Marvin - Testing with Python

> *The original version of this document is available on github with syntax highlighting*

Marvin - is a python testing framework for Apache CloudStack. Tests written using the framework use the unittest module under the hood. The unittest module is the Python version of the unit-testing framework originally developed by Kent Beck et al and will be familiar to the Java people in the form of JUnit. The following document will act as a tutorial introduction to those interested in testing CloudStack with python. This document does not cover the python language and we will be pointing the reader instead to explore some tutorials that are more thorough on the topic. In the following we will be assuming basic python scripting knowledge from the reader. The reader is encouraged to walk through the tutorial in full and explore the existing tests in the repository before beginning to write tests.

- Installation
    - Dependencies
    - Windows specifics
    - Compiling and Packaging
    - Installation
- Writing the test
- Parts of the test
    - imports
    - TestData
    - TestDeployVM
- Test Categories
- Running the test
    - IDE - Eclipse and PyDev
    - CLI - shell run
- Test Pattern
- Advanced test cases
    - setUpClass, tearDownClass
    - Checkin/smoke Tests
        - Building
        - Deploying Database
        - Start the management server
        - Sync Marvin APIs
        - Run the Tests
        - Including your own
    - Existing Tests
        - Smoke
        - Component
        - Devcloud Tests
- User Tests
- Logging
- Marvin Deployment Configurations
    - What does it look like?
    - How do I generate it?
- Nose Plugins
    - Nose-timer
    - Marvin plugin
    - Attribute selection
- Guidelines to choose scenarios for integration
    - Scenario
    - Backend Verification with paramiko/and other means
- Python Resources
- Acknowledgements

Installation

Marvin requires Python 2.6 for installation but the tests written using marvin utilize Python 2.7 to the fullest. You should follow the test environment setup instructions here before proceeding further.

Dependencies

- Marvin dependencies are installed automatically by the installer
- python-paramiko - remote ssh access, ssh-agent
- mysql-connector-python - db connectivity
- nose - test runner, plugins
- marvin-nose - marvin plugin for nose

Windows specifics

- For Windows (non-cygwin) development environment, following additional steps are needed to be able to install Marvin:
- The mysql-connector-python for Windows can be found here
- pycrypto binaries are required since Windows does not have gcc/glibc. Download here

Compiling and Packaging

The `developer` profile compiles and packages Marvin. It does *NOT* install marvin your default PYTHONPATH.

```
bash
$ mvn -P developer -pl :cloud-marvin
```

Alternately, you can fetch marvin from jenkins as well: packages are built every hour here

Installation

```
bash
$ pip install tools/marvin/dist/Marvin-*.tar.gz
```

To upgrade an existing marvin installation and sync it with the latest APIs on the managemnet server, call this command

```
bash
$ pip install --upgrade tools/marvin/dist/Marvin-*.tar.gz
```

First Things First

You will need a CloudStack management server that is already deployed, configured and ready to accept API calls. You can pick any management server in your lab that has a few VMs running on it or you can use DevCloud or the simulator environment deployed for a checkin test #checkin. Create a sample json config file `demo.cfg` telling marvin where your management server and database server are.

The `demo.cfg` json config looks as shown below:

```
{
    "dbSvr": {
        "dbSvr": "marvin.testlab.com",
        "passwd": "cloud",
        "db": "cloud",
        "port": 3306,
        "user": "cloud"
    },
    "logger": {
         "LogFolderPath": "/tmp/"
    },
    "mgtSvr": [
        {
            "mgtSvrIp": "marvin.testlab.com",
            "port": 8096,
            "user": "root",
            "passwd": "password",
            "hypervisor": "XenServer",
        }
    ]
}
```

> *- Note: `dbSvr` is the location where mysql server is running and passwd is the password for user cloud*

- open up the integration.port on your management server `iptables -I INPUT -p tcp --dport 8096 -j ACCEPT`

- Change the global setting `integration.api.port` on the CloudStack GUI to `8096` and restart the management server

Writing the test

Without much ado, let us jump into test case writing. Following is a working scenario we will test using Marvin.

> *You are encouraged to write your tests in a Pydev+Eclipse environment as this*

features auto-completion for the test that follows. We will explain how to
run the tests in eclipse later.

- create a testcase class
- setUp a user account - name: user, passwd: password
- deploy a VM into that user account using the default small service offering and CentOS template
- verify that the VM we deployed reached the 'Running' state
- tearDown the user account - basically delete it to cleanup acquired resources
- If you have test data pertaining to a test suite, use the file under tools/marvin/marvin/config/test_data.py. Please refer existing smoke test suites for the same.

Here is our test_deploy_vm.py module:

```
python
#All tests inherit from cloudstackTestCase
from marvin.cloudstackTestCase import cloudstackTestCase

#Import Integration Libraries

#base - contains all resources as entities and defines create, delete, list operations on them
from marvin.integration.lib.base import Account, VirtualMachine, ServiceOffering

#utils - utility classes for common cleanup, external library wrappers etc
from marvin.integration.lib.utils import cleanup_resources

#common - commonly used methods for all tests are listed here
from marvin.integration.lib.common import get_zone, get_domain, get_template


class TestDeployVM(cloudstackTestCase):
    """Test deploy a VM into a user account
    """


        @classmethod
    def setUpClass(cls):
        super(TestDeployVM, cls)



    def setUp(self):
        self.apiclient = self.testClient.getApiClient()
        self.testdata = self.testClient.getParsedTestDataConfig()


        # Get Zone, Domain and Default Built-in template
        self.domain = get_domain(self.apiclient)
        self.zone = get_zone(self.apiclient, self.testClient.getZoneForTests())


        self.testdata["mode"] = self.zone.networktype
        self.template = get_template(self.apiclient, self.zone.id, self.testdata["ostype"])


        #create a user account
        self.account = Account.create(
            self.apiclient,
            self.testdata["account"],
            domainid=self.domain.id
        )

        #create a service offering
        small_service_offering = self.testdata["service_offerings"]["small"]
        small_service_offering['storagetype'] = 'local'
        self.service_offering = ServiceOffering.create(
            self.apiclient,
            small_service_offering
        )
        #build cleanup list
        self.cleanup = [
            self.service_offering,
            self.account
        ]


        def tearDown(self):
        try:
            cleanup_resources(self.apiclient, self.cleanup)
        except Exception as e:
            self.debug("Warning! Exception in tearDown: %s" % e)


    def test_deploy_vm(self):
        """Test Deploy Virtual Machine

        # Validate the following:
        # - listVirtualMachines returns accurate information
        """
        self.virtual_machine = VirtualMachine.create(
            self.apiclient,
            self.testdata["virtual_machine"],
            accountid=self.account.name,
            zoneid=self.zone.id,
            domainid=self.account.domainid,
            serviceofferingid=self.service_offering.id,
            templateid=self.template.id
        )

        list_vms = VirtualMachine.list(self.apiclient, id=self.virtual_machine.id)

        self.debug(
            "Verify listVirtualMachines response for virtual machine: %s"\
            % self.virtual_machine.id
        )

        self.assertEqual(
            isinstance(list_vms, list),
            True,
            "List VM response was not a valid list"
        )
        self.assertNotEqual(
            len(list_vms),
            0,
            "List VM response was empty"
        )

        vm = list_vms[0]
        self.assertEqual(
            vm.id,
            self.virtual_machine.id,
            "Virtual Machine ids do not match"
        )
        self.assertEqual(
            vm.name,
            self.virtual_machine.name,
            "Virtual Machine names do not match"
        )
        self.assertEqual(
            vm.state,
            "Running",
            msg="VM is not in Running state"
        )
```

Parts of the test

imports

- cloudstackTestCase - All the cloudstack marvin tests inherit from this class. The class provides the tests with apiclient and dbclients using which calls can be made to the API server. dbclient is useful for verifying database values using SQL
- lib.base - the base module contains all the resources one can manipulate in cloudstack. eg: VirtualMachine, Account, Zone, VPC, etc. Each resource defines a set of operations. For eg: VirtualMachine.deploy, VirtualMachine.destroy, VirtualMachine.list, etc. Each operation makes an API call: For eg: VirtualMachine.recover -> recoverVirtualMachineCmd returning a recoverVirtualMachineResponse. When dealing with the tests one only has to identify the set of resources one will be using and all the operations will autocomplete if you are using an IDE like PyDev/PyCharm
- lib.utils - utility classes for performing verification/actions outside the API. Eg. ssh utilities, random_string generator, cleanup etc.
- lib.common - simple methods that are commonly required for most of the tests, eg: a template -get_template, a zone-get_zone, etc

TestData

The test data class carries information in a dictionary object. (*key*, *value*) pairs in this class are needed to be externally supplied to satisfy an API call. For eg: In order to create a VM one needs to give a displayname and the vm name. These are externally supplied data. It is not mandatory to use the testdata class to supply to your test. In all cases you can simply send the right arguments to the `Resource.operation(` method of your resource without using testdata dictionaries. The advantage of testdata is keeping all data to be configurable in a single place.

Mention all test data related to a test suite under tools/marvin/marvin/config/test_data.py. In our case we have identified that we need an `account` (firstname,lastname etc), a `virtual_machine` (with name and displayname) and a `service_offering` (with cpu: 128 and some memory) as test data. Please refer to current test suites under smoke for specific examples

TestDeployVM

- TestDeployVM is our test class that holds the suite of tests we want to perform. All test classes start with Capital caseing and the `Test` prefix. Ideally only one test class is contained in every module
- The comment in triple quotes is used to describe the scenario being tested.
- `setUp()` - the setup method is run before every test method in the class and is used to initialize any common data, clients, resources required in our tests. In our case we have initialized our testclients - apiclient and dbclient identified the zone, domain and template we will need for the VM and created the user account into which the VM shall be deployed into.
- `self.cleanup =[]`. The cleanup list contains the list of objects which should be destroyed at the end of our test run in the tearDown method
- `tearDown()` - the teardown method simply calls the cleanup (delete) associated with every resource thereby garbage collecting resources of the test
- `test_deploy_vm` - our test scenario. All methods must begin with the `test_` prefix
- VirtualMachine.creat(-> this is the deployVirtualMachineCmd)
- triple quote comments talk about the scenario in detail
- Multiple asserts are made with appropriate failure messages
- Read up about asserts here
- You can also use the should_dsl to do asserts as it is included with the Marvin install and is more readable

Test Categories

Marvin supports test categories which enables you to run specific tests for product areas. For example if you have made a change in the accounts product area, there's a way to trigger all accounts related tests in both smoke and component tests directories. Just put a tag 'accounts' and point the directories /files

Example:

nosetests --with-marvin --marvin-config=[config] --hypervisor=xenserver -a tags=accounts [file(s)]

More info on this wiki page: Categories

Running the test

IDE - Eclipse and PyDev

In PyDev you will have to setup the default test runner to be nose. For this:

- goto Window->Preferences->PyDev

- PyUnit - set the testrunner to nosetests
- In the options window specify the following

- --with-marvin
- --marvin-config=/path/to/demo.cfg

- Save/Apply

Now create a Debug Configuration with the project set the one in which you are writing your tests. And the main module to be your `test_deploy_vm.py` script we defined earlier. Hit Debug and you should see your test run within the Eclipse environment and report failures in the *Debug Window*. You will also be able to set breakpoints, inspect values, evaluate expressions while debugging like you do with Java code in Eclipse.

CLI - shell run

Create a new python module `test_deploy_vm` of type unittest, copy the above test class and save the file using your favorite editor. On your shell environment you can run the tests as follows:

```
bash
tsp@cloud:~/cloudstack# nosetests --with-marvin --marvin-config=demo.cfg test_deploy_vm.py
Test Deploy Virtual Machine  ok


----
Ran 1 test in 10.396s


OK
```

Congratulations, your test has passed!

Running from the CLI you can also experiment with the various plugins provided by Nose described in a later section.

Test Pattern

An astute reader would by now have found that the following pattern has been used in the test examples shown so far and in most of the suites in the `test /integration` directory:

- creation of an account
- deploying Vms, running some test scenario
- deletion of the account

This pattern is useful to contain the entire test into one atomic piece. It helps prevent tests from becoming entangled in each other ie we have failures localized to one account and that should not affect the other tests. Advanced examples in our basic verification suite are written using this pattern. Those writing tests are encouraged to follow the examples in `test/integration/smoke` directory.

Advanced test cases

Many more advanced tests have been written in the `test/integration/` directory of the codebase. Suites are available for many features in cloudstack. You should explore these tests for reference.

setUpClass, tearDownClass

Sometimes it is not favourable for tests do have setUp and teardown run after each test. You want the setup to run only once per module and remain commonly available for all tests in the suite. In such cases you will use the `@classmethod setUpClass` defined by python unittest (since 2.7)

for eg:

```
python
class TestDeployVM(cloudstackTestCase):



    @classmethod
    def setUpClass(cls):
        cls.apiclient = super(TestDeployVM, cls).getClsTestClient().getApiClient()
```

Note that the testclient is available from the superclass using getClsTestClient in this case.

Checkin/smoke Tests

The agent simulator and marvin are integrated into maven build phases to help you run basic tests before pushing a commit. These tests are integration tests that will test the CloudStack system as a whole. Management Server will be running during the tests with the Simulator Agent responding to hypervisor commands. For running the checkin tests, your developer environment needs to have Marvin installed and working with the latest CloudStack APIs. These tests are lightweight and should ensure that your commit doesnt break critical functionality for others working with the master branch. The checkin-tests utilize marvin and a one-time installation of marvin will be done so as to fetch all the related dependencies. Further updates to marvin can be done by using the sync mechanism described later in this section. In order for these tests to run on simulator, we need to add an attribute <required_hardware="false"> to these test cases.

These build steps are similar to the regular build, deploydb and run of the management server. Only some extra switches are required to run the tests and should be easy to recall and run anytime:

Building

Build with the -Dsimulator switch to enable simulator hypervisors

```
$ mvn -Pdeveloper -Dsimulator clean install
```

Deploying Database

In addition to the regular deploydb you will be deploying the simulator database where all the agent information is stored for the mockvms, mockvolumes etc.

```
$ mvn -Pdeveloper -pl developer -Ddeploydb
$ mvn -Pdeveloper -pl developer -Ddeploydb-simulator
```

Start the management server

Same as regular jetty:run.

```
$ mvn -pl client jetty:run -Dsimulator
```

> *To enable the debug ports before the run*
> ***export MAVEN_OPTS="-XX:MaxPermSize=512m -Xmx2g -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y, suspend=n"***

Sync Marvin APIs

Marvin also provides the sync facility which contacts the API discovery plugin on a running cloudstack server to rebuild its API classes and integration libraries:

You can install/upgrade marvin using the sync mechanism as follows.

```
$ sudo mvn -Pdeveloper,marvin.sync -Dendpoint=localhost -pl :cloud-marvin
```

Run the Tests

Once the simulator is up, In a separate session you can use the following two commands to bring up n zone(below example creates an advanced zone) and run tests.

command 1: Below command deploys a datacenter.

```
$ python tools/marvin/marvin/deployDataCenter.py \
    -i setup/dev/advanced.cfg
```

> *Example configs are available in setup/dev/advanced.cfg and setup/dev/basic.cfg*

command 2: Below command runs the test.

```
$ export MARVIN_CONFIG=setup/dev/advanced.cfg
$ export TEST_SUITE=test/integration/smoke
$ export ZONE_NAME=Sandbox-simulator
$ nosetests-2.7 \
  --with-marvin \
  --marvin-config=${MARVIN_CONFIG} \
  -w ${TEST_SUITE} \
  --with-xunit \
  --xunit-file=/tmp/bvt_selfservice_cases.xml \
  --zone=${ZONE_NAME} \
  --hypervisor=simulator \
  -a tags=advanced,required_hardware=false
```

The `--zone` argument should match the name of the zone defined in the config file (currently Sandbox-simulator for basic.cfg and advanced.cfg).

Including your own

Check-In tests are the same as any other tests written using Marvin. The only additional step you need to do is ensure that your test is driven entirely by the API only. This makes it possible to run the test on a simulator. Once you have your test, you need to tag it to run on the simulator so the marvin test runner can pick it up during the checkin-test run. Then place your test module in the `test/integration/smoke` folder and it will become part of the checkin test run.

For eg:

```
@attr(tags =["advanced", "smoke"],required_hardware="false")
def test_deploy_virtualmachine(self):
"""Tests deployment of VirtualMachine
"""
```

The sample simulator configurations for advanced and basic zone is available in setup/dev/ directory. The default configuration setup/dev/advanced.cfg deploys an advanced zone with two simulator hypervisors in a single cluster in a single pod, two primary NFS storage pools and a secondary storage NFS store. If your test requires any extra hypervisors, storage pools, additional IP allocations, VLANs etc - you should adjust the configuration accordingly. Ensure that you have run all the checkin tests in the new configuration. For this you can directly edit the JSON file or generate a new configuration file. The setup/dev/advanced.cfg was generated as follows

```
$ cd tools/marvin/marvin/sandbox/advanced
$ python advanced_env.py -i setup.properties -o advanced.cfgThese configurations are generated using the marvin configGenerator module. You can write your own configuration by following the examples shown in the configGenerator module:
```

- describe_setup_in_basic_mode()
- describe_setup_in_advanced_mode()
- describe_setup_in_eip_mode()

More detailed explanation of how the JSON configuration works is shown later sections of this tutorial.

Existing Tests

Tests with more backend verification and complete integration of suites for network, snapshots, templates etc can be found in the `test/integration/smoke` and `test/integration/component`. Almost all of these test suites use common library wrappers written around the test framework to simplify writing tests. These libraries are part of the `marvin.lib` package. Ensure that you have gone through the existing tests related to your feature before writing your own.

The integration library takes advantage of the fact that every resource - VirtualMachine, ISO, Template, PublicIp etc follows the pattern of

- create - where we cause creation of the resource eg: deployVirtualMachine
- delete - where we delete our resource eg: deleteVolume
- list - where we look for some state of the resource eg: listPods

Marvin can auto-generate these resource classes using API discovery. The auto-generation ability is being added as part of this refactor

Smoke

This is our so-called *BVT* - basic verification tests. Tests here include those that check the basic sanity of the cloudstack features. Include only simple tests for your feature here. If you are writing a check-in test, this the where the test module should be put.

Component

More in-depth tests drilling down the entire breadth of a feature can be found here. These are used for regression testing.

Devcloud Tests

Some tests have been tagged to run only for devcloud environment. In order to run these tests you can use the following command after you have setup your management server and the devcloud vm is running with `tools/devcloud/devcloud.cfg` as its deployment configuration.

```
tsp@cloud:~/cloudstack# nosetests --with-marvin --marvin-config=tools/devcloud/devcloud.cfg  -a tags='devcloud'
test/integration/smoke


Test Deploy Virtual Machine  ok
Test Stop Virtual Machine  ok
Test Start Virtual Machine  ok
Test Reboot Virtual Machine  ok
Test destroy Virtual Machine  ok
Test recover Virtual Machine  ok
Test destroy(expunge) Virtual Machine  ok


----

Ran 7 tests in 10.001s


OK
```

User Tests

The test framework by default runs all its tests under 'admin' mode which means you have admin access and visibility to resources in cloudstack. In order to run the tests as a regular user/domain-admin - you can apply the @user decorator which takes the arguments (account, domain, accounttype) at the head of your test class. The decorator will create the account and domain if they do not exist.

If the decorator is not suitable for you, for instance, you have to make some API calls as an admin and others as a user you can get two apiClients in your test. One for the admin accessiblity operations and another for user access only. The `getUserApiClient()` can be used to obtain a user apiclient instead of the default admin `getApiClient()`

Logging

The logs from the test client detailing the requests sent by it and the responses fetched back from the management server can be found under `/tmp/testclient.log`. By default all logging is in `INFO` mode. In addition, you may provide your own set of `DEBUG` log messages in tests you write. Each `cloudstackTestCase` inherits the debug logger and can be used to output useful messages that can help troubleshooting the testcase when it is running.

nosetests will capture all the logs from running a single test and show them as part of a failure. This allows isolating the sequence of calls made that caused the failure of the test. Successful tests do not show any logs.

Marvin Deployment Configurations

Marvin can be used to configure a deployed Cloudstack installation with Zones, Pods and Hosts automatically in to *Advanced* or *Basic* network types. This is done by describing the required deployment in a hierarchical json configuration file. But writing and maintaining such a configuration is cumbersome and error prone. Marvin's configGenerator is designed for this purpose. A simple hand written python description passed to the configGenerator will generate the compact json configuration of our deployment.

Examples of how to write the configuration for various zone models is within the configGenerator.py module in your marvin source directory. Look for methods `describe_setup_in_advanced_mode()/ describe_setup_in_basic_mode()`.

What does it look like?

Shown below is such an example describing a simple one host deployment:

```
json
{
    "zones": [
        {
            "name": "Sandbox-XenServer",
            "guestcidraddress": "10.1.1.0/24",
            "physical_networks": [
                {
                    "broadcastdomainrange": "Zone",
                    "name": "test-network",
                    "traffictypes": [
                        {
                            "typ": "Guest"
                        },
                        {
                            "typ": "Management"
                        },
                        {
                            "typ": "Public"
                        }
                    ],
                    "providers": [
                        {
                            "broadcastdomainrange": "ZONE",
                            "name": "VirtualRouter"
                        }
                    ]
                }
            ],
            "dns1": "10.147.28.6",
            "ipranges": [
                {
                    "startip": "10.147.31.150",
                    "endip": "10.147.31.159",
                    "netmask": "255.255.255.0",
                    "vlan": "31",
                    "gateway": "10.147.31.1"
                }
            ],
            "networktype": "Advanced",
            "pods": [
                {
                    "endip": "10.147.29.159",
                    "name": "POD0",
                    "startip": "10.147.29.150",
                    "netmask": "255.255.255.0",
                    "clusters": [
                        {
                            "clustername": "C0",
                            "hypervisor": "XenServer",
                            "hosts": [
                                {
                                    "username": "root",
                                    "url": "http://10.147.29.58",
                                    "password": "password"
                                }
                            ],
                            "clustertype": "CloudManaged",
                            "primaryStorages": [
                                {
                                    "url": "nfs://10.147.28.6:/export/home/sandbox/primary",
                                    "name": "PS0"
                                }
                            ]
                        }
                    ],
                    "gateway": "10.147.29.1"
                }
            ],
            "internaldns1": "10.147.28.6",
            "secondaryStorages": [
                {
                    "url": "nfs://10.147.28.6:/export/home/sandbox/secondary"
                }
            ]
        }
    ],
    "dbSvr": {
        "dbSvr": "10.147.29.111",
        "passwd": "cloud",
        "db": "cloud",
        "port": 3306,
        "user": "cloud"
    },
    "logger":{
         "LogFolderPath": "/tmp/"
    },
    "globalConfig": [
        {
            "name": "storage.cleanup.interval",
            "value": "300"
        },
        {
            "name": "account.cleanup.interval",
            "value": "600"
        }
    ],
    "mgtSvr": [
        {
            "mgtSvrIp": "10.147.29.111",
            "port": 8096
        }
    ]
}
```

What you saw in the beginning of the tutorial was a condensed form of this complete configuration file. If you are familiar with the CloudStack installation you will recognize that most of these are settings you give in the install wizards as part of configuration. What is different from the simplified configuration file are the sections *zones* and *globalConfig*. The *globalConfig* section is nothing but a simple listing of (*key*, *value*) pairs for the *Global Settings* section of CloudStack.

The `zones` section defines the hierarchy of our cloud. At the top-level are the availability zones. Each zone has its set of pods, secondary storages, providers and network related configuration. Every pod has a bunch of clusters and every cluster a set of hosts and their associated primary storage pools. These configurations are easy to maintain and deploy by just passing them through marvin.

```
bash
tsp@cloud:~/cloudstack# nosetests --with-marvin --marvin-config=advanced.cfg -deploy -w /tmp #Empty directory
where there are no tests to be discovered
```

Here we have pointed to a likely empty directory so as to only deploy and configure the zone.

How do I generate it?

The above one host configuration was described as follows:

```
import random
import marvin
from marvin.configGenerator import *

def describeResources():
    zs = cloudstackConfiguration()

    z = zone()
    z.dns1 = '10.147.28.6'
    z.internaldns1 = '10.147.28.6'
    z.name = 'Sandbox-XenServer'
    z.networktype = 'Advanced'
    z.guestcidraddress = '10.1.1.0/24'

    pn = physicalNetwork()
    pn.name = "test-network"
    pn.traffictypes = [traffictype("Guest"), traffictype("Management"), traffictype("Public")]
    z.physical_networks.append(pn)

    p = pod()
    p.name = 'POD0'
    p.gateway = '10.147.29.1'
    p.startip =  '10.147.29.150'
    p.endip =  '10.147.29.159'
    p.netmask = '255.255.255.0'

    v = iprange()
    v.gateway = '10.147.31.1'
    v.startip = '10.147.31.150'
    v.endip = '10.147.31.159'
    v.netmask = '255.255.255.0'
    v.vlan = '31'
    z.ipranges.append(v)

    c = cluster()
    c.clustername = 'C0'
    c.hypervisor = 'XenServer'
    c.clustertype = 'CloudManaged'

    h = host()
    h.username = 'root'
    h.password = 'password'
    h.url = 'http://10.147.29.58'
    c.hosts.append(h)

    ps = primaryStorage()
    ps.name = 'PS0'
    ps.url = 'nfs://10.147.28.6:/export/home/sandbox/primary'
    c.primaryStorages.append(ps)

    p.clusters.append(c)
    z.pods.append(p)

    secondary = secondaryStorage()
    secondary.url = 'nfs://10.147.28.6:/export/home/sandbox/secondary'
    z.secondaryStorages.append(secondary)

    '''Add zone'''
    zs.zones.append(z)

    '''Add mgt server'''
    mgt = managementServer()
    mgt.mgtSvrIp = '10.147.29.111'
    zs.mgtSvr.append(mgt)

    '''Add a database'''
    db = dbServer()
    db.dbSvr = '10.147.29.111'
    db.user = 'cloud'
    db.passwd = 'cloud'
    zs.dbSvr = db

    '''Add some configuration'''
    [zs.globalConfig.append(cfg) for cfg in getGlobalSettings()]

    ''''add loggers'''
    testLogger = logger()
    testLogger.logFolderPath = '/tmp/'
    zs.logger = testLogger
        return zs

def getGlobalSettings():
   globals = { "storage.cleanup.interval" : "300",
               "account.cleanup.interval" : "60",
            }
   for k, v in globals.iteritems():
        cfg = configuration()
        cfg.name = k
        cfg.value = v
        yield cfg

if __name__ == '__main__':
    config = describeResources()
    generate_setup_config(config, 'advanced_cloud.cfg')
```

The `zone(), pod(), cluster(), host()` are plain objects that carry just attributes. For instance a *zone* consists of the attributes - `name, dns entries, network type` etc. Within a zone I create `pod()s` and append them to my zone object, further down creating `cluster()s` in those pods and appending them to the pod and within the clusters finally my `host()s` that get appended to my cluster object. Once I have defined all that is necessary to create my cloud I pass on the described configuration to the `generate_setup_config()` method which gives me my resultant configuration in JSON format.

Nose Plugins

Nose extends unittest to make testing easier. Nose comes with plugins that help integrating your regular unittests into external build systems, coverage, profiling etc. Marvin comes with its own nose plugin for this so you can use nose to drive CloudStack tests. The plugin is installed on installing marvin. Running `nosetests -p` will show if the plugin registered successfully.

```
bash
$ nosetests -p
Plugin xunit
Plugin multiprocess
Plugin capture
Plugin logcapture
Plugin coverage
Plugin attributeselector
Plugin doctest
Plugin profile
Plugin collect-only
Plugin isolation
Plugin pdb
Plugin marvin
```

Nose-timer

There is an interesting plugin named Nose-timer, which helps you to generate the execution time of individual python test. Link here: https://github.com/mahmoudimus/nose-timer

Running nosetest with --with-timer flag.

If you execute testing with marvin, add this parameter in cloud-marvin/pom.xml like this:

```
<executable>nosetests</executable>
<arguments>
  <argument>--with-marvin</argument>
  <argument>--marvin-config</argument>
  <argument>${resolved.user.dir}/${resolved.marvin.config}</argument>
  <argument>-a</argument>
  <argument>tags=${tag}</argument>
  <argument>${resolved.user.dir}/${test}</argument>
  <argument>-v</argument>
  <argument>--with-timer</argument>
</arguments>
```

Marvin plugin

```
bash
$ nosetests --with-marvin --marvin-config=/path/to/basic_zone.cfg  /path/to/tests
```

Attribute selection

The smoke tests and component tests contain attributes that can be used to filter the tests that you would like to run against your deployment. You would use nose attrib plugin for this. Following tags are available for filtering:

- advanced - Typical Advanced Zone
- basic - a basic zone without security groups
- sg - a basic zone with security groups
- eip - an elastic ip basic zone
- advancedns - advanced zone with a netscaler device
- advancedsg - advanced zone with security groups
- devcloud - tests that will run only for the basic zone on a devcloud setup done using tools/devcloud/devcloud.cfg
- speed = 0/1/2 (greater the value lesser the speed)
- multihost/multipods/mulitcluster (test requires multiple set of hosts/pods/clusters)

```
bash
$ nosetests --with-marvin --marvin-config=/path/to/config.cfg -w <test_directory> -a tags=advanced # run tests
tagged to run on an advanced zone


#Use below options to run all test cases under smoke directory on advanced zone "and" are provisioning cases, i.
e., require hardware to run them. See "hypervisor" option to specify against which hypervisor to run them
against, provided your zone and cluster has multiple hosts of various hypervisors type.
$ nosetests-2.7 --with-marvin --marvin-config=/home/abc/softwares/cs_4_4_forward/setup/dev/advanced.cfg  -w
/home/abc/softwares/cs_4_4_forward/test/integration/smoke/ --with-xunit --xunit-file=/tmp/bvt_provision_cases.
xml --zone=<zone_in_cfg> --hypervisor=<xenserver\kvm\vmware> -a tags=advanced,required_hardware=true


#Use below options to run all test cases under smoke directory on advanced zone "and" are selfservice test
cases, i.e., "not" requiring hardware to run them, and can be run against simulator.
$ nosetests-2.7 --with-marvin --marvin-config=/home/abc/softwares/cs_4_4_forward/setup/dev/advanced.cfg  -w
/home/abc/softwares/cs_4_4_forward/test/integration/smoke/ --with-xunit --xunit-file=/tmp/bvt_selfservice_cases.
xml --zone=<zone_in_cfg> --hypervisor=simulator -a tags=advanced,required_hardware=false

#Use below options to run all test cases under smoke directory on advanced zone "and" are selfservice test
cases, i.e., "not" requiring hardware to run them, and can be run against simulator. As well, below "deploy"
option takes care to deploy datacenter as well.
$ nosetests-2.7 --with-marvin --marvin-config=/home/abc/softwares/cs_4_4_forward/setup/dev/advanced.cfg  -w
/home/abc/softwares/cs_4_4_forward/test/integration/smoke/ --with-xunit --xunit-file=/tmp/bvt_provision_cases.
xml --zone=<zone_in_cfg> --hypervisor=simulator -a tags=advanced,required_hardware=false --deploy
```

Guidelines to choose scenarios for integration

There are a few do's and don'ts in choosing the automated scenario for an integration test. These are mostly for the system to blend well with the continuous test infrastructure and to keep environments pure and clean without affecting other tests.

Scenario

- Every test should happen within a CloudStack user account. The order in which you choose the type of account to test within should be
  User > DomainAdmin > Admin
  At the end of the test we delete this account so as to keep tests atomic and contained within a tenant's users space.

- Every test case should clean up it's resource ASAP, after it's no longer in use.
  - Any Create operation should be followed by adding the newly created object to the cleanup set. Otherwise any later failure before cleanup set updated would result in previous allocated object won't be freed.
  - Any resources allocated in setUp() for each test case must be released in tearDown(). Suggest using the variable "self.cleanup" to track the test case resources.
  - Any resources allocated in setUpClass() for whole test class must be released in tearDownClass(). Suggest using the variable "cls._cleanup" to track the class resources.
  - Any resources allocated (excepted in setUpClass()) must be added to the cleanup list then released in tearDown().

- All tests must be written with the perspective of the API. UI directions are often confusing and using the rich API often reveals further test scenarios. You can capture the API arguments using cloudmonkey/firebug.

- Tests should be generic enough to run in any environment/lab - under any hypervisor. If this is not possible then it is appropriate to mark the test with an @attr attribute to signify any specifics. eg: @attr(hypervisor='vmware') for runs only on vmware
- Every resource should be creatable in the test from scratch. Referring to an Ubuntu template is probably not a good idea. Your test must show how to fetch this template or give a static location from where the test can fetch it.
- Do not change global settings configurations in between a test. Make two separate tests for this. All tests run against one given deployment and altering the settings midway is not effective

Backend Verification with paramiko/and other means

- Verifying status of resources within hypervisors is fine. Most hypervisors provide standard SSH server access
- Your tests should include the complete command and its expected output. e.g. `iptables -L INPUT` to list the INPUT chain of iptables
- If you execute multiple commands then all of them should be chained together on one line e.g: `service iptables stop; service iptables start #to stop and start iptables`
- Your script must execute over ssh because this is how Marvin will execute it `ssh <target-backend-machine> "<your script>"`
- Most external devices like F5/ NetScaler have ssh open to execute commands. But you must include the command and its expected output in the test as not everyone is aware of the device's CLI
- If you are using a UI like vCenter to verify something most likely you cannot automate what you see there because there are no ASLv2 licensed libraries for vmware/esx as of today.

Python Resources

- The single largest python resource is the python website itself
- Mark Pilgrim's - 'Dive Into Python' - is another great resource. The book is available for free online. Chapter 1 to 6 cover a good portion of language basics and Chapter 13 & 14 are essential for anyone doing test script development.
- You can also use Zed Shaw's book, Learn Python the Hard Way, if programming is something completely new to you
- To read more about the assert methods the language reference is the ideal place - http://docs.python.org/library/unittest.html.

Acknowledgements

- The original author of the testing framework - Edison Su
- Maintenance and bug fixes - Prasanna Santhanam
- Documentation - Prasanna and Edison

For any feedback, typo corrections please email the dev@cloudstack.apache.org list.