

**Automated Testing for Your Network**

**HOLPRG-2004**

**Speakers:**

**Sergey Sazhin**, Technical Leader, CX

**Alexey Sazhin**, Technical Leader, CX

**Vladimir Savostin**, Technical Leader, CX

**Monther Koujeh**, Customer Support Engineer

# Learning Objectives

During this lab, you will get hands-on experience with pyATS which is a vendor-agnostic suite of libraries for Python. You will learn how to build your automated tests, using those frameworks.

pyATS opens a wide variety of opportunities and soon you will see it’s not hard to start using them.

During this lab, we will show you real-world examples that you can use to start implementing the automation of tests in your network.

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

1. Create a testbed file.
2. Understand the main capabilities of pyATS suites.
3. Start writing automated tests for your network.
4. Render test results with XPRESSO (pyATS Web Dashboard).

Throughout this lab you will work with and learn the following features and concepts:

1. pyATS testbed file
2. pyATS shell
3. Device.connect() and Device.execute() methods
4. pyATS test script structure
5. pyATS testcases
6. Logging in pyATS
7. pyATS Parse
8. pyATS Learn
9. find\_links method to find all the links between devices.
10. pyATS Run
11. pyATS job file
12. pyATS Log Viewer
13. XPRESSO (pyATS Web Dashboard)

# Scenario

Imagine the following situation is happening:

Josh was awakened by an early call today. Something is broken again after his works this night.

Josh has realized it couldn't continue like this, and thought came to him that if he could learn to test his network automatically before and after work, his life would become happier.

He would feel calmer, more cheerful and he would have time for a cup of espresso he likes so much.

This lab guide caught his eye and he began to study pyATS following it. Let’s start learning pyATS with Josh!

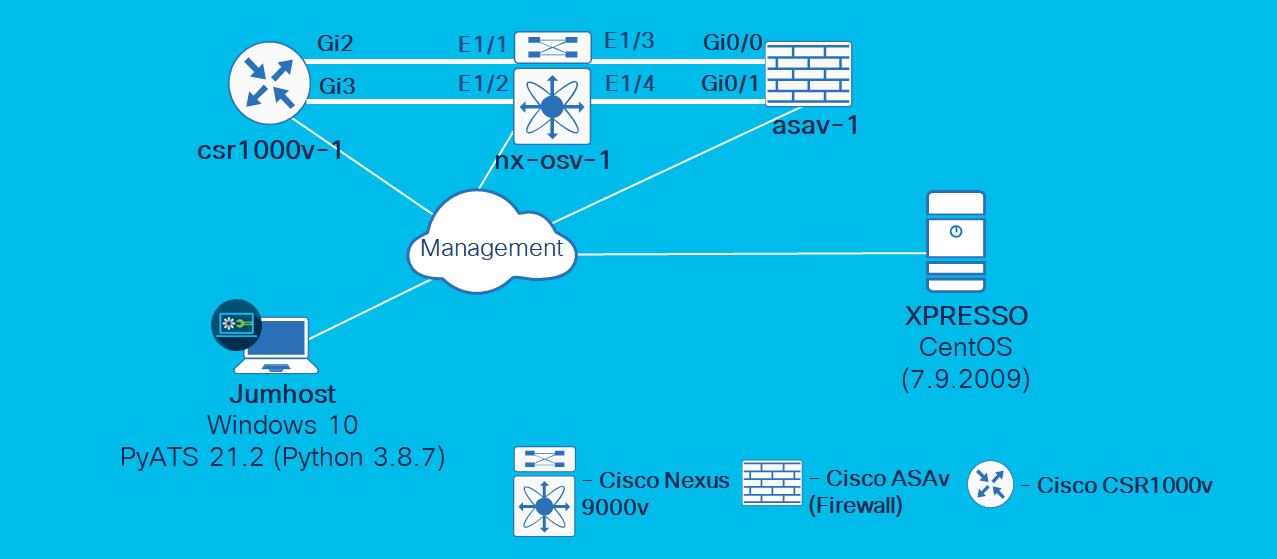
Figure 1: Josh's Adventure Today



This lab includes a CML server on which simulation of the lab network will be running.

All lab tasks will be done from the Jumphost (a Windows 10 machine with WSL installed).

# Network Diagram



# Equipment Details

|  |  |  |  |
| --- | --- | --- | --- |
| Hostname | Credentials | IP address | Connection protocol |
| Jumphost | **Username: administrator**  **Password: C1sco12345** | 198.18.133.252 | Microsoft Remote Desktop (RDP) |
| asav-1 | Password: **cisco**  Enable Password: **cisco** | 198.18.1.202 | Telnet |
| csr1000v-1 | Password: **cisco**  Enable Password: **cisco** | 198.18.1.201 | Telnet |
| nx-osv-1 | login: **cisco**  password: **cisco** | 198.18.1.203 | Telnet |
| xpresso | login: **xpresso**  password: **C1sco12345** | 198.18.134.50 | SSH |

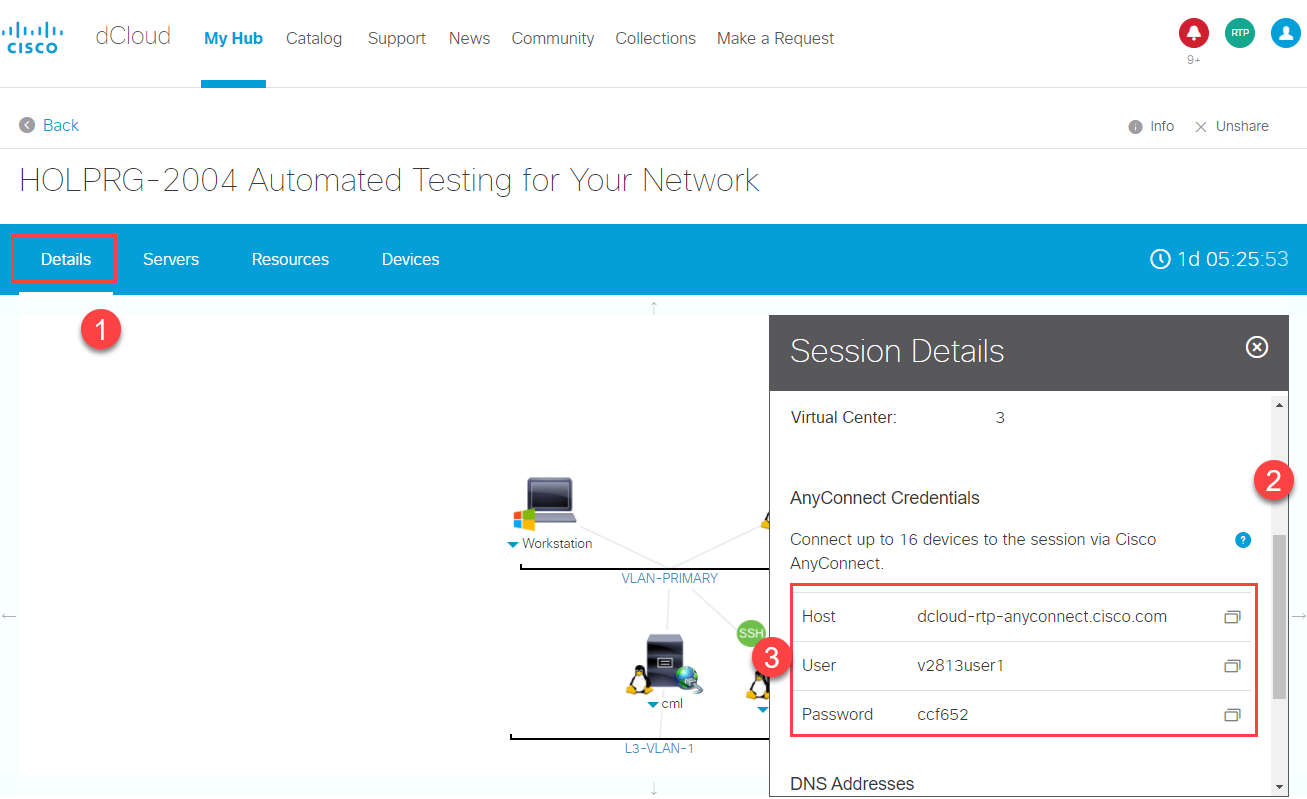
# Component Details

|  |  |  |
| --- | --- | --- |
| Hostname | Product | Operating System |
| Jumphost | Microsoft Windows 10  WSL 1 with Ubuntu 18.04 | Microsoft Windows 10 |
| asav-1 | Cisco Adaptive Security Virtual Appliance | ASA OS 9.14(1) |
| csr1000v-1 | Cisco Cloud Services Router 1000v | IOS-XE 17.3.1a |
| nx-osv-1 | Cisco Nexus 9000v Switch | NS-OS 9.2(4) |
| xpresso | XPRESSO  (pyATS Web Dashboard) | CentOS Linux release 7.9.2009 |

# Get Started

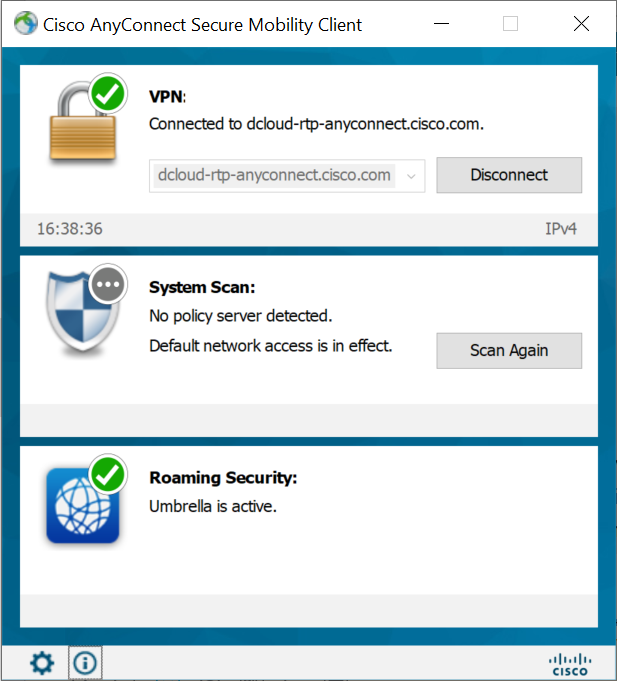
1. Go to dCloud <https://dcloud.cisco.com/>
2. Authenticate using your Cisco (CCO) credentials
3. Paste Event URL into your browser and click Enter, you will be forwarded to your session page
4. On the session, page click **Details** (1) and scroll down (2) for Anyconnect Credentials (3). You will need these credentials to get access to your lab, using Cisco AnyConnect client.

Figure 2: Copy dCloud Info



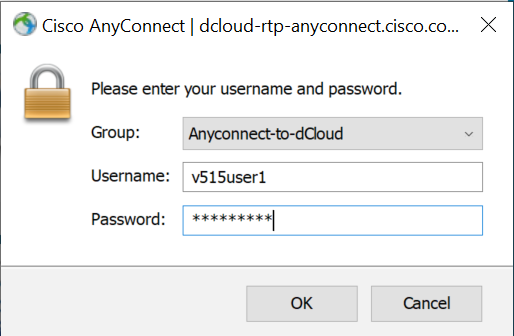
1. Cisco AnyConnect client and **Host** address from the previous step.

Figure 3: Cisco AnyConnect Client Connection Window



1. When the login banner will appear, enter Username/Password from the previous step.

Figure 4: Cisco AnyConnect Client Login Window

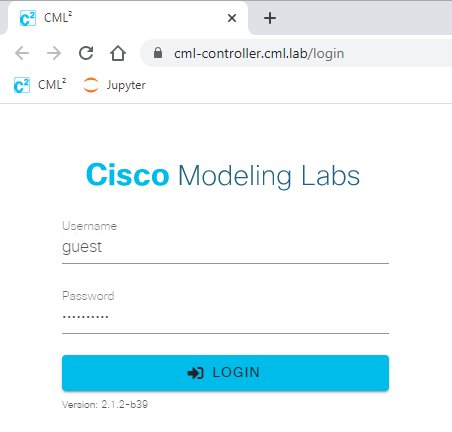


1. For best performance, connect to the workstation with **Cisco AnyConnect VPN** [[Show Me How](https://dcloud-cms.cisco.com/help/install_anyconnect_pc_mac)] and the **local RDP client on your laptop** [[Show Me How](https://dcloud-cms.cisco.com/help/local_rdp_mac_windows)]

Workstation 1: **198.18.133.252**, Username: **administrator**, Password: **C1sco12345**

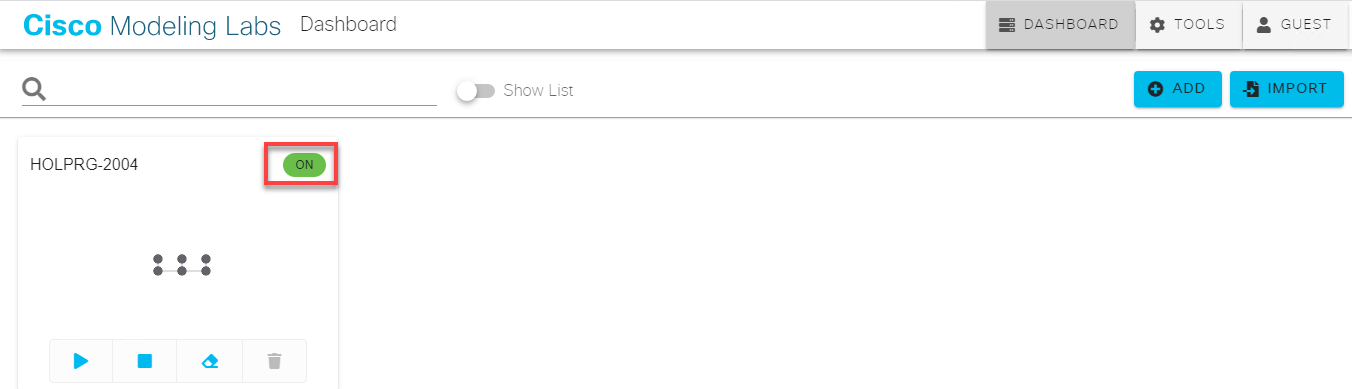
1. Open Google Chrome browser, startup page <https://cml-controller.cml.lab/login> would be opened (web interface of Cisco Modeling Labs server).
2. Press Login button:

Figure 5: Sign in to Cisco Modeling Labs (CML)



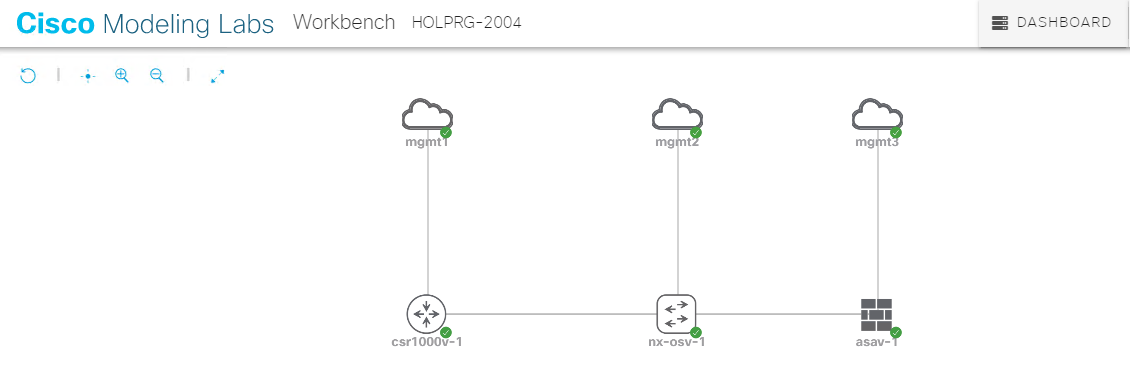
1. On the opened page ensure the lab HOLPRG-2004 is in ON state:

Figure 6: Ensure the lab is in ON state



1. Click on the topology and ensure on the opened page status for all devices is green:

Figure 7: Ensure status of all devices is green



If the status of any device is not green 10 minutes after CML topology has been started, refer to the lab’s proctor for assistance.

1. On the remote desktop, double-click the PuTTY shortcut icon.

Verify connectivity by launching the **asav-1**, **csr100v-1**, and **nx-osv-1** devices from the remote desktop and logging in. Username/password for all three devices: **cisco**/**cisco**.

If all devices are reachable and you can log in, close the **PuTTY** sessions and proceed with **Scenario 1**.

1. Exploring the Lab Structure

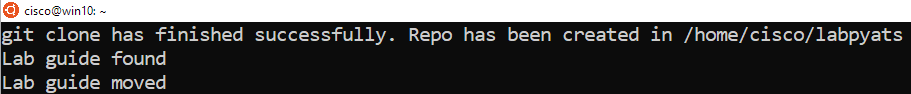
Value Proposition: Understand the tools used in the lab and the lab’s structure.

Steps

1. On the remote desktop, double-click the **Ubuntu 18.04 LTS** shortcut.



1. Ubuntu will run on our RDP Jumphost on top of Windows 10. The bash Linux shell appears.



When Linux shell is opened for the first time, a script will automatically download the current version of Lab Guide to the desktop of Jumphost. The Lab Guide would be saved with the following name: Lab\_Guide.docx on a desktop of Jumphost:

You might prefer to open a Lab Guide from the desktop of Jumphost and follow it there, especially if you have one display (in this case it might be inconvenient to switch between RDP and Lab Guide on your PC).

Throughout the lab, you will be working from a virtual environment. The virtual environment provides the following major advantages over running Python scripts globally:

* Project Isolation: Avoids installing Python packages globally which could break system tools or other projects.
* Dependency Management: Makes the project self-contained and reproducible by capturing all package dependencies in a requirements file.

Cisco recommends that you run pyATS scripts from the virtual environment. The keyword (pyats) at the beginning of each line indicates that you are working from a virtual environment.

1. Change to the directory that contains the lab files:

(pyats) cisco@win10$ cd ~/labpyats

(pyats) cisco@win10$~/labpyats

1. Check the lab’s structure (before running the command shown below, ensure that you have changed to the correct directory: **~/labpyats**).

(pyats) cisco@win10:~/labpyats$ ls -l

Check the list of files and refer to the description of each file depicted in the table below (see Table 1).

Files used in the lab

**NOTE:** The following files are in the **labpyats** directory and will be used throughout the lab.

Table 1: Files used in the lab

|  |  |  |
| --- | --- | --- |
| Filename | Description | Task # |
| pyats\_testbed.yaml | Testbed file for pyATS (in YAML) | - |
| task10\_runtestsjob.py | pyATS job file to run tests from Task 10 | Scenario 10 |
| Task4\_labpyats.py | The task for collection of show commands | Scenario 4 |
| Task5\_labpyats.py | The initial task for pyATS exploration | Scenario 5 |
| task61\_labpyats.py | Test to verify log messages (asav-1 only) | Scenario 6 |
| task62\_labpyats.py | Test to verify log messages (all devices) | Scenario 6 |
| task7\_labpyats.py | Test to verify service contracts coverage | Scenario 7 |
| task8\_labpyats.py | Test to verify routing information | Scenario 8 |
| task9\_labpyats.py | Test to verify reachability between devices (ping) | Scenario 9 |

Command Prompt Conventions

Throughout this lab we use the following prompt conventions at the beginning of the first line of a command or code statement:

$ - commands appearing after the $ sign are input in the bash shell (be sure you copy only the commands, and not the $ sign).

In [1]: - code going after this statement is input into the pyATS shell (be sure you copy only the code, and not the In [1]: statement). This statement goes only on the first line of code.

Out [1]: - signifies output in pyATS shell (in reaction to the code input into pyATS shell, which follows and In [1]: statement)

This concludes Scenario 1.

1. Explore pyATS Testbed File

Value Proposition: In this task, we will explore pyATS testbed file used in the lab.

For pyATS to be able to work with network topology, it must know the following basic information: management interfaces, IP addresses, connection protocol and connections between network devices.

This information is stored in pyATS testbed file (in YAML format).

Could this information be gathered automatically and pyATS testbed file prepared for us?

Sure, since the lab is running in Cisco Modeling Labs (CML), cmlutils**[[1]](#footnote-2)** tool could be used to prepare pyATS testbed file from CML topology.

The testbed file was already prepared and you **don’t** need to call **cmlutils** now. But if you are curious how you can prepare pyATS testbed file from CML topology, then the needed commands are:

source /var/lib/virlutils/venv/bin/activate

cml ls (get id from **ID** column)

cml use --id XXX

cml generate pyats

source ~/pyats/bin/activate

Steps

The Testbed YAML file for pyATS has been pre-created for this lab and is named **pyats\_testbed.yaml**.

1. Change to the directory that contains the lab files:

(pyats) cisco@win10$ cd ~/labpyats

1. Open the pre-created **pyats\_testbed.yaml** file in the Nano editor by entering the **nano pyats\_testbed.yaml** command.

(pyats) cisco@win10$~/labpyats $ nano pyats\_testbed.yaml

1. The output of the command should contain the following:

testbed:

name: labpyats <-- the name of topology simulation

credentials: <-- credentials for CLI access (stored as environment variables)

default: <-- username/password used by default

username: "%ENV{PYATS\_USERNAME}"

password: "%ENV{PYATS\_PASSWORD}"

enable: <-- enable password (if required)

password: "%ENV{PYATS\_AUTH\_PASS}"

line: <-- line (VTY/console) password (if required)

password: "%ENV{PYATS\_AUTH\_PASS}"

devices: <-- All necessary information to connect to devices is in this block

asav-1:

alias: asav-1

os: asa

type: ASAv

platform: ASAv

connections:

console:

protocol: ssh

ip: 198.18.1.202

port: 22 <-- connection to a device would be done via SSH port

<…>

topology: <-- All information about links between devices is in this block

asav-1:

interfaces:

GigabitEthernet0/0:

ipv4: 10.0.0.5/30

link: asav-1-to-nx-osv-1

type: ethernet

GigabitEthernet0/1:

ipv4: 10.0.0.9/30

link: asav-1-to-nx-osv-1#1

type: ethernet

<…>

Now we have all the required information to start our tests with pyATS.

NOTE: Note that username and passwords to access devices are not stored in the YAML file:

credentials:

default:

username: "%ENV{PYATS\_USERNAME}"

password: "%ENV{PYATS\_PASSWORD}"

enable:

password: "%ENV{PYATS\_AUTH\_PASS}"

line:

password: "%ENV{PYATS\_AUTH\_PASS}"

We recommend that you store credentials separately and at least as environmental variables.

1. Exit Nano without saving, pressing:

Ctrl + X

1. In the lab environment, variables for PYATS with credentials to access all devices are preconfigured. Check these environment variables from a Bash shell:

$ echo $PYATS\_USERNAME $PYATS\_PASSWORD $PYATS\_AUTH\_PASS

1. The output of the command should contain the following:

cisco cisco cisco

This concludes Scenario 2.

1. Observe pyATS Capabilities using the pyATS Shell

Value Proposition: We believe it’s always faster to start learning about pyATS in an interactive format where you can try different pyATS functions. It allows you to run commands and see results immediately, instead of making small changes in Python code and re-running it after every minor change.

pyATS has an interactive shell for developing tests. It’s run from Bash shell in the following way:

pyats shell --testbed-file <testbed\_file\_name>

We will call the pyATS interactive shell “pyATS shell” throughout this guide.

Let’s begin with the pyATS shell, using it with pyats\_testbed.yaml, which we’ve seen in the previous scenario.

The following pyATS methods would be used in this scenario:

* pyATS connect
* pyATS execute
* pyATS parse
* pyATS learn
* pyATS diff

Steps

1. Enter the following command to run the pyATS shell command from the Bash shell and specify the YAML testbed file as the parameter. After about 10 seconds, the interactive shell opens. This is where you can input the Python code.

$ pyats shell --testbed-file pyats\_testbed.yaml

It might take up to 10-15 seconds for pyATS shell to load

1. The output of the command should contain the following (version of Python might be different):

Welcome to pyATS Interactive Shell

==================================

Python 3.8.7 (default, Mar 30 2021, 10:31:15)

[GCC 7.4.0]

1. Check the devices included in the lab’s testbed.

In [1]: testbed.devices

1. The output of the command should contain the following:

Out[1]: TopologyDict({'asav-1': <Device asav-1 at 0x7f5342e17210>, 'csr1000v-1': <Device csr1000v-1 at 0x7f5342deced0>, 'nx-osv-1': <Device nx-osv-1 at 0x7f5341998890>})

1. Create variables (Python objects) to call devices easily (nx - 'nx-osv-1' device, asa -'asav-1' device):

In [1]: nx = testbed.devices['nx-osv-1']

asa = testbed.devices['asav-1']

1. Connect and collect raw output from devices.
   1. Connect to devices from the pyATS shell:

In [1]: nx.connect()

asa.connect()

* 1. Let's prepare ourselves for our first test and collect show inventory command output from the devices.

In [1]: nx.execute('show inventory')

asa.execute('show inventory')

1. Verify the collected information in the output of each command. Pay attention to the output of both execute methods returned as plain text (string type in Python):

nx-osv-1#

Out[6]: 'NAME: "Chassis", DESCR: "Nexus9000 9000v Chassis" \r\nPID: N9K-9000v , VID: V02 , SN: 9OQ8QSK7JX1 \r\n\r\nNAME: "Slot 1", DESCR: "Nexus 9000v Ethernet Module" \r\nPID: N9K-9000v , VID: V02 , SN: 9OQ8QSK7JX1 \r\n\r\nNAME: "Fan 1", DESCR: "Nexus9000 9000v Chassis Fan Module" \r\nPID: N9K-9000v-FAN , VID: V01 , SN: N/A \r\n\r\nNAME: "Fan 2", DESCR: "Nexus9000 9000v Chassis Fan Module" \r\nPID: N9K-9000v-FAN , VID: V01 , SN: N/A \r\n\r\nNAME: "Fan 3", DESCR: "Nexus9000 9000v Chassis Fan Module" \r\nPID: N9K-9000v-FAN , VID: V01 , SN: N/A'

asav-1#

Out[7]: 'Name: "Chassis", DESCR: "ASAv Adaptive Security Virtual Appliance"\r\nPID: ASAv , VID: V01 , SN: 9AWXBH2QJP7'

1. Collect structured data output using **parse** command.
   1. Import pprint python module to represent collected output in pretty format.

In [1]: from pprint import pprint

* 1. Run parse command to convert device output into a Python dictionary, which stores the device data as a set of key-value pairs.

In [1]: out = nx.parse('show inventory')

* 1. Verify collected information using **pprint** command.

In [1]: pprint(out)

{'name': {'Chassis': {'description': 'Nexus9000 9000v Chassis',

'pid': 'N9K-9000v',

'serial\_number': '9EIFZPG7ZAM',

'slot': 'None',

'vid': 'V02'},

'Fan 1': {'description': 'Nexus9000 9000v Chassis Fan Module',

'pid': 'N9K-9000v-FAN',

'serial\_number': 'N/A',

'slot': 'None',

'vid': 'V01'},

'Fan 2': {'description': 'Nexus9000 9000v Chassis Fan Module',

'pid': 'N9K-9000v-FAN',

'serial\_number': 'N/A',

'slot': 'None',

'vid': 'V01'},

'Fan 3': {'description': 'Nexus9000 9000v Chassis Fan Module',

'pid': 'N9K-9000v-FAN',

'serial\_number': 'N/A',

'slot': 'None',

'vid': 'V01'},

'Slot 1': {'description': 'Nexus 9000v Ethernet Module',

'pid': 'N9K-9000v',

'serial\_number': '9EIFZPG7ZAM',

'slot': '1',

'vid': 'V02'}}}

* 1. Since information is collected in a Python dictionary, we can call any value, using its key. Collect a serial number of the chassis using its key.

In [1]: print(out['name']['Chassis']['serial\_number'])

9EIFZPG7ZAM

1. Collect features state using **learn** command.
   1. Run **learn** command to get the state of the feature (**ospf** in our case) into a Python dictionary, which stores the device data as a set of key-value pairs.

In [1]: ospf\_state\_before = nx.learn('ospf')

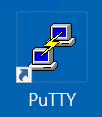
* 1. Print collected output to observe the structure of the Python dictionary.

In [1]: pprint(ospf\_state\_before.info)

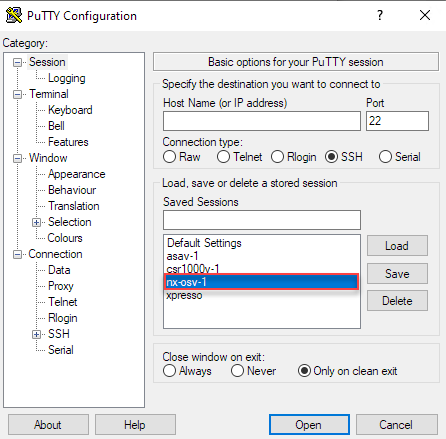
* 1. Run parse command to collect interfaces output when network.

In [1]: int\_before = nx.parse('show interface')

* 1. Now impose failure in the topology, shutting down the interface **Ethernet1/1** on device **nx-osv-1**.
     1. Open Putty terminal



* + 1. Connect to **nx-osv-1** using password **cisco**



* 1. Disable (input **shutdown** command) interface **Ethernet1/1** on **nx-osv-1**.

Input the following commands in the console of **nx-osv-1**:

configure terminal

interface Ethernet1/1

shutdown

* 1. In pyATS shell, run **learn** the command to get the state of the feature (**ospf** in our case) into a Python dictionary, which stores the device data as a set of key-value pairs.

In [1]: ospf\_state\_after = nx.learn('ospf')

* 1. Import PyATS Diff package and compare previous (working) and current state (failed) to understand what has changed and further troubleshoot the problem.

In [1]: from genie.utils.diff import Diff

diff = Diff(ospf\_state\_before.info, ospf\_state\_after.info)

diff.findDiff()

print(diff)

* 1. PyATS Diff can compare outputs of structured data collected by parse command.
  2. Parse “show interface” to collect interfaces output into a Python dictionary.

In [1]: int\_after = nx.parse('show interface')

* 1. Compare before and after outputs, using PyATS Diff package.

In [1]: diff2 = Diff(int\_before, int\_after)

diff2.findDiff()

print(diff2)

* 1. Enable (input no shutdown command) interface Ethernet1/1onnx-osv-1.

Input the following commands in the console of nx-osv-1:

configure terminal

interface Ethernet1/1

no shutdown

1. Exit the pyATS shell by using the **exit** command and proceed to step 11.
2. PyATS **parse/learn** and **diff** commands can be run right from Linux Shell, and you can start using PyATS without coding skills.
   1. Observe PyATS capabilities from Linux Shell
   2. Run pyATS parse command from Linux Shell:

pyats parse "show interface" --devices nx-osv-1 --testbed-file pyats\_testbed.yaml --output parse-work/

* 1. Run pyATS learn command from Linux Shell for feature OSPF:

pyats learn ospf --devices nx-osv-1 --testbed-file pyats\_testbed.yaml --output working/

* 1. Disable (input shutdown command) interface **Ethernet1/1** onnx-osv-1**.**

configure terminal

interface Ethernet1/1

shutdown

* 1. Return to Linux Shell and collect outputs after failure:
  2. Run pyATS parse command from Linux Shell:

pyats parse "show interface" --devices nx-osv-1 --testbed-file pyats\_testbed.yaml --output parse-failed/

* 1. Run pyATS learn command from Linux Shell for feature OSPF:

pyats learn ospf --devices nx-osv-1 --testbed-file pyats\_testbed.yaml --output failed/

* 1. Run pyATS diff for parsed commands from Linux Shell:

pyats diff parse-work parse-failed

cat ./diff\_nx-osv-1\_show-interface\_parsed.txt

* 1. Run pyATS diff for learned states from Linux Shell:

pyats diff working failed

cat ./diff\_ospf\_nxos\_nx-osv-1\_ops.txt

* 1. Don’t forget to enable (input no shutdown command) interface **Ethernet1/1** onnx-osv-1**:**

configure terminal

interface Ethernet1/1

no shutdown

This concludes Scenario 3.

1. Collect Show Commands from the Network Devices

Josh is curious about more practical use-cases for pyATS so let’s start with them.

**That’s’ fine that I have known so far, but how I can use this knowledge in my day-to-day work?**

**That’s’ fine that I have known so far, but how I can use this knowledge in my day-to-day work?**

Value Proposition: In this task, we will use the knowledge we have gained from the previous task to write a script that collects 'show inventory' command from each device in the testbed. The output of this command will be saved in the file, created by the script **collected\_task4.** These outputs can be used later if you want to compare the future state of the network with the current one. Since it’s required to collect outputs from all the devices in the testbed, in this task we will work with the **testbed.devices** object, and iterate over all the devices contained in this object to collect an output of the required commands from each device.

Steps

1. Let's connect to pyATS and check parts of the code before running the final script. In the beginning, we will check the structure of **testbed.devices** object.

$ pyats shell --testbed-file pyats\_testbed.yaml

In [1]: print(testbed.devices)

1. Check the output.

Out[1]: TopologyDict({'asav-1': <Device asav-1 at 0x7f60bef1da90>, 'csr1000v-1': <Device csr1000v-1 at 0x7f60beee73d0>, 'nx-osv-1': <Device nx-osv-1 at 0x7f60bda8d850>})

**NOTE:** As you can see from the output in the previous step, **'Device <device\_name>'** objects are contained as dictionary values in the object of **TopologyDict** class. The device names are used as dictionary keys.

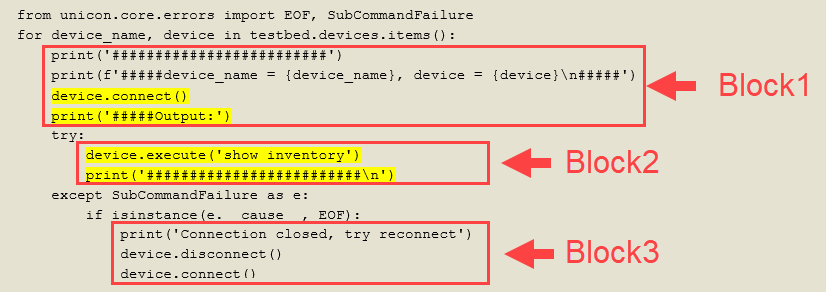
1. In this task, we will apply the standard dictionary method: **items()** to get **keys** (device names) and **values** (respective device objects). For iteration, the **for** loop will be used:

Table 2: Code Description

|  |  |
| --- | --- |
| for **device\_name**, **device** in testbed.devices.items():  device.connect() | **device\_name** - stores hostname of a device  **device** - stores device object |
| try:  **device.execute**('show inventory')  print('#########################\n') | **device.execute()** method - will be used to get the output of **'show inventory'** command |

NOTE: Python uses **Indentation** (the spaces at the beginning of a code line) to specify code blocks.

Figure 8: Python Indentation Example



NOTE: To preserve indentation during copy and paste, we will use the approach with **%cpaste** below.

1. Paste the following snippet to pyATS console:
   1. Place the following iPython command in the beginning of code:

%cpaste

* 1. Copy and paste the code into the pyATS console:

from unicon.core.errors import EOF, SubCommandFailure

for device\_name, device in testbed.devices.items():

print('#########################')

print(f'#####device\_name = {device\_name}, device = {device}')

print(f'#####device\_name = {device\_name}, device\_object\_type = {type(device)}')

device.connect(log\_stdout=False)

print('#####Output:')

try:

out = device.execute('show inventory')

print(f'{out}')

except SubCommandFailure as e:

if isinstance(e.\_\_cause\_\_, EOF):

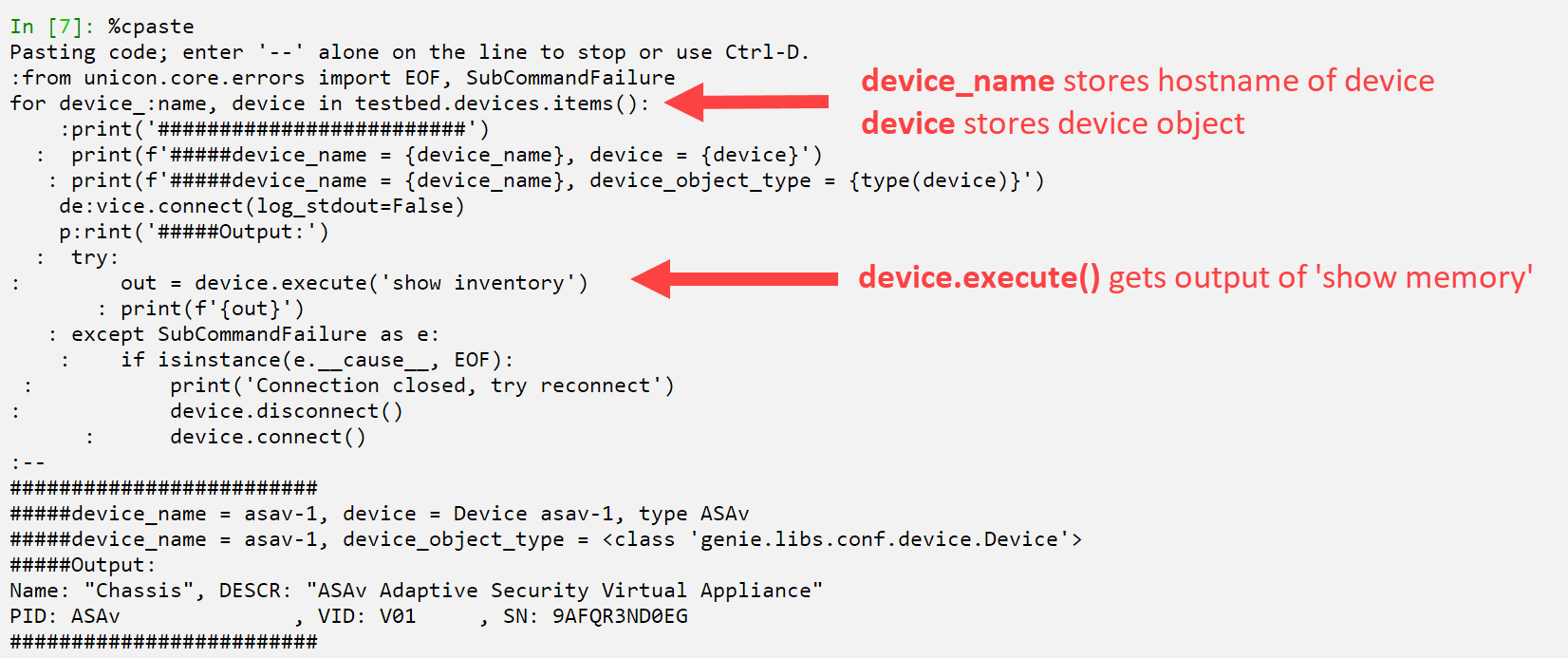
print('Connection closed, try reconnect')

device.disconnect()

device.connect()

* 1. End the code with --

Figure 9: Code Insertion Result



1. Check the result of this code. Now each device should return the output of **'show inventory'** command.

**NOTE:** If a device connection is closed or terminated unexpectedly after it has already connected to a device, there will be multiple errors generated (for example, the Python **EOF** exception would be invoked) at the time of command execution.

To handle this situation, it’s required to add the following code to reconnect to a device in case a broken connection to a device has been detected:

from unicon.core.errors import EOF, SubCommandFailure

try:

device.execute('show inventory')

except SubCommandFailure as e:

if isinstance(e.\_\_cause\_\_, EOF):

print('Connection closed, try reconnect')

device.disconnect()

device.connect()

Also, refer to the “EOF Exception handling” chapter in the following document for more details:

<https://pubhub.devnetcloud.com/media/unicon/docs/user_guide/services/generic_services.html>

1. Exit the pyATS shell by using the exit command. Now we are ready to go through the final version of the script gathering commands specified from all the devices in the testbed and saving them to file on Linux (proceed to step 7).
2. Open the prepared script task4\_labpyats.py in Nano editor.

$ nano task4\_labpyats.py

1. Before diving into the details of the code, study the explanation of the code given below.

The script task4\_labpyats.py has the following Python functions:

Table 3: task4\_labpyats.py Main Functions

|  |  |
| --- | --- |
| def main() | The main function (see “1” on the illustration that follows). It calls the collect\_device\_commands function (see “2”). |
| def collect\_device\_commands( command\_to\_gather, output\_filename) | The function that does most of the job. The main tasks this function performs include the following:   1. Iterates over devices - see “3” (in the way it has been done in the previous steps). For each device: 2. Connects to the device (see “4”). 3. Writes output of the commands for this device to a file – see “5”. |
| def write\_commands\_to\_file(abs\_filename, command\_output) | This is a supplementary function and it’s used to write the output of commands to a file (see “5”). |

NOTE: To simplify the script, the name of the testbed is hard-coded into the main():

testbed\_filename = '/home/cisco/labpyats/pyats\_testbed.yaml'

In further scripts, the name of a testbed file will be input as a parameter of the script.

Figure 10: TASK4\_LABPYATS.PY Code Structure



NOTE:

log\_stdout=False option in device.connect call:

device.connect(log\_stdout=False)

It will disable all logging to a screen for the whole connection session to this device (until disconnection takes place or until log\_stdout is set to True).

For the script collecting many commands, it would be preferred to prune the output of the commands to the console using this method.

1. Exit Nano without saving, pressing:

**Ctrl + X**

1. Now run the script:

$ python task4\_labpyats.py

1. Check that there is a new file created: **collected\_task4**. Check the time when it was created.

$ ls -l ~/labpyats | grep collected\_task4

Sample output in a Bash shell:

-rw-rw-rw- 1 cisco cisco 1126 Mar 21 13:02 collected\_task4

1. Check content of collected\_task4 file.

$ cat collected\_task4

This concludes Scenario 4.

1. Write the Test Script using pyATS Library

Value Proposition: Now let’s write our first test script on Python using the pyATS library. Our test script will connect to all the devices in the testbed and print the results of the connection. If the connections to all the devices are successful, then the test will pass, else it will fail. Using this simple test script, we will learn the structure of the pyATS test script file.

The pyATS test script is a file with the Python code which uses the pyATS library.

The structure of the pyATS test script is modular and straightforward.

Each test script is written in a Python file and split into three major sections (Python classes) – see the illustration below for a graphical representation:

* Common Setup: The first section in the test script, run at the beginning. It performs all the "common" setups required for the script.
* Testcase(s): A self-contained individual unit of testing. Each testcase is independent of the other testcases.
* Common Cleanup: The last section in the test script, performs all the "common" cleanups at the end of execution.

Each of these sections is further broken down into smaller subsections (Python methods of the class).

**NOTE:** Both Common Setup and Common Cleanup could be only one in a script, whereas there might be multiple test cases in one test script.

Figure 11: Structure of the pyATS test script



Steps

1. Let's verify our first test script. Open the file **task5\_labpyats.py** and observe its structure:

$ nano task5\_labpyats.py

1. Pay special attention to the following part of the code. Whereas it's not related to only this task, it will help you understand the logging capabilities of pyATS that would be used in other tasks in this lab:

# Import of pyATS logging banner

from pyats.log.utils import banner

<..>

# This section sets up logging (ensure the log.level is the same or higher than

# level of log.info where a banner is used)

global log

log = logging.getLogger(\_\_name\_\_)

log.setLevel(logging.INFO)

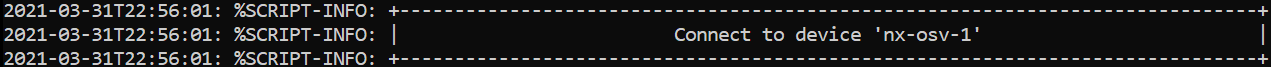
<..>

# Use pyATS logging banner to format the output

log.info(banner(f"Connect to device '{device.name}'"))

1. When the pyATS logging banner is used, the following format of message would be shown in the output of the test.

Figure 12: Output of pyATS Logging Banner



**NOTE:** The pyATS logging banner itself does not perform logging, and instead only performs style formatting of its input messages. Hence, the **log.info(banner("logging message")**) construction is used in the code for logging.

Since the banner is logged with INFO logging level, it’s required to set logging level up to INFO (default is WARNING):

log.setLevel(logging.INFO)

1. Let’s look at the main contents of this example.

Python class **common\_setup** which is inherited from **aetest.CommonSetup** represents the major section “Common Setup” (see the following illustration). The Python class **common\_setup** is where initializations and preparations before the actual script’s testcases should be performed. For this reason, code in class **common\_setup** is always run first, before all the testcases.

Refer to the description of the code of this Python class shown below:

class common\_setup(aetest.CommonSetup):

@aetest.subsection

def establish\_connections(self, pyats\_testbed):

# Pass testbed file into class method

device\_list = []

# Load all devices from testbed file and try to connect to them

for device in pyats\_testbed.devices.values():

log.info(banner(f"Connect to device '{device.name}'"))

try:

device.connect(log\_stdout=False)

except errors.ConnectionError:

self.failed(f"Failed to establish "

f"connection to '{device.name}'")

device\_list.append(device)

# Pass list of devices to testcases

self.parent.parameters.update(dev=device\_list)

The following code is used to load testbed file from filename specified as command-line option (**--testbed** is a command line key, **dest** – specifies name of object that would represent testbed file in code):

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

# Load testbed file which is passed as command-line argument

parser.add\_argument('--testbed', dest='pyats\_testbed',

type=loader.load)

args, unknown = parser.parse\_known\_args()

aetest.main(\*\*vars(args))

1. Exit Nano without saving, pressing:

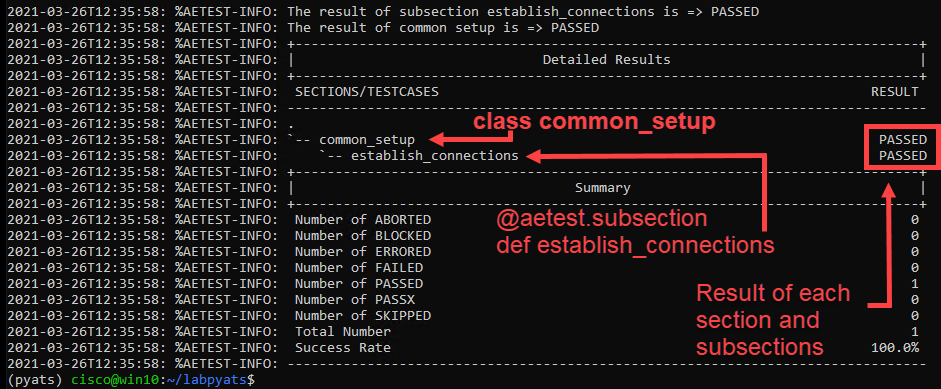
**Ctrl + X**

1. Let's run our first test script. This test script will try to connect to all the devices in the testbed and print the results of these attempts:

$ python task5\_labpyats.py --testbed pyats\_testbed.yaml

1. Upon finishing the test script, pyATS generates a report of Success/Failed testcases, the **common\_setup** section is also treated as the testcase with subsection **establish\_connections**. Since all the devices are reachable, the testcases should finish successfully (PASSED). Refer to the following illustration.

Figure 13: Scenario 5 output



This concludes Scenario 5.

1. Verify Log Messages

Value Proposition: In this test case, we will verify that the logging messages with ERROR or WARN are not present on the devices in the testbed.

The high-level logic of the test case will be as follows:

* Connect to each device in the testbed.
* Collect the output of show logging | i ERROR|WARN.
* If the output contains more than 0 strings, then ERROR messages were found and the test should fail for this device. Otherwise, the test should succeed.

Steps

1. Before creating our testcase, connect to ASA. Launch **PuTTY** and connect to **asav-1**.

User Access Verification

Password: cisco

asav-1> **enable**

Password: cisco

asav-1#

asav-1# clear logging buffer

asav-1#

1. Let's open the pyATS shell and check it out.

$ pyats shell --testbed-file pyats\_testbed.yaml

1. Input the following code into pyATS shell:

In [1]: csr = testbed.devices['csr1000v-1']

asa = testbed.devices['asav-1']

csr.connect()

asa.connect()

1. Let's verify whether there are any errors or warning messages in the logs:

In [1]: out1 = csr.execute('show logging | i ERROR|WARN')

out2 = asa.execute('show logging | i ERROR|WARN')

**NOTE:** The output for ASA should be empty.

If you don’t see any ERROR logs on the **csr1000v-1** device, then:

1. Connect to CSR:

Launch PuTTY and connect to **csr1000v-1**. Username: **cisco**, password: **cisco**

1. Generate a test ERROR message:

csr1000v-1# send log 'Test ERROR message for pyATS'

1. Repeat step 3 above for CSR in the pyATS shell:

out1 = csr.execute('show logging | i ERROR|WARN')

1. To check whether there is an empty or non-empty output, we will use the Python **len()** built-in function, which returns the length of the given string. If the collected output is empty, then **len()** of the output will be 0, otherwise, the result will be greater than 0.

Input into pyATS shell:

In [1]: **len**(out2)

The resulting length is 0, which means output from ASA is empty:

Out [1]: **0**

Input into pyATS shell:

In [1]: len(out1)

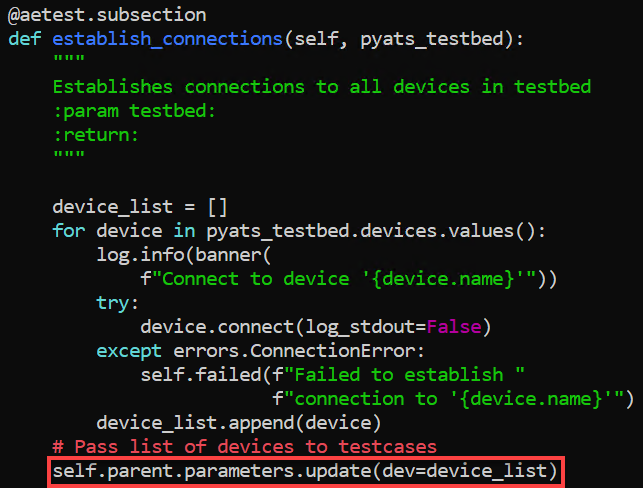
The resulting length is greater than 0, which means the output from CSR is not empty:

Out [1]: **664**

1. Exit pyATS shell using the **exit** command.
2. Open the file **task61\_labpyats.py** in Nano editor:

$ nano task61\_labpyats.py

This file reuses the **establish\_connections(self, testbed)** method from **task5\_labpyats.py** (used in previous scenarios), which make connections to all devices in the testbed.

NOTE: Pay special attention to the method self.parent.parameters.update(dev=device\_list), located at the end of the establish\_connections(self,testbed) method.

Where **self.parent.parameters** is an attribute of class **aetest**, and **aetest** is the class from which all the testcase classes and **MyCommonSetup** class are inherited from:

class MyCommonSetup(aetest.CommonSetup):

<…>

class VerifyLogging(aetest.Testcase):

<…>

Using **self.parent.parameters** attribute arguments can be passed between different classes.

As an example, in the class **MyCommonSetup**, we store all the devices from the variable **device\_list** in the parameter **parameters['dev']**.

self.parent.parameters.update(dev=device\_list)

Then we can access all the devices in class VerifyLogging, using the method **self.parent.parameters['dev']**.

1. Pay special attention to the code in class **VerifyLogging**, which is used to implement the approach that has been tested using the pyATS shell: if the length of the output is greater than zero, it means that output contains ERROR or WARN message. Testcase should be marked as Failed in this case.

NOTE: device.connect(log\_stdout=False) is used in this example (see def establish\_connections).

This code (log\_stdout=False) - disables all logging to a screen for the whole connection session. To make the execution of the command on a device visible (show logging | i ERROR|WARN) in the output of the test, the following code is used:

any\_device.log\_user(enable=True)

class VerifyLogging(aetest.Testcase):

<..>

@aetest.test

def error\_logs(self):

any\_device = self.parent.parameters['dev'][0]

any\_device.log\_user(enable=True)

output = any\_device.execute('show logging | i ERROR|WARN')

if len(output) > 0:

self.failed('Found ERROR in log, review logs first')

else:

pass

1. Note that the Setup section of the test case is not used, therefore a **pass** is written in this function. We will use the Setup section of the test case later when we execute the **show logging | i ERROR|WARN** command on multiple devices.

