Yukhe Lavinia

Testing Tools Manual

Our project OneClickVitals is developed in Python and tested using Pylint for static unit testing, Unittest for dynamic unit testing, and Selenium IDE and Selenium WebDriver for functional testing. This manual will give an overview of the installation and guidelines to write and run each testing tool.

Static Testing: Pylint

Pylint is a Python tool that checks our code against Python coding standard and detects errors without code execution. (Logilab, 2014)

Installation

The easiest way to install Pylint is through the command line:

OS X

```
pip install pylint
```

Running Pylint

The following describes the steps in running Pylint:

1. Go to the directory in which our Python module is located.
2. Type pylint ./<module_name> (e.g. models.py)

```
Kashermans-MacBook-Pro:oneclickvitals kasherman$ pylint ./models.py
```

One important note about Pylint is that it is known to give false positives as it will alert regarding spacing or details that are not strictly Python's PEP8 style guide. The good news is that Pylint is fully customizable. We can disable certain messages that don't abide to our need. For instance, in our project OneClickVitals we decided to use more whitespace to improve readability, and thus we disabled the messages C0303, C0330, W0611, C0111, E1101, C0326, C0301, and W0311. The full list of messages can be found at http://docs.pylint.org/features.html.

3. To disable messages, we only need to include --disable=<message_code or name>, like so:

```
Kashermans-MacBook-Pro:oneclickvitals kasherman$ pylint --disable=C0303,C0330,W0
611,C0111,E1101,C0326,C0301,W0311 ./models.py
```

Pylint will generate a thorough report on module type statistics, external dependencies, raw metrics, duplication (which is useful for factoring), and the categories of messages generated.
The most important part, however, is the final score of our code:

```
Global evaluation
-----------------
Your code has been rated at 8.58/10 (previous run: -2.00/10, +10.58)
```

Here we can see that Pylint also compares the current score with the previous one and quantifies the score improvement (in this case it is +10.58).

Dynamic Testing: Unittest

Python unittest is a built-in Python standard library unit testing module. It supports test automation and also the setups and shutdowns of tests. (Python Software Foundation, 2015)

Installation

We used Django as the framework to develop OneClickVitals. When we create a Django app, Django automatically creates a module called tests.py, which is initially empty.

Writing unittest Test Cases

1. On module, type the following:

```
1  from django.test import TestCase
2
```

Django.test.TestCase is a subset of Python's unittest.TestCase. (Django Software Foundation, 2015) Beside Python's basic unittest functionalities, Django's TestCase enables web application-specific tests such as creating TestClient instances and Django's specific assertions for testing form errors and redirections.

2. Import the modules we wish to test, e.g. `models.py` (which defines all models), `forms.py` (which defines the forms and fields in the forms).
3. We want to create a test for each model. Name the class using the model name and attach the word "test."

```
1  from django.test import TestCase
2  from django.utils import timezone
3  from django.shortcuts import render, get_object_or_404, redirect, render_to_response,
   RequestContext
4  from .models import *
5  from .forms import *
6  from django.http import HttpRequest
7  from medical_project.settings import *
8  from django.conf import settings
9  from django.utils.importlib import import_module
10
11 class UserDetailTest(TestCase):
12
```

4. First we want to create the instance of the model.

```
11 class UserDetailTest(TestCase):
12     def create_user_object(self, username="me"):
13         return User.objects.create(username=username)
14
15     def create_user_detail_object(self, user="me", gender="female"):
16         the_user = self.create_user_object(user)
17         return UserDetail.objects.create(user=the_user, gender=gender)
18
```

Since our model UserDetail is built upon another model called User, we need to create the User instance first before creating the UserDetail instance.

