* Created by [AJAYKUMAR NATTUDURAI](https://confluence-eng-rtp2.cisco.com/conf/display/~anattudu), last modified on [Aug 10, 2024](https://confluence-eng-rtp2.cisco.com/conf/pages/diffpagesbyversion.action?pageId=586942954&selectedPageVersions=8&selectedPageVersions=9)

***What is FTL?***

*FTL - > Feature test Library, as the name itself suggests that FTL is nothing but a concept to test the Feature of the FMC using Libraries.*

*It is using pyATS as a test framework.*

***pyATS:****It is a test framework which uses a testbed to define devices and topologies targeted by the test cases.Test beds are often defined as a YAML file. pyATS currently supports Python 3.4+ on Linux and ac systems. Windows platform are not yet supported.*

*So, to make pyATS more flexible we will create a linux based container which will comes with python as default. So we will upload our code base to the container and make the ssh connection to container from Pycharm(IDE) and we will install the necessary packages and run our code.*

***Easypy:***

*Easypy provides a standardized runtime environment for testscript execution in pyATS. It offers a simple, straight-forward way for users to aggregate testscripts together into****jobs****, integrates various pyATS modules together into a collectively managed ecosystem, and archives all resulting information for post-mortem debugging. Main advantage of Easypy is store all the output log into single link called****TaskLog.****We will get an email notification at then end of each run. It is installed by default as a part of pyATS module.*

***Jobs:***

*In pyATS, the aggregation of multiple testscripts together and executed within the same runtime environment is called a job. It allows aggregation of multiple testscripts to run under the same environment as tasks, sharing testbeds, and archiving their logs and results together.*

**Tasks**

*An Easypy****task****is essentially a testscript being executed by a test-harness like aetest in a child process. They exhibit the following properties:*

* *each task is encapsulated in its own child process, forked from main easypy program.*
* *each task contains a single [TaskLog](http://wwwin-pyats.cisco.com/documentation/latest/log/concept.html" \l "tasklog) where all messages are logged to.*
* *all tasks report to their results via*[*Reporter*](http://wwwin-pyats.cisco.com/documentation/latest/reporter/index.html#reporter)*.*

***Flow:****FTL  → pyATS → Easypy → Jobfile & Task → Test\_execution → Email\_notification*

***Definition:****Feature-test-library is making use of pyATS to run the testcases. Here Easypy comes into picture and help pyATS to setup a environment to run the testscript execution. Now we know that we are having multiple scripts(include python and yaml). so inorder to integrate all scripts into a single job easypy will make use of jobfile and tasks.*

***Folder Structure:***

*Rule1: Folder name should have proper meaning.*

*Rule2: We need to maintain all our yaml files under the subfolder called****data***

*Rule3: Main script should not include repeated lines of codes*

*Rule4: Common functions should be placed in a utility folder so that any one can make use of it. we can have more than one python files if necessary.*

*Rule5: proper definition should be added in a command line for complex testcases.*

***MAIN\_FOLDER***

***⁄                     \***

***data****mainscript.py, utility\_file.py*

*⁄*

*yaml files*

***Main\_Script.py***

*This is the starting point of the program at top we will have all the import statement*

*then, it will have code to import yaml files*

*CommonSetup class*

*Testcase classes*

*CommonCleanup Class*

***Utility\_file.py***

It will contain all the common methods. So that all the team members can able to make use of it.

***YAML files***

Yaml files are like heart of the FTL. it includes all the configuration to create policy, license, interfaces, Zones etc..

Main\_script.py will get the configuration from Yaml and create the policy/zone/interface with the help of library functions.

All the yaml files will be stored under **data** folder.

***How  Configurations are getting created?***

***Backend-Flow:***

***Example  code:***

ac\_policy = api\_service\_fmc1.create(Ac\_Policy\_Config)  
log.info('\*\*\*\*\* Successfully created AC policy \*\*\*\*\*')  
ac\_policy\_assignment = PolicyAssignment()  
ac\_policy\_assignment.targets = [device]  
ac\_policy\_assignment.policy = ac\_policy  
api\_service\_fmc1.create(ac\_policy\_assignment)  
log.info('\*\*\*\*\* Successfully assigned AC policy to the FTD \*\*\*\*\*')

**APIService:**

*This service is designed to handle all the CRUD operations for models, It should be included in the setup class parameter in CommonSetup.*

*It will collect the configuration detail from data.yaml file and create the respective item in the FMC.*

**PolicyAssignment();**

*We are creating instance for the PolicyAssignment class, this class contains Adapter, serializer and deserializer.*

***Adapter:****It will give the URL/REST API*

***Serialier and Deserializer:****Basically, the Configuration we are collecting from the data yaml and sharing to the FMC model is object format, but sometimes the model in the FMC is created in such a way that it will accept the inputs in the format of class. Also while we are retrieving config from model class we need the config in the format of object.So, serializer and deserializer will take the responsibility of converting request and response  to the format which is required.*

**Note:**We can directly create the instance of the required model class and do the testcase or else we can create the instance for factory class which will link us to the respective model.

***Developer Flow Diagram:***

***Script Structure:***

*FTL is all about maintaining the proper Formatted structural Script structures*

*In high level we can divide the main script into three different parts:*

1. *CommonSetup*
2. *Testcase class*
3. *CommonCleanup*

***1.CommonSetup:***

*CommonSetup is where all the common configurations, prerequisites and initializations shared between the script’s testcases should be performed. It is inherited from****aetest.CommonSetup****.It will connect to all testbed devices & check whether they are ready.*

*Common setup will consists of n number of subsection. we will define the general initalization in the CommonSetup class. For Example: cloning files from Git, creating interfaces, etc..*

*CommonSetup class -  format:*

*from pyats import aetest*

*# define a common setup section by inheriting from aetest*

*class ScriptCommonSetup(aetest.CommonSetup):*

*@aetest.subsection*

*def check\_script\_arguments(self):*

*pass*

*@aetest.subsection*

*def connect\_to\_devices(self):*

*pass*

*@aetest.subsection*

*def configure\_interfaces(self):*

*pass*

***NOTE:****Do not forget to mention the****@aetest.subsection****above every method definition in CommonSetup class*

***2.Testcase Class:***

*After the Commonsetup class we need to define the test cases in the separate class. It is upto us to whether declare all the test cases in a single class or we are also allowed to create a separate class for each testcase.*  
  
*This testcase class is mostly consist of library class to create policy, sending traffic, verifying events get generated in FMC.*

*In test case class each method definition should have a decorater****@aetest.test****this decorater helps pytest to identify that it is a test case.*

*Testcase class - format:*

class SimpleTestcase(aetest.Testcase):

@aetest.test

def trivial\_test(self):

assert 1 + 1 == 2

***NOTE:****Do not forget to mention the****@aetest.test****above every method definition in methods of TestCase class*

**3.CommonCleanup:**

*CommonCleanup  is the last section to run within each testscript. Any configurations, initializations and environment changes that occurred during this script run should be cleaned up (removed) here. Eg, the testbed/environment should be returned to the same state as it was before the script run. whatever configuration happened in the CommonCleanup class will get cached and in the CommonClean class all the cache will get removed.*

*CommonCleanup class - format:*

*class ScriptCommonCleanup(aetest.CommonCleanup):*

*@aetest.subsection*

*def remove\_testbed\_configurations(self):*

*pass*

*@aetest.subsection*

*def disconnect\_from\_devices(self):*

*pass*

***NOTE:****Do not forget to mention the****@aetest.subsection****above every method definition in CommonCleanup class*

***What is Subsection?***

*CommonSetup and CommonCleanup is made up of Subsections.*

*Subsections are independent and each subsection runs regardless of any previous section’s result. The control of whether to abort/skip/continue after an unexpected result is entirely in the hands of the user.*

**Key Matrices used in FTL:**

**ConfigProvider:**

It is responsible for all the connections to FMC and FTD and Endpoints. It will establish a connection based on the credentials which we are giving through Testbed file.

**Store:**

It is responsible for collecting the configuration details from the yaml files and stores in the variable. Later it will be used to create the policy interface zones etc...

**with:**

with statement is used in testclass. Mainly it is used for resource management. it more or less works like try/finally.

**steps:**

It comes default with pyATS package. we can further divide subsection into smaller units called steps. it will help us to understand the  code in a more clear manner. It also gives the clear vision to the output log. Each step is started by calling steps.start(), and providing it a short descriptive name for that new step. A longer description can also be provided, which will be included in the final report. You can find the details under OUTPUT:

Example Format for with and steps:

@aetest.test

def says(self, steps):

# breaking down this test into two steps

# using python "with" statement and steps parameter

with steps.start('first step',

description = 'this is the first step'):

print('one small step for [a] man')

with steps.start('second step',

description = 'this is the second step'):

print('one giant leap for mankind')

***loop:***

*We are allowed to perform section looping in pyATS. It will help us to run a particular set of codes fro different test cases.  decorator used for looping is****@loop****each loop will give us a separate output in a log file.*

*Example format for loop:*

@aetest.loop(uids=['testcase\_one', 'testcase\_two'])

class Testcase(aetest.Testcase):

# setup section of this testcase is run once

# every time the testcase is looped.

@aetest.setup

def setup(self):

pass

***OUTPUT:***

*When aetest finishes running a testscript, an overall summary report is provided to the user. This report provides an outlook of what testcases were run, and what their results are. Depending on the*[*execution mode*](https://pubhub.devnetcloud.com/media/pyats/docs/aetest/run.html#running-aetest-script)*, this reporting behavior may differ.*

# Example

# -------

#

# the following is an example standalone reporter output

+------------------------------------------------------------------------------+

| Detailed Results |

+------------------------------------------------------------------------------+

SECTIONS/TESTCASES RESULT

--------------------------------------------------------------------------------

.

|-- common\_setup PASSED

| |-- sample\_subsection\_1 PASSED

| `-- sample\_subsection\_2 PASSED

|-- tc\_one PASSED

| |-- prepare\_testcase PASSED

| |-- simple\_test\_1 PASSED

| |-- simple\_test\_2 PASSED

| `-- clean\_testcase PASSED

|-- TestcaseWithSteps ERRORED

| |-- setup PASSED

| | |-- Step 1: this is a description of the step PASSED

| | `-- Step 2: another step PASSED

| |-- step\_continue\_on\_failure\_and\_assertions FAILED

| | |-- Step 1: assertion errors -> Failed FAILED

| | `-- Step 2: allowed to continue executing FAILED

| |-- steps\_errors\_exits\_immediately ERRORED

| | `-- Step 1: exceptions causes all steps to skip over ERRORED

| `-- steps\_with\_child\_steps PASSED

| |-- Step 1: test step one PASSED

| |-- Step 1.1: substep one PASSED

| |-- Step 1.1.1: subsubstep one PASSED

| |-- Step 1.1.1.1: subsubsubstep one PASSED

| |-- Step 1.1.1.1.1: running out of indentation PASSED

| |-- Step 1.1.1.1.1.1: definitely gone too far... PASSED

| |-- Step 1.2: substep two PASSED

| |-- Step 2: test step two PASSED

| |-- Step 2.1: function step one PASSED

| |-- Step 2.2: function step two PASSED

| `-- Step 2.3: function step three PASSED

`-- common\_cleanup PASSED

`-- clean\_everything PASSED

***Architecture:***

+--------------+

| TestScript |

+-------+------+

|

+----------------------------+---------------------------+

| | |

+------+------+ +--------+-------+ +-------+-------+

| CommonSetup | | Testcases | | CommonCleanup |

+------+------+ +--------+-------+ +-------+-------+

| | |

+------+------+ | +------+------+

| subsections | +----------+-----------+ | subsections |

+-------------+ | | | +-------------+

+---+---+ +---+---+ +----+----+

| setup | | tests | | cleanup |

+-------+ +-------+ +---------+

**Theoretical Explanation:**

Let's Assume We are having Folder TEST under this we re having main\_script.py(python file) and subfolder called data, in which all the yaml files will be present. Once we triggered the test in pycharm, pyATS initiate a Easypy to set an environment for which we need a Jobfile i.e(JOBS) [responsible for aggregating multiple scripts and run under a environment]. Once the Easypy process gets passed it will start with CommonSetup and then TestCase classes and finally the CommonCleanup class.

For More Clarification Kindly go through the below link:  
  
<https://developer.cisco.com/docs/pyats/api/>