@aetest.setup

def setup(self):

pass

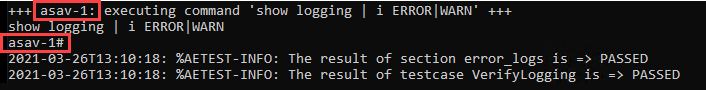
1. Exit Nano without saving, pressing:

**Ctrl + X**

1. Execute the test script task6**1\_labpyats.py** and check the results section.

$ python task61\_labpyats.py --testbed pyats\_testbed.yaml

The Testcase error\_log will run only for one device. Scroll above the results section and you will see which device is related to this output.



We have learned how to run a testcase for only one device, now we need to get familiar with the aetest.loop method, which will let us repeat an elementary test case (written for one device) for every device in the testbed.

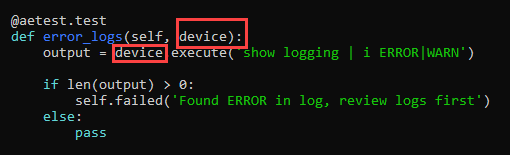
1. Open the file task62\_labpyats.py.

$ nano task62\_labpyats.py

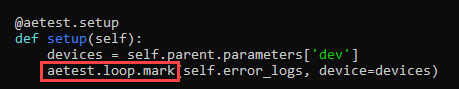
In this task, we will learn dynamic loops, which will create loops based on information that is only available during a script run.

This approach is helpful if we don’t want to hardcode device names inside our testcase, but want to dynamically load devices from testbed file and run testcases across them.

1. Pay special attention to the code in the error\_logs method. It receives the device object as an argument on input and collects the command from this device.



1. Next, check the setup(self) method of class VerifyLogging. Method setup(self) is used to load all the devices from the testbed and to run the error\_logs method for each device.



NOTE: aetest.loop.mark() instructs method self.error\_logs to take an argument for input variable 'device', one-by-one from the devices list and run a testcase for each device separately.

1. Exit Nano without saving, pressing:

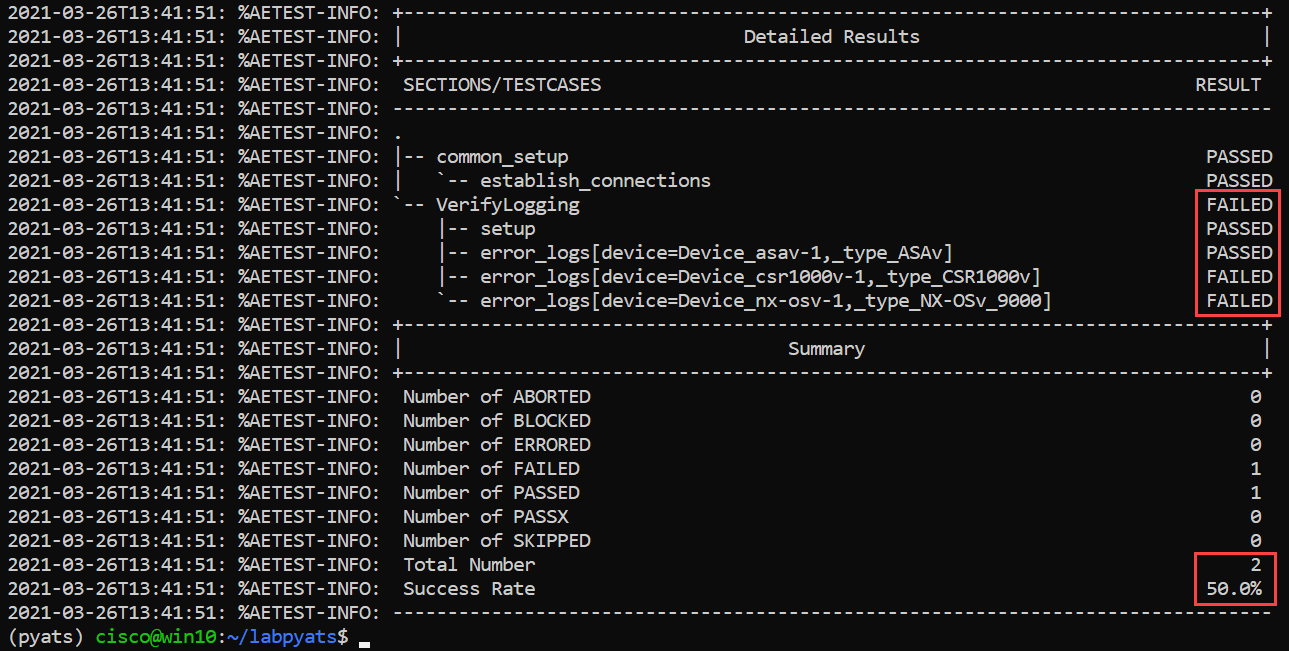
Ctrl + X

1. Execute the test script; testcase error\_logs will run for all the devices in the testbed:

$ python task62\_labpyats.py --testbed pyats\_testbed.yaml

1. Check the VerifyLogging results section. The test for asav-1 should pass, whereas for csr1000v-1 and nx-osv-1 should fail, because these devices have error messages in the logs.

Figure 14: Scenario 6 Output



This concludes Scenario 6.

1. Verify the Service Contracts Coverage

Value Proposition: In this test case, we have the list of devices’ serial numbers, covered by the service contracts, and we must verify that all the devices in the testbed are covered by the service contracts. This ensures you will be able to open a TAC case if something goes wrong when the network is in production.

The high-level logic of the test will be as follows:

* Connect to each device in the testbed.
* Parse the output of show inventory to find the device’s serial number (SN).
* Verify whether SN is in the list, covered by the service contracts.

Steps

1. Let's use the pyATS shell to check our idea.

$ pyats shell --testbed-file pyats\_testbed.yaml

1. Input the following code into the pyATS shell:

In [1]: csr = testbed.devices['csr1000v-1']

asa = testbed.devices['asav-1']

nx = testbed.devices['nx-osv-1']

csr.connect()

asa.connect()

nx.connect()

1. pyATS uses the **parse** method to collect the output of different show commands and parses it into a structured format (Python dictionary). Let's collect the output of 'show inventory' commands and parse it, using the **parse** method.

In [1]: out1 = csr.parse('show inventory')

out2 = asa.parse('show inventory')

out3 = nx.parse('show inventory')

1. Now we can observe the structure of parsed outputs. We are starting with the parsed output for csr1000v-1. Review it and pay special attention to the highlighted sections.

**NOTE:** The Python library **pprint** will be imported in this task. This is used to break the output (Python dictionary) onto multiple lines, which is easier to check, instead of having it all on one line.

In [1]: import pprint

pprint.pprint(out1)

1. Observe the output:

Out [1]: {'main': {'chassis': {'CSR1000V': {'descr': 'Cisco CSR1000V Chassis',

'name': 'Chassis',

'pid': 'CSR1000V',

'sn': '9KZZ4X737UP',

'vid': 'V00'}}},

1. Get the serial number of csr1000v-1:

In [1]: out1['main']['chassis']['CSR1000V']['sn']

1. The result of the code should contain a serial number collected in the previous step[[2]](#footnote-3):

Out [1]: '9KZZ4X737UP'

1. Obtain the parsed output for asav-1.

In [1]: pprint.pprint(out2)

1. Observe the output:

Out [1]: {'Chassis': {'description': 'ASAv Adaptive Security Virtual Appliance',

'pid': 'ASAv',

'sn': '9ABUANH9G5F',

'vid': 'V01'}}

1. Get the serial number of asav-1:

In [1]: out2['Chassis']['sn']

1. The result of the code should contain a serial number collected in the previous step2:

Out [1]: '9ABUANH9G5F'

1. Obtain the parsed output for nx-osv-1.

In [1]: pprint.pprint(out3)

1. Observe the output:

Out [1]: {'name': {'Chassis': {'description': 'Nexus9000 9000v Chassis',

'pid': 'N9K-9000v',

'serial\_number': '9712TV4C2JF',

'slot': 'None',

'vid': 'V02'},

1. Get the serial number of nx-osv-1:

In [1]: out3['name']['Chassis']['serial\_number']

1. The result of the code should contain a serial number collected in the previous step2:

Out [1]: '9712TV4C2JF'

Now we have all the needed information to write the next test script.

1. Exit the pyATS shell using the **exit** command.
2. Open the file **task7\_labpyats.py** in Nano editor:

$ nano task7\_labpyats.py

1. Review the content of the **Inventory** test case, note that we use the data structure learned from pyATS shell in the previous step, to extract a serial number from the output of **show** **inventory**:

@aetest.test

def inventory(self,device):

if device.os == 'iosxe':

out1 = device.parse('show inventory')

chassis\_sn = out1['main']['chassis']['CSR1000V']['sn']

**NOTE:** The path to fetch the serial number from the structures has been explored in the previous step with pyATS shell. Variables out2 and out3 are used:

elif device.os == 'nxos':

out2 = device.parse('show inventory')

chassis\_sn = out2['name']['Chassis']['serial\_number']

elif device.os == 'asa':

out3 = device.parse('show inventory')

chassis\_sn = out3['Chassis']['sn']

1. Exit Nano without saving, pressing:

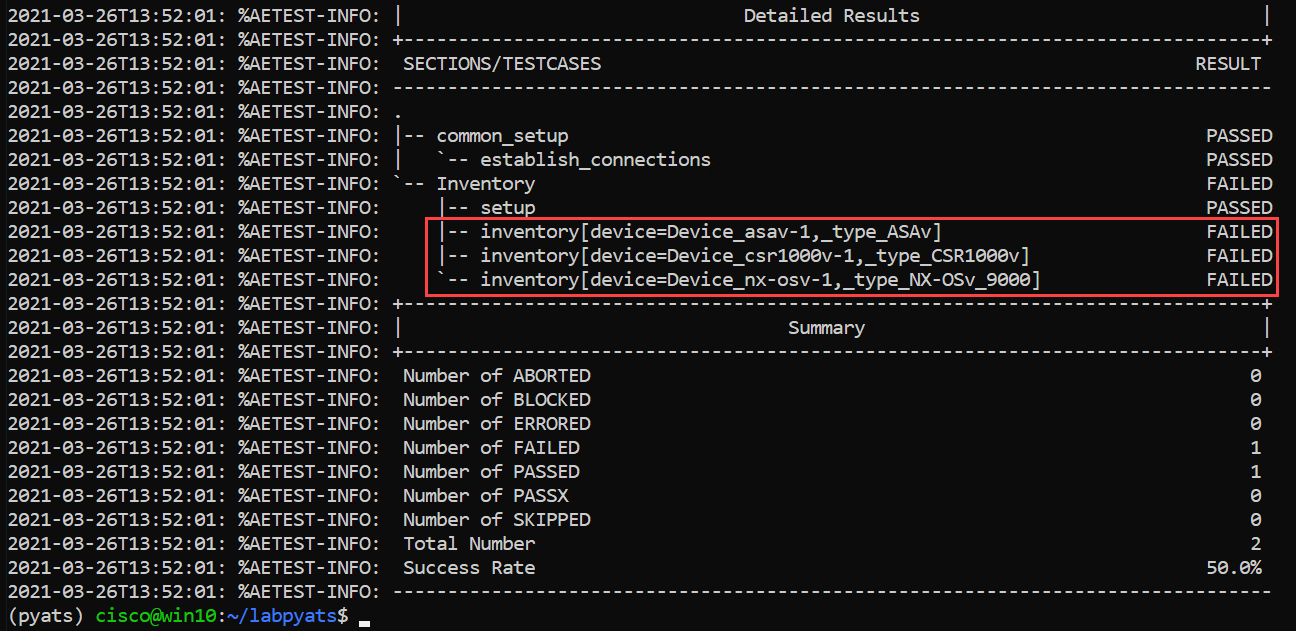
**Ctrl + X**

1. Execute the test script and check the **Detailed Results** section.

$ python task7\_labpyats.py --testbed pyats\_testbed.yaml

What are the results of these testcases? All fail? Do you have a clue as to why? Continue for the correct answer.

Figure 15: Failed Tests Output



All the tests have failed since we have serial numbers from a different network in our contract SNs list at the beginning of the **task7\_labpyats.py** file.

contract\_sn = ['9AQHSSAS8AU', '9Q3YV06WJ71', '9IFUH4GPSGL']

1. Open the file **task7\_labpyats.py** in Nano editor.

$ nano task7\_labpyats.py

1. Replace the serial numbers in the list contract\_sn with SNs from our testbed's equipment.
2. When you finish, save changes to file **task7\_labpyats.py**. By pressing:

**Ctrl + O**

File Name to Write: **task7\_labpyats.py**

Hit **[Enter]**

**NOTE:** Correct SNs from the testbed can obtained also from the previous script’s output:

2020-01-23T13:20:24: %AETEST-ERROR:Failed reason: 9AQHSSAS8AU is not covered by contract

<…>

2020-01-23T13:20:25: %AETEST-ERROR:Failed reason: 9Q3YV06WJ71 is not covered by contract

<…>

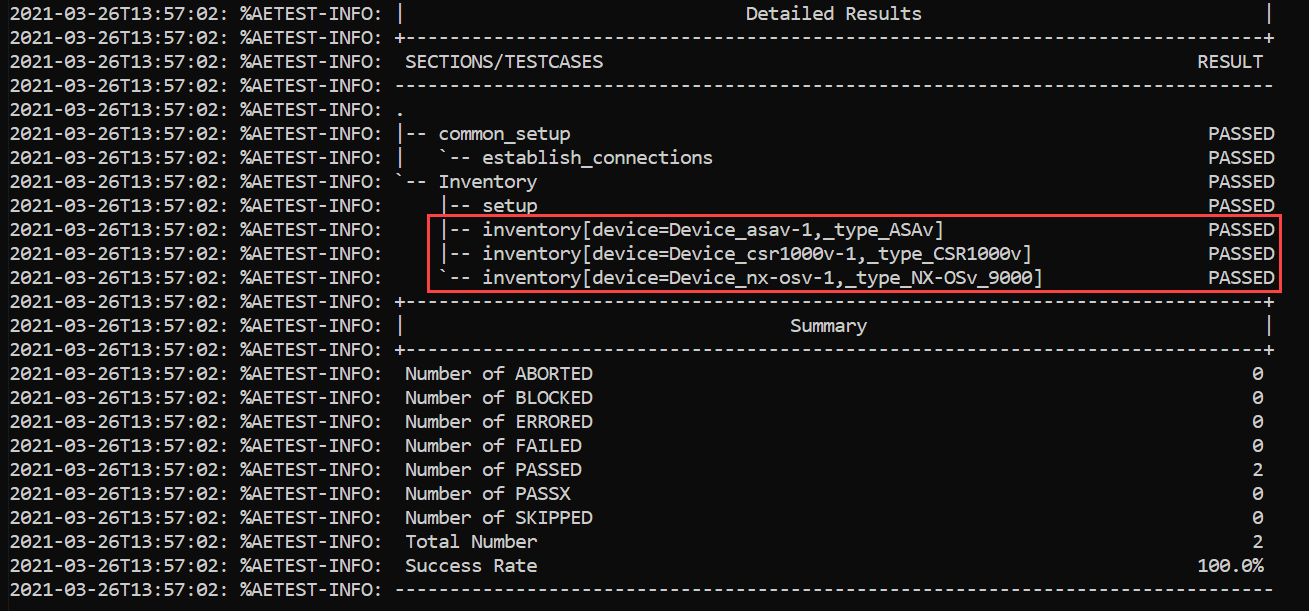
2020-01-23T13:20:26: %AETEST-ERROR:Failed reason: 9IFUH4GPSGL is not covered by contract

1. Еxecute the changed test script once again:

$ python task7\_labpyats.py --testbed pyats\_testbed.yaml

Now all the testcases should succeed:

Figure 16: Passed Tests Output



This concludes Scenario 7.

1. Verify the Routing Information using Parsers and pyATS Learn

Value Proposition: In this test case, we have the list of critical routes (usually this is a device’s loopback interface) and we must check that these loopbacks are installed in the routing information base (RIB) of all the devices in the testbed.

The high-level logic of the test will be as follows:

* Connect to each device in the testbed.
* Learn routing information from RIB of the devices.
* Verify whether all the critical routes are presented in the device’s RIB.

Steps

1. Let’s connect to the pyATS shell and check our idea.

$ pyats shell --testbed-file pyats\_testbed.yaml

1. Input the following code into pyATS shell:

In [1]: csr = testbed.devices['csr1000v-1']

asa = testbed.devices['asav-1']

nx = testbed.devices['nx-osv-1']

csr.connect()

asa.connect()

nx.connect()

pyATS uses the **learn** method to collect the set of show commands output for a feature configured on the device, to get its snapshot and store it into a structured format (Python dictionary).

In [1]: csr\_routes = csr.learn('routing')

nx\_routes = nx.learn('routing')

Now we can observe the structure of the parsed outputs. We are starting with the parsed output for csr1000v-1.

1. Input the following code into pyATS shell:

In [1]: import pprint

pprint.pprint(csr\_routes.info)

1. Observe the output in pyATS shell:

Out [1]:

{'vrf': {'default': {'address\_family': {'ipv4': {'routes':

{'10.0.0.12/30': {'active': True,

'next\_hop': {'outgoing\_interface': {'GigabitEthernet2': {'outgoing\_interface': 'GigabitEthernet2'}}},

'route': '10.0.0.12/30',

'source\_protocol': 'connected',

'source\_protocol\_codes': 'C'},

'10.0.0.13/32': {'active': True,

'next\_hop': {'outgoing\_interface': {'GigabitEthernet2': {'outgoing\_interface': 'GigabitEthernet2'}}},

'route': '10.0.0.13/32',

'source\_protocol': 'local',

'source\_protocol\_codes': 'L'},

<…>

Now we understand that RIB routes for csr1000v-1 are stored under the following path:

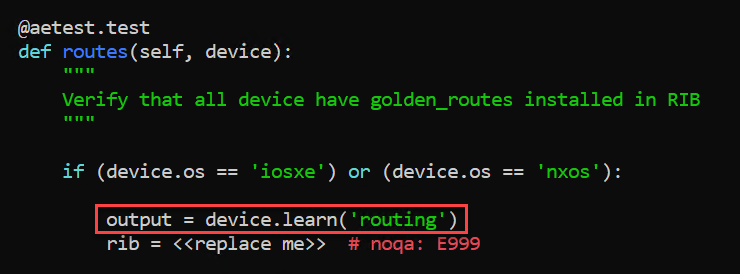
In [1]: pprint.pprint(csr\_routes.info['vrf']['default']['address\_family']['ipv4']['routes'])

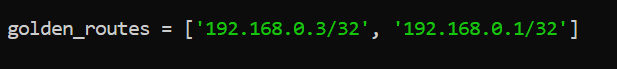
For nx-osv-1, RIB routes are stored under the same path as for csr1000v-1:

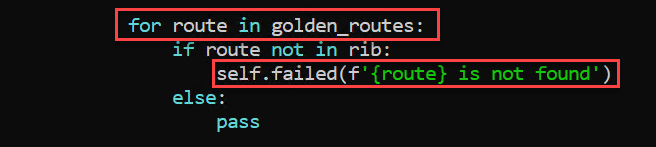
In [1]: pprint.pprint (nx\_routes.info['vrf']['default']['address\_family']['ipv4']['routes'])

1. Exit the pyATS shell using the **exit** command.
2. Open the file **task8\_labpyats.py** in Nano editor.

$ nano task8\_labpyats.py

1. Review the content of the **routes** testcase, note that we use the path to routes in RIB from the previous step to get the routing information. First, we’ll get a snapshot of the **routing** feature.
2. Then we compare the loopback routes stored in **golden\_routes list**, with the content of **rib**. If the loopback route is not found, then we force the test case to fail.





Golden routes are /32 networks of loopback interfaces on csr1000v-1 and nx-osv-1.

Loopback0 on csr1000v-1:

csr1000v-1#sh ip int br

Interface IP-Address OK? Method Status Protocol

GigabitEthernet1 198.18.1.201 YES TFTP up up

GigabitEthernet2 10.0.0.13 YES TFTP up up

GigabitEthernet3 10.0.0.17 YES TFTP up up

Loopback0 192.168.0.3 YES TFTP up up

Loopback0 on NX-OS:

nx-osv-1# sh ip interface brief vrf all

IP Interface Status for VRF "default"(1)

Interface IP Address Interface Status

Lo0 192.168.0.1 protocol-up/link-up/admin-up

Eth1/1 10.0.0.14 protocol-up/link-up/admin-up

Eth1/2 10.0.0.18 protocol-up/link-up/admin-up

Eth1/3 10.0.0.6 protocol-up/link-up/admin-up

IP Interface Status for VRF "management"(2)

Interface IP Address Interface Status

mgmt0 198.18.1.203 protocol-up/link-up/admin-up

IP Interface Status for VRF "inside"(3)

Interface IP Address Interface Status

Lo100 192.168.100.1 protocol-up/link-up/admin-up

Eth1/4 10.0.0.10 protocol-up/link-up/admin-up

1. Complete this test case by replacing the <<replace me>> statement with a rib variable. To accomplish this, you must paste the path to the rib routes, which was explored in the previous step:

output.info['vrf']['default']['address\_family']['ipv4']['routes']

Before insertion:

After insertion:

1. When you finish, save changes to file **task8\_labpyats.py**. By pressing:

**Ctrl + O**

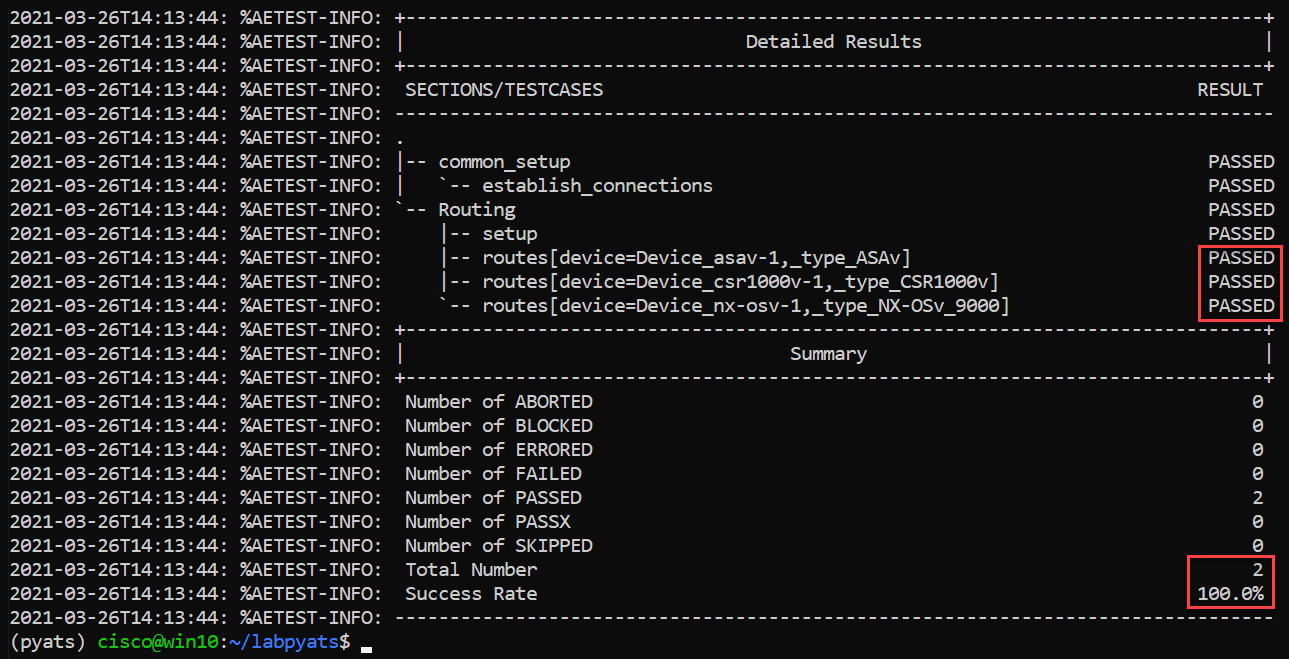
File Name to Write: **task8\_labpyats.py**

Hit **[Enter]**

1. Execute the test script and check the results section:

$ python task8\_labpyats.py --testbed pyats\_testbed.yaml

Figure 17: Passed Tests Output



This concludes Scenario 8.

1. Run PING to Verify Reachability

Value Proposition: In this testcase we must test reachability between devices (**nx-osv-1** and **csr1000v-1**), using the ping command.

The high-level logic of the test will be as follows:

* Connect to each device in the testbed.
* Find links between nx-osv-1 and csr1000v-1.
* Collect IP addresses from both ends of these links.
* Run the ping commands from nx-osv-1 for IP addresses, discovered in the previous step.

Steps

1. Let's connect to the pyATS shell and check our idea:

$ pyats shell --testbed-file pyats\_testbed.yaml

1. Input the following code into pyATS shell:

In [1]: csr = testbed.devices['csr1000v-1']

nx = testbed.devices['nx-osv-1']

1. pyATS has a **find\_links(device\_name)** method to find all the links between two devices in the topology. Let’s find the links between csr1000v-1 and nx-osv-1.

In [1]: nx.find\_links(csr)

1. Observe the output:

Out [1]: {<Link object 'csr1000v-1-to-nx-osv-1' at 0x7f445194b050>,

<Link object 'csr1000v-1-to-nx-osv-1#1' at 0x7f445194b150>,

<Link object 'flat' at 0x7f445194b410>}

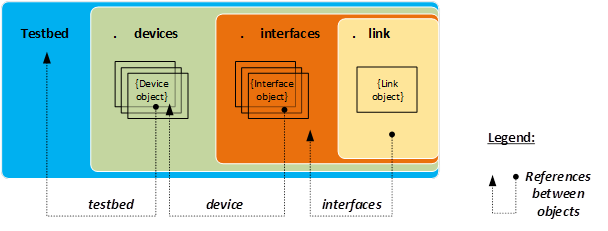
1. cc

Before studying the code and running the next script, let’s dive into the details of how information about a topology is stored in a testbed object (see the illustration that follows for a graphical representation of the explanation).

Things to know about the structure of the testbed object (created from the testbed YAML file specified: **testbed.yaml**):

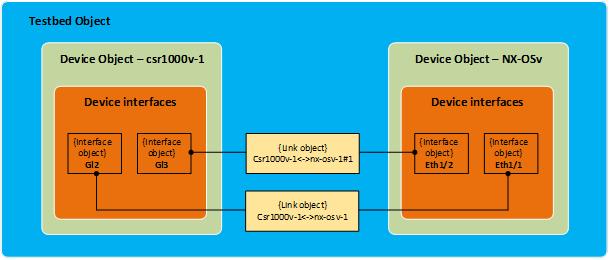
* The pyATS **Testbed** object contains the Python dictionary **devices**.
* Elements of the **devices** dictionary are the **Device** objects.
* Each object in the **devices** dictionary stores dictionary **interfaces** (contains **interface** objects).
* Each **interface** object stores the **link** object.

Figure 18: Testbed Object Structure



* The **Testbed** object is the top container object, containing all the testbed devices and all the subsequent information that is generic to the testbed.
* **Device** objects represent physical and/or virtual hardware in a testbed topology.
* **Interface** objects represent a physical/virtual interface that connects to a link of some sort (for example, Ethernet, ATM, Loopback, and so on).
* **Link** objects represent the connection (wire) between two or more interfaces within a testbed topology.

Figure 19: Links Representation



Let’s check the structure depicted above using our topology. We will find all the links connected between nx-osv-1 and csr1000v-1.

**IMPORTANT**: We can get the value of an attribute for each object. For example, we can get a link to which an interface object belongs by calling a **link** attribute. We can also reference interfaces that belong to this link, by calling the **interfaces** attribute in step 6 (see code below).

$ pyats shell --testbed-file pyats\_testbed.yaml

1. Paste the following snippet to pyATS console:
   1. Place the following iPython command in the beginning of code:

%cpaste

* 1. Copy and paste the code into the pyATS console:

csr = testbed.devices['csr1000v-1']

nx = testbed.devices['nx-osv-1']

links = nx.find\_links(csr)

for link in links:

print(f'#{link}')

for link\_iface in link.interfaces:

print(f'##{link\_iface}')

print(f'###link\_iface.ipv4 = {link\_iface.ipv4}, {type(link\_iface.ipv4)}')

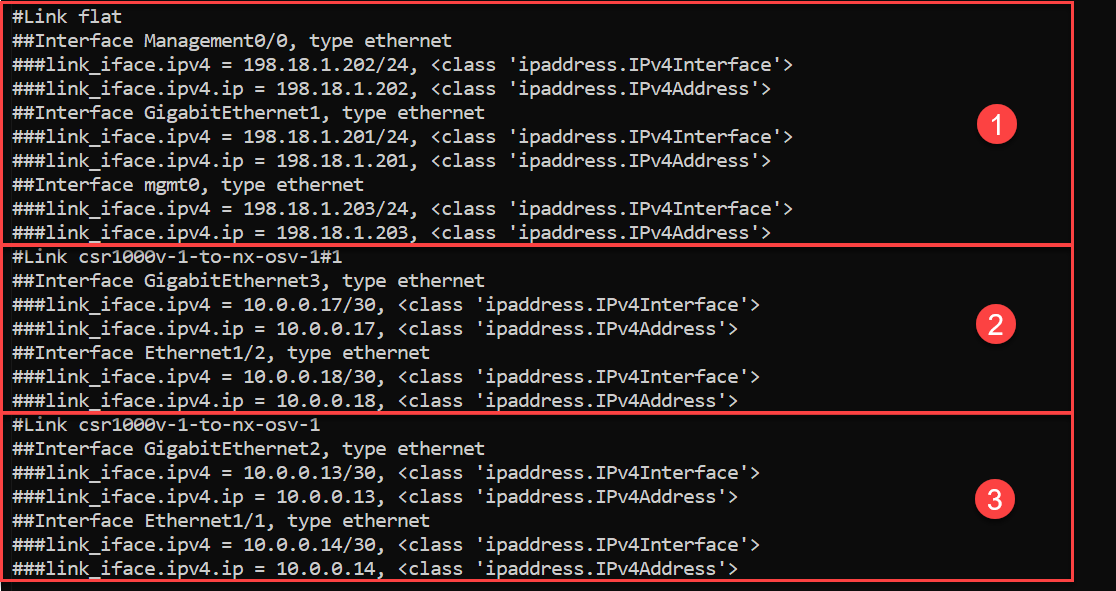
print(f'###link\_iface.ipv4.ip = {link\_iface.ipv4.ip}, {type(link\_iface.ipv4.ip)}')

* 1. End the code with **--**

Refer to the command output:

* **#Link csr1000v-1-to-nx-osv-1** – represents interfaces of all devices connected to the first link between **csr1000v-1** and **nx-osv-1**.
* **#Link flat** – represents interfaces of all devices (**asav-1**, **csr1000v-1**, **nx-osv-1**) connected to a management network.
* **#Link csr1000v-1-to-nx-osv-1**#1 - represents interfaces of all devices connected to the second link between **csr1000v-1** and **nx-osv-1**.

Figure 20: Scenario 9 Script Output



* 1. Exit the pyATS shell using the **exit** command.

1. Open the file **task9\_labpyats.py** in Nano editor:

$ nano task9\_labpyats.py

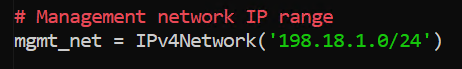
1. Review the content of the **PingTestcase** test case, look at the def setup(self) function. Code in this function follows the logic used in the previous step:

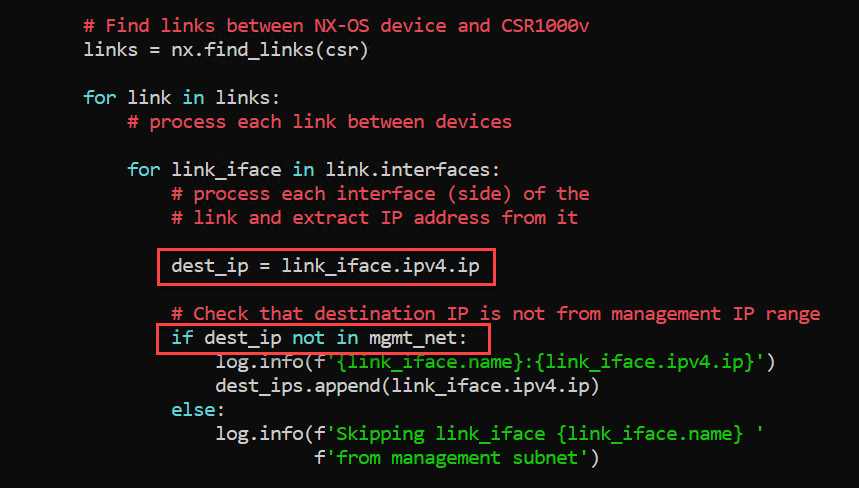
* Get all the links between nx-osv-1 and csr1000v-1 (nx.find\_links(csr).
* Get interfaces for each link (**for iface in links.interfaces**) and append its IPv4 address (**iface.ipv4.ip**) into list **dest\_ips**, to use them further in **ping** commands.

To exclude management IP addressing space, there is a check whether an IP address on a link is from a management address space (if dest\_ip not in mgmt\_net). If an IP address is from a management IP address, it’s not appended to the list dest\_ips.

**NOTE:** Note that the IP address in the **link\_iface.ipv4.ip** object is of the IPv4Address type so we can check whether it overlaps with IPv4Network without any conversion of type (hence **if dest\_ip not in** **mgmt\_net** is used).

The code of the **setup(self)** function is shown below:





A **ping** command for each IPv4 address of both ends of the links between nx-osv-1 and csr1000v-1 is executed in the function **def ping(self, dest\_ip)**.

**NOTE:** Note the following:

1. In this task we are not passing **Device** objects into **@aetest.test** from **@aetest.setup** using **aetest.loop.mark** as it has been done in previous tasks:

aetest.loop.mark(self.error\_logs, device=devices)

2. In this task we are passing **dest\_ip** one-by-one from the **dest\_ips** list:

aetest.loop.mark(self.ping, dest\_ip=dest\_ips)

3. To get a Device object we call the **self.parent.parameters** attribute:

nx = self.parent.parameters['testbed'].devices['nx-osv-1']

The string returned by a ping operation is shown below. The field that must be extracted is marked yellow:

5 packets transmitted, 5 packets received, 0.00% packet loss

To check this field, we use a regular expression, it extracts packet loss from the ping command’s output. If the loss rate is less than 20% (to accommodate the potential first ping drop due to ARP resolution) then the test case should pass successfully:

nx = self.parent.parameters['testbed'].devices['nx-osv-1']

try:

result = nx.ping(dest\_ip)

<…>

else:

m = re.search(r"(?P<rate>\d+)\.\d+% packet loss", result)

loss\_rate = m.group('rate')

if int(loss\_rate) < 20:

self.passed(f'Ping loss rate {loss\_rate}%')

else:

self.failed('Ping loss rate {loss\_rate}%')

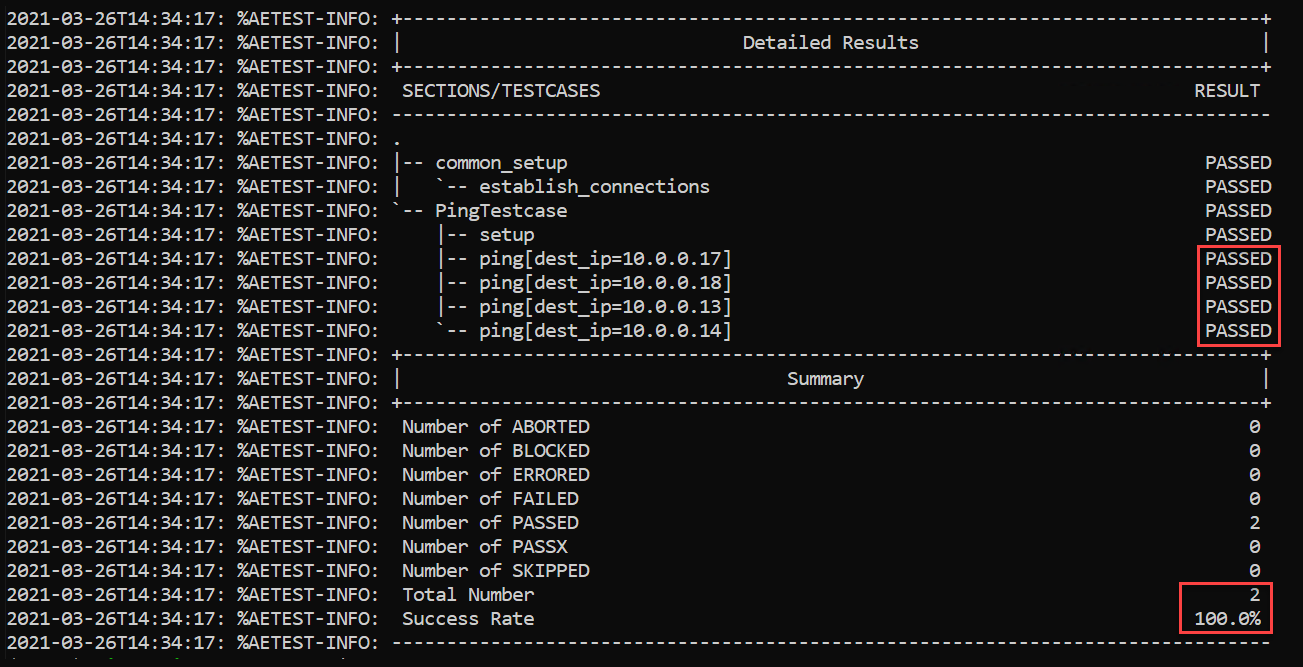
1. Exit Nano without saving, pressing:

**Ctrl + X**

1. Execute the created test script and check the **results** section; all pings should succeed:

$ python task9\_labpyats.py --testbed pyats\_testbed.yaml

Figure 21: Passed Tests Output



This concludes Scenario 9.

1. Show the Results of Tests in a Browser

Value Proposition: In this last task, we will see how to show the results of the tests in a more user-friendly way in a browser. For this, the pyats run job command will be used in a Bash shell.

When a test is run using pyats run job it adds the following advantages:

* Logs of test runs are saved into the archive
* Graphical representation of test results in a browser
* Ability to run tests in different Python scripts

To use **pyats run job**, a special file “job file” (written in Python) should be created.

A job file looks as shown below:

**<test\_name1>** - specifies the path in the system to the Python file with the first list of tests (for example **task8\_labpyats.py**).

**<test\_name2>** - specifies the path to the Python file with the second list of tests (for example **task9\_labpyats.py**).

The method **run** from the imported library **ats.easypy** instructs the system to run tests in sequence.

import os

from ats.easypy import run

def main():

# Find the location of the script in relation to the job file

<test\_name1> = os.path.join('<file\_with\_tests1.py>')

<test\_name2> = os.path.join('<file\_with\_tests2.py>')

# Execute the testscript

run(testscript=<test\_name1>)

run(testscript=<test\_name2>)

To call **pyats run job**, use the following command in a Bash shell:

$ pyats run job <job-file> --testbed <testbed-file>

Schematically, the process of **pyats run job** can be shown as follows:

Figure 22: PyATS Run Job Process



Steps

Let’s use **pyats job run** to execute tests from Task 9. PyATS job file **task10\_runtestsjob.py** has been pre-configured for this.

1. Open the **runtestsjob.py** file in Nano and check it (the structure is per the one shown above).

$ nano task10\_runtestsjob.py

1. Exit Nano without saving, pressing:

**Ctrl + X**

1. Execute the pyATS job file with the **pyats run job** command:

$ pyats run job task10\_runtestsjob.py --testbed pyats\_testbed.yaml

1. After completion of the job, check the results:

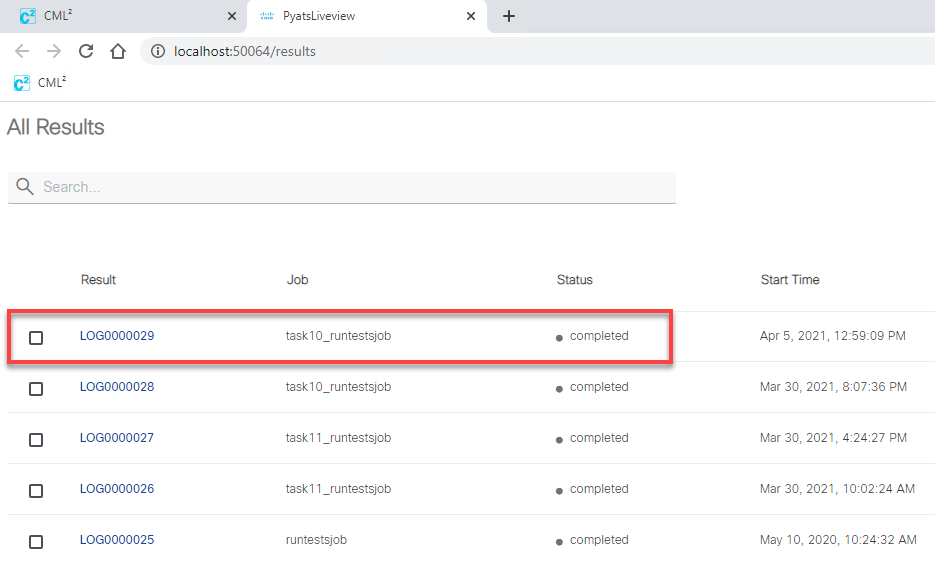
$ pyats logs view

1. Google Chrome would be opened to show the last jobs run. Minimize Linux shell window.

Don’t close the Linux shell, otherwise, it will stop the local pyATS web server.

Click the upper line in a list to open results of the last job run:

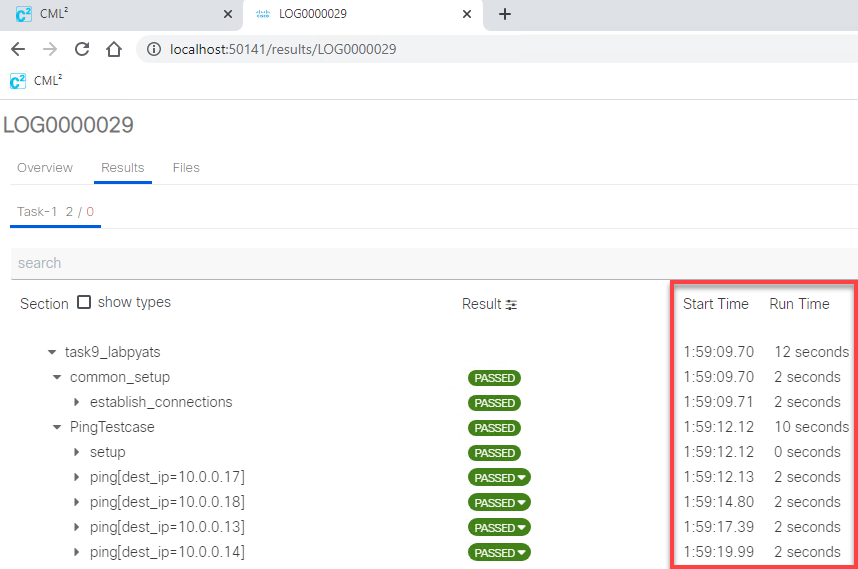
Figure 23: PyATS Log Viewer List of jobs run



1. Detailed results of the tests comprising the last run job would be shown (see Figure 20).

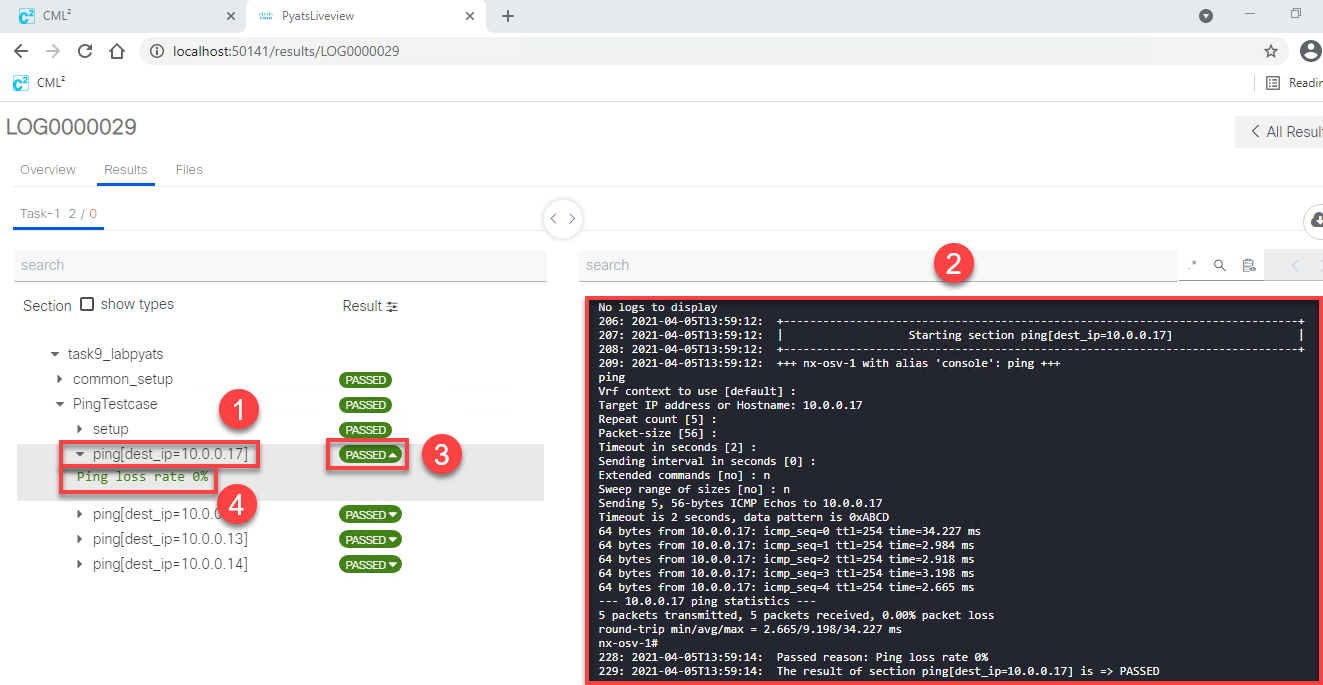
**NOTE:** Pay special attention to the result of each test, which is shown along with the start time and run time of each test.

Figure 24: PyATS Log Viewer Results Page



1. Click on the test **ping[dest\_ip=10.0.0.17]** (see “1” on Figure 21), a detailed log from the execution of this test will be shown on the right side of the window (see “2”).
2. Click on the PASSED button for the test **ping[dest\_ip=10.0.0.17]** (see “3” on Figure 21), ensure the test passed message is shown (see “4”).

Figure 25: PyATS Log Viewer Ping Test Results



You can open detailed results of the last job, without opening the list of previous jobs, using the following command in the shell:

$ **pyats logs view --latest**

To test this option, follow the next steps.

1. Maximize Linux shell, minimized in step 5.

Stop running pyATS web server:

**Ctrl + C**

Open the web page with the detailed results of the last job:

$ **pyats logs view --latest**

Ensure detailed results of the tests comprising the last run job would be shown right away.

1. Open Linux shell again, stop running pyATS web server:

**Ctrl + C**

What you should learn from this Scenario:

**pyATS run** is a very handy tool and it’s recommended that you use it to run your pyATS tests.

You might also check the official documentation for the details at this site:

<https://pubhub.devnetcloud.com/media/pyats/docs/cli/pyats_run.html#pyats-run-job>



Josh is happy now since he knows how to demonstrate test results.

This concludes Scenario 10.

1. Run tests and compare results from XPRESSO dashboard

Value Proposition: In this last task, we will learn how to use XPRESSO dashboard to run test job and compare test results. To simplify the scenario basic configuration was already done in XPRESSO. Test Harness, Execution Engine, Testbed and Job are pre-configured.

Due to security restrictions in dCloud on Jumphost, access to XPRESSO dashboard is provided via RDP session to CentOS VM running XPRESSO.

Steps

1. Locate XPRESSO.rdp shortcut on the desktop of Workstation, double-click to start Remote Desktop Protocol (RDP) session to XPRESSO VM. Login with the following credentials (see Figure 22):

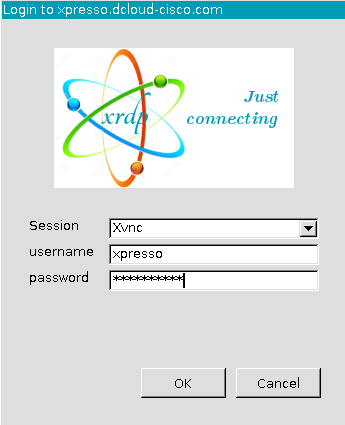
Username: **xpresso**

Password: **C1sco12345**



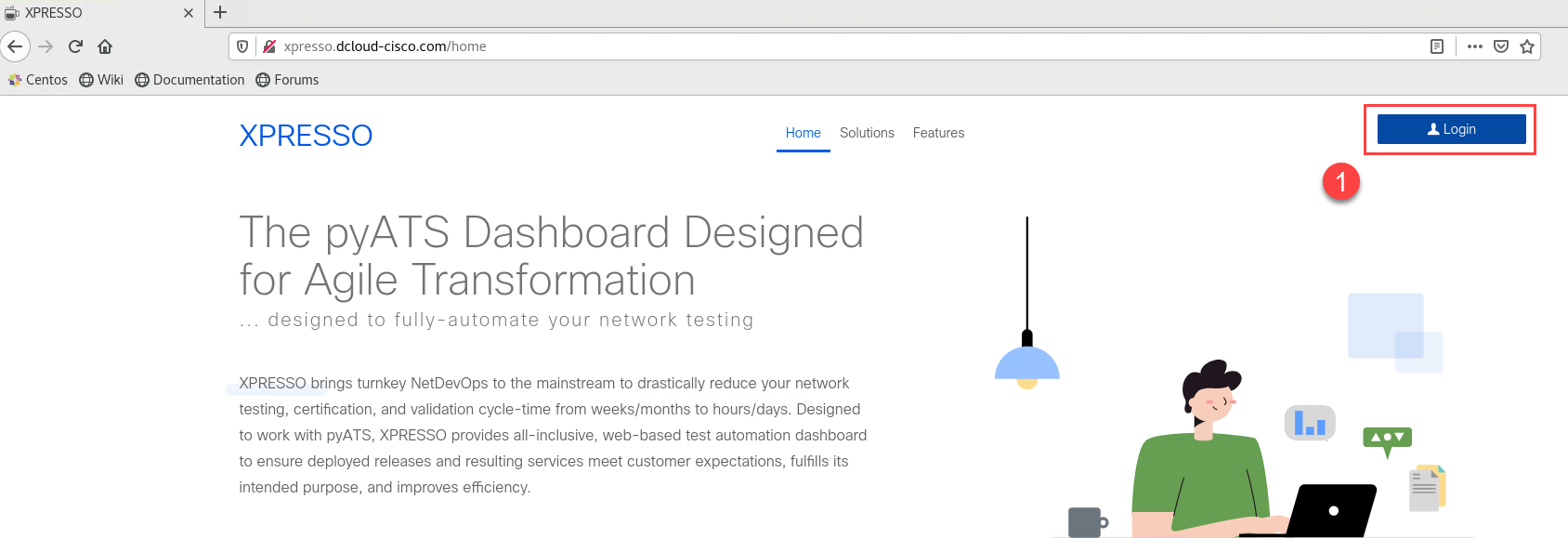
If you are using AnyConnect VPN and have Microsoft RDP client installed, you can connect directly from your PC via RDP to address XPRESSO VM (use IP address: **198.18.134.50)**.