5. Create a test method by using the word "test" in front of the method name. For instance, to test whether the UserDetail instance is properly created, we name the method test_user_detail_object_creation. The test method name should be as descriptive as possible to make it more readable if the test fails.

```
19      def test_user_detail_object_creation(self):
20          the_user_detail = self.create_user_detail_object()
21          self.assertTrue(isinstance(the_user_detail, UserDetail))
22          self.assertEqual(the_user_detail.__str__(), "me")
23          self.assertEqual(the_user_detail.gender, "female")
```

6. To test object retrieval, we wrote:

```
25      def test_get_user_detail_object(self, pk=1):
26          the_user_detail = self.create_user_detail_object()
27          you = User.objects.get(pk=pk)
28          user_detail_instance = UserDetail.objects.get(user=you)
29          self.assertEqual(user_detail_instance.__str__(), "me")
```

7. To test another model, create a separate class, model instances, and test methods.

```
81  class VitalSignsTest(TestCase):
82
83      def create_user_object(self, username="me"):
84          return User.objects.create(username=username)
85
86      def create_vitalsigns_object(self, user="me", heart_rate="100", blood_pressure="120/70"):
87          the_user = self.create_user_object(user)
88          return VitalSigns.objects.create(user=the_user, heart_rate=heart_rate,
    blood_pressure=blood_pressure)
89
90      def test_vitalsigns_object_creation(self):
91          the_vitalsigns= self.create_vitalsigns_object()
92          self.assertTrue(isinstance(the_vitalsigns, VitalSigns))
93          self.assertEqual(the_vitalsigns.user.username, "me")
94          self.assertEqual(the_vitalsigns.heart_rate, "100")
95          self.assertEqual(the_vitalsigns.blood_pressure, "120/70")
96
97      def test_get_vitalsigns_object(self, pk=1):
98          the_vitalsigns = self.create_vitalsigns_object()
99          me = User.objects.get(pk=pk)
100         vitalsigns_instance = VitalSigns.objects.get(user=me)
101         self.assertEqual(vitalsigns_instance.user.username, "me")
```

8. To test that the HTTP request of the Vital Signs list is responded properly and that the requested detail is correct, we wrote:

```
103     def test_load_vitalsigns_list(request):
104             context = {'vitalsigns': VitalSigns.objects.all().order_by('-visit_date')}
105             return render(request, 'oneclickvitals/vitalsigns_list.html', context)
```

```
107     def test_load_vitalsigns_details(self, request=HttpRequest(), pk=1):
108         the_vitalsigns = self.create_vitalsigns_object()
109         the_request=request
110         vitalsigns_instance = VitalSigns.objects.get(pk=pk)
111         return render(the_request, 'oneclickvitals/vitalsigns_details.html',
    {'vitalsigns_instance': vitalsigns_instance})
```

Running unittest

1. Go to the directory where we have the manage.py module.
2. Make sure that we are running in the virtual environment. If not, type the following:

```
Kashermans-MacBook-Pro:MedicalProject kasherman$ source myvenv/bin/activate
```

3. Once the virtual environment is activated, we are ready to run the test:

```
(myvenv) Kashermans-MacBook-Pro:MedicalProject kasherman$ python manage.py
test oneclickvitals
```

   We simply type “test” followed by the application name, which in our case is oneclickvitals.

4. When it runs, Django unittest will create a separate test database and will destroy it after it is done. This ensures that we are not touching the actual database.

```
(myvenv) Kashermans-MacBook-Pro:MedicalProject kasherman$ python manage.py
test oneclickvitals
Creating test database for alias 'default'...
..................................
----------------------------------------------------------------------
Ran 34 tests in 1.189s

OK
Destroying test database for alias 'default'...
```

Functional Testing: Selenium

Selenium is a browser automation tool. There are generally two types and we used both: 1) Selenium IDE and 2) Selenium WebDriver.

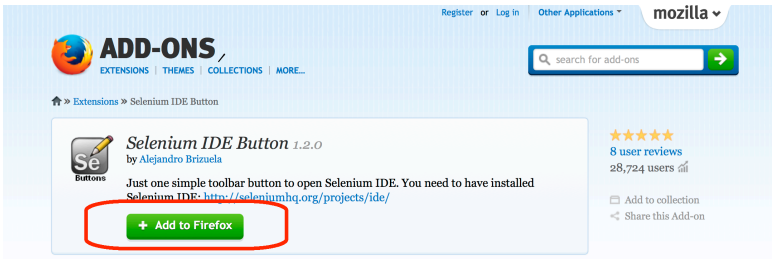
Selenium IDE

Selenium IDE is a complete integrated development environment for Selenium scripts in the form of Firefox extension. (Selenium Project, 2015)

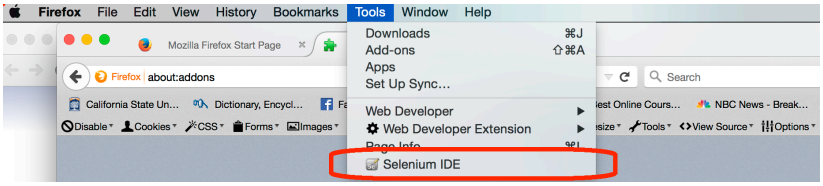
Installation

1. Open Firefox browser.
2. Go to https://addons.mozilla.org/en-US/firefox/addon/selenium-ide-button/

3. Click on “Add to Firefox.”

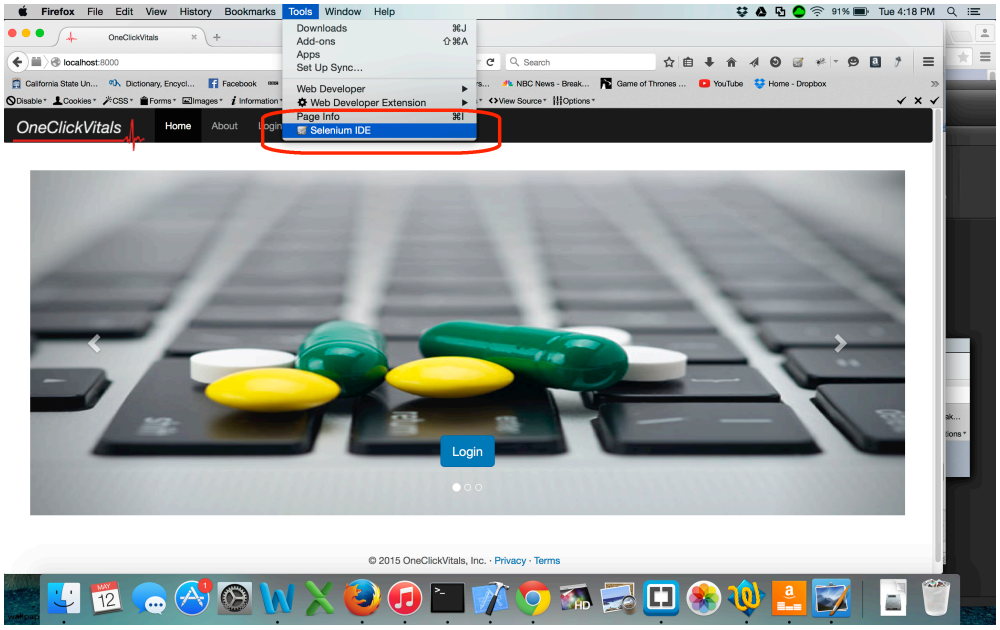


4. Check that Selenium is installed properly.

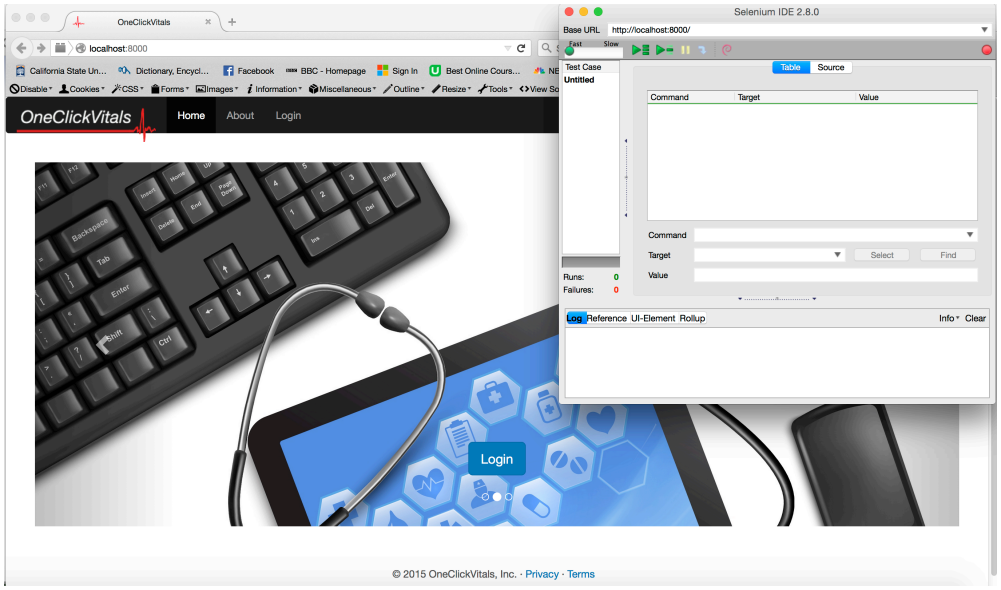


Creating Test Suites or Cases

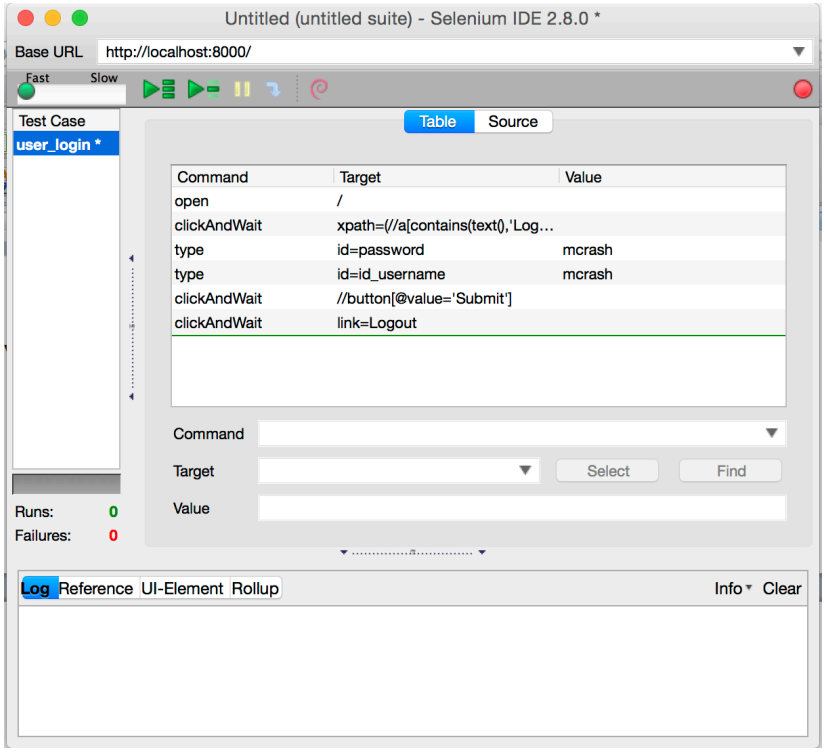
1. Go to the web page we wish to test
2. Open Selenium IDE by clicking Tools > Selenium IDE



3. Once clicked, Selenium will start recording

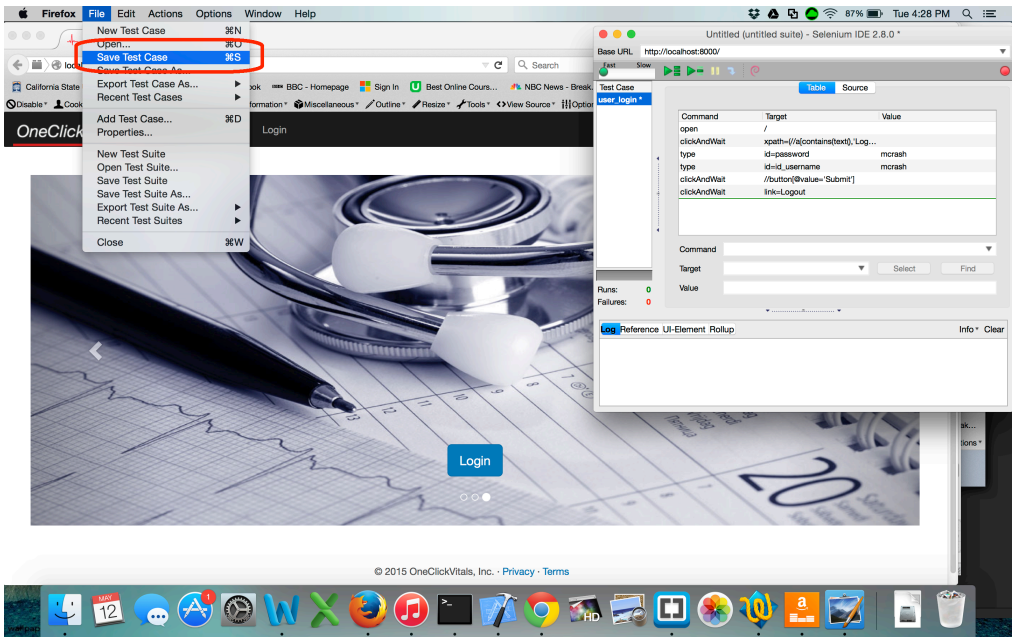


4. As we navigate to do the desired functionality, Selenium will record the steps in three categories: Command (e.g. clickAndWait), Target (the browser element on which we target our commands), and Value (to input the values for the type command).

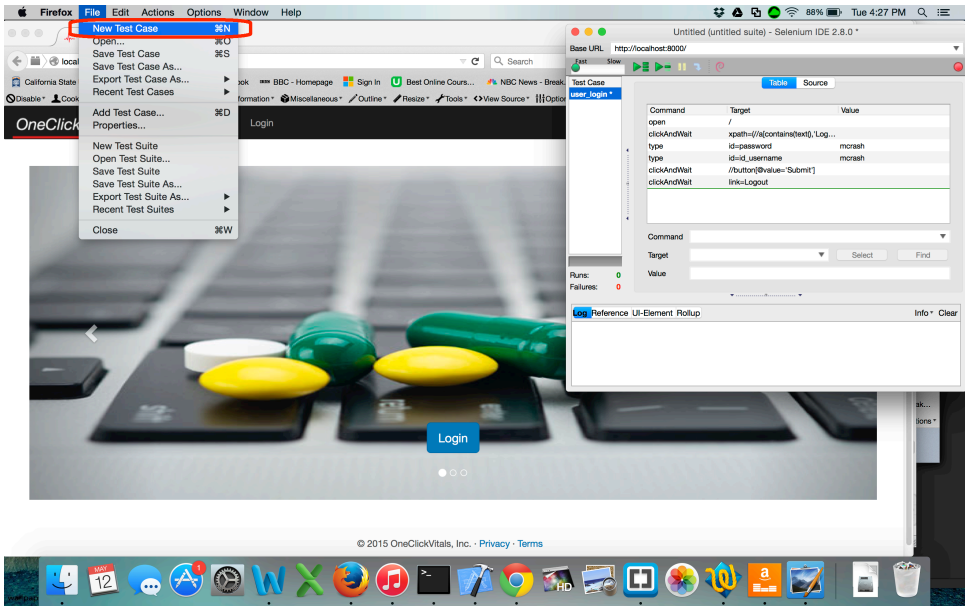


In this example, we test user login. As we navigate to click the “Login” button, get directed to a login page where we inputs our username and password and click “Enter”, get directed to the homepage, and finally logout, Selenium records each step by saving the commands, targets, and values.

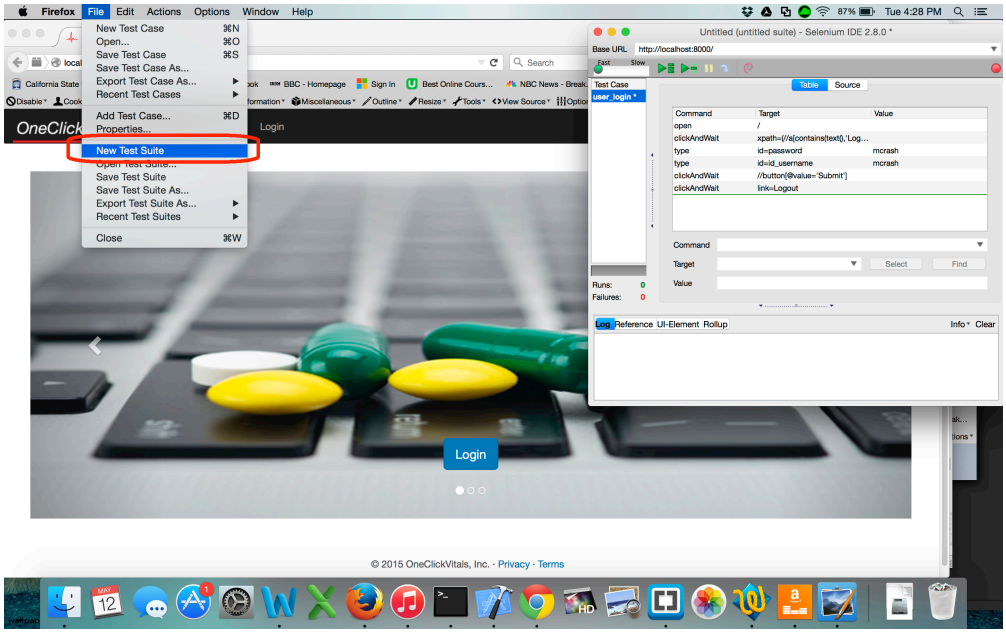
5. To save test cases, click on the Selenium pop-up window and click File > Save Test Case



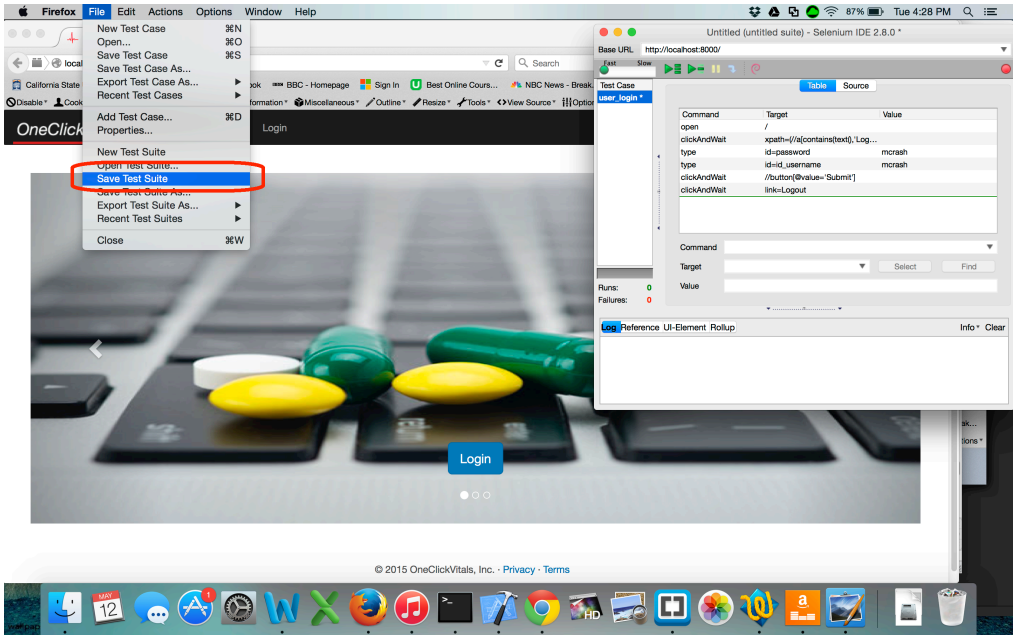
6. The create new test cases, click on the Selenium pop-up window and click File > New Test Case



7. To create a new test suite, click on the Selenium pop-up window and click File > New Test Suite

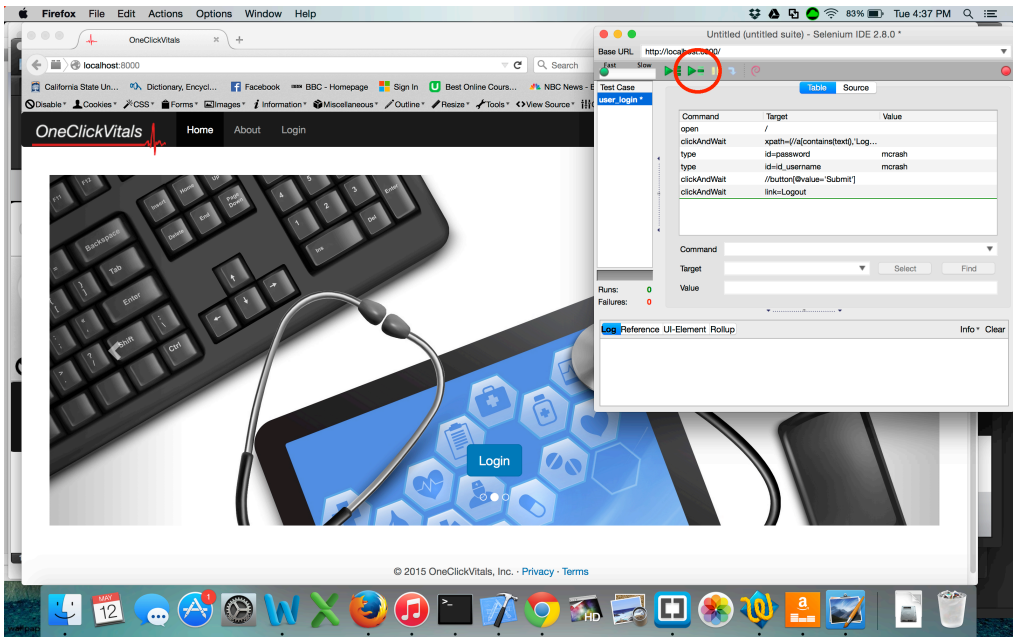


8. To save the test suite, click on the Selenium pop-up window and click File > Save Test Suite



Running the Tests

To run the tests, we need only click on the play button.



Selenium WebDriver

Selenium WebDriver is a language-specific binding to drive a web browser geared to provide better support for dynamic web pages and more robust programming interface. (Selenium Project, 2015)

Installation

We install Selenium WebDriver through the terminal by typing pip install –U selenium, like so:

```
Kashermans-MBP:MedicalProject kasherman$ pip install -U selenium
```

This will install the latest Selenium WebDriver compatible with the latest Firefox browser, Firefox 37. Incompatibility with Firefox browser will prevent the WebDriver test cases to run properly.

Writing Test Cases

1. Create a new directory within the application root directory. We call it selenium_script.
2. Install a new virtual environment in the new directory.

```
Kashermans-MBP:selenium_script kasherman$ python -m venv myvenv
```

3. Create a new Python module in which we are writing the Selenium scripts
4. Import the following:

```
1  import unittest
2  from selenium import webdriver
3  from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
4  from selenium.webdriver.common.keys import Keys
5  from selenium.webdriver.common.action_chains import ActionChains
```

5. Create a class with the parameter unittest.TestCase

```
7  class OneclickvitalsTest(unittest.TestCase):
```

6. Write the setUp class

```
7  class OneclickvitalsTest(unittest.TestCase):
8
9      @classmethod
10     def setUpClass(cls):
11         cls.driver = webdriver.Firefox()
12         cls.driver.maximize_window()
```

Here we specify that we are using Firefox as the web browser. Also, we tell Selenium that it should maximize the browser window once it runs.

7. Write the tearDown class

```
144     @classmethod
145     def tearDownClass(cls):
146         cls.driver.quit()
```

8. Write the test cases

```
21     def test_login(self):
22         self.driver.get('http://localhost:8000/accounts/login/')
23
24         loginform = self.driver.find_element_by_class_name('form-signin')
25
26         username = self.driver.find_element_by_name('username')
27         password = self.driver.find_element_by_name('password')
28
29         username.send_keys('victor.vitals')
30         password.send_keys('vitals')
31         password.send_keys(Keys.RETURN)
```

Here we specify the URL link that the browser will open, tell Selenium to find the elements by class name ("form-signin") and by name ("username" and "password"). Then we provide the values for the username and password, and finally hit the Return key (equivalent to the Enter key on Windows).

Running the Tests

1. To discover the test module, we use another well-known Python unit test framework: nose. Install it by typing "pip install nose."
2. Make sure that the server is running. If not, type the following:

```
(myvenv) Kashermans-MacBook-Pro:MedicalProject kasherman$ python manage.py
runserver
```

3. Open another tab in the terminal by clicking command-T, and go to the directory where we have the selenium script module. In our case, it is the selenium_script directory.
4. Activate the virtual environment for selenium_script by typing the following:

```
(myvenv) Kashermans-MBP:selenium_script kasherman$ source myvenv/bin/activate
```

5. Finally, to run the WebDriver script, run the following in the selenium_script directory:

```
(myvenv) Kashermans-MBP:selenium_script kasherman$ nosetests ocvtest.py
```

6. It will open a browser window, maximize the window, and run the tests in the selenium_script python module, which we call ocvtest.py. It will look as if we ran the Selenium IDE on the web browser.
7. Upon completion, we will see:

```
(myvenv) Kashermans-MBP:selenium_script kasherman$ nosetests ocvtest.py
.....
----------------------------------------------------------------------
Ran 5 tests in 22.946s
```

Bibliography

Django Software Foundation. (2015). *Writing and running tests*. Retrieved April 2015, from Django Documentation: https://docs.djangoproject.com/en/1.8/topics/testing/tools/#django.test.TestCase

Logilab. (2014). *Pylint*. Retrieved April 2015, from Pylint: http://www.pylint.org/

Python Software Foundation. (2015, March 24). *26.3. unittest — Unit testing framework*. Retrieved April 2015, from Python: https://docs.python.org/3/library/unittest.html

Selenium Project. (2015). *Selenium IDE*. Retrieved April 2015, from SeleniumHQ: Browser Automation: http://www.seleniumhq.org/projects/ide/

Selenium Project. (2015, May 11). *Selenium WebDriver Documentation*. Retrieved April 2015, from SeleniumHQ: Browser Automation: http://docs.seleniumhq.org/docs/03_webdriver.jsp