

Topic 12: Testing

CITS3403 Agile Web Development

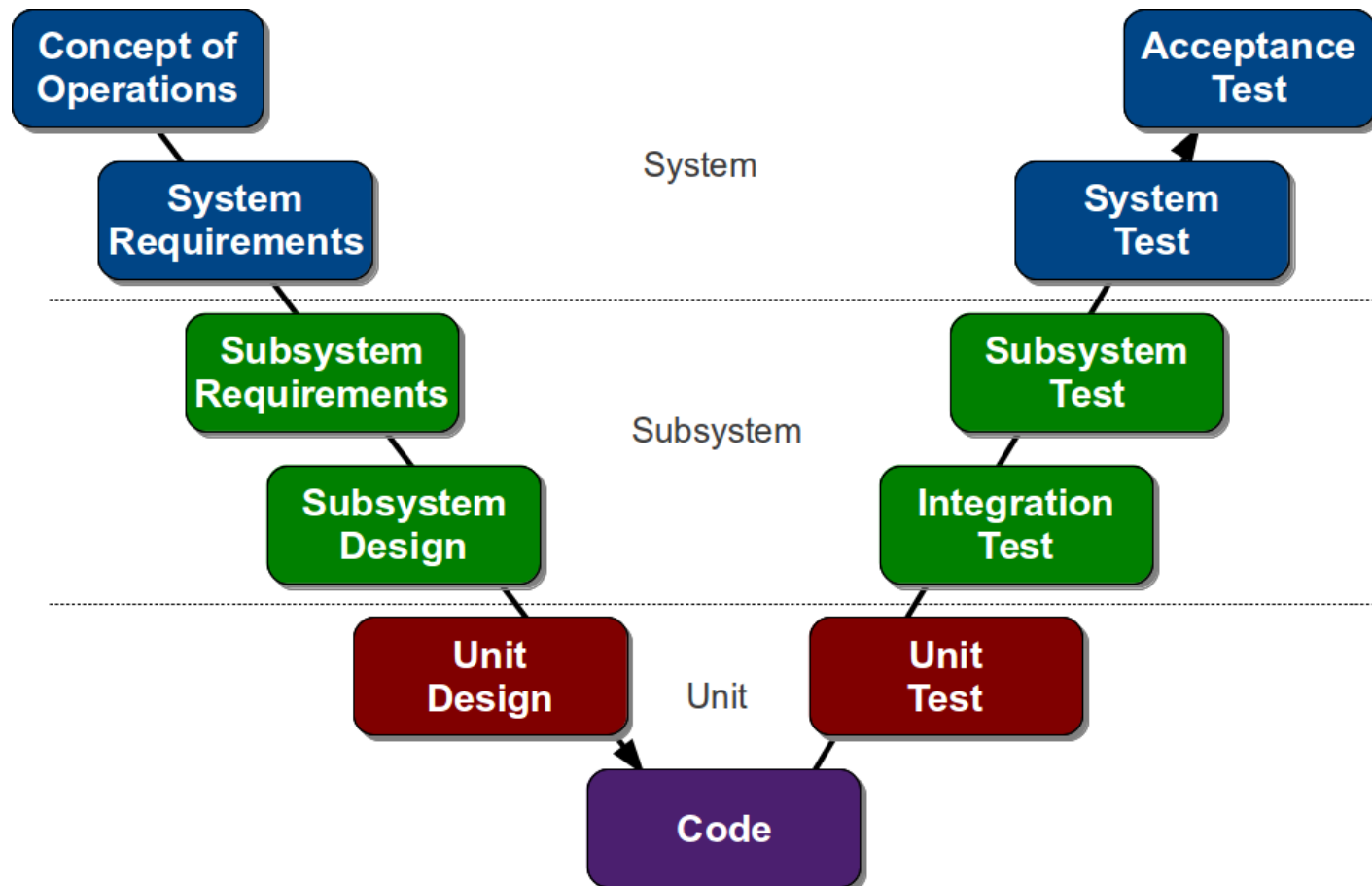
Getting MEAN with Mongo,
Express, Angular and Node,
Chapter 11

Semester 1, 2019

- Writing a bug free application is critical to the success of that application.
- There are various ways to eliminate bugs.
 - Code inspections: having peers critically examine your code and make suggestions.
 - Formal verification: building precise specifications of correctness, and proving the code meets these specs.
 - Testing: Providing test cases of inputs and actions, and expected behaