Automated testing of CS UI using Selenium and Python

Parth Jagirdar

Software Test Engineer

Datacenter & Cloud Division

Citrix Systems – Powering Mobile Workstyles and Cloud Services.

Introduction

You would need "very basic" understanding of Python (or coding in general), Selenium and HTML/XML (and similar technologies heavily used in web pages) to get started.

1) What is selenium?

   http://seleniumhq.org

   Selenium is a very useful tool and can be used to create scripts to reproduce specific bugs through UI or can also be used for regression.

   Selenium Webdriver which resides within the client browser; Can record user actions on UI; including data entry. These actions can later be played back N number of times to mimic user actions on UI.

   With logical organization of such recorded scripts we can create and execute full test plans.

2) Selenium has language bindings with Python.

   http://selenium-python.readthedocs.org/en/latest/

3) General HTML / XML / Python / Scripting

   http://www.w3schools.com/

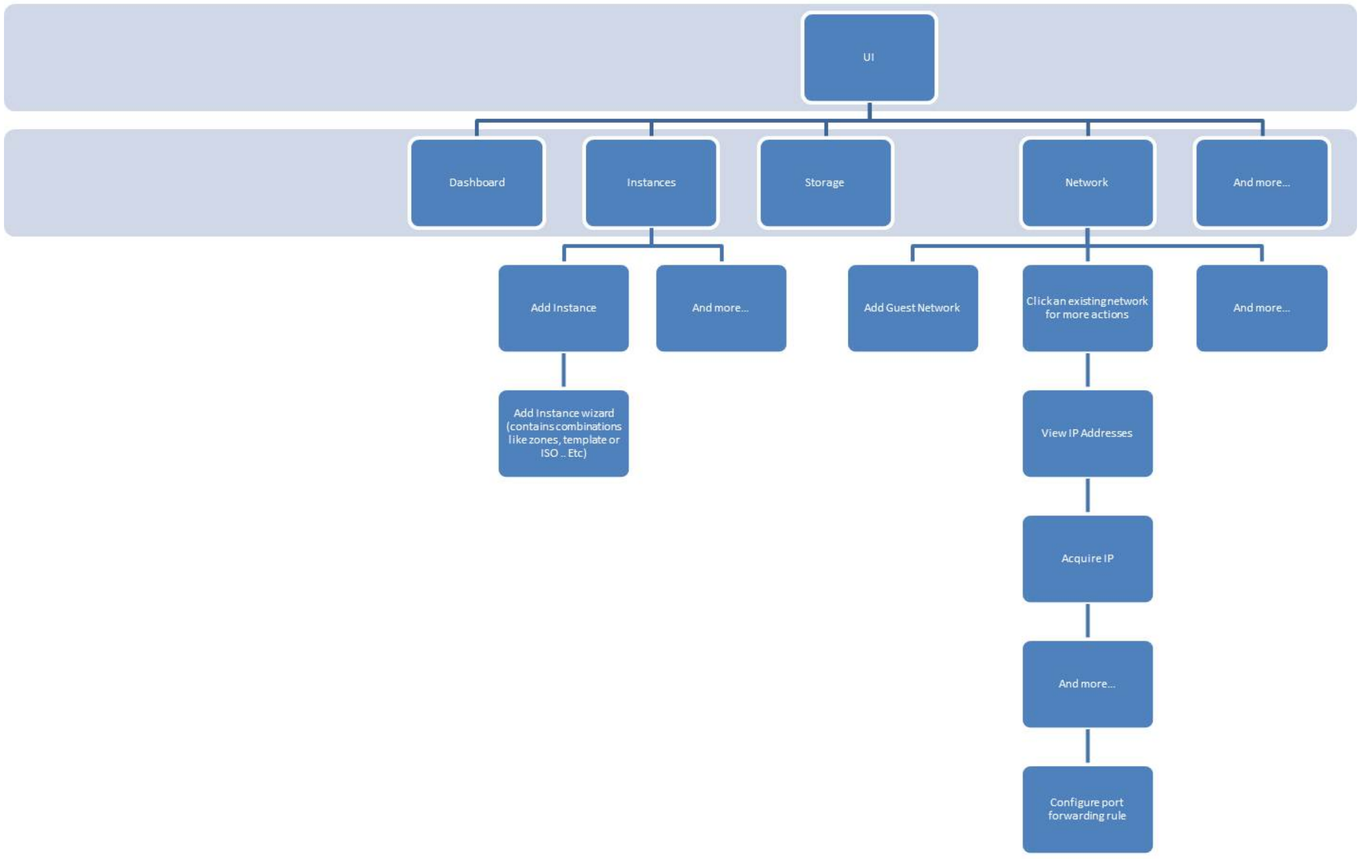
Project outline and framework

Our primary goal is to exercise as many UI elements as possible which are interactive.

We intend Selenium Automation to run as and when required with minimal supervision / intervention from an engineer.

We can break UI into a Tree like structure and think of our goal as to traverse as many leaf nodes as possible with least inter-script dependency as possible.

Following is a brief example with a few nodes.


UI
Dashboard
Instances
Storage
Network
And more...
Add Instance
And more...
Add Guest Network
Click an existing network for more actions
And more...
Add Instance wizard (contains combinations like zones, template or ISO .. Etc)
View IP Addresses
Acquire IP
And more...
Configure port forwarding rule

We can see that two of the farthest leaves have one way dependency.

PF -> Instance: We need a VM Instance present, if we were to configure and verify a PF rule.

So we need to Organize test actions into groups such that it addresses serialization and dependency requirements.

For Example:

A simple, hypothetical test case: log into FTP server 10.1.1.0 and download CentOS ISO from there. (Web-based)

Now we break this test case into multiple actions:

1) Open Browser
2) Type in Server IP/URL
3) Wait till page is loaded
4) Locate CentOS ISO (Within webpage contents)
5) Click on link to download the ISO
6) Verify data integrity by running md5 and comparing it with the one on webpage.

Now let us compile a dependency list; assume that This server is running on one of the VM's managed by CS.

Some of the easy ones we can think of right away are:

1) Need a VM for server (Of course)
2) To host VM we need a infrastructure
3) To create a VM we will also need a template or ISO
4) We need to host a server on this VM
5) We need to upload our CentOS ISO which we will download in our test case
6) Our Server (VM) should be reachable from other networks
7) So we create Port Forwarding and Firewall rules.
8) And so on and on…

It becomes clear that to run 1 test case we will have to ensure that at least 10 test cases have ran successfully; before we begin execution of our original test case.

We can follow a top down approach (the tree above) and start with some golden minimum configuration as part of test initialization. And as we go on, we serialize our test cases such that majority of the dependencies are resolved.

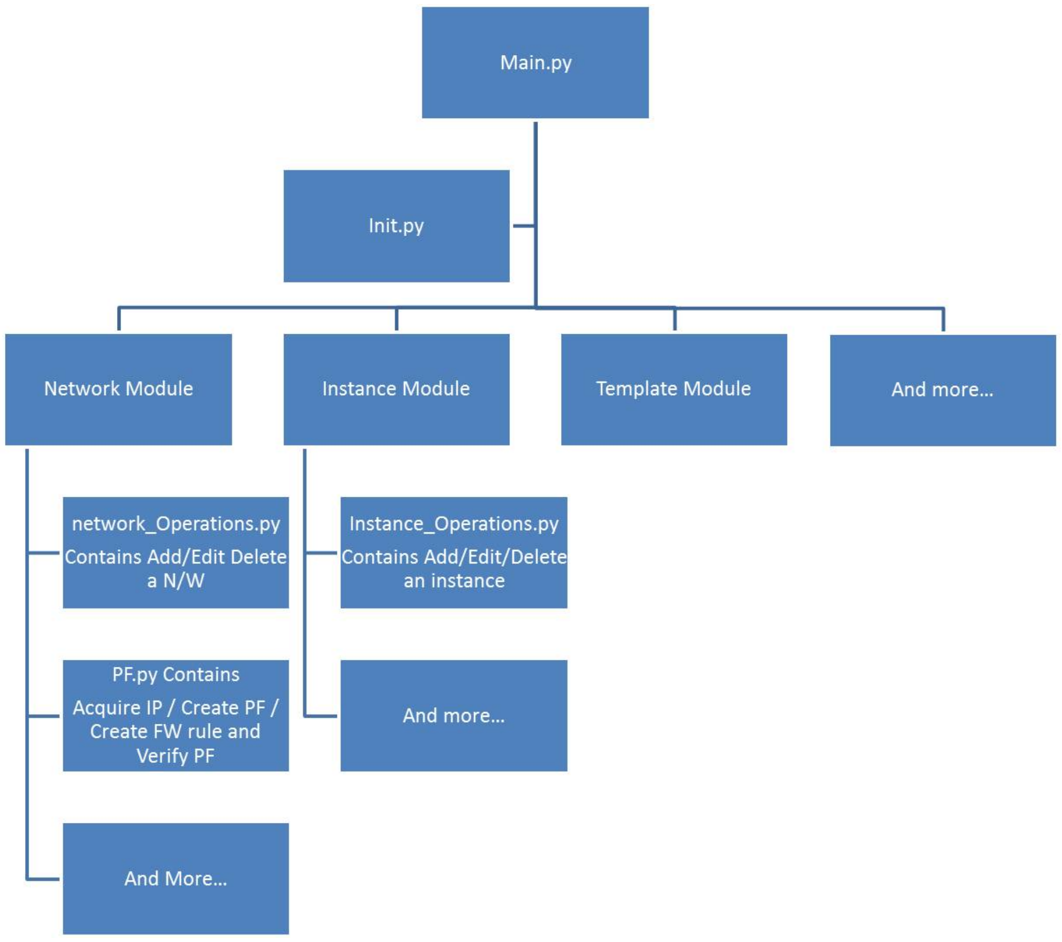
Now let us put all that we have discussed so far into following:

1) Find bare minimum pre requisites for these test cases. (Few of them are)
    a. Zone configuration
    b. Adding Template
    c. Adding VM's
    And so on… We will now arrange these into an Initialization script.

2) Initialization script will grow as Automation efforts progresses. Contributors must identify and add these dependencies as needed.

3) Every script should have dependency only on Initialization script and Global Locators file (Discussed in next section).
4) Scripts will be serialized into main script to resolve serialization dependencies.

So our project organization will look like this. (Consider Global Locators file as one being parallel to Init.py)


Main.py
Init.py
Network Module
Instance Module
Template Module
And more...
network_Operations.py Contains Add/Edit Delete a N/W
Instance_Operations.py Contains Add/Edit/Delete an instance
PF.py Contains Acquire IP / Create PF / Create FW rule and Verify PF
And more...
And More...

Main.py :: We will create test suits here and Serialize Test Cases. Now pass this test suit to Test Runner, which will begin the test execution.

Init.py :: This will be the first one to be executed in Main.py. Contains minimum golden configuration on which rest of the test cases depend upon. Init.py also contains configuration verification.

Modules :: These are nothing but folders to organize Test cases in some logical order. We will use each menu item on CS UI Dashboard as a module.

***.py** :: These are individual files. They must contain test steps and verification both. It is logical to have an organization such that; Create_Instance –> Edit –> Restart –> Delete.

This way at the end of the script execution script cleans for it self.

** We are also considering organizing subsequent test runs into separate Accounts. Thus we can tear down anything left over from the Test by deleting that account.

Challenges and Mitigations

1) Element Locators

UI automation heavily depends on location of interactive elements on UI screen. And just as there are number of different ways to reach your office from your home; we have multiple different ways to identify/locate an element on UI screen.

We have following ways to identify an element on the screen.

(Refer to http://mestachs.wordpress.com/2012/08/13/selenium-best-practices/)

In order of their preference. id > name > css > xpath

Unfortunately we do not have ID's or Name's for most of the elements within CS UI. So we will be relying on xpath for now with provisioning for future ID/Name inclusion into the code.

So to make scripts more robust and portable across code versions with UI changes

We will have a Global Locator file. And every script must have dependency on this file.

This file will contain:

Element Name	Locator	Variable Name
Add Instance	/html/body/div/div/div[2]/div[2]/div[2]/div/div[2]/div/div[2]/span	Instance_add_Instance
Add Guest Network	/html/body/div/div/div[2]/div[2]/div[2]/div/div[2]/div/div[3]/span	Network_add_Guest_network

- Variable names are unique.
- Variable name succeeds its module name
- They should more or less reflect exact actions they represent on UI.
- Only variable must be used in scripts and not full xpath locators.

This way we can change locators at one single location when UI code changes. This will also help adding ID's or names as and when they are available.

2) Dynamically Generated elements

Some elements on UI will be generated dynamically while testing. The challenge is to identify such elements and then find a method to access them.

Now as these elements are generated dynamically at run time we do not have their xpath and thus we need to find a way to identify this dynamically generated element by means other than direct xpath.

Say while creating port forwarding rule we acquired a new IP. This IP was generated dynamically hence we do not have its xpath locator. Careful observation shows that this newly generated IP will be the first one in the table. However if we navigate away and back to the page then this IP may not be the first one in the table. Table may get sorted upon each page load.

Here is how we can deal with it.

1) Navigate to the page and click acquire IP
2) Wait for (worst case scenario) for IP to get acquired and displayed. Now store this IP into a variable.
3) Re-Navigate to this page and using locator of the table (Not the element; we will use table now) search through IP's to find the one we just acquired.
4) Once found Click for more actions on that IP.

Following snippet will clarify::

Server IP will host the acquired IP in our example.

Linkclass will point to the Table's First Column

Comparison takes place For each link in First Column without acquired IP. If Match found click this link. (Else Error will be reported).

```
Server_Ip = None
Server_Ip = driver.find_element_by_xpath("/html/body/div/div/div[2]/div[2]/div[2]/div[3]/div[2]/div/div[2]/table/tbody/tr/td").text

linkclass = None
linkclass = driver.find_elements_by_xpath('/html/body/div/div/div[2]/div[2]/div[2]/div[3]/div[2]/div/div[2]/table/tbody/tr/td/span')

for link in linkclass:
    if link.text == Server_Ip:
        print "found link to IP %s" % Server_Ip
        link.click()
```

There are numerous tables in CS UI and this snippet can be reused.

That's all for now ☺

Current Project Plans

I will be working on BVT (Build Verification Tests) Test plan found @ http://wiki.cloudstack.org/display/QA/Manual+BVT

It has about 90 individual test cases.

We will start with Xenserver, 1 Host and 1 Advance Zone. This will cover about 70% of BVT Test cases.

Thanks you for your time. Please pass on your valuable suggestions.