Figure 26: Login to XPRESSO VM via RDP



1. Inside the RDP session open Firefox from the desktop or Application menu on top of the screen. Open XPRESSO dashboard using <http://xpresso.dcloud-cisco.com>

Figure 27: Expresso Login Page



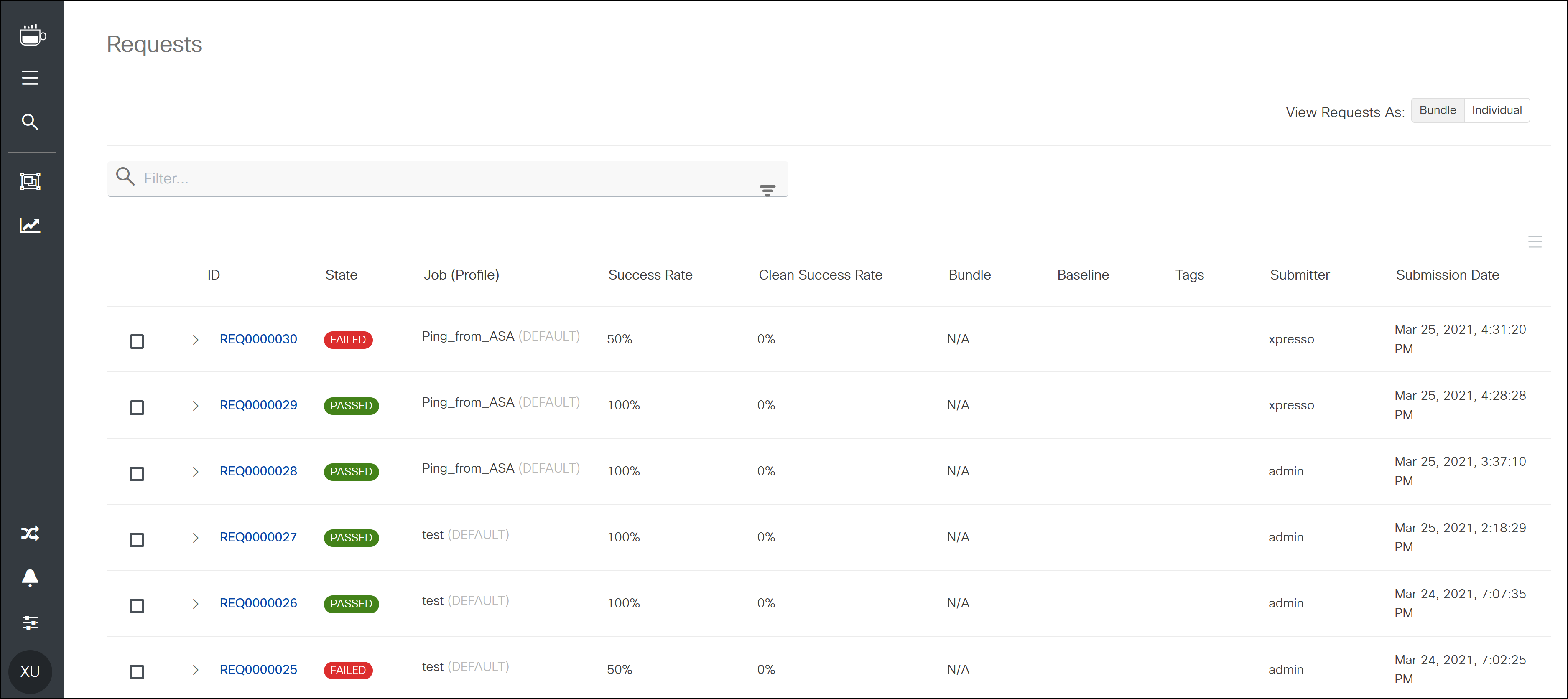
1. Login with credentials:

Username (if prompted): **xpresso**

Password: **C1sco12345**

You should be automatically logged into XPRESSO dashboard and see the Requests page:

Figure 28: XPRESSO DASHBOARD LOGIN PAGE



In rare cases for some dCloud PODs authentication to XPRESSO dashboard might fail.

If you've tried 3 times (to eliminate the issue is due to wrong credentials), try to restart Docker client container on XPRESSO VM.

Follow the procedure (only if authentication to XPRESSO dashboard fails):

1. Login to XPRESSO VM via SSH (from Jumphost[[3]](#footnote-4) or your PC).

IP address: 198.18.134.50

username: root

password: C1sco12345

1. After successful authentication, run the following commands in the Linux shell:

# cd /workspace/xpresso

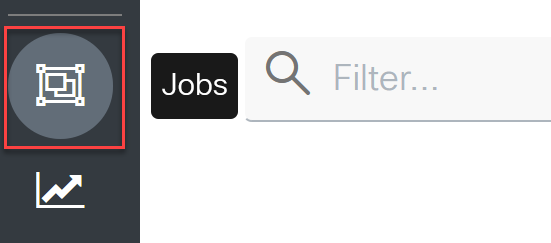
# docker-compose restart client

After successful execution of the last command, you should see the output:

“Restarting xpresso\_client\_1 ... done”

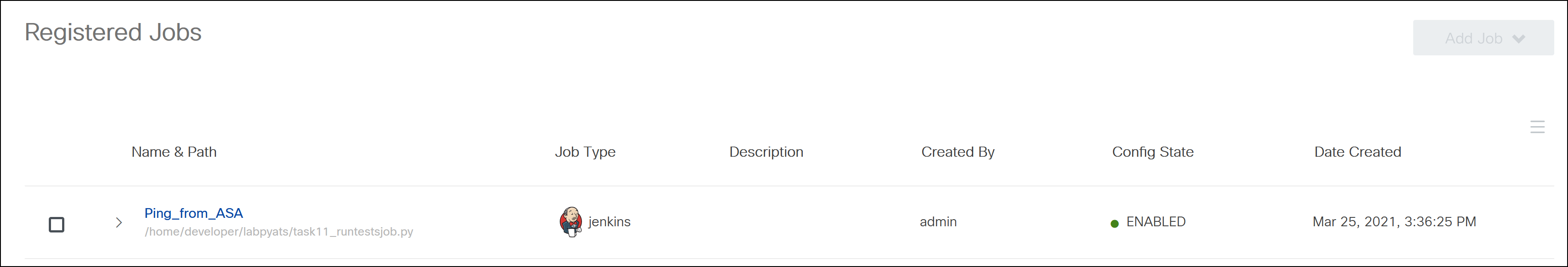


1. After 40 seconds (to ensure Docker client container has come up) you should be able to login into XPRESSO dashboard successfully.
2. From the menu icons on the left locate the Jobs item and click on it:

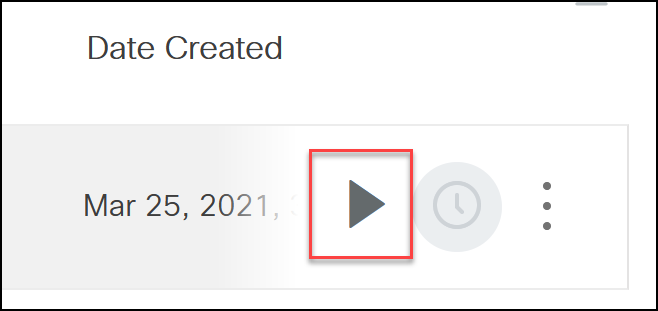


You will see the pre-configured job **Ping\_from\_ASA** which executes the task10\_runtestsjob.py script you’ve used in Scenario 10:

Figure 29: XPRESSO Job list



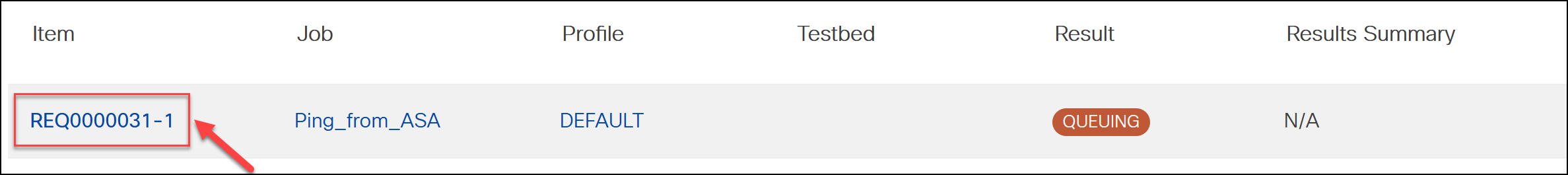
1. Hover mouse over the job row and you will see Execute icon on the right, click it:



You will be presented with a *“You are configuring a new group job request”* page where you can customize job run settings. Leave all settings by default and click **Submit** button. Once done, the job will be submitted for execution.

On the bottom of the job execution page, you will see request item which will go through the different stages: PREPARING, QUEUING, QUEUED, RUNNING, PASSED, ERRORED or FAILED:

Figure 30: Request execution status



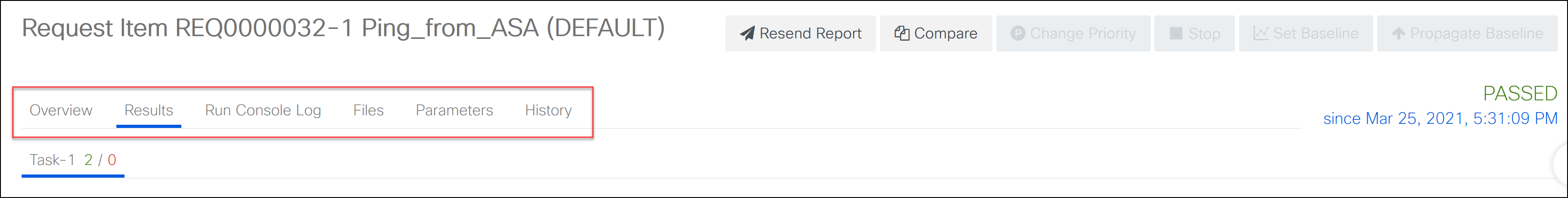
1. Click on the Request Item while the job is running, you will see how pyATS is executing every test defined in the job file one by one in real-time:



**NOTE:** If you click on the Request item while the job is going through PREPARING, QUEUING, QUEUED stages there would be no results visible as the job is not running yet. Once the job transitions to the RUNNING stage, the page will be updated and you start getting test execution results.

1. Once job execution is completed, you will see the results, can check raw console output, job history with timestamps, download archive with results or compare test execution with another job run:

Figure 31 Request details



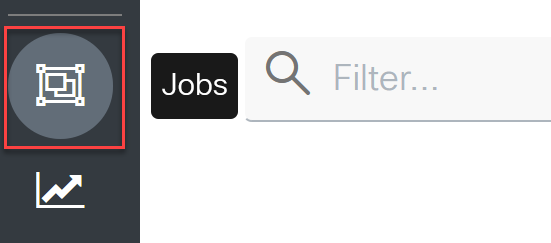
1. Let’s introduce a network failure by connecting to csr1000v-1 and shutting down interface GigabitEthernet2. From Admin Workstation launch Putty, login to csr1000v-1 and execute commands:

configure terminal

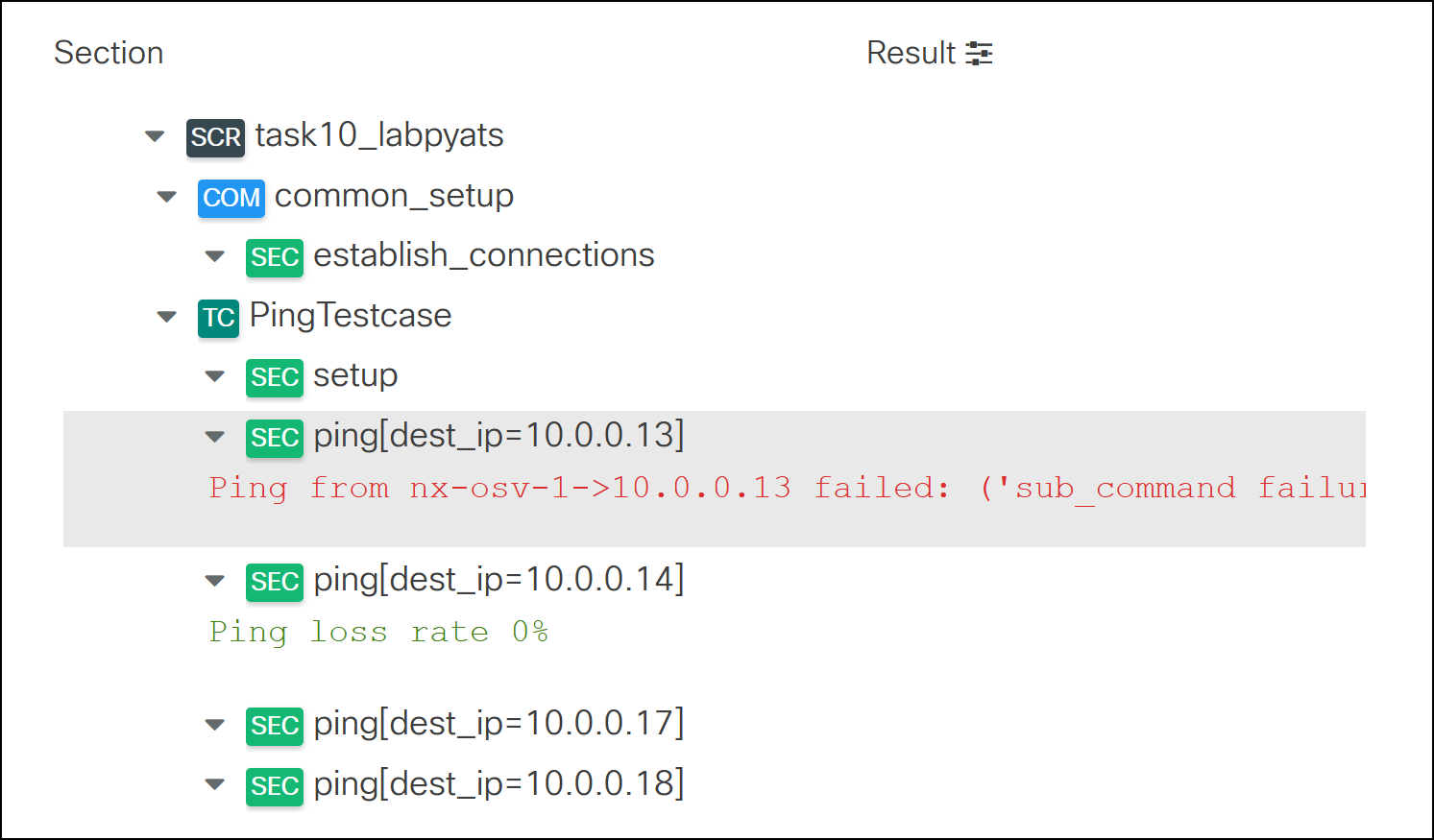
interface gigabitEthernet 2

shutdown

1. Go back to XPRESSO dashboard and click on the Jobs menu item:

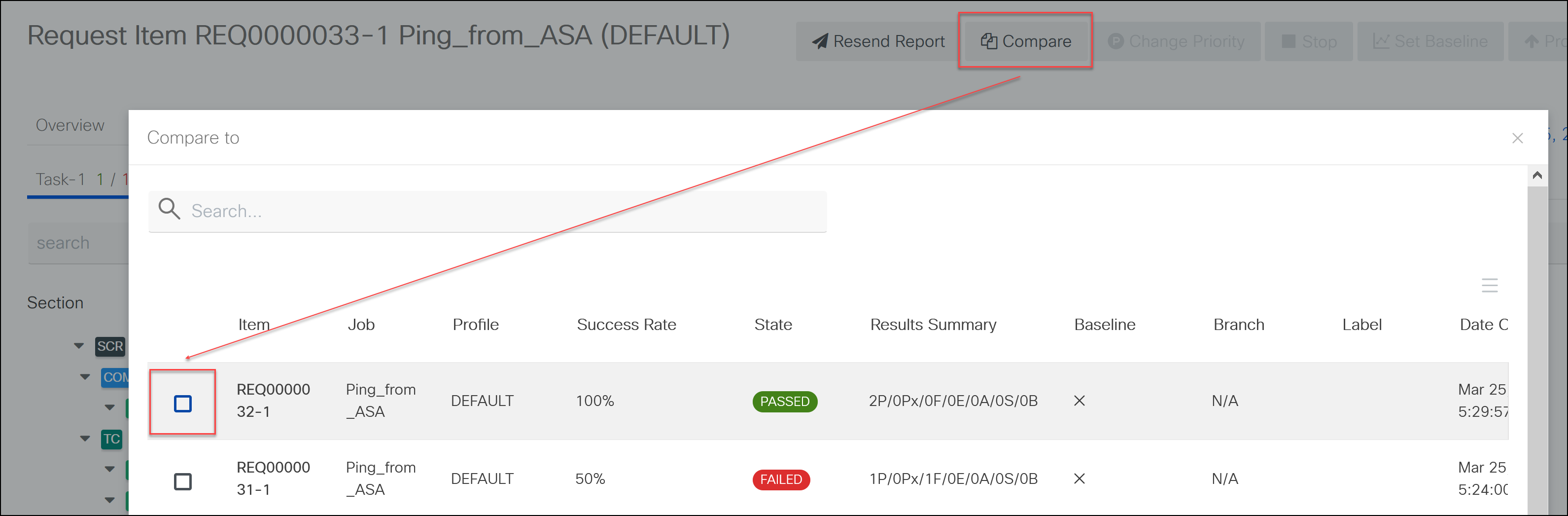


1. Run Ping\_from\_ASA job again by repeating steps 4 – 7 above. This time you will notice that one of the tests is failing:



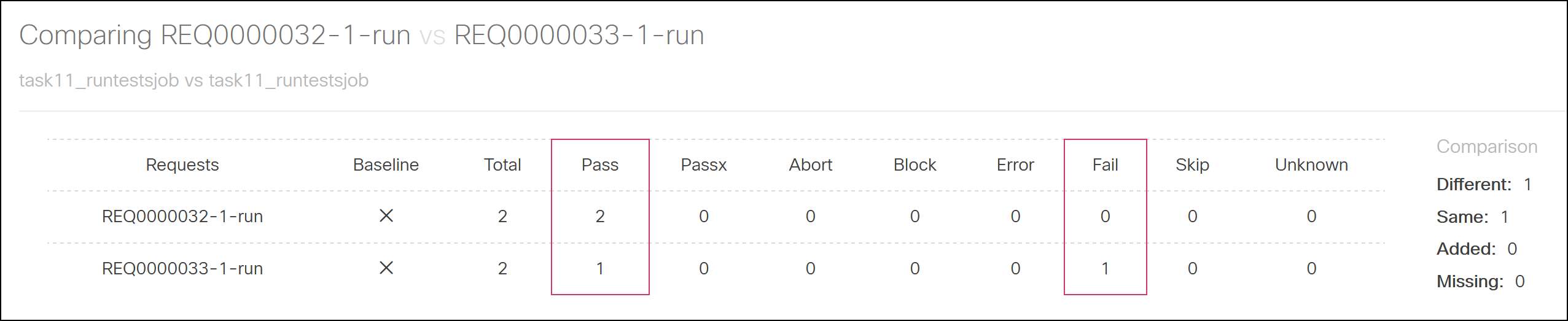
1. Now let’s compare job results. On the top of the page click on Compare button and check the last job run that was successful with status PASSED:

Figure 32: Compare request results

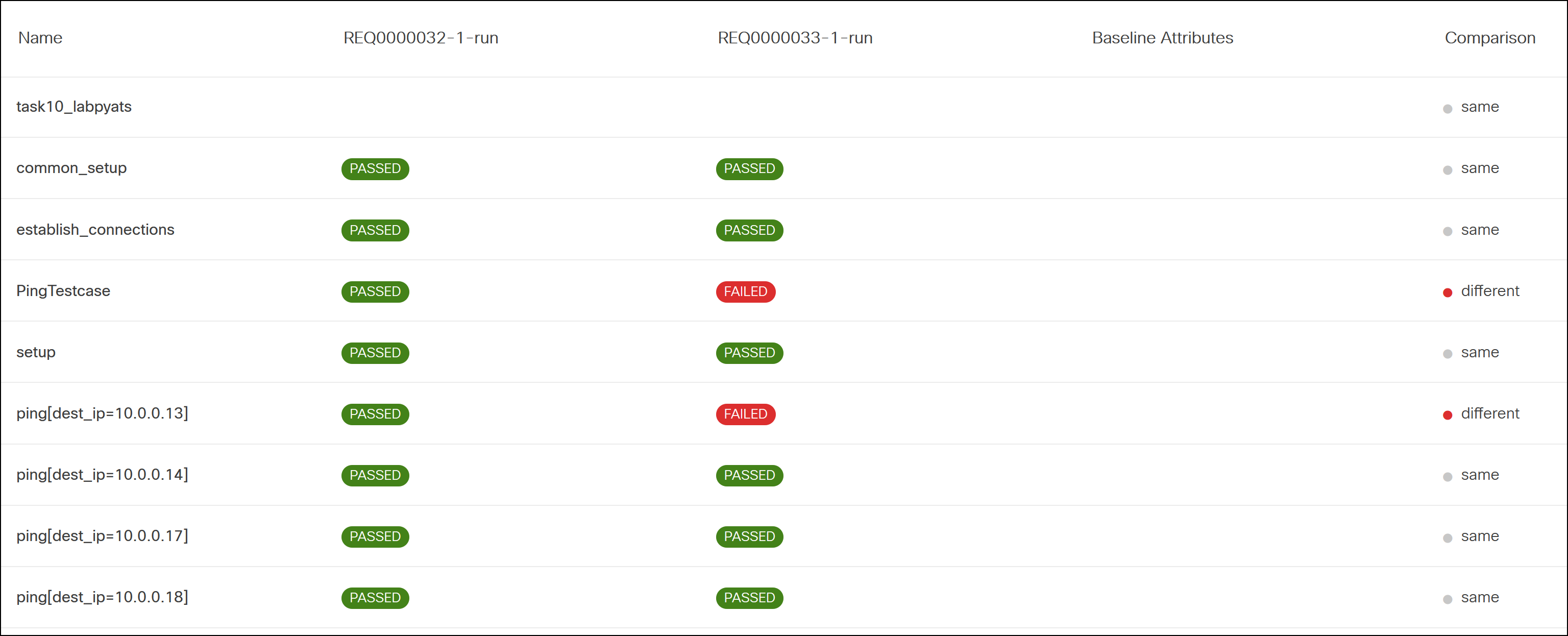


1. You will see the summary of comparison for both job runs and some tests passed and failed:

Figure 33 Result comparison page



Followed by detailed test to test comparison:



1. Hover mouse over the failing test line ping[dest\_ip=10.0.0.13] and click the Testcase Diff icon on the right to see the test result in diff format:

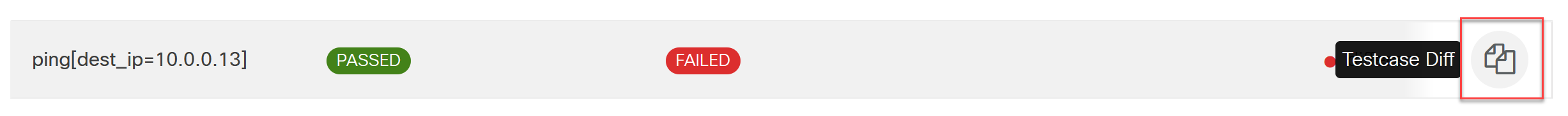
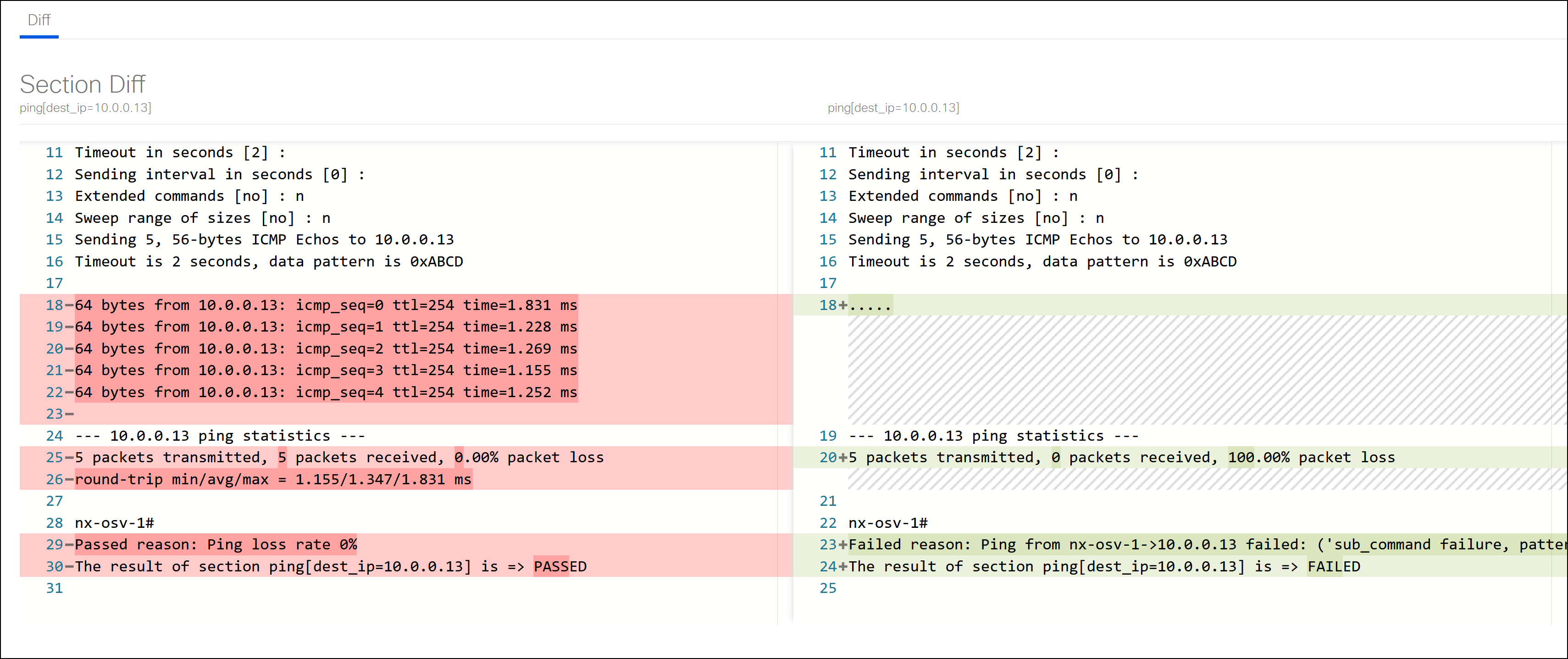
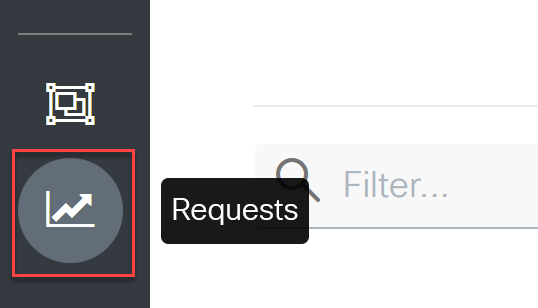
Section diff page will open and load diff plugin:

Figure 34 Results section diff

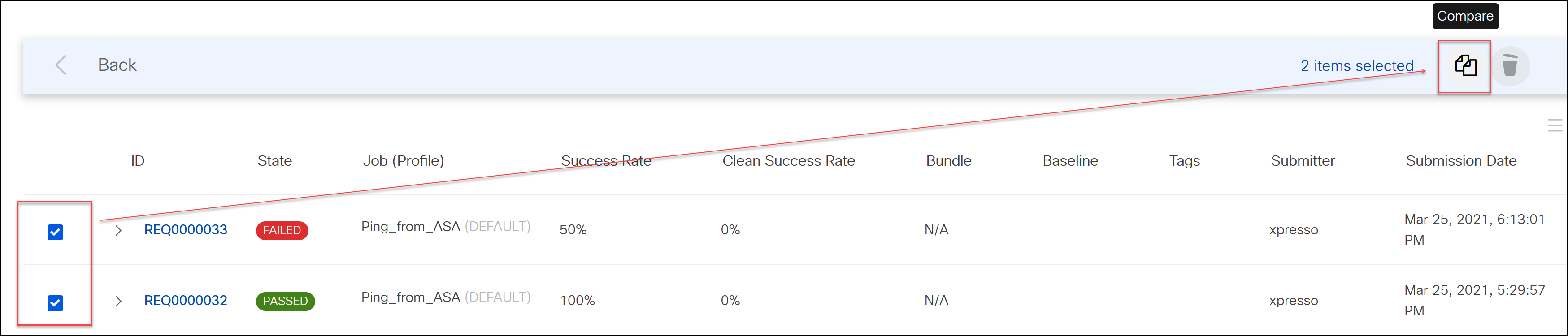


**NOTE:** Alternatively, you can compare test results by going to the Requests page and selecting 2 requests for comparison as described below (see steps 14 – 16 below).

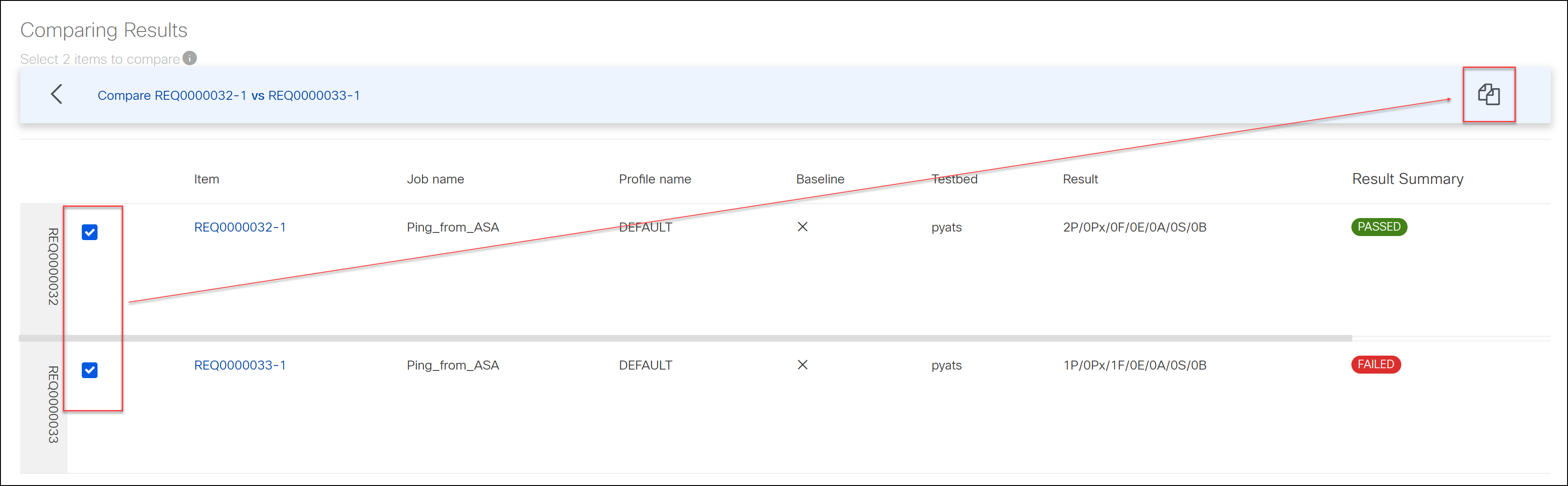
1. Click on the **Requests** menu item:



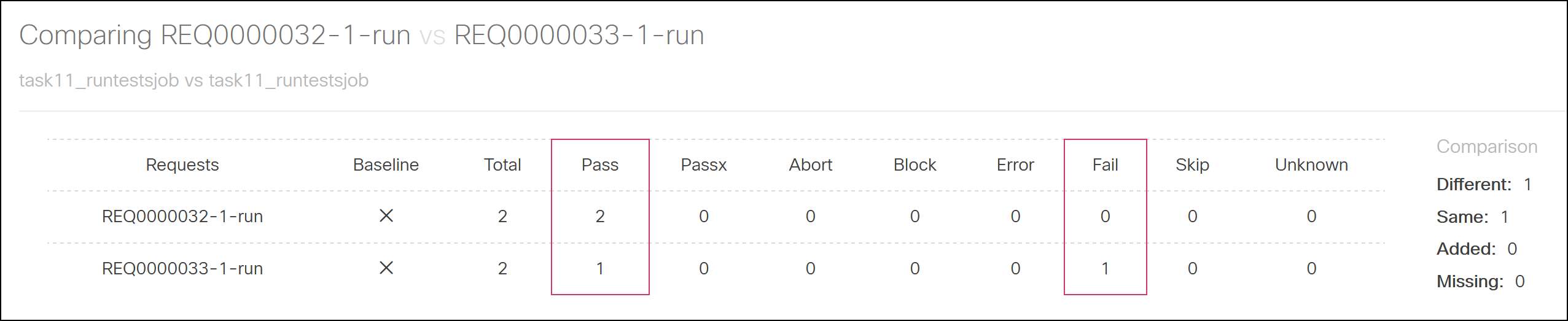
1. Select 2 requests - PASSED and FAILED and click **Compare** icon on the top right of the page. Compare icon will be visible only if you select exactly 2 items:



1. Select 2 results for comparison and click **Compare** icon. This additional step is required as Job can include several requests ran as Job Bundle:



1. You will be brought to the results comparison page:



**This concludes Scenario 11.**

# Conclusion

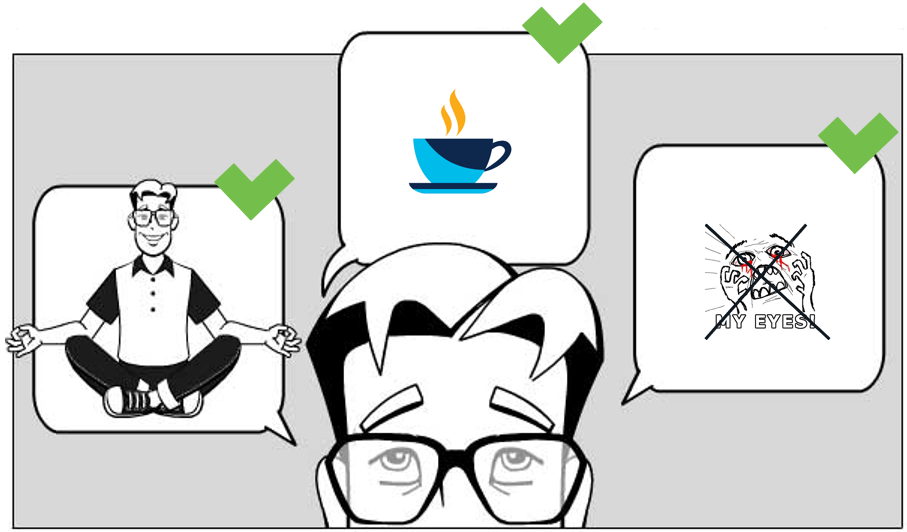
You and Josh have learned how to build automated tests using the pyATS framework.

Tools included in pyATS provide a wide variety of opportunities, and it’s not hard to start using them.

This lab has introduced you to real-world examples and, we hope, has given you a head start on the automation of tests in your network.

The main points we wanted to highlight in this lab include:

* Test automation for network operations is available today.
* It's easy to implement automation in your network, with little programming experience.
* pyATS is a rather simple and extensible framework for automation.
* You can use pyATS Dashboard (XPRESSO) to render results of tests.



Now Josh feels calm, much cheerful. Tomorrow he would have time for a cup of espresso after checking the results of tests after last night works in XPRESSO.

Hope you have enjoyed our lab and are happy with how you can leverage pyATS for automated testing for your network! Thank you for following this lab guide with Josh.

1. <https://github.com/CiscoDevNet/virlutils> [↑](#footnote-ref-2)
2. Serial number shown below is provided for example and would be different on equipment in a lab. [↑](#footnote-ref-3)
3. From Jumphost use Putty shortcut saved on a desktop [↑](#footnote-ref-4)