Resource	community/iosxr.robot

a. Add community/iosxr./robot as a Resource in the *** Settings *** field:

```
*** Settings ***
Library cxta
Resourse cxta.robot
```

```
Resource community/generic.robot
Resource community/iosxr.robot
```

b. Using the Keyword create a new test case to verify the Software Version of iosxrv-1 device:

```
Verify software version using iosxr Keywords
    Verify Software Release  ${iosxr01_device}
${iosxr_software_version}
```

NOTE: All in one line

c. Run again the test cases suite using:

```
Robot device_test.robot
```

In that task, you created a Keyword and used it and also navigated a bit around the documentation file. Sometimes it is necessary either to add another Library either to Resource additional code. Finally, as you can see many of the actions and verifications you want to do are part of CXTA Library and created by the CXTA community members, so firstly search for the Keyword you need, most probably somebody has already developed it!

Task 3: Configure devices using files and validate config

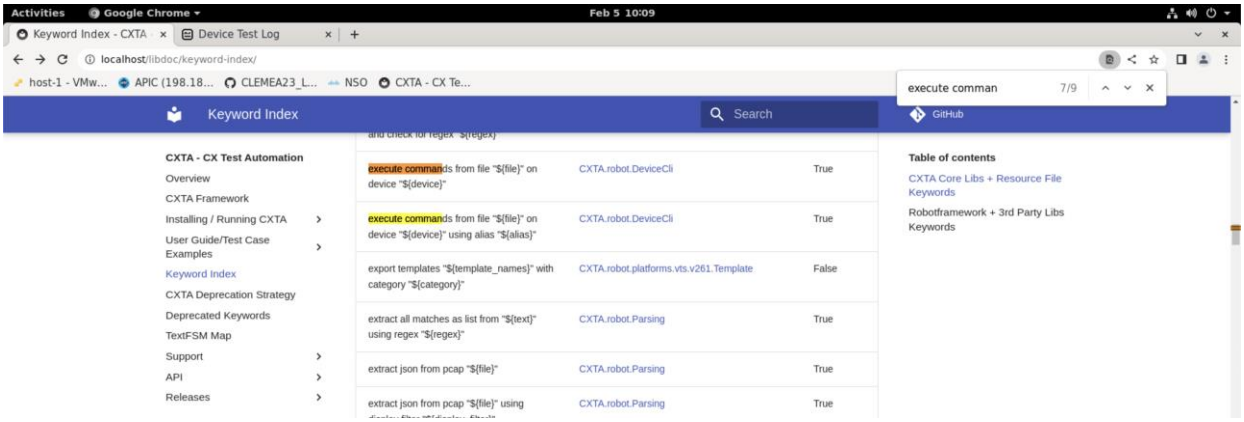
In this Task you will push configuration to devices using a txt file and also validate the configuration with your expected configuration.

1. Open device_test.robot in Visual Studio Code
2. In the same folder as devices_tests.robot file there is a folder with name configuration. Get in that folder and open file with name iosxr_configuration.txt. The content should be:

```
configure
interface GigabitEthernet 0/0/0/1
ipv4 address 10.10.10.1/24
no shutdown
commit
exit
exit
```

As you can see those are all the necessary commands in order to configure an IP address for interface GigabitEthernet 0/0/0/1, bring the interface up and commit that configuration. That will be the configuration to be pushed in iosxrv-1 device.

3. Create a Test Case that will push the file configuration in iosxrv-1 device. Search for the Keyword that is helping you with that tasks in Keywords Index:



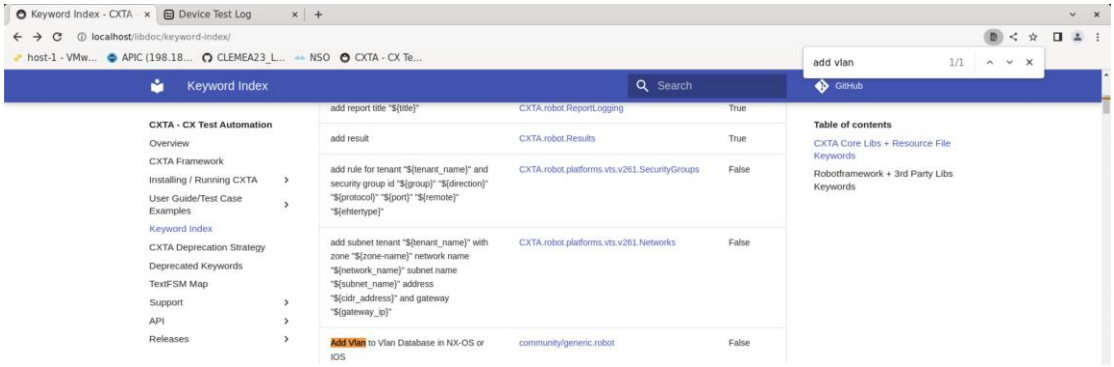

```
Send configuration to device using file
    execute commands from file
"configuration/iosxr_configuration.txt" on device
"${iosxr01_device}"
```

4. Create another Test Case that will add the VLANs of range 9-11 to NX-OS-1 device and then will validate if the configuration applied is the expected. The expected configuration is stored in a file.

    a. Create the Test Case name : Add vlan in nxos device and validate:

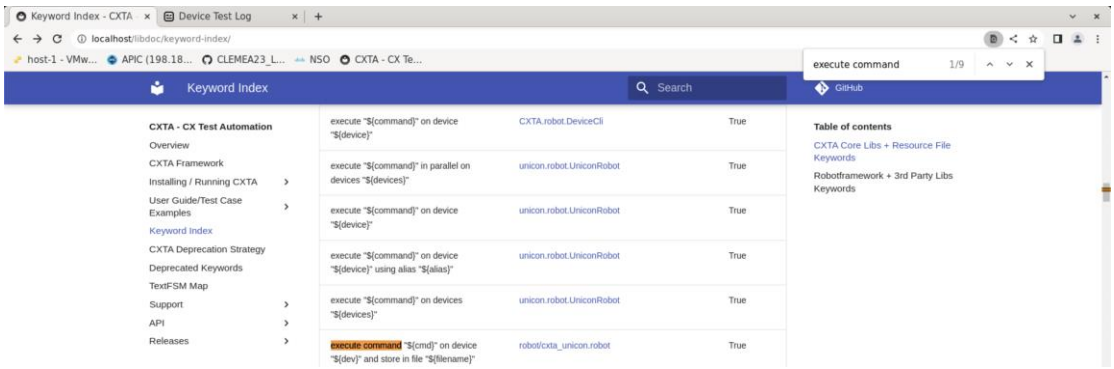
```
Add vlan in nxos device and validate
```

    b. Use the correct Keyword to add a range of VLANs:




```
Add vlan in nxos device and validate
    Add Vlan to Vlan Database in NX-OS or IOS
${nxos01_device}    9-11
```

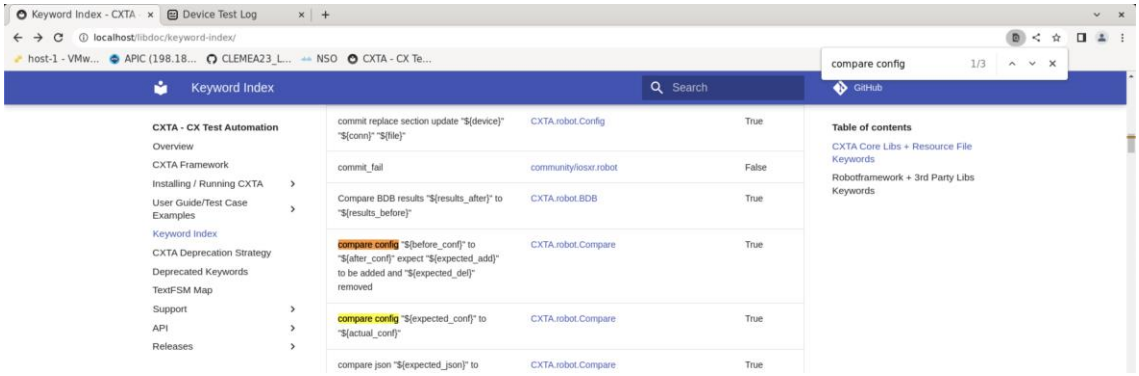
c. Execute a show vlan command on NX-OS-1 and store the output under the foler configuration in a file named vlan_config.txt:



Now test case looks like:

```
Add vlan in nxos device and validate
    Add Vlan to Vlan Database in NX-OS or IOS
${nxos01_device}    9-11
    execute command "show vlan" on device
"${nxos01_device}" and store in file
"configuration/vlan_config.txt"
```

d. User compare config keyword to compare the output of the show vlan command of previous step with the expected configutration:



Bringing everything together the test case looks like:

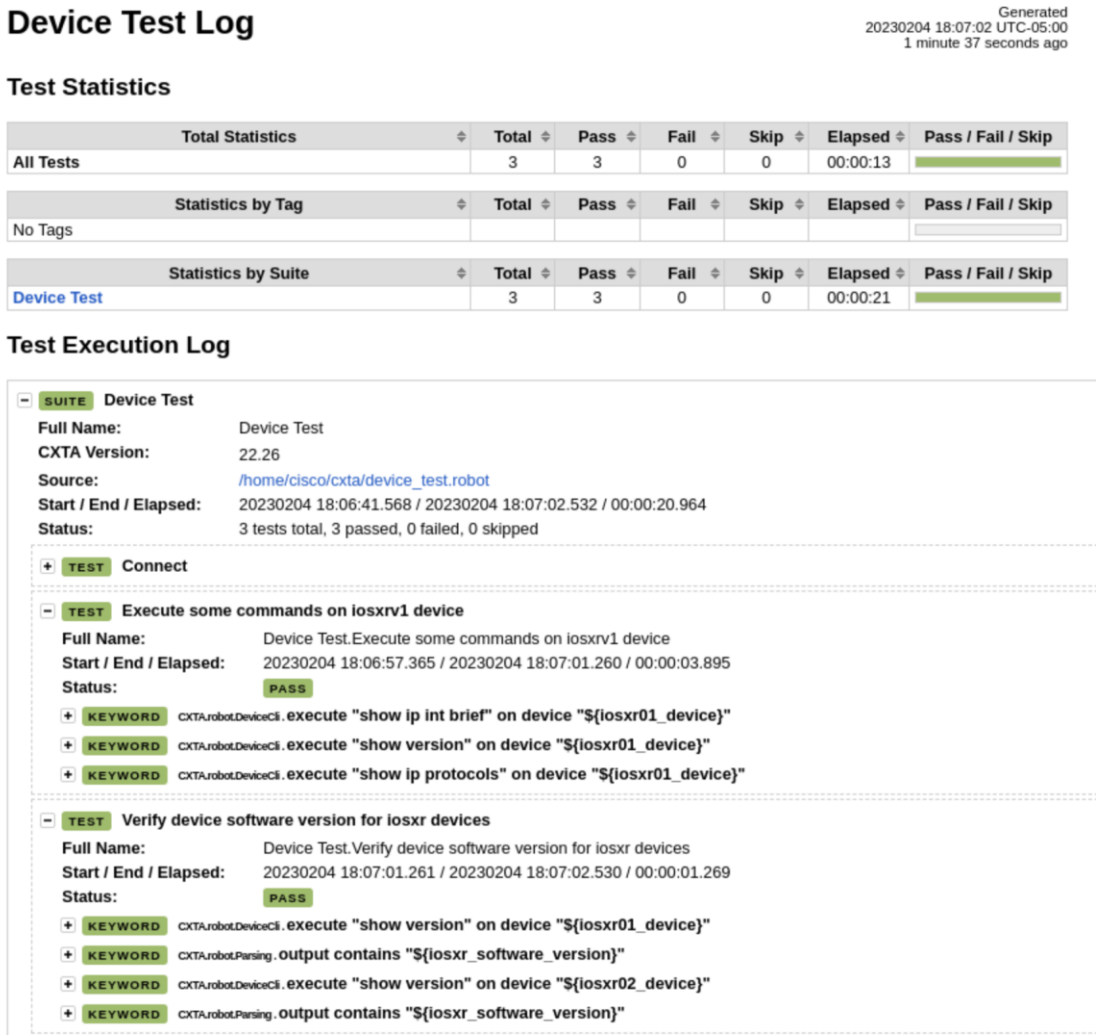
```
Add vlan in nxos device and validate
    Add Vlan to Vlan Database in NX-OS or IOS
${nxos01_device}    9-11
    execute command "show vlan" on device
"${nxos01_device}" and store in file
"configuration/vlan_config.txt"
    compare config "validation_files/vlan_config.txt" to
"configuration/vlan_config.txt"
```

5. Run the test cases from CXTA container:

```
robot device_test.robot
```

```
PROBLEMS 5   OUTPUT   DEBUG CONSOLE   TERMINAL   ROBOT DOCUMENTATION   ROBOT OUTPUT

cisco@6e95cb041675:~/cxta$ robot device_test.robot
==============================================================================
Device Test
==============================================================================
Connect                                                               | PASS |
------------------------------------------------------------------------------
Execute some commands on iosxrv1 device                               | PASS |
------------------------------------------------------------------------------
Verify device software version for iosxr devices                      | PASS |
------------------------------------------------------------------------------
Device Test                                                           | PASS |
3 tests, 3 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

6. Check again the log.html file. This is how it looks like when it is all passing:



Device Test Log

Generated
20230204 18:07:02 UTC-05:00
1 minute 37 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	3	0	0	00:00:13	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Device Test	3	3	0	0	00:00:21	

Test Execution Log

SUITE **Device Test**

Full Name: Device Test
CXTA Version: 22.26
Source: /home/cisco/cxta/device_test.robot
Start / End / Elapsed: 20230204 18:06:41.568 / 20230204 18:07:02.532 / 00:00:20.964
Status: 3 tests total, 3 passed, 0 failed, 0 skipped

TEST **Connect**

TEST **Execute some commands on iosxrv1 device**

Full Name: Device Test.Execute some commands on iosxrv1 device
Start / End / Elapsed: 20230204 18:06:57.365 / 20230204 18:07:01.260 / 00:00:03.895
Status: PASS

KEYWORD CXTA.robot.DeviceCli . **execute "show ip int brief" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show ip protocols" on device "${iosxr01_device}"**

TEST **Verify device software version for iosxr devices**

Full Name: Device Test.Verify device software version for iosxr devices
Start / End / Elapsed: 20230204 18:07:01.261 / 20230204 18:07:02.530 / 00:00:01.269
Status: PASS

KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr01_device}"**
KEYWORD CXTA.robot.Parsing . **output contains "${iosxr_software_version}"**
KEYWORD CXTA.robot.DeviceCli . **execute "show version" on device "${iosxr02_device}"**
KEYWORD CXTA.robot.Parsing . **output contains "${iosxr_software_version}"**

In this task, you configured device using a simple txt file and also you configured a device and verified that the expected configuration is present.

Task 4: Negative Test Cases

In general we are happy when our Test Cases are all green and indicate PASS. What is happening though when the configuration is different that the expected one? In this case, it is important to have a good way of identifying the faulty configuration parts. The log.html file that is created every time that you are running a Robot test case is helpful on that.

1. Go to the Test Case that you created before in order to verify the iosxr Software Version:

```
Verify device software version using Keyword
```

2. This time the expected software version of the iosxrv-2 device is 7.3.3, that value is stored under the variable “iosxr02_software_version”. Change the test case to correspond to that:

```
Verify device software version using Keyword
    Connect to device "${iosxr01_device}" and verify is running
"${iosxr_software_version}"
    Connect to device "${iosxr02_device}" and verify is running
"${iosxr02_software_version}"
```

3. Configuration comparison fails
    a. Create a new Test Case with name “Verify vlan configuration”

```
Verify vlan configuration
```

    b. Use the compare config keyword you used also in Task 3, Step 4d, this time the expected configuration will be under validation folder and in the vlan_config_negative.txt file. Test Case looks like:

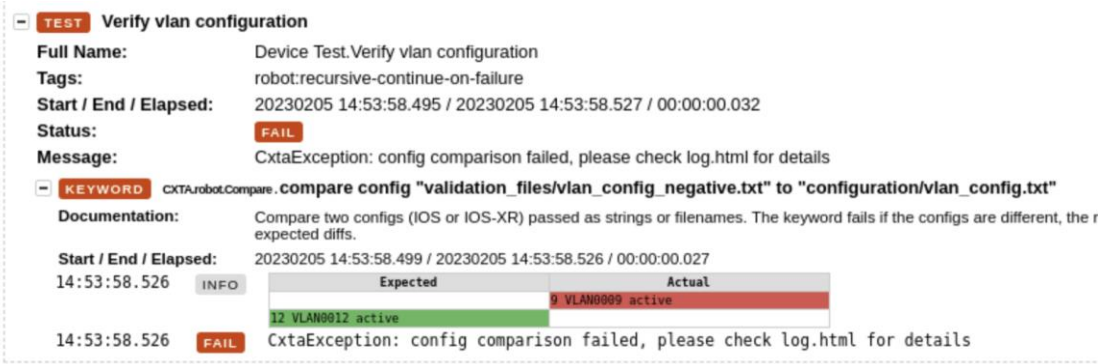
```
Verify vlan configuration
        compare config
"validation_files/vlan_config_negative.txt" to
"configuration/vlan_config.txt"
```

4. Run robot test cases:

```
PROBLEMS 5   OUTPUT   DEBUG CONSOLE   TERMINAL   ROBOT DOCUMENTATION   ROBOT OUTPUT

Verify device software version for iosxr devices                      | PASS |
------------------------------------------------------------------------------
Verify device software version using Keyword                          | PASS |
------------------------------------------------------------------------------
Verify software version using iosxr Keywords                          | FAIL |
No keyword with name 'Verify Software Release' found.
------------------------------------------------------------------------------
[ WARN ] Multiple tests with name 'Verify device software version using Keyword' executed in suite 'Device Test'.
Verify device software version using Keyword                          | FAIL |
CxtaException: Output does not contain "7.3.3"!
------------------------------------------------------------------------------
Verify vlan configuration                                             | FAIL |
CxtaException: config comparison failed, please check log.html for details
------------------------------------------------------------------------------
Device Test                                                           | FAIL |
7 tests, 4 passed, 3 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

   As you can see, the robot test cases Failed. In the first case it is clearly stated even from CLI the reason of Failure. It is informing the user that the Output does not contain 7.3.3 that is our required version.

5. Open the log.html file and observe the output under the Verify vlan configuration Test Case:



   The log.html report is informin you that the entry for VLAN 12 that was expected is missing from the configuration and the configuration contains also the VLAN 9, that was not expected to be configured.

Nice job! Now you have also explored how you can user Robot and CXTA Library in order to:

1. Configure devices
2. Display devices' configuration
3. Verify device configuration
4. Create your own Keyword
5. Read and get information of log.html files.

Exercise 3: ACI REST Test Cases

In this exercise, you will explore how CXTA can interact with ACI using REST API calls. Application Centric Infrastructure (ACI) is Cisco's core Software-Defined Networking (SDN) solution for the data center. The infrastructure is centrally managed by a cluster of controllers, the Application Policy Infrastructure Controllers (APICs).

Task 1: Verify overlay-1 VRF Configuration

1. Open aci_test_cases.robot file:

CLEMEA23_LTROPS-1964 > aci_test_cases.robot

```
*** Settings ***
Library     CXTA
Resource    cxta.robot
Suite Setup    setup-test

*** Variables ***
${testbed}    testbed.yaml
${apic}       apic1
${nxos01_device}    NX-OS-1

*** Test Cases ***

Verify in-band BD Configuration
    ${tenant_name} =  Set Variable  mgmt
    ${bd_name} =  Set Variable  inb
    ${uri} =  Set Variable  /api/mo/uni/tn-${tenant_name}/BD-${bd_name}
    ACI REST login on "${apic}"
    @{return}=  via ACI REST API retrieve "${uri}" from "${apic}" as "xml"
    Should Be Equal as Integers     ${return}[0]    200
    Should Contain  ${return}[1]  <imdata totalCount="1">
    Should Contain  ${return}[1]  dn="uni/tn-mgmt/BD-inb"


ACI logout
    ACI REST logout on apic1

*** Keywords ***
setup-test
    load testbed "${testbed}"

```

Observe that in this Exercise, the communication of Robot with ACI is happening using REST API call. Using the Keywords: Should Contain and Should Be Equal as Integers user can validated the response of the call.

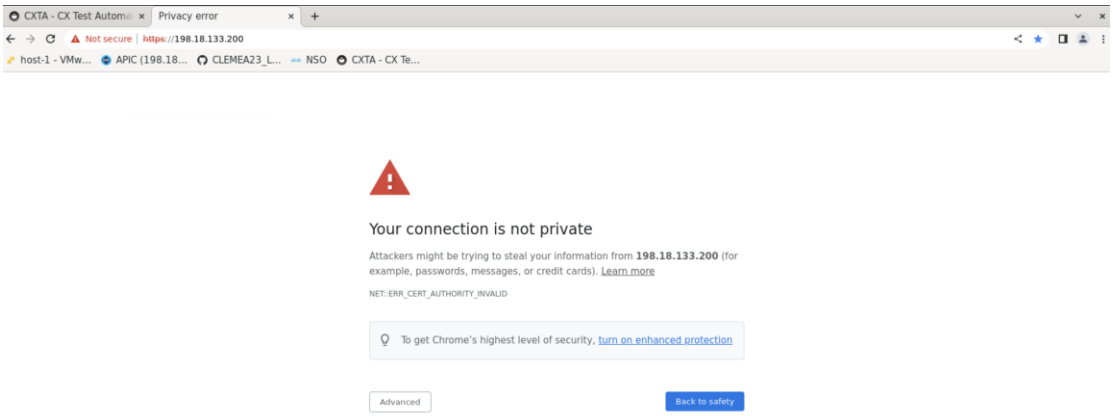
That test case verifies that the Domain Bridge "inb" belongs to tenant with name "mgmt". Later on, when the test is executed, there is a logout of CXTA from apic1.

2. Run the test case:

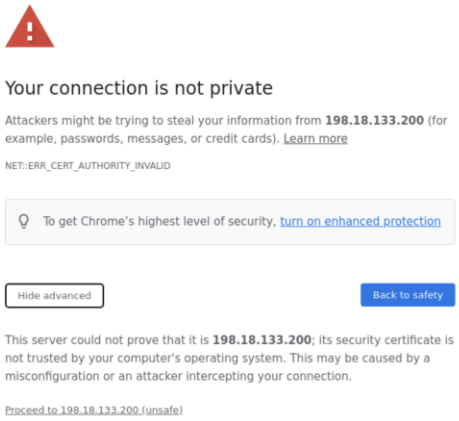
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ROBOT DOCUMENTATION  ROBOT OUTPUT
cisco@6e95cb041675:~/cxta$ robot aci_test_cases.robot
==============================================================================
Aci Test Cases
==============================================================================
Verify in-band BD Configuration                                       | PASS |
------------------------------------------------------------------------------
ACI logout                                                            | PASS |
------------------------------------------------------------------------------
Aci Test Cases                                                        | PASS |
2 tests, 2 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

Since all test cases are passing. There is the bridge domain with in APIC with name apic1.

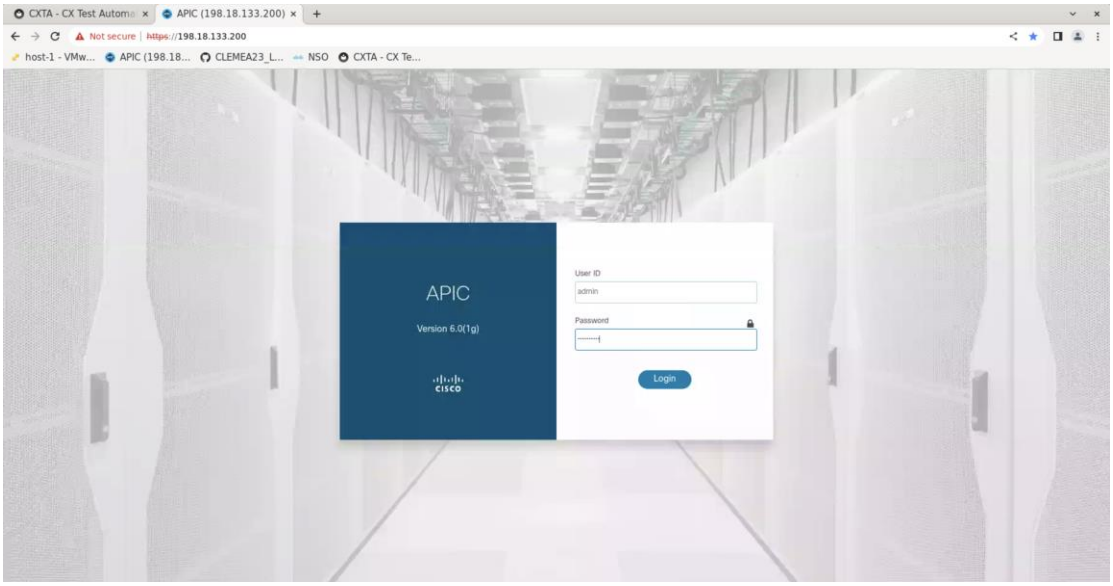
3. Verify the test using APIC web page. Access to APIC using Google Chrome. Open a new tab and type the APIC IP in the browser: 198.18.133.200. This page will appear:



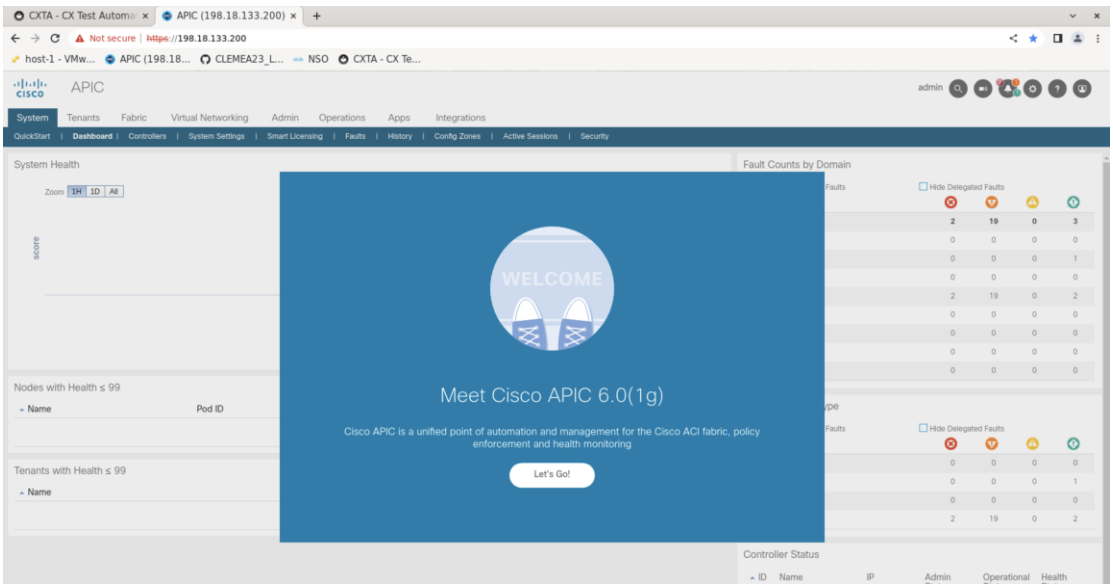
Push the Advanced button:



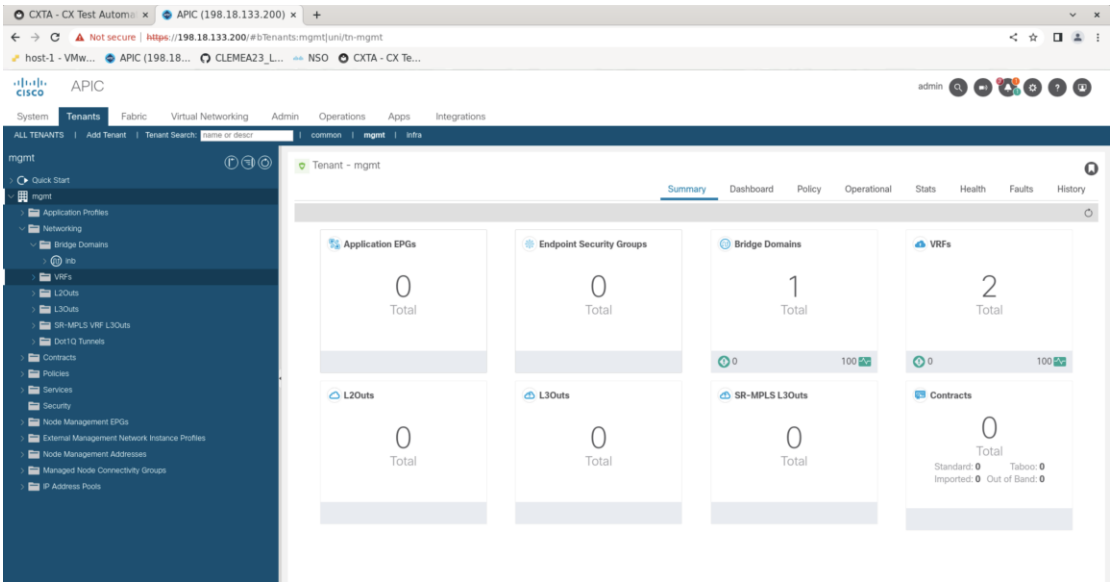
Select : Proceed to 198.18.133.200 (unsafe). It will take you to APIC page:



Insert the same credentials as in the testbed : Username: admin, Password: C1sco12345 and Login:



Navigate to Tenants → mnmt → Networking → Bridge Domains and you will see inb to be there:

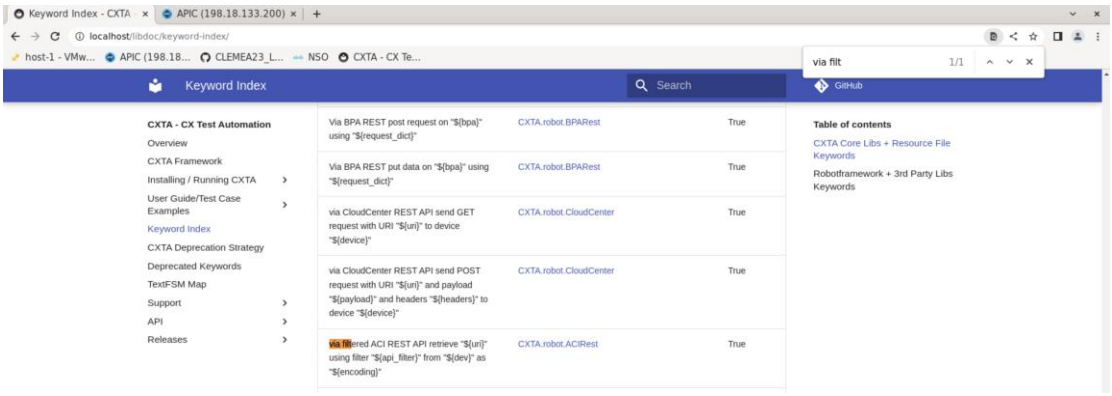


Task 2: Verify in-band Bridge Domain Configuration

1. Go back to the file with name aci_test_cases.robot. Create a new Test Case with name: Verify overlay-1 VRF Configuration.

```
Verify overlay-1 VRF Configuration
```

2. That test case will have to examine if an overlay is present in VRF configuration. For that we will need to filter the API Response. Using that information, search in Keywords Index and find the Keyword to be used:



3. You can observce that the Keyword is expecting 4 input parameters:
    a. uri (the uri towards which will send the call)
    b. api_filter (how to filter the response)
    c. dev (device name)
    d. encoding (json or xml, the expected encoding)
4. Define the above variables inside the Test Case:
    a. Device that is receiving the Request is apic1,
    b. Both the response and the request will be in json encoding,
    c. The filter based on ACI API documentation is :

```
query-target-filter=eq(fvCtx.name,"${vrf}")
```

    d. The uri based on ACI API documentation is :

```
/api/class/fvCtx
```

Your test case so far should look like:

```
Verify overlay-1 VRF Configuration
    ${vrf} =  Set Variable  overlay-1
    ${filter} =  Set Variable  query-target-
filter=eq(fvCtx.name,"${vrf}")
    ${uri} =  Set Variable  /api/class/fvCtx
```

5. Use the Keyword you explored before in order to filter the REST API response. Save the output of the Keyword to a variable with name result:

```
    @{return}=  via ACI REST API configure device
"${apic}" at URI "${uri}" using "json" payload
"${payload}"
```

Studying the documentation of the Keyword, you can see that the output of will contain two parameters:

via filtered ACI REST API retrieve "${uri}" using filter "${api_filter}" from "${dev}" as "${encoding}"

Documentation

Retrieves an URI from ACI using REST using an ACI API filter.

Login on the APIC controller and maintenance of the login session are implicitly taken care of by this method. It is therefore not needed to perform a login on the APIC before using this method.

Parameters:	**param dev:**	(str) APIC device name as defined in testbed.yaml
	param uri:	(str) ACI API URI excluding filters (example: /api/mo/uni/tn-<tenant_name>)
	param api_filter:	
		(str) ACI API filter (example: query-target-filter=eq(fvCtx.name,"overlay-1")
	param encoding:	(str) json, xml or object
Returns:	**returns:**	(list) The function returns a list of two values: the return code (i.e. 201, 404, etc.) and the returned text.

This method handles API session creation using the APIC's REST address/port and credentials from the unicon testbed (ex: "testbed.yaml"), so it needs to be loaded first (using 'load testbed ...')

It returns the status code and return text as a list if the encoding is json or xml. If object is set as encoding type, one single object is returned. This object has a property status indicating the operation status (200 = success). The payload property contains the data return as 'imdata' in object/property format. Should the property have a new that is also a PYTHON keyword, the name will be prefixed with an underscore (e.g. "_if").

Returns the return code and the returned text. That is the case that the output of the Keyword is a list. The annotation to access the list elements in robot is:

```
${list}[position_of_element_in_list]
```

The positions start from element 0. So if for example you want to access the first element of the a list with name rest_response, you will do like:

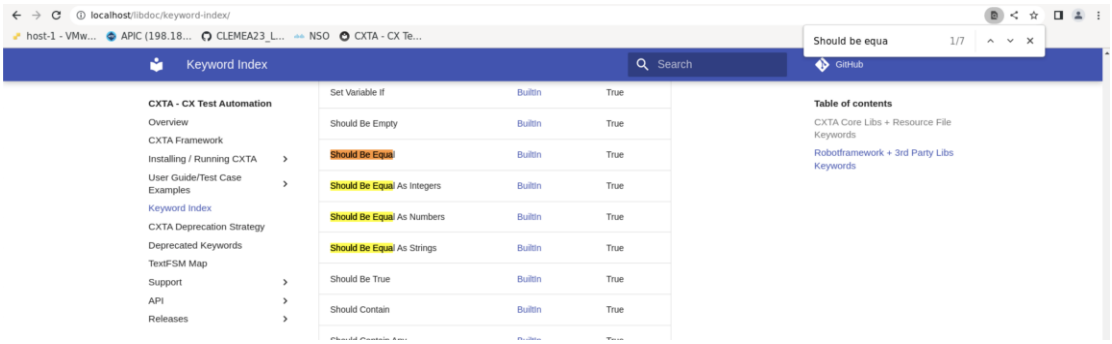
```
${rest_response}[0]
```

6. Verify the Response Code of the Keyword was OK = 200.

Since the ${return} parameter, in this case, is a list and contains two values, in order to access the content of the first parameter use the annotation:

```
${return}[0]
```

This way, you will be able to see the return code. Search for a Keyword that is validating Integers:



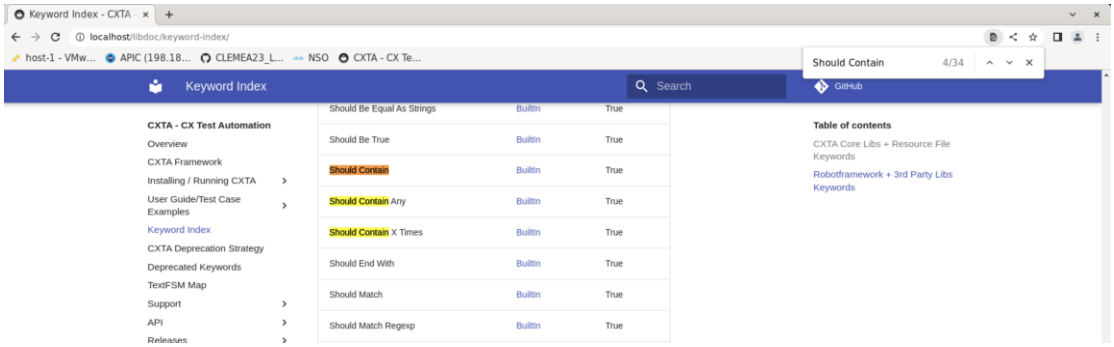
Now your test case should look like:

```
Verify overlay-1 VRF Configuration
    ${vrf} =  Set Variable  overlay-1
    ${filter} =  Set Variable  query-target-
filter=eq(fvCtx.name,"${vrf}")
```

```
    ${uri} =  Set Variable  /api/class/fvCtx
    @{return}=  via filtered ACI REST API retrieve "${uri}" using
filter "${filter}" from "${apic}" as "json"
    Should Be Equal as Integers     ${return}[0]    200
```

7. Verify the Response text includes "dn":"uni/tn-infra/ctx-overlay-1".

For that use the Keyword: “Should Contain” as you can see in the list below:



Use the same keyword to verify that there is only one entry. That means that the totalCount variable of the response body contains: “"totalCount":"1"”. Your test case now looks like:

```
Verify overlay-1 VRF Configuration
    ${vrf} =  Set Variable  overlay-1
    ${filter} =  Set Variable  query-target-
filter=eq(fvCtx.name,"${vrf}")
    ${uri} =  Set Variable  /api/class/fvCtx
    @{return}=  via filtered ACI REST API retrieve "${uri}" using
filter "${filter}" from "${apic}" as "json"
    Should Be Equal as Integers     ${return}[0]    200
    Should Contain  ${return}[1]  "totalCount":"1"
    Should Contain  ${return}[1]  "dn":"uni/tn-infra/ctx-overlay-
1"
```

8. Now test case is ready, execute the test cases using the cxta container:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ROBOT DOCUMENTATION  ROBOT OUTPUT

cisco@6e95cb041675:~/cxta$ robot aci_test_cases.robot
==============================================================================
Aci Test Cases
==============================================================================
Verify in-band BD Configuration                                       | PASS |
------------------------------------------------------------------------------
Verify overlay-1 VRF Configuration                                    | PASS |
------------------------------------------------------------------------------
ACI logout                                                            | PASS |
------------------------------------------------------------------------------
Aci Test Cases                                                        | PASS |
3 tests, 3 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

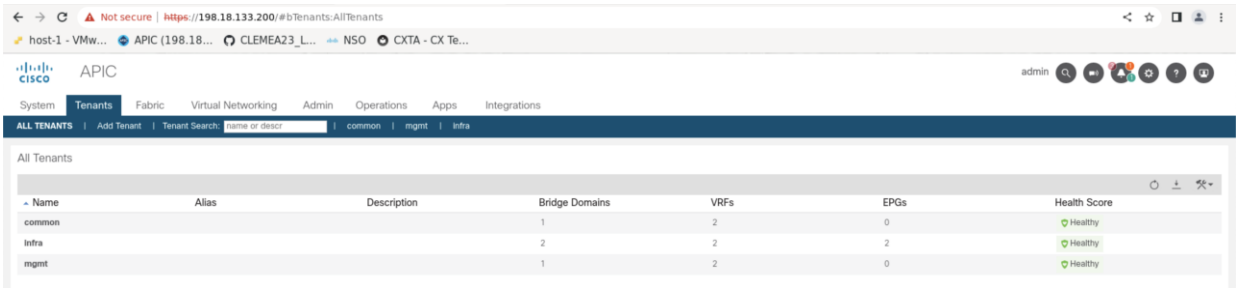
9. Open log.html file and observe how the output of the API call looks like:

Completing that tasks, you explored how to use output variables that are lists in order to verify their content.

Task 3: Configure Tenant

In this Task, you will configure a new Tenant in APIC using REST API call through CXTA.

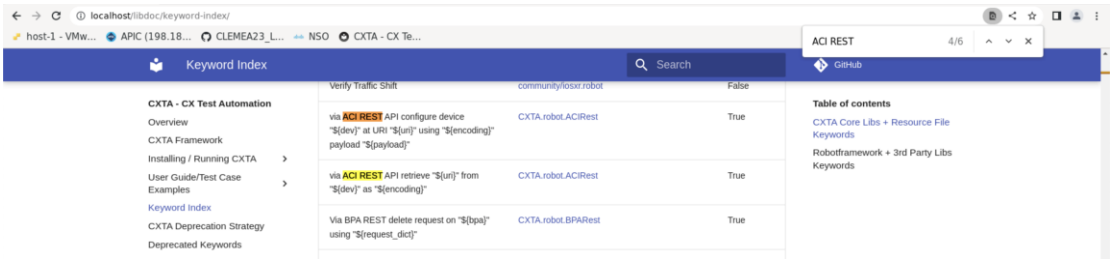
1. Go back to APIC Graphical Interface and in the tenant page observe how many tenants exist and what are their names:



2. Create a new test case with name: “Configure Tenant”:

```
Configure Tenant
```

3. Search in the Keyword Index for the Keyword to configure a device through ACI:



The necessary inputs for the Keyword are:

a. dev (device name)

b. uri (the uri towards which will send the call)

c. encoding (json or xml, the expected encoding)

d. payload (the content of the call – when it is necessary)

4. Since nothing has changed, device name and encoding will be the same. Now uri and payload are changed since different parts of APIC are going to be used. For uri use: “/api/mo/uni”.

   In order to create a Tenant with name CXTA-TESTING, as a payload you need to provide:

```
{"fvTenant": {"attributes": {"name": "CXTA-TESTING", "status":
""} } }
```

5. Bringing everything together until now your test case looks like:

```
Configure Tenant
    ${payload}=  Set Variable  {"fvTenant": {"attributes":
{"name": "CXTA-TESTING", "status": ""} } }
    ${uri} =  Set Variable  /api/mo/uni
    @{return}=  via ACI REST API configure device "${apic}" at
URI "${uri}" using "json" payload "${payload}"
```

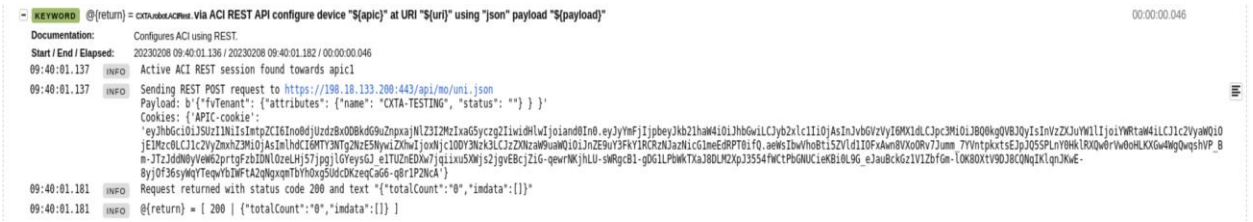
6. In order to verify that the call went to APIC and everything was OK, the response code should be 200 and there should not exist any message in the response body. As before use the same Keywords in order to validate the 200 code and the response body. Since now there will not be a response body, expect in the total_count to have 0 and in the imdata to be an empty list:

```
Configure Tenant
    ${payload}=  Set Variable  {"fvTenant": {"attributes":
{"name": "CXTA-TESTING", "status": ""} } }
    ${uri} =  Set Variable  /api/mo/uni
    @{return}=  via ACI REST API configure device "${apic}" at
URI "${uri}" using "json" payload "${payload}"
    Should Be Equal as Integers     ${return}[0]    200
    Should Contain  ${return}[1]
{"totalCount":"0","imdata":[]}
```

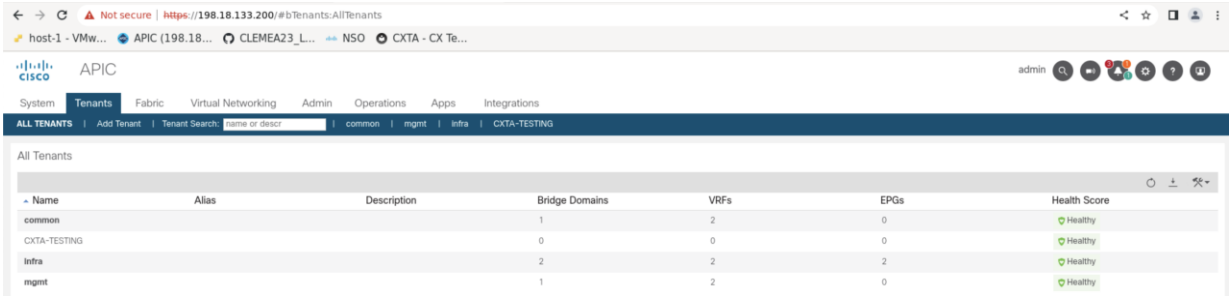
7. Run the test cases:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ROBOT DOCUMENTATION  ROBOT OUTPUT

cisco@6e95cb041675:~/cxta$ robot aci_test_cases.robot
==============================================================================
Aci Test Cases
==============================================================================
Verify in-band BD Configuration                                       | PASS |
------------------------------------------------------------------------------
Verify overlay-1 VRF Configuration                                    | PASS |
------------------------------------------------------------------------------
Configure Tenant                                                      | PASS |
------------------------------------------------------------------------------
ACI logout                                                            | PASS |
------------------------------------------------------------------------------
Aci Test Cases                                                        | PASS |
4 tests, 4 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

8. Investigate the log.html file:



9. Go back to APIC Graphical Interface and verify that the tenant was added:



Well done! After finishing this exercise you are familiar with:

1. Ouput manipulation when output is a list,
2. Using ACI Keywords to Verify and Configure elements using REST API calls,

Since the purpose of the exercise is to verify objects of ACI using REST and not particularly the ACI API specifications, the necessary inputs are provided. If you would like to explore ACI API specifications, please visit developer.cisco.com so that you get more information regarding ACI documentation.

Exercise 4: NSO Test Cases

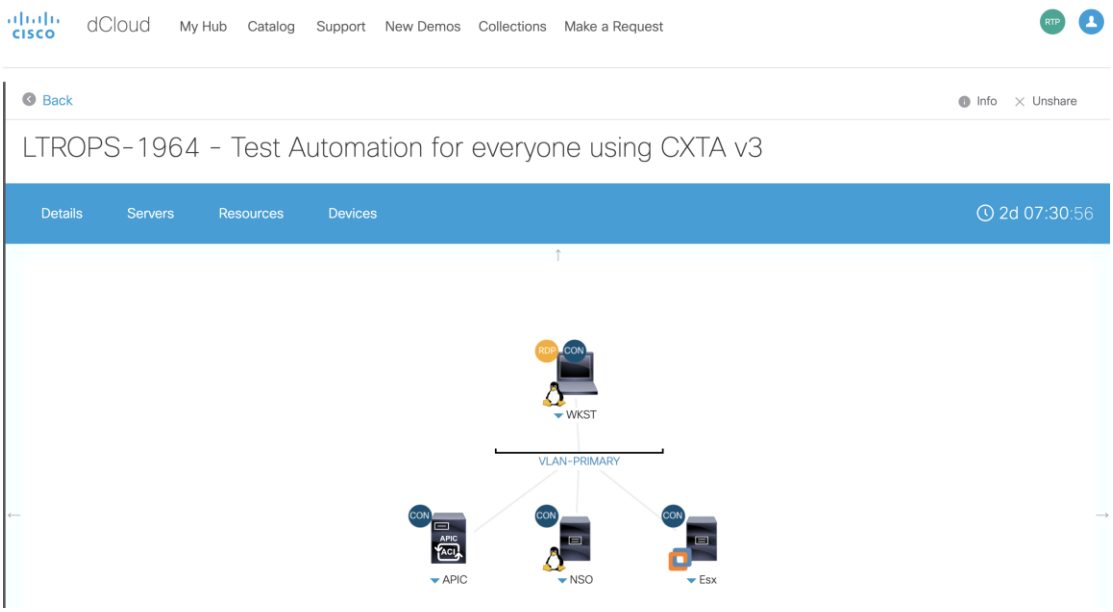
Let's explore NSO testing and CXTA basic capabilities regarding checking NSO executed services directly on devices.
This kind of test might be very useful not only for NSO services tests but also for Compliance Testing and connectivity checks if tools like ping or traceroute are used.
In complex scenarios (not covered in this guide) test case input might be generated automatically based on NSO CDB data or any other data source.

Task 1: Create basic Robot file

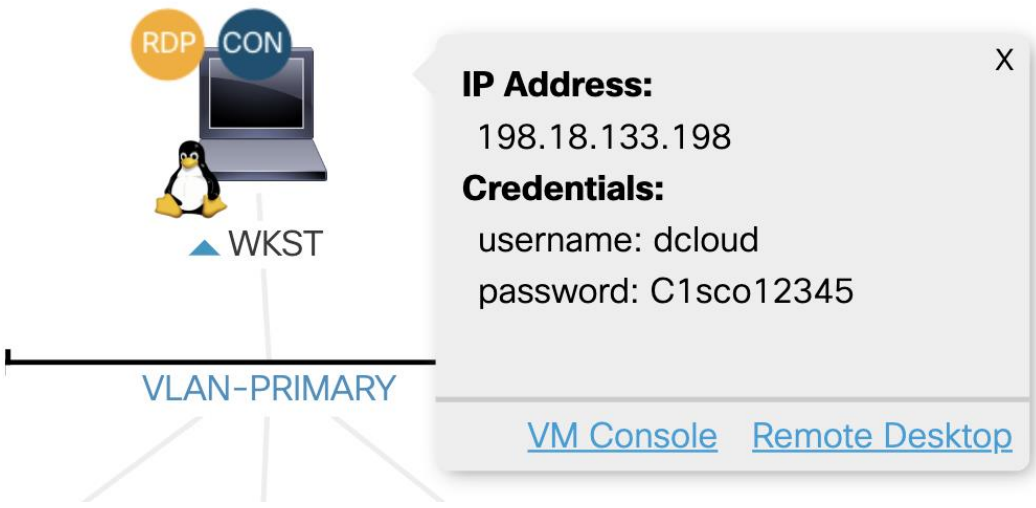
This is a preparation task that prepares nso_test_cases.robot file and its structure.

Step 1: Open Visual Studio Code and create file

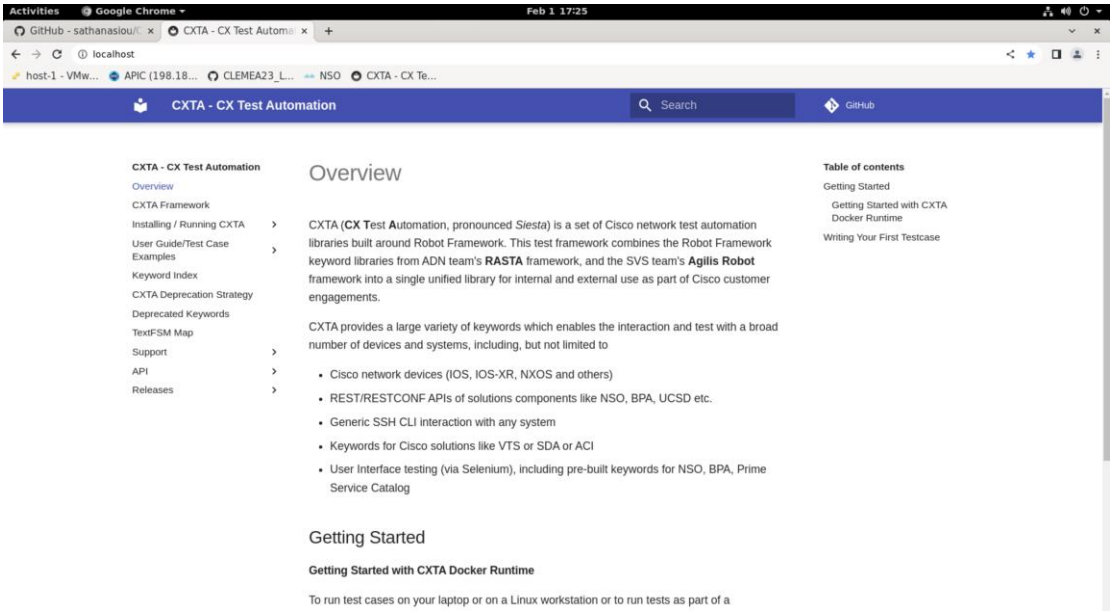
1. Use Dcloud interface to see the topology



2. **O**pen WKST object



3. Select Remote Desktop to access Workstation interface



4. Create new test file (menu bar: File => New Text File)
5. Save it as 'nso_test_cases.robot'

Step 2: Create Robot structure inside the file

1. Create '*** Settings ***' section
2. Insert CXTA Library and Resources

```
*** Settings ***
Library     CXTA
Resource    cxta.robot
```

3. Create '*** Variables ***' section
4. Insert '${nso} variable pointing to nso structure in testbed file

```
*** Variables ***
${nso}          nso
```

5. Create '*** Test Cases ***' and '*** Keywords***' sections

```
*** Test Cases ***

*** Keywords ***
```

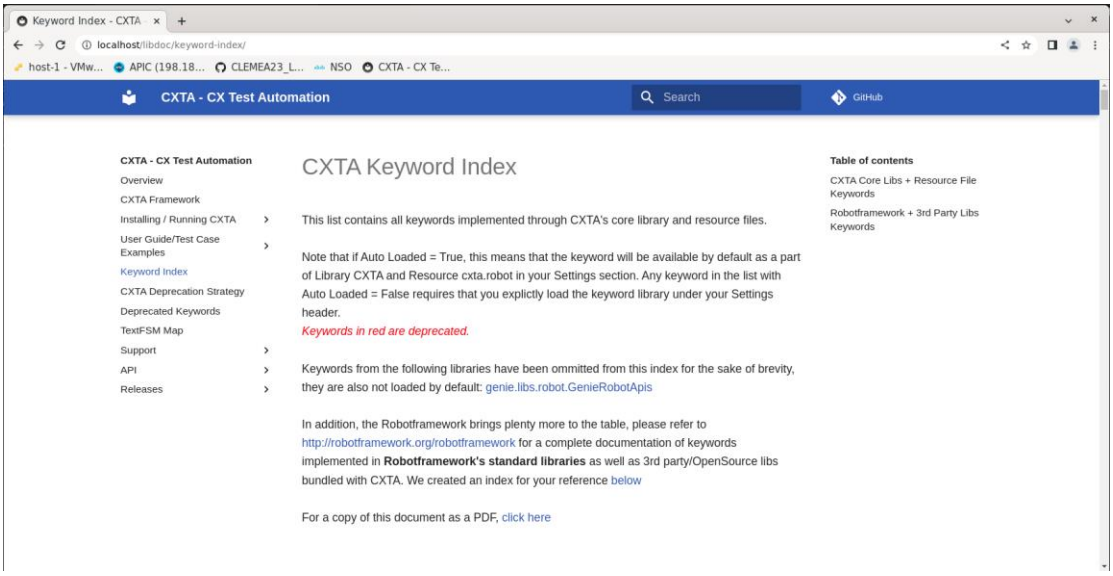
Task 2: Create Setup and Teardown Keywords and attach them to Test Suite

The purpose of this task is to create appropriate Setup and Teardown sections for NSO test to be able to setup the suite, check connectivity and rollback NSO and network state to previous state when tests are done.

Step 1: Create 'Suite Setup Actions' Keyword and set to be 'Suite Setup'

1. Add 'Suite Setup Actions' keyword in Keywords section

2. Use ‘Keywords Index’ in CXTA Documentation to find proper keywords (can be accessed using Chrome in WKST Remote Desktop):



3. Add keywords that
    a. Point to testbed file
    b. Catch and store latest NSO commit
    c. Connect to all devices
    d. Set NSO CLI to Cisco

```
Suite Setup Actions
    use testbed "testbed.yaml"
    retrieve latest NSO rollback number from "${nso}"
    connect to all devices
    Set NSO cli style to "cisco"
```

4. Add Setup Suite to ‘*** Resources ***’ section

```
Suite Setup       Suite Setup Actions
```

Step 2: Create ‘Suite Teardown Actions’ Keyword and set it to be ‘Suite Teardown’

1. Add ‘Suite Teardown Actions’ keyword in Keywords section
2. Add keyword that rollbacks NSO to state captured in Suite Setup

```
Suite Teardown Actions
    rollback NSO "${nso}" to rollback retrieved
```

3. Add Setup Suite to ‘*** Resources ***’ section

```
Suite Teardown      Suite Teardown Actions
```

Step 3: Test Setup and Teardown actions

NOTE: As Suite has to contain at least one testcase, an empty testcase needs to be created for this test. This test case should be removed when test is finished

1. Add empty testcase in ‘*** Test Cases ***’ section

```
Empty test case
    Log to Console      \nHello World!!!
```

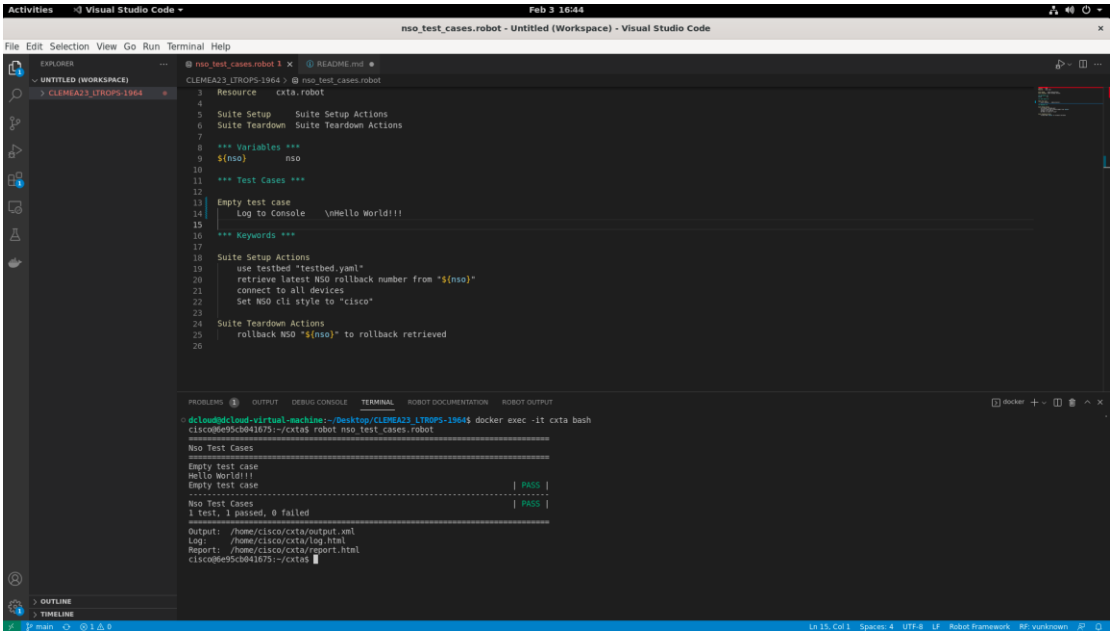
2. Save the file
3. Go to Terminal window
4. Start bash shell in CXTA container

```
dclould@dcloud-virtual-machine:~$ docker exec -it cxta bash
```

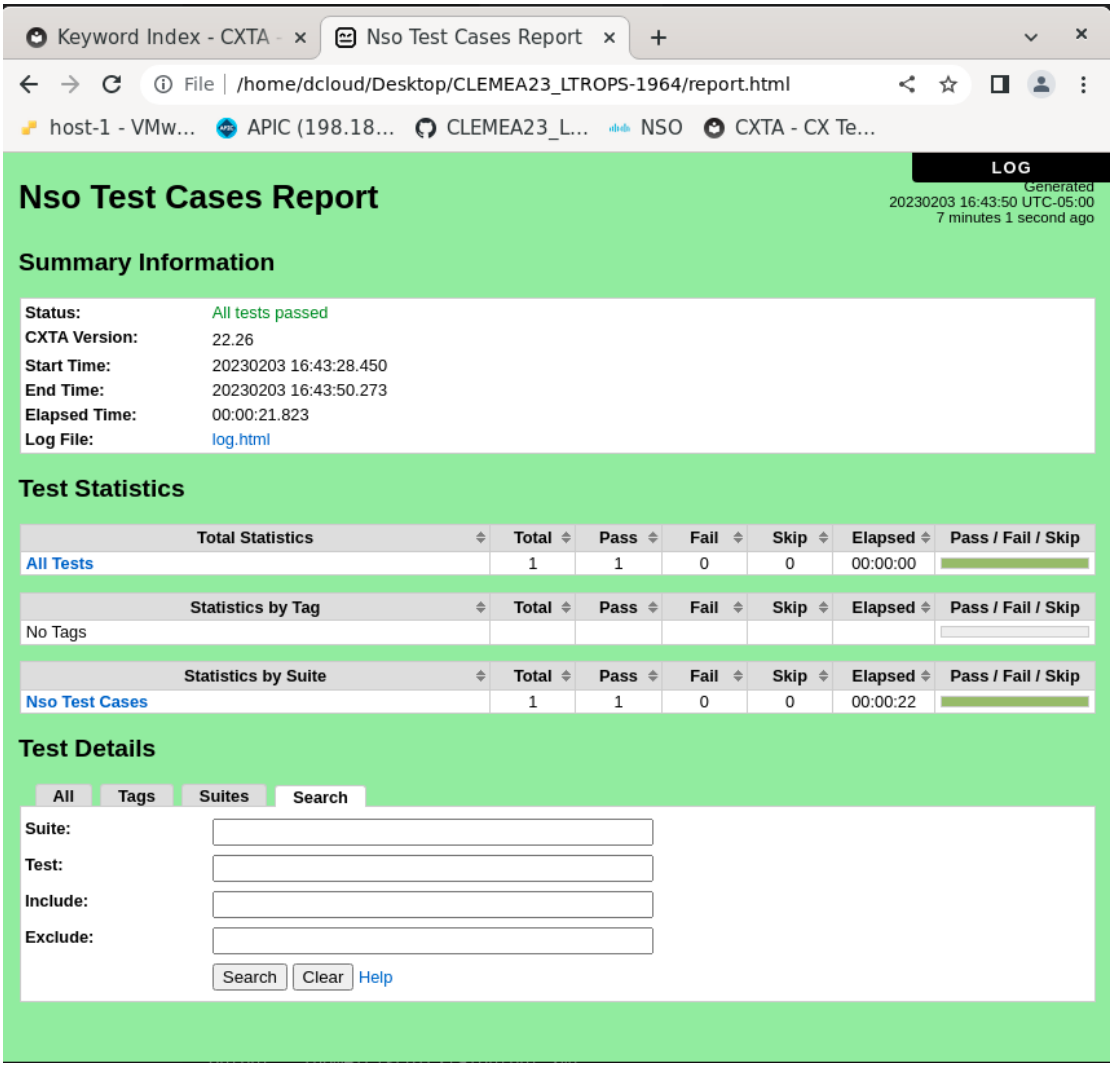
5. Execute test suite

```
cisco@1234567890abcdef:~/cxta$ robot nso_testcases.robot
```

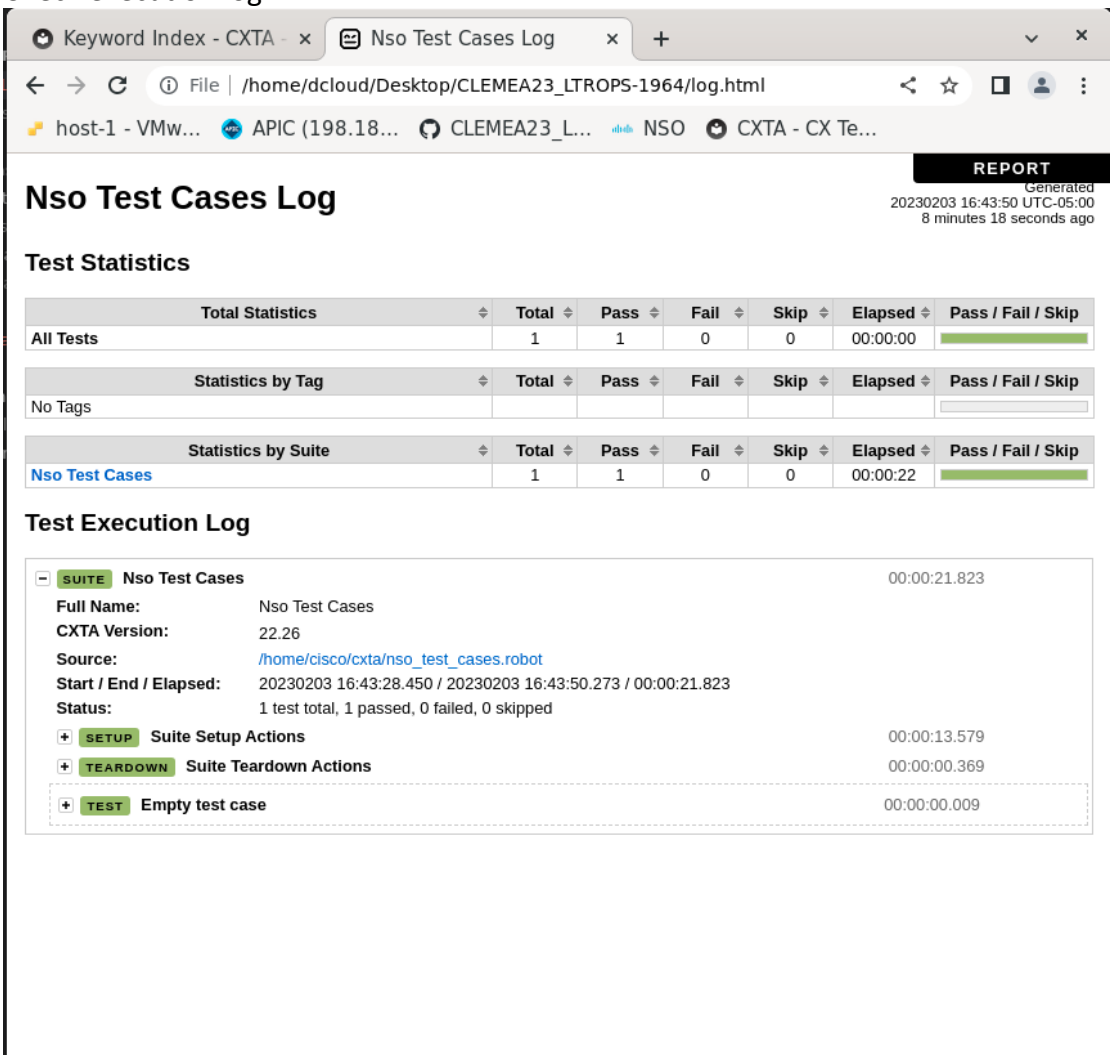
6. Check the results



7. Check results.html in the browser



8. Check execution log



9. Drill down SETUP and TEARDOWN sections to see the process step-by-step

Test Execution Log

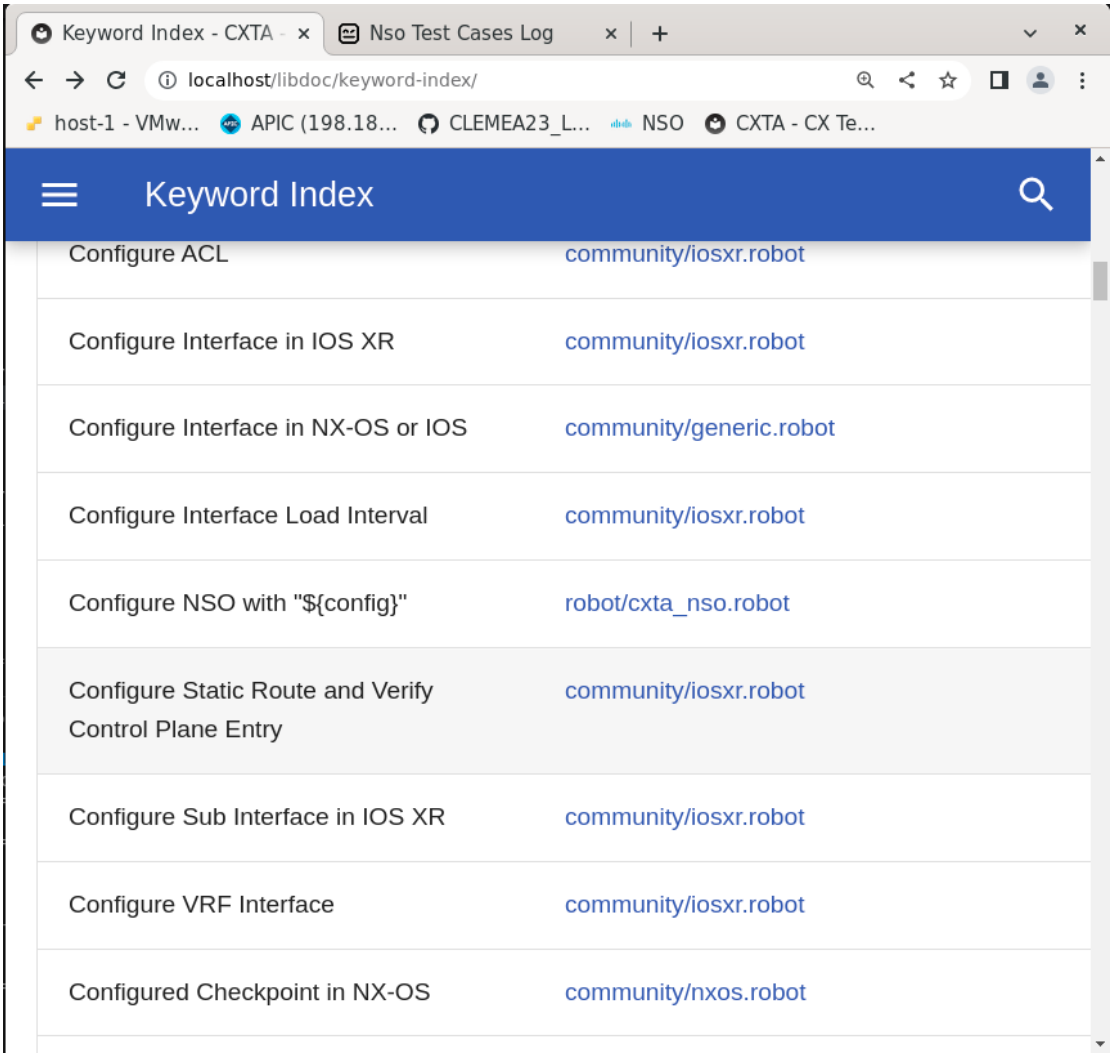
Task 3: Create and execute First NSO Test Case

This task covers simple NSO service test (if-description NSO package allows to set description on GigabitEthernet interface). Results are confirmed by doing SSH connection and show command on the target router.

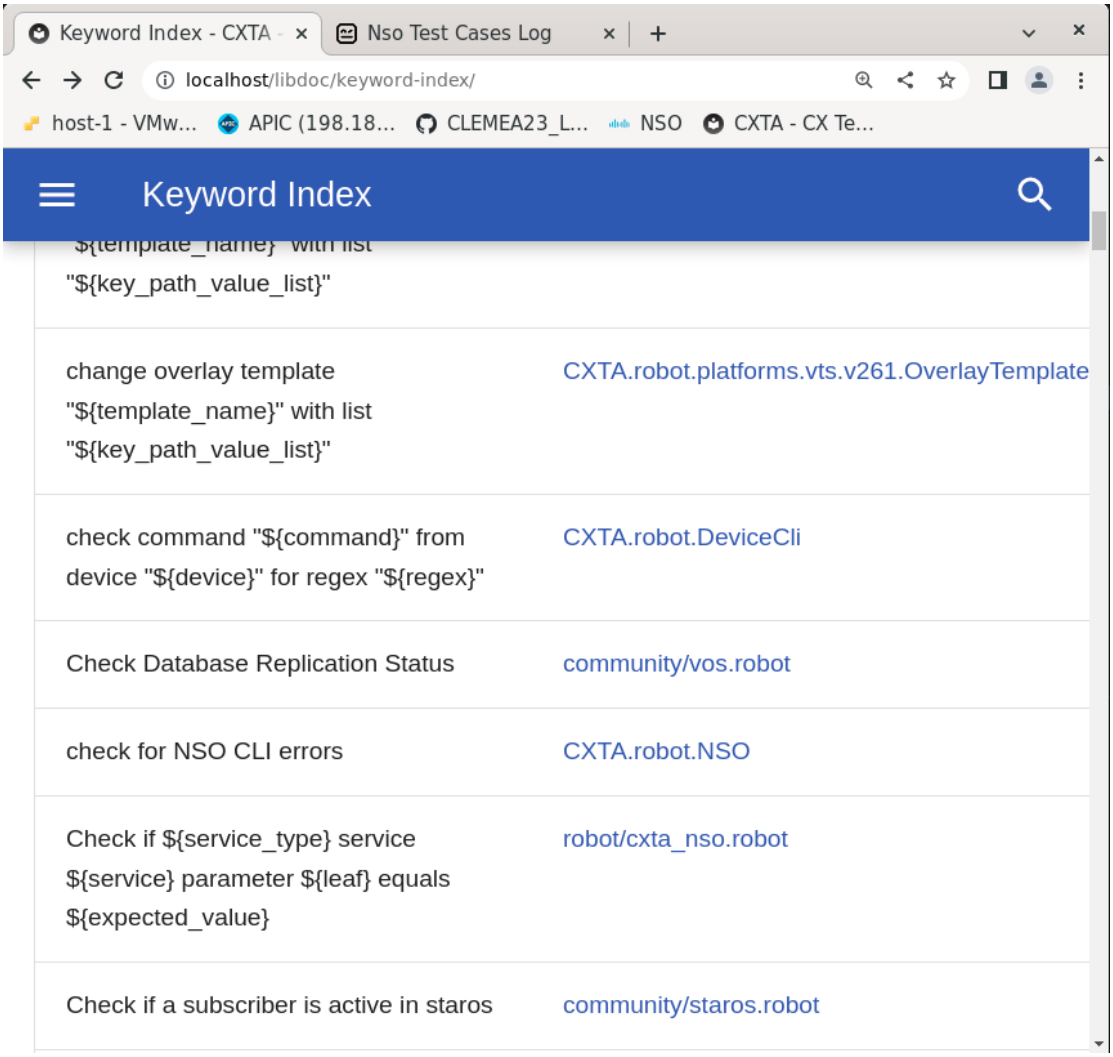
Step 1: Create new keyword and define test case

1. Add ‘NSO if-description Test case 1’ to ‘Test Cases’ section

2. Find 'Configure NSO with “${config}”’ command in Keywords Index and review details



3. Find ‘check command “${command}” from device “${device}” for regex “${regex}”’ and review details



4. Write test case in using these commands
    a. NSO configuration command: “services if-description Test1 device iosxrv-1 interface GigabitEthernet id 0/0/0/1 description Test1”
    b. IOS-XR show command: “show interfaces GigabitEthernet 0/0/0/1 description | include 0/0/0/1" from device "iosxrv-1”

```
NSO if-description Test case 1
    Configure NSO with “services if-description Test1 device
iosxrv-1 interface GigabitEthernet id 0/0/0/1 description
Test1”
    check command “show interfaces GigabitEthernet 0/0/0/1
description | include 0/0/0/1” from device “iosxrv-1” for regex
“GigabitEthernet 0/0/0/1 Test1”
```

5. Save the file

Step 2: Execute test

1. Go to Terminal window
2. Start bash shell in CXTA container

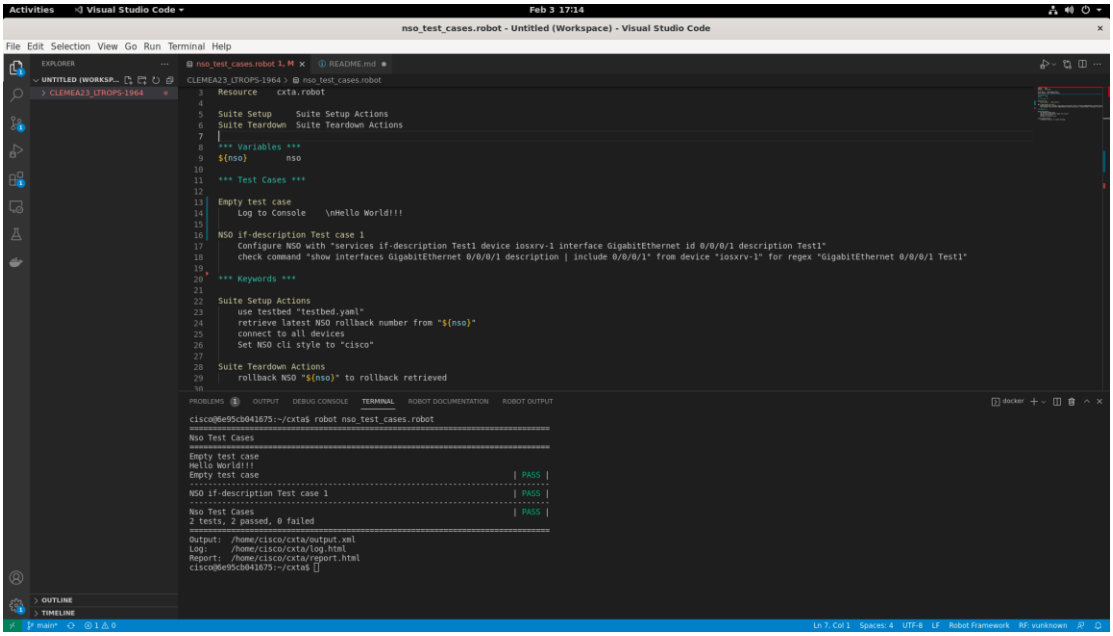
```
dclould@dcloud-virtual-machine:~$ docker exec -it cxta bash
```

3. Execute test suite

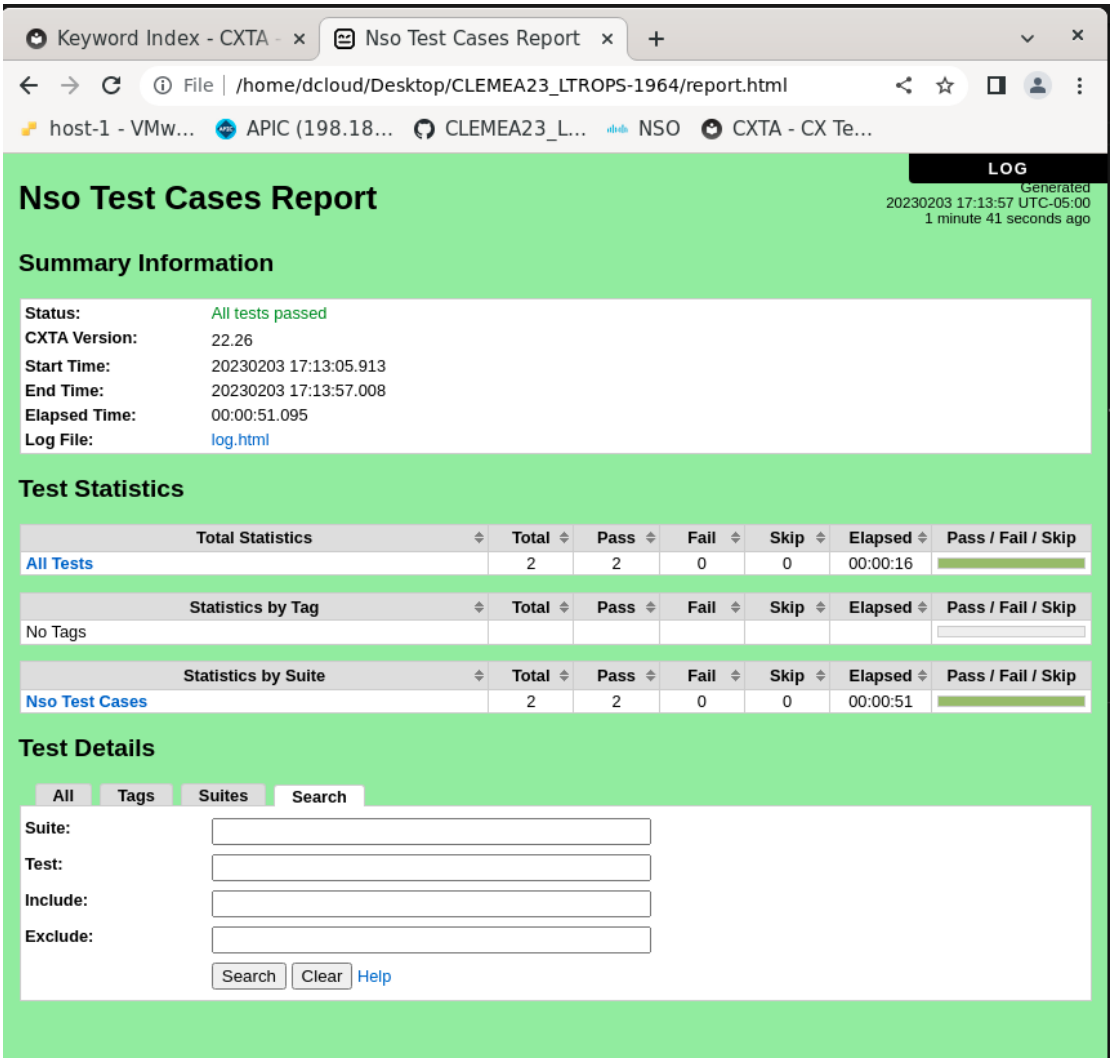
```
cisco@1234567890abcdef:~/cxta$ robot nso_testcases.robot
```

Step 3: Review Results

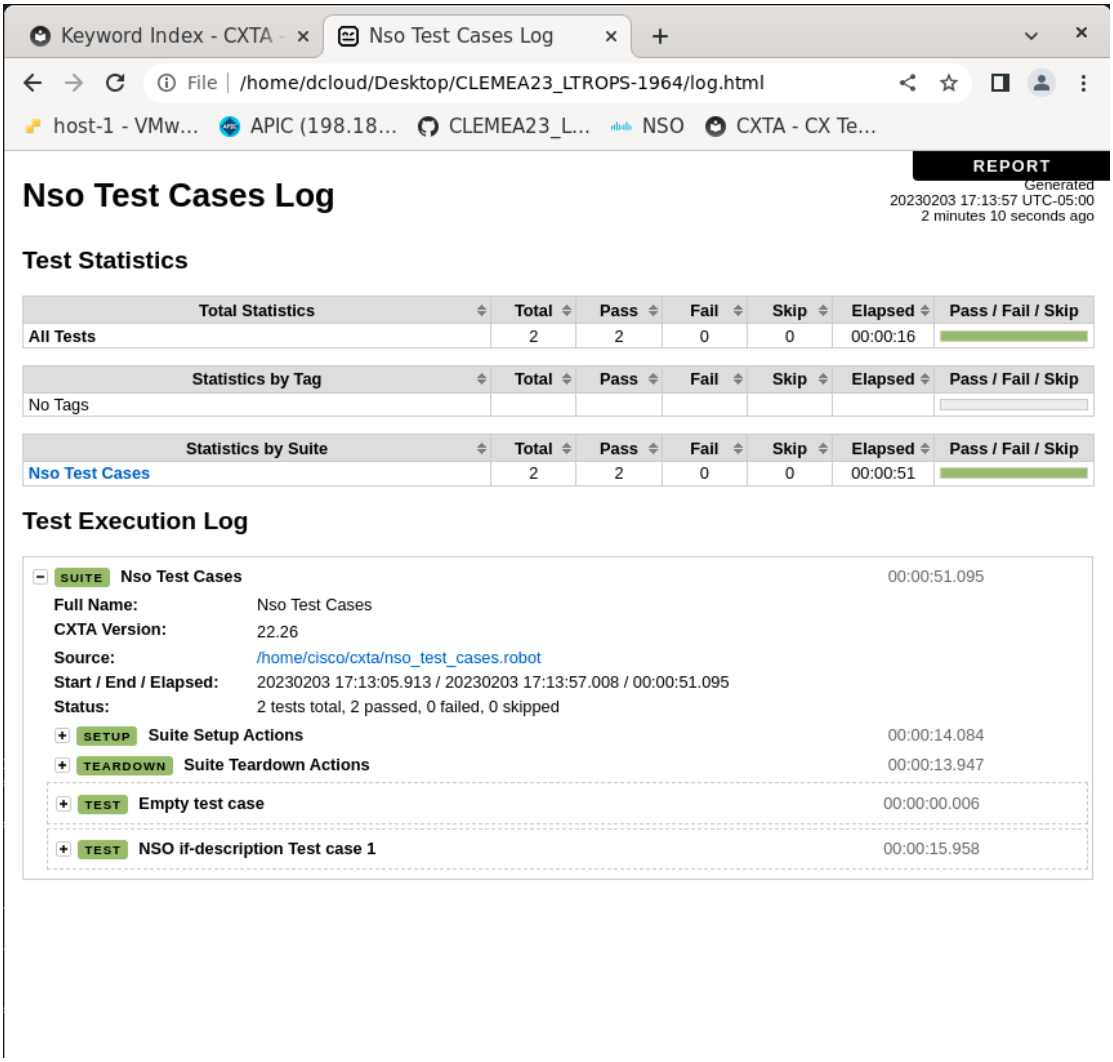
1. Check the results



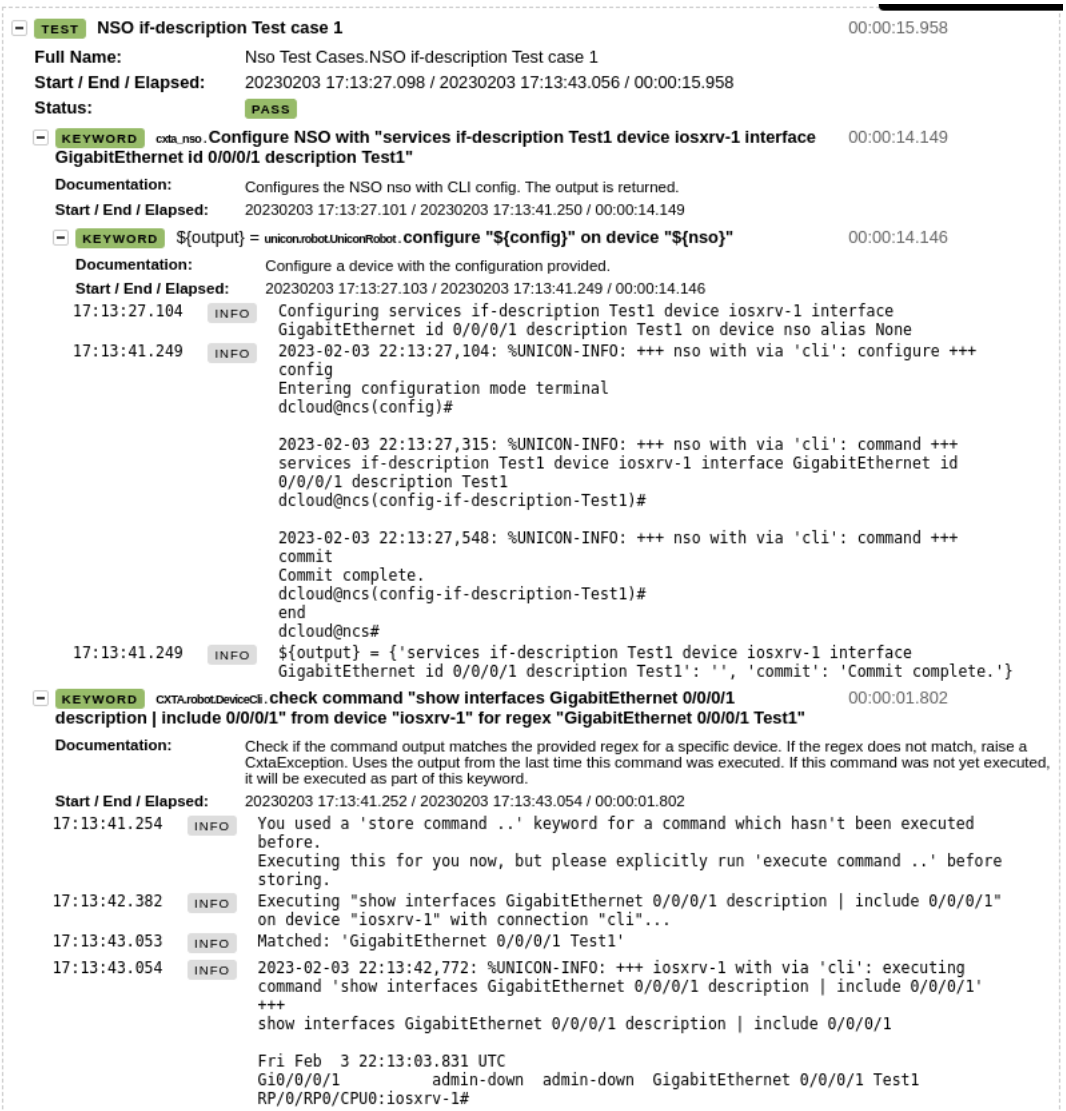
2. Review report.html in Chrome



3. Review log.html in Chrome



4. Drilldown test case for detailed results



Task 4: Create and execute Second NSO Test Case

This test case is very similar to the previous one but there is a small difference.

Step 1: Create second keyword

1. Add ‘NSO if-description Test case 2’ to ‘Test Cases’ section
2. Write test case in using these commands
    a. NSO configuration command: “services if-description Test1 device iosxrv-2 interface GigabitEthernet id 0/0/0/2 description Test2”
    b. IOS-XR show command: “show interfaces GigabitEthernet 0/0/0/2 description | include 0/0/0/2" from device "iosxrv-2”

Or copy-paste Test case 1 changing all ‘1’ to ‘2’ ☺

```
NSO if-description Test case 2
```

```
    Configure NSO with "services if-description Test2 device
iosxrv-2 interface GigabitEthernet id 0/0/0/2 description
Test2"
    check command "show interfaces GigabitEthernet 0/0/0/2
description | include 0/0/0/1" from device "iosxrv-2" for regex
"GigabitEthernet 0/0/0/1 Test2"
```

3. Save the file

Step 2: Execute test

1. Go to Terminal window
2. Start bash shell in CXTA container

```
dclould@dcloud-virtual-machine:~$ docker exec -it cxta bash
```

3. Execute test suite

```
cisco@1234567890abcdef:~/cxta$ robot nso_testcases.robot
```

Step 3: Review Results

1. Check results in terminal

```
cisco@6e95cb041675:~/cxta$ robot nso_test_cases.robot
==============================================================================
Nso Test Cases
==============================================================================
Empty testcase
Hello World!!!
Empty testcase                                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 1                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 2                                        | FAIL |
CxtaException: 'GigabitEthernet 0/0/0/2 Test2' not found in output
------------------------------------------------------------------------------
Nso Test Cases                                                        | FAIL |
3 tests, 2 passed, 1 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

2. Review report in Chrome

LOG

Generated
20230203 17:56:24 UTC-05:00
2 minutes 21 seconds ago

Nso Test Cases Report

Summary Information

Status:	1 test failed
CXTA Version:	22.26
Start Time:	20230203 17:55:13.254
End Time:	20230203 17:56:24.468
Elapsed Time:	00:01:11.214
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	2	1	0	00:00:34	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Nso Test Cases	3	2	1	0	00:01:11	

Test Details

All | Tags | Suites | Search

Suite:

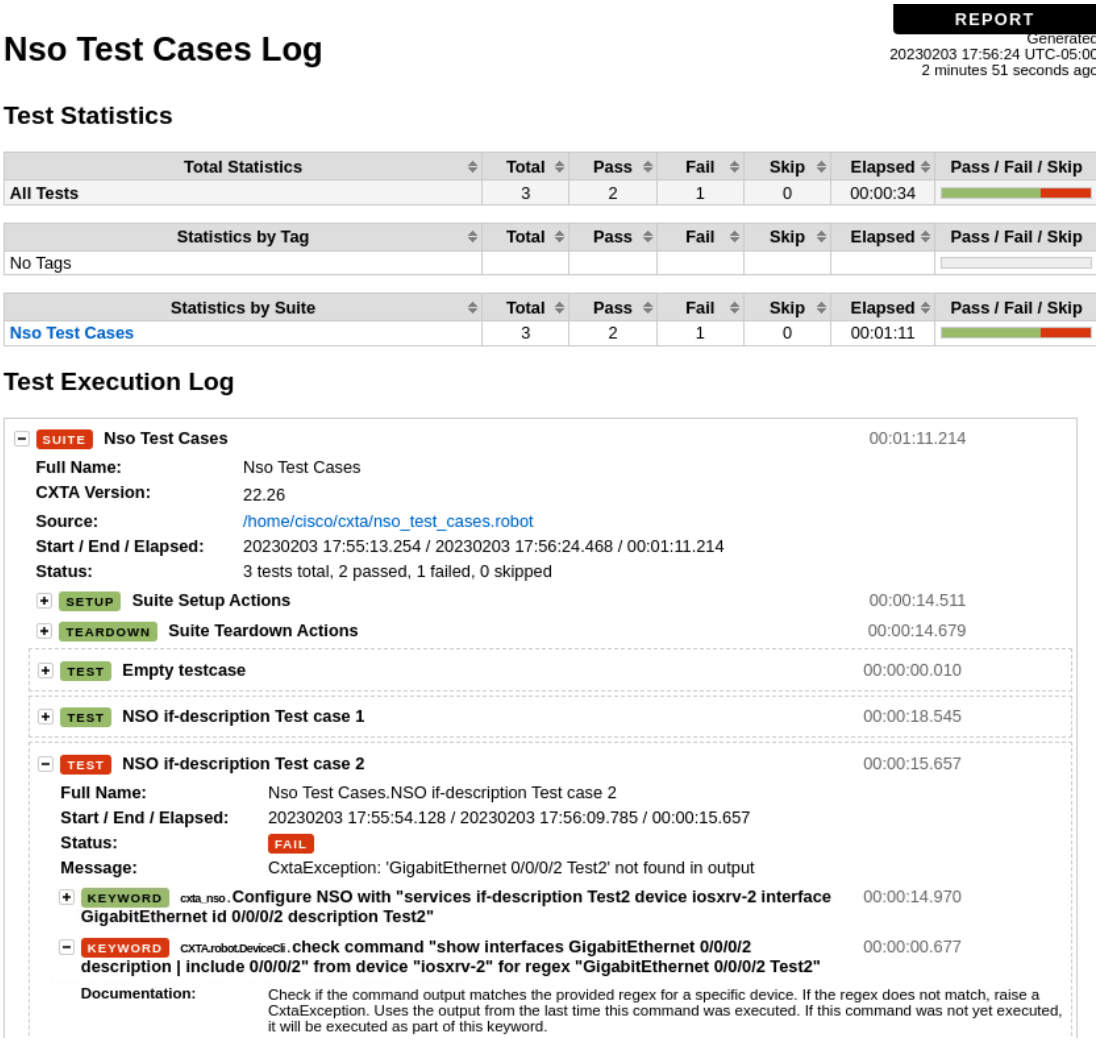
Test:

Include:

Exclude:

Search | Clear | Help

3. Review log



REPORT

Generated
20230203 17:56:24 UTC-05:00
2 minutes 51 seconds ago

Nso Test Cases Log

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	3	2	1	0	00:00:34	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Nso Test Cases	3	2	1	0	00:01:11	

Test Execution Log

SUITE **Nso Test Cases** 00:01:11.214

Full Name: Nso Test Cases
CXTA Version: 22.26
Source: /home/cisco/cxta/nso_test_cases.robot
Start / End / Elapsed: 20230203 17:55:13.254 / 20230203 17:56:24.468 / 00:01:11.214
Status: 3 tests total, 2 passed, 1 failed, 0 skipped

SETUP **Suite Setup Actions** 00:00:14.511

TEARDOWN **Suite Teardown Actions** 00:00:14.679

TEST **Empty testcase** 00:00:00.010

TEST **NSO if-description Test case 1** 00:00:18.545

TEST **NSO if-description Test case 2** 00:00:15.657

Full Name: Nso Test Cases.NSO if-description Test case 2
Start / End / Elapsed: 20230203 17:55:54.128 / 20230203 17:56:09.785 / 00:00:15.657
Status: FAIL
Message: CxtaException: 'GigabitEthernet 0/0/0/2 Test2' not found in output

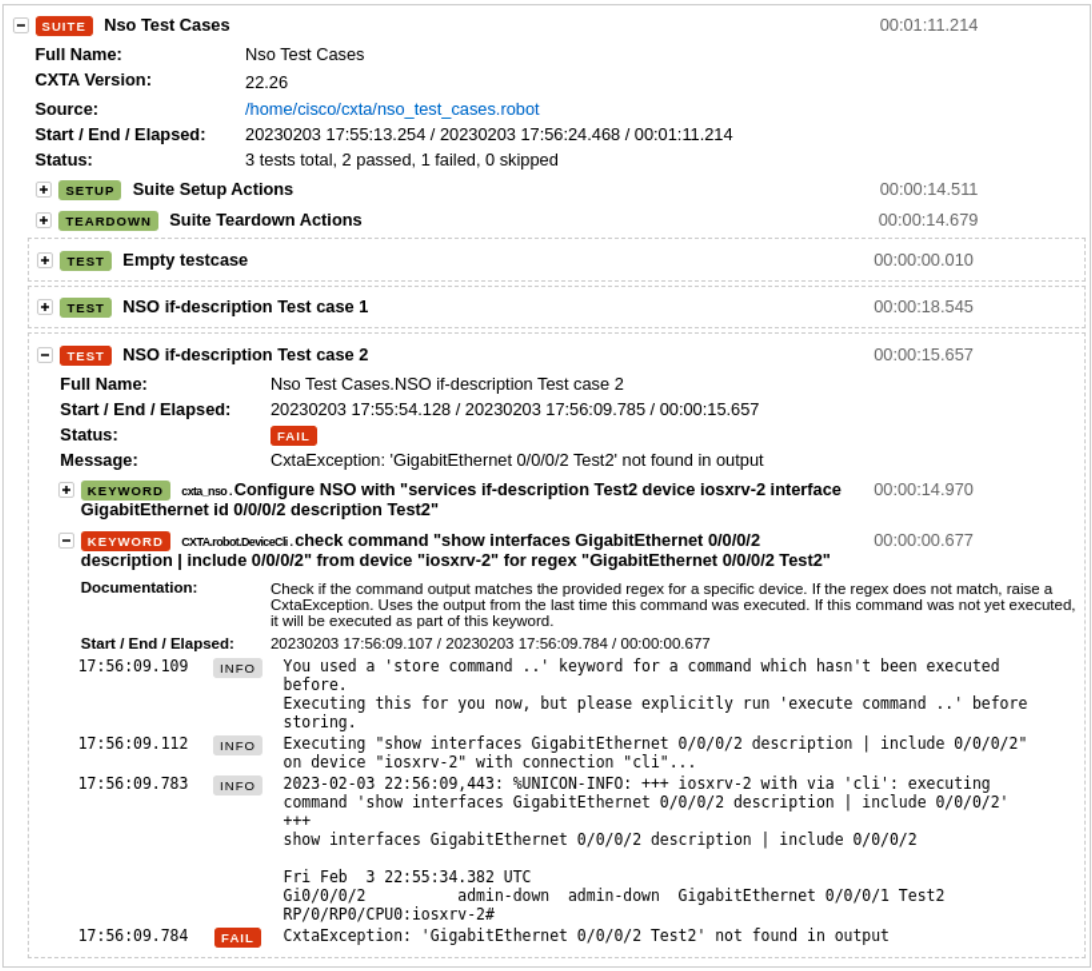
KEYWORD cxta_nso . **Configure NSO with "services if-description Test2 device iosxrv-2 interface GigabitEthernet id 0/0/0/2 description Test2"** 00:00:14.970

KEYWORD CXTA.robot.DeviceCli . **check command "show interfaces GigabitEthernet 0/0/0/2 description | include 0/0/0/2" from device "iosxrv-2" for regex "GigabitEthernet 0/0/0/2 Test2"** 00:00:00.677

Documentation: Check if the command output matches the provided regex for a specific device. If the regex does not match, raise a CxtaException. Uses the output from the last time this command was executed. If this command was not yet executed, it will be executed as part of this keyword.

4. Check exact results (errors are already drilled down on open)

Test Execution Log

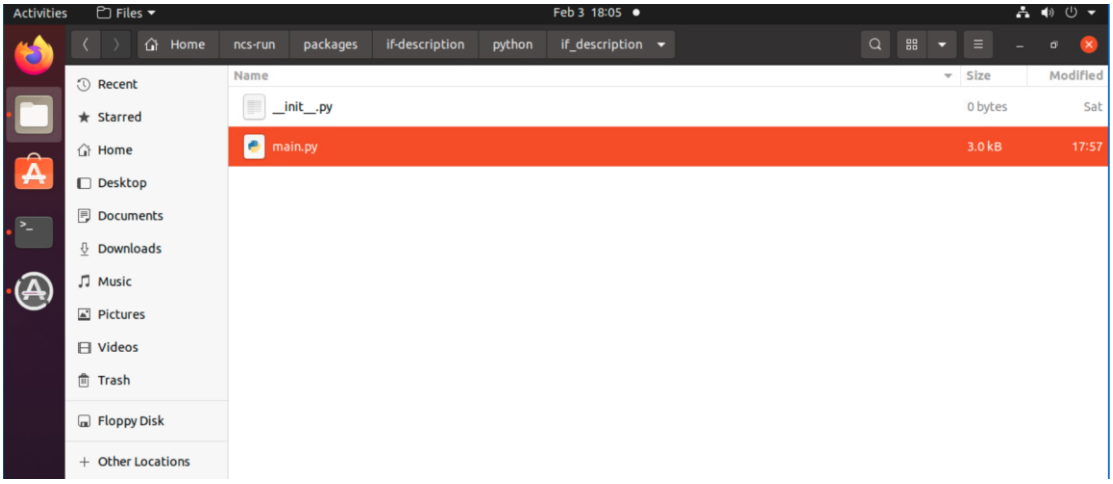


5. Review show command result in the log

Step 4: Check and fix NSO root cause

1. Go to NSO Console as in Lab introduction section
2. Go to Activities in the Console

3. Open Files application and find NSO if-description main python file in NSO



4. Open main.py with default editor

5. Comment out following lines (lines 17, 21 and 22)

```
#         test_id = '0/0/0/2';

        description = f'{service.interface} {service.id}
{service.description}'

#         if service.id == test_id:
#             description = f'{service.interface} 0/0/0/1
{service.description}'
```

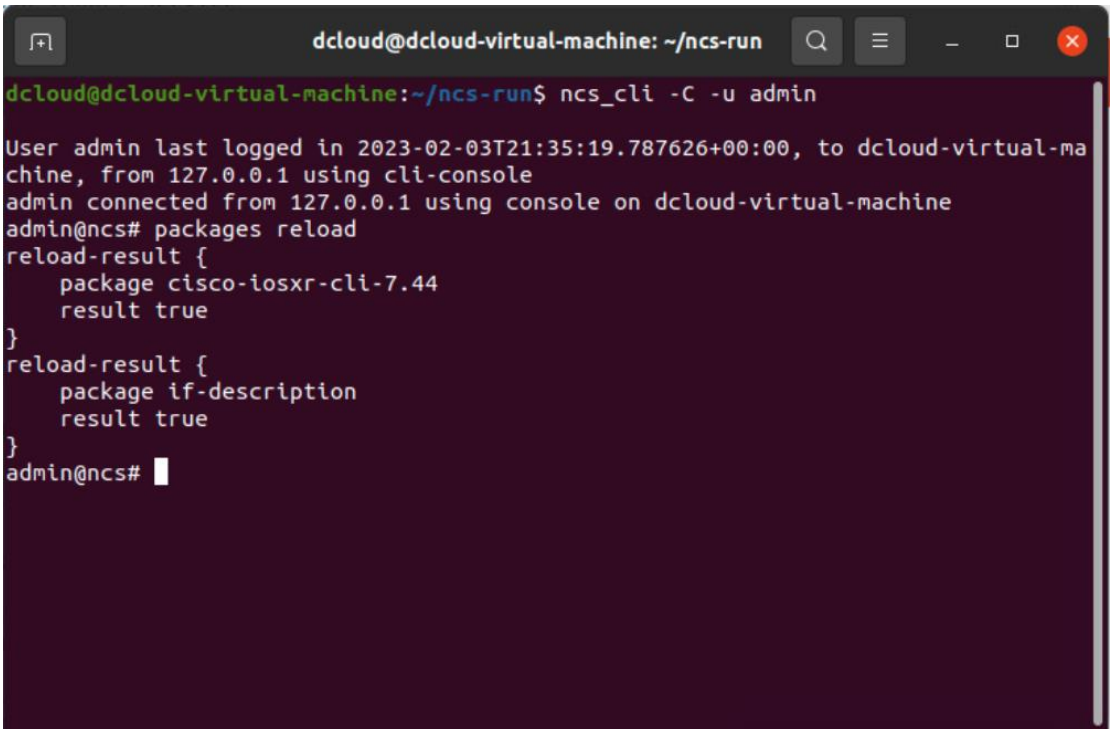
6. File should look like this

```
17 #         test_id = '0/0/0/2';
18
19          description = f'{service.interface} {service.id} {service.description}'
20
21 #         if service.id == test_id:
22 #             description = f'{service.interface} 0/0/0/1 {service.description}'
23
```

7. Save the file
8. Switch to NSO terminal
9. Execute following commands to reload NSO packages and pick up changes

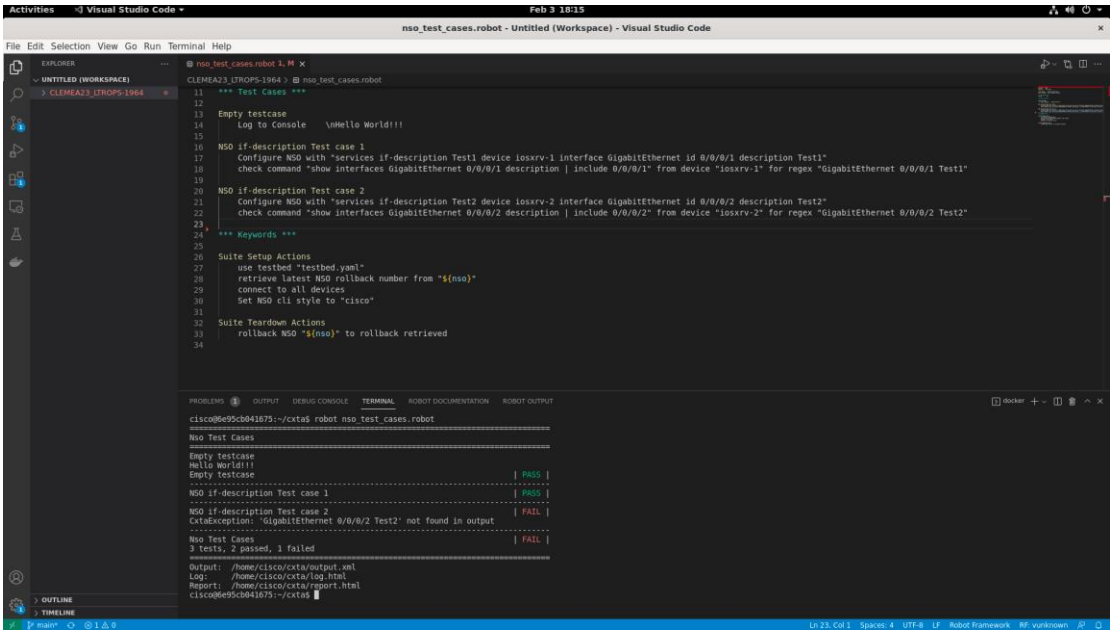
```
dcloud@dcloud-virtual-machine:~/ncs-run$ ncs_cli -C -u admin
…
admin@ncs# packages reload
```

10. Result should look like this one



Step 5: Execute test again

1. Switch back to WKST view



2. Rerun robot test

```
cisco@1234567890abcdef:~/cxta$ robot nso_testcases.robot
```

3. Review results

```
cisco@6e95cb041675:~/cxta$ robot nso_test_cases.robot
==============================================================================
Nso Test Cases
==============================================================================
Empty testcase
Hello World!!!
Empty testcase                                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 1                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 2                                        | PASS |
------------------------------------------------------------------------------
Nso Test Cases                                                        | PASS |
3 tests, 3 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

Task 4: Create and execute Negative NSO Test Case

This test case covers scenario where we expect executed commands to fail so this describes negative test scenario.

Step 1: Create negative test keyword

1. Add ‘NSO if-description Test case 3 - negative’ to ‘Test Cases’ section
2. Write test case in using these commands
    a. Built in Robot Keyword: “Run Keyword and Expect Error REGEXP: …” (expected error contains ‘syntax error’ phrase as device does not have any TenGigE interface)

b. NSO configuration command: “services if-description Test3 device iosxrv-1 interface TenGigE id 1/0/0/1 description Test3”

```
NSO if-description Test case 3 - negative
    Run Keyword and Expect Error    REGEXP: .*syntax error.*
    ...    Configure NSO with "services if-description Test3
device iosxrv-1 interface TenGigE id 1/0/0/1 description
Test3"
```

3. Save the file

Step 2: Execute test

1. Go to Terminal window
2. Start bash shell in CXTA container

```
dclould@dcloud-virtual-machine:~$ docker exec -it cxta bash
```

3. Execute test suite

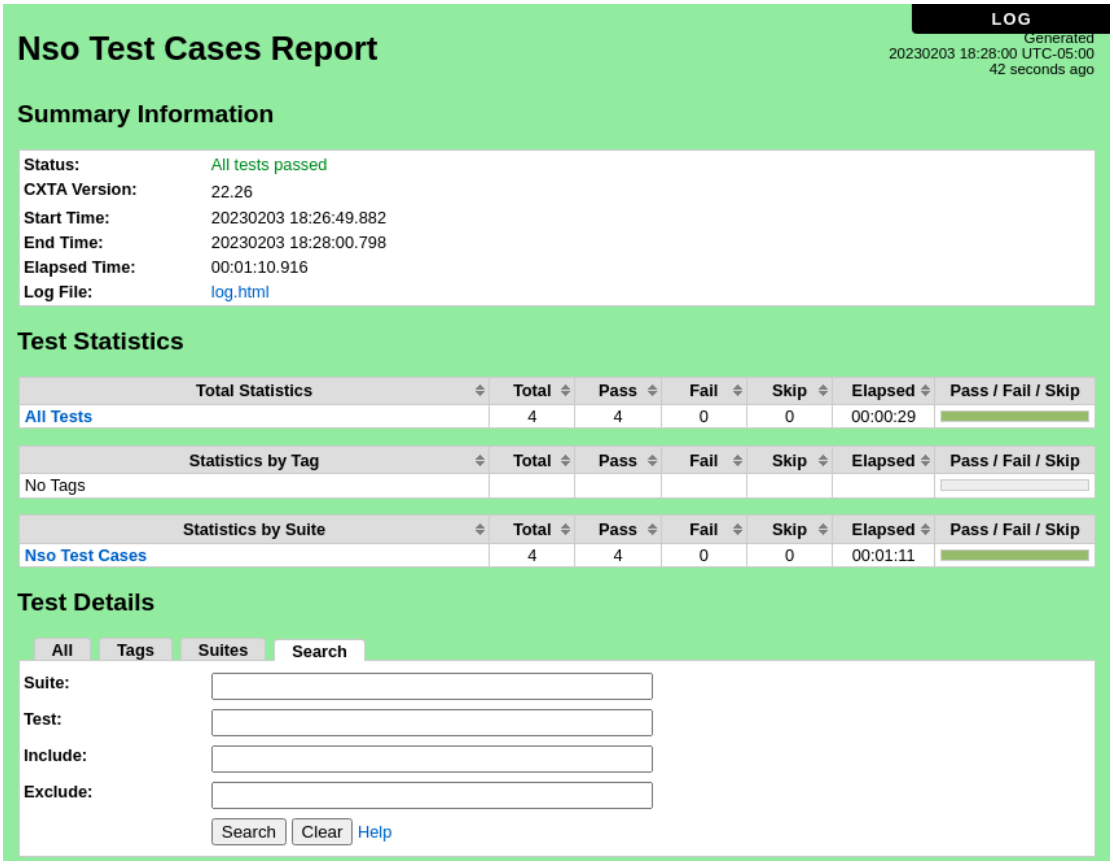
```
cisco@1234567890abcdef:~/cxta$ robot nso_testcases.robot
```

Step 3: Review Results

1. Check results in Terminal

```
cisco@6e95cb041675:~/cxta$ robot nso_test_cases.robot
==============================================================================
Nso Test Cases
==============================================================================
Empty testcase
Hello World!!!
Empty testcase                                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 1                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 2                                        | PASS |
------------------------------------------------------------------------------
NSO if-description Test case 3 - negative                             | PASS |
------------------------------------------------------------------------------
Nso Test Cases                                                        | PASS |
4 tests, 4 passed, 0 failed
==============================================================================
Output:  /home/cisco/cxta/output.xml
Log:     /home/cisco/cxta/log.html
Report:  /home/cisco/cxta/report.html
cisco@6e95cb041675:~/cxta$
```

2. Review report in Chrome



LOG
Generated
20230203 18:28:00 UTC-05:00
42 seconds ago

Nso Test Cases Report

Summary Information

Status:	All tests passed
CXTA Version:	22.26
Start Time:	20230203 18:26:49.882
End Time:	20230203 18:28:00.798
Elapsed Time:	00:01:10.916
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	4	4	0	0	00:00:29	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Nso Test Cases	4	4	0	0	00:01:11	

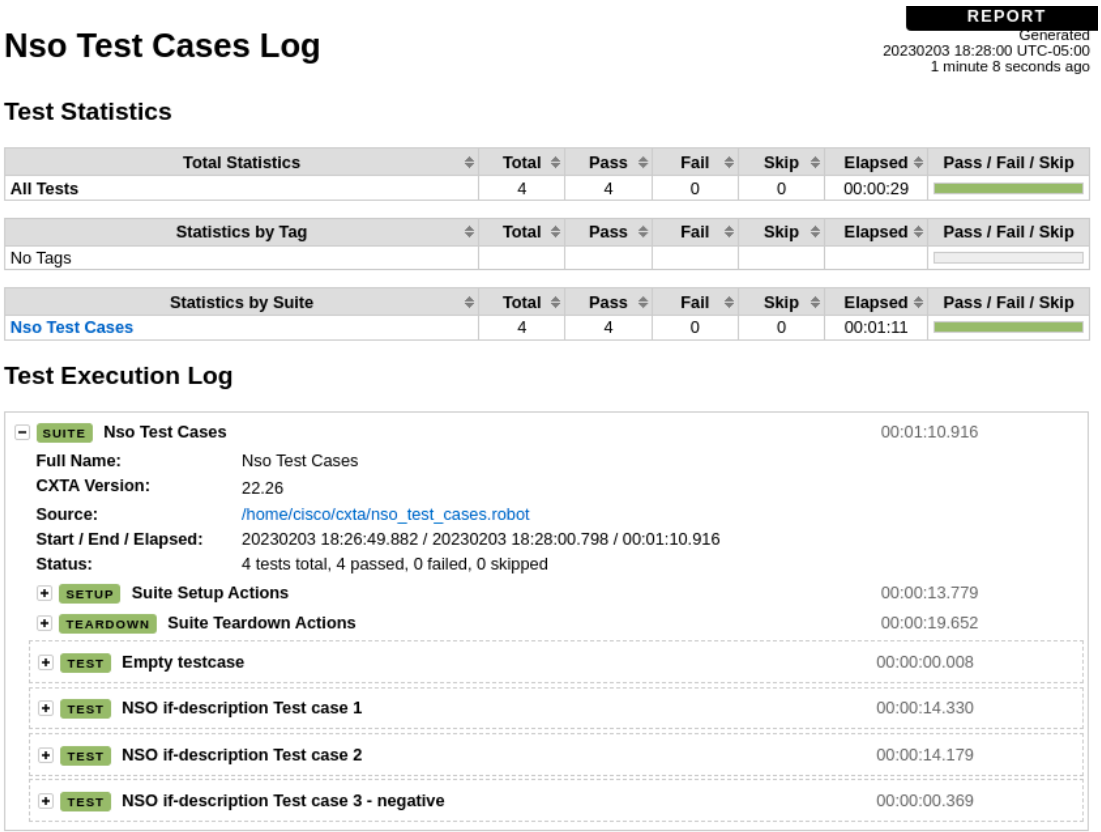
Test Details

All | Tags | Suites | Search

Suite:
Test:
Include:
Exclude:

Search | Clear | Help

3. Review log in Chrome



REPORT
Generated
20230203 18:28:00 UTC-05:00
1 minute 8 seconds ago

Nso Test Cases Log

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	4	4	0	0	00:00:29	

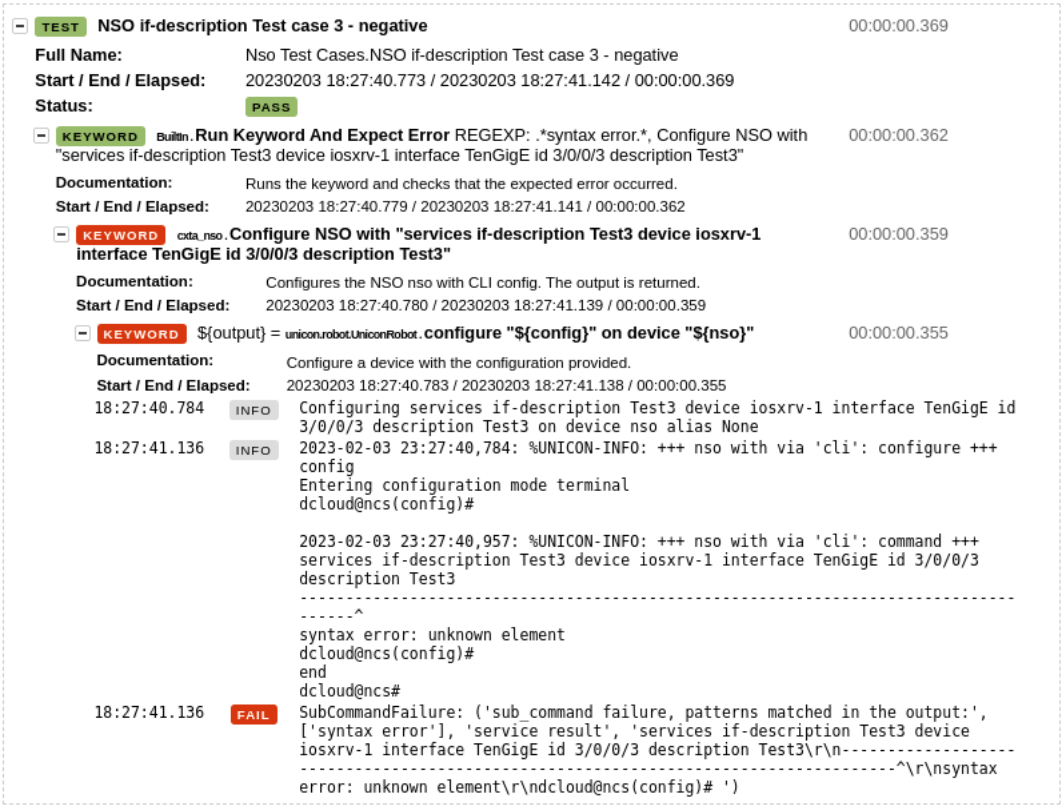
Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Nso Test Cases	4	4	0	0	00:01:11	

Test Execution Log

SUITE	**Nso Test Cases**	00:01:10.916
	Full Name: Nso Test Cases	
	CXTA Version: 22.26	
	Source: /home/cisco/cxta/nso_test_cases.robot	
	Start / End / Elapsed: 20230203 18:26:49.882 / 20230203 18:28:00.798 / 00:01:10.916	
	Status: 4 tests total, 4 passed, 0 failed, 0 skipped	
SETUP	**Suite Setup Actions**	00:00:13.779
TEARDOWN	**Suite Teardown Actions**	00:00:19.652
TEST	**Empty testcase**	00:00:00.008
TEST	**NSO if-description Test case 1**	00:00:14.330
TEST	**NSO if-description Test case 2**	00:00:14.179
TEST	**NSO if-description Test case 3 - negative**	00:00:00.369

4. Drilldown and review Test case 3 results



Next Steps

1. Explore the open-source library of robot framework.
2. Define test cases to be automated
3. Start creating your test scenarios parameters and requirements
4. Contact Cisco for further support and for CXTA framework.

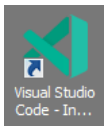
Summary

Appendix A: Use Visual Studio Code

Notice that we connect to a Windows Host, but Cisco NSO software runs in a remote server reachable through IP 198.18.134.28.

You can find the initial packages and other required files locally in Windows under "C:\dcloud\HOLOPS-1806" (reachable through shortcut in Desktop) and as well inside NSO Host under "/home/cisco/nso572/ncs-run".

To avoid having to edit every file locally and then upload it to NSO host using SFTP (Filezilla configured for it) the lab comes with 'Visual Studio Code – Insiders' installed (available from the Desktop) with a plugin that automatically connects to NSO and allows you to work with the remote files locally.



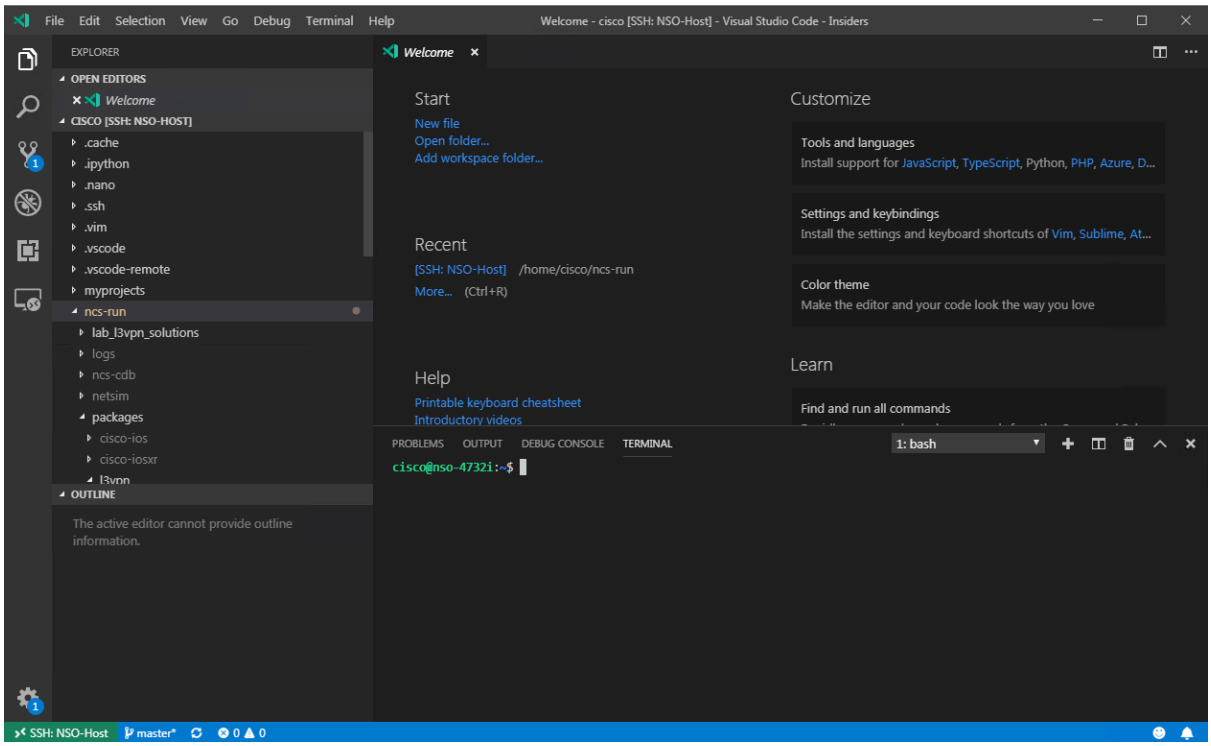
When you start 'Visual Studio Code – Insiders' you can verify that you are connected to NSO by looking at the bottom left corner for 'SSH: NSO-Host'. You should see as well in the left panel the /home/cisco directory with all the main files for NSO in directory /home/cisco/nso572/ncs-run. From there you can view, edit and save all the required files

To open a Terminal to NSO Host go to Terminal > New Terminal.

From the terminal you can run the required commands in NSO Host. Or you can run the following command to connect to NSO CLI

```
cisco@nso-572i:~/ncs-run$ ncs_cli -C -u admin
```

Where '-C' is for Cisco CLI mode



NOTE: Do not confuse 'Visual Studio Code – Insiders' with 'Visual Studio Code', that it is as well installed in Windows, but doesn't have a plugin to edit remote files

Appendix B: Install NSO

This section will help you understand how to install NSO Software in your laptop/machine for development and learning purposes. The installation is called local-install.

For production environment NSO Software installation should be done on a server with recommended specification. This installation is called system-install.

Download free NSO: https://developer.cisco.com/docs/nso/#!getting-nso/getting-nso

Note: At the time of writing this section NSO 5.2.1 was available in the above link.

In this lab you can find it downloaded in NSO under /home/cisco/Downloads

Note: Make sure the user of the system has root privileges. If not, then use command with sudo in case of any issues.

Install NSO 5.2.1

1. Extract the installation file:

```
developer@nso-host:~/Downloads$ sh nso-
5.2.1.linux.x86_64.signed.bin
Unpacking...
Verifying signature...
Downloading CA certificate from
http://www.cisco.com/security/pki/certs/crcam2.cer ...
Successfully downloaded and verified crcam2.cer.
Downloading SubCA certificate from
http://www.cisco.com/security/pki/certs/innerspace.cer ...
Successfully downloaded and verified innerspace.cer.
Successfully verified root, subca and end-entity certificate
chain.
Successfully fetched a public key from tailf.cer.
Successfully verified the signature of nso-
5.2.1.linux.x86_64.installer.bin using tailf.cer
nso@ubuntu:~/Downloads$
```

2. Install NSO

```
developer@nso-host:~/Downloads$ sh nso-
5.2.1.linux.x86_64.installer.bin /home/cisco/nso521/nso521 --
local-install
INFO  Using temporary directory /tmp/ncs_installer.11530 to stage
NCS installation bundle
INFO  Unpacked ncs-5.2.1 in /home/cisco/nso521/nso521/
INFO  Found and unpacked corresponding DOCUMENTATION_PACKAGE
INFO  Found and unpacked corresponding EXAMPLE_PACKAGE
INFO  Generating default SSH hostkey (this may take some time)
INFO  SSH hostkey generated
INFO  Environment set-up generated in
/home/cisco/nso521/nso521//ncsrc
INFO  NCS installation script finished
INFO  Found and unpacked corresponding NETSIM_PACKAGE
INFO  NCS installation complete
```

3. Source the binaries:

```
developer@nso-host:~/Downloads$ cd ..
developer@nso-host:~$ source nso521/nso521/ncsrc
```

This sets up paths and environment variables in order to run NSO.

As this must be done before running NSO. It is recommended to put it in your profile.

4. Create running directory:

```
developer@nso-host:~$ ncs-setup --dest nso521/ncs-run-521
```

The runtime directory should look similar to below:

```
developer@nso-host:~/nso521/ncs-run-521$ ls -lrt
total 36
drwxrwxr-x 2 developer developer 4096 Nov 27 03:05 state
drwxrwxr-x 2 developer developer 4096 Nov 27 03:05 packages
drwxrwxr-x 2 developer developer 4096 Nov 27 03:05 ncs-cdb
drwxrwxr-x 2 developer developer 4096 Nov 27 03:05 logs
drwxrwxr-x 4 developer developer 4096 Nov 27 03:05 scripts
-rw-rw-r-- 1 developer developer 9951 Nov 27 03:05 ncs.conf
-rw-rw-r-- 1 developer developer  636 Nov 27 03:05 README.ncs
```

Install the NEDs

1. Link NED packages from install directory to running directory (in our lab this is already done):

```
nso@ubuntu:~/ncs-run-5.2.1$ cd ../Downloads/
nso@ubuntu:~/Downloads$ ls -lrt
total 449576
-rwxr-xr-x 1 nso nso 183649615 Nov 30 09:13 nso-5.2.1.linux.x86_64.installer.bin
-rw-r--r-- 1 nso nso      1383 Nov 30 09:13 tailf.cer
-rw-r--r-- 1 nso nso     10696 Nov 30 09:13 cisco_x509_verify_release.py
-rw-r--r-- 1 nso nso       256 Nov 30 09:13 nso-
5.2.1.linux.x86 64.installer.bin.signature
-rw-r--r-- 1 nso nso      1822 Nov 30 09:13 README.signature
-rw-rw-r-- 1 nso nso 183661414 Jan 19 02:15 nso-5.2.1.linux.x86_64.signed.bin
-rw-rw-r-- 1 nso nso  32476407 Jan 19 02:17 ncs-5.2.1-cisco-ios-6.39.signed.bin
-rw-rw-r-- 1 nso nso  24417815 Jan 19 02:17 ncs-5.2.1-cisco-iosxr-7.17.signed.bin
-rw-rw-r-- 1 nso nso  36116185 Jan 19 02:18 ncs-5.2.1-cisco-nx-5.13.signed.bin
nso@ubuntu:~/Downloads$
```

2. Extract the NEDs:

```
developer@nso-host:~/Downloads$ sh ncs-5.2.1-cisco-ios-
6.39.signed.bin && sh ncs-5.2.1-cisco-iosxr-7.17.signed.bin && sh
ncs-5.2.1-cisco-nx-5.13.signed.bin
developer@nso-host:~/Downloads$ ls -lrt
total 419972
-rw-r--r-- 1 developer developer      1383 May  6  2019 tailf.cer
-rw-r--r-- 1 developer developer     12381 May  6  2019
cisco x509 verify release.py
-rw-r--r-- 1 developer developer  36267747 May 16  2019 ncs-5.2.1-cisco-iosxr-
7.17.tar.gz
-rw-r--r-- 1 developer developer  39632795 May 16  2019 ncs-5.2.1-cisco-nx-
5.13.tar.gz
-rw-r--r-- 1 developer developer  46103905 May 16  2019 ncs-5.2.1-cisco-ios-
6.39.tar.gz
-rw-r--r-- 1 developer developer       256 May 16  2019 ncs-5.2.1-cisco-iosxr-
7.17.tar.gz.signature
-rw-r--r-- 1 developer developer       256 May 17  2019 ncs-5.2.1-cisco-ios-
6.39.tar.gz.signature
-rw-r--r-- 1 developer developer       256 May 17  2019 ncs-5.2.1-cisco-nx-
5.13.tar.gz.signature
-rw-r--r-- 1 developer developer      1832 May 17  2019 README.signature
-rwxrwxrwx 1 developer developer 186085011 Nov 27 02:38 nso-
5.2.1.linux.x86_64.installer.bin
-rw-r--r-- 1 developer developer  46077897 Nov 27 03:33 ncs-5.2.1-cisco-ios-
6.39.signed.bin
-rw-r--r-- 1 developer developer  36256645 Nov 27 03:40 ncs-5.2.1-cisco-iosxr-
7.17.signed.bin
-rw-r--r-- 1 developer developer  39577109 Nov 27 05:08 ncs-5.2.1-cisco-nx-
5.13.signed.bin
```

3. Untar the NEDs:

```
developer@nso-host:~/Downloads$ tar -xvf ncs-5.2.1-cisco-iosxr-7.17.tar.gz && tar -xvf ncs-5.2.1-cisco-nx-5.13.tar.gz && tar -xvf ncs-5.2.1-cisco-ios-6.39.tar.gz
```

```
developer@nso-host:~/Downloads$ ls -lrt
total 419984
-rw-r--r-- 1 developer developer      1383 May  6  2019 tailf.cer
-rw-r--r-- 1 developer developer     12381 May  6  2019
cisco_x509_verify_release.py
-rw-r--r-- 1 developer developer  36267747 May 16  2019 ncs-5.2.1-cisco-iosxr-
7.17.tar.gz
drwxr-xr-x 9 developer developer      4096 May 16  2019 cisco-iosxr-cli-7.17
-rw-r--r-- 1 developer developer  39632795 May 16  2019 ncs-5.2.1-cisco-nx-
5.13.tar.gz
-rw-r--r-- 1 developer developer  46103905 May 16  2019 ncs-5.2.1-cisco-ios-
6.39.tar.gz
drwxr-xr-x 6 developer developer      4096 May 16  2019 cisco-nx-cli-5.13
drwxr-xr-x 8 developer developer      4096 May 16  2019 cisco-ios-cli-6.39
-rw-r--r-- 1 developer developer       256 May 16  2019 ncs-5.2.1-cisco-iosxr-
7.17.tar.gz.signature
-rw-r--r-- 1 developer developer       256 May 17  2019 ncs-5.2.1-cisco-ios-
6.39.tar.gz.signature
-rw-r--r-- 1 developer developer       256 May 17  2019 ncs-5.2.1-cisco-nx-
5.13.tar.gz.signature
-rw-r--r-- 1 developer developer      1832 May 17  2019 README.signature
-rwxrwxrwx 1 developer developer 186085011 Nov 27 02:38 nso-
5.2.1.linux.x86 64.installer.bin
-rw-r--r-- 1 developer developer  46077897 Nov 27 03:33 ncs-5.2.1-cisco-ios-
6.39.signed.bin
-rw-r--r-- 1 developer developer  36256645 Nov 27 03:40 ncs-5.2.1-cisco-iosxr-
7.17.signed.bin
-rw-r--r-- 1 developer developer  39577109 Nov 27 05:08 ncs-5.2.1-cisco-nx-
5.13.signed.bin
developer@nso-host:~/Downloads$
nso@ubuntu:~/Downloads$
```

4. Link the NEDs to NSO runtime directory packages folder:

```
developer@nso-host:~/nso521$ cd ncs-run-521/packages/
developer@nso-host:~/nso521/ncs-run-521/packages$ ln -s ~/Downloads/cisco-iosxr-cli-7.17/ /home/cisco/nso521/ncs-run-521/packages/
developer@nso-host:~/nso521/ncs-run-521/packages$ ln -s ~/Downloads/cisco-ios-cli-6.39/ /home/cisco/nso521/ncs-run-521/packages/
developer@nso-host:~/nso521/ncs-run-521/packages$ ln -s ~/Downloads/cisco-nx-cli-5.13/ /home/cisco/nso521/ncs-run-521/packages/
```

Start NSO and verify status

1. Start NSO from within running-directory:

```
developer@nso-host:~/nso521/ncs-run-521/packages$ cd /home/cisco/nso521/ncs-run-521/
developer@nso-host:~/nso521/ncs-run-521$ ncs
```

2. Verify NSO status:

```
developer@nso-host:~/nso521/ncs-run-521$ ncs --status
vsn: 5.2.1
SMP support: yes, using 4 threads
Using epoll: yes
available modules: backplane,netconf,cdb,cli,snmp,webui
running modules: backplane,netconf,cdb,cli,snmp,webui
status: started
…
```

3. Check the ncs.conf file for default NSO configuration parameters:

```
developer@nso-host:~/nso521/ncs-run-521$ cat ncs.conf

<!-- -*- nxml -*- -->
<!-- Example configuration file for ncs. -->

<ncs-config xmlns="http://tail-f.com/yang/tailf-ncs-config">

  <!-- NCS can be configured to restrict access for incoming
connections -->
  <!-- to the IPC listener sockets. The access check requires that
-->
  <!-- connecting clients prove possession of a shared secret. -->
  <ncs-ipc-access-check>
    <enabled>false</enabled>
    <filename>${NCS_DIR}/etc/ncs/ipc_access</filename>
  </ncs-ipc-access-check>

  <!-- Where to look for .fxs and snmp .bin files to load -->
 …
```

4. Login to NSO CLI Juniper mode (default password for user admin is admin):

```
developer@nso-host:~/nso521/ncs-run-521$ ncs_cli -u  admin

admin connected from 192.168.234.3 using ssh on nso-host
admin@ncs>
```

OR

```
developer@nso-host:~/nso521/ncs-run-521$ ssh -l admin -p 2024
localhost
The authenticity of host '[localhost]:2024 ([127.0.0.1]:2024)'
can't be established.
RSA key fingerprint is
SHA256:ZLWvfBSWDj4yqS1a68ZpTT4nTVsrrCC8CVTBlDPJuO0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:2024' (RSA) to the list of
known hosts.
admin@localhost's password:

admin connected from 127.0.0.1 using ssh on nso-host
admin@ncs>
```

5. Switch to Cisco style CLI:

```
admin@ncs> switch cli
admin@ncs#  << Cisco style CLI>>
admin@ncs#

admin@ncs#
admin@ncs# switch cli
[ok][2019-11-27 05:38:02]
admin@ncs>   << Juniper Style>>
admin@ncs>
admin@ncs>
```

OR

```
## Cisco Style CLI

developer@nso-host:~/nso521/ncs-run-521$ ncs_cli -u admin -C

admin connected from 192.168.234.3 using ssh on nso-host
admin@ncs#
```

```
## Juniper Style CLI

developer@nso-host:~/nso521/ncs-run-521$ ncs_cli -u admin -J

admin connected from 192.168.234.3 using ssh on nso-host
admin@ncs>
```

6. Explore different show commands from NSO CLI

```
 # show packages packages package-version
 # show devices list
 # show running-config
```

7. Observe NSO startup process through different logs: ncs-java-vm.log

```
developer@nso-host:~/nso521/ncs-run-521$
developer@nso-host:~/nso521/ncs-run-521$ cd logs/
developer@nso-host:~/nso521/ncs-run-521/logs$ ls -lrt
total 240
-rw-rw-r-- 1 developer developer      0 Nov 27 05:30 netconf.log
-rw-rw-r-- 1 developer developer      0 Nov 27 05:30 snmp.log
-rw-rw-r-- 1 developer developer     13 Nov 27 05:30
ncserr.log.siz
-rw-rw-r-- 1 developer developer     18 Nov 27 05:30
ncserr.log.idx
-rw-rw-r-- 1 developer developer      8 Nov 27 05:30 ncserr.log.1
-rw-rw-r-- 1 developer developer      0 Nov 27 05:31 ncs-python-
vm.log
-rw-rw-r-- 1 developer developer    826 Nov 27 05:31 rollback10001
-rw-rw-r-- 1 developer developer      0 Nov 27 05:31
localhost:8080.access
-rw-rw-r-- 1 developer developer   7274 Nov 27 05:31 ncs-java-
vm.log
-rw-rw-r-- 1 developer developer  21401 Nov 27 05:32 ncs.log
-rw-rw-r-- 1 developer developer  53451 Nov 27 05:40 xpath.trace
-rw-rw-r-- 1 developer developer 127736 Nov 27 05:40 devel.log
-rw-rw-r-- 1 developer developer   6097 Nov 27 05:40 audit.log
developer@nso-host:~/nso521/ncs-run-521/logs$
```

Appendix C: Manage Netsim Devices

NSO already has 6 simulated devices (called Netsim devices across the lab guide) configured and ready to start from /home/cisco/nso572/ncs-run directory.

During Lab Introduction and Verification section you are asked to start them by running command 'ncs-netsim start' from that directory. If everything goes well, you will see an output like the following:

```
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim start
DEVICE PE_00 OK STARTED
DEVICE PE_01 OK STARTED
DEVICE PE_10 OK STARTED
DEVICE PE_11 OK STARTED
DEVICE P_20 OK STARTED
DEVICE P_21 OK STARTED
cisco@ubuntu:~/nso572/ncs-run$
```

If see the following output, everything it's OK, that means the devices had already been started.

```
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim start
Cannot bind to internal socket 127.0.0.1:5010 : address already in
use
Daemon died status=20
DEVICE PE_00 FAIL
Cannot bind to internal socket 127.0.0.1:5011 : address already in
use
Daemon died status=20
DEVICE PE_01 FAIL
Cannot bind to internal socket 127.0.0.1:5012 : address already in
use
Daemon died status=20
DEVICE PE_10 FAIL
Cannot bind to internal socket 127.0.0.1:5013 : address already in
use
Daemon died status=20
DEVICE PE_11 FAIL
Cannot bind to internal socket 127.0.0.1:5014 : address already in
use
Daemon died status=20
DEVICE P_20 FAIL
Cannot bind to internal socket 127.0.0.1:5015 : address already in
use
Daemon died status=20
DEVICE P_21 FAIL
cisco@ubuntu:~/nso572/ncs-run$
```

In case you are experiencing some issues, you can stop and start again the devices. To stop the devices run the following command.

```
cisco@nso-572i:~/ncs-run$ ncs-netsim stop
DEVICE PE_00 STOPPED
DEVICE PE_01 STOPPED
DEVICE PE_10 STOPPED
DEVICE PE_11 STOPPED
DEVICE P_20 STOPPED
DEVICE P_21 STOPPED

cisco@nso-572i:~/ncs-run$
```

If by mistake you have deleted the devices from NSO, once they have been started you can add them back by running the following command from /home/cisco/nso572/ncs-run.

```
cisco@nso-572i:~/ncs-run$ ncs-netsim ncs-xml-init > devices.xml
cisco@nso-572i:~/ncs-run$ ncs_load -l -m devices.xml
```

If you have to re-create the netsim devices, please do as follows:

```
cisco@ubuntu:~/nso572/ncs-run$ cd /home/cisco/nso572/ncs-run
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim create-network
packages/cisco-ios 2 PE_0
DEVICE PE_00 CREATED
DEVICE PE_01 CREATED
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim add-to-network
packages/cisco-iosxr 2 PE_1
DEVICE PE_10 CREATED
DEVICE PE_11 CREATED
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim add-to-network
packages/juniper-junos 2 P_2
DEVICE P_20 CREATED
DEVICE P_21 CREATED
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim start
DEVICE PE_00 OK STARTED
DEVICE PE_01 OK STARTED
DEVICE PE_10 OK STARTED
DEVICE PE_11 OK STARTED
DEVICE P_20 OK STARTED
DEVICE P_21 OK STARTED
cisco@ubuntu:~/nso572/ncs-run$ ncs-netsim ncs-xml-init > devices.xml
cisco@ubuntu:~/nso572/ncs-run$ ncs_load -l -m devices.xml
```

Related Sessions at Ciscolive

You can search CiscoLive Amsterdam content catalog with specific keywords and recommend sessions that are relevant to your lab. In this case CXTA

Session Catalog - NSO related Sessions

1. Robot Framework - Automating Cisco NSO testing - TSCSPG-2011, Maciej Godlewski, Software Consulting Engineer
2. Significance of CX Test Automation Framework for Validating Service Provide Technology – DEVNET-2959, Ronak Gera, Cisco Consulting Engineer, Ravi Dhotarad, Cisco Consulting Engineer.
3. Infrastructure-as-Code with NSO - Maintain and deploy network services via NSO using a CICD Pipeline - DEVWKS-3984, Oliver Boehmer, Principal Architect, - **Distinguished Speaker,** Maciej Małysz, Customer Delivery Software Architect
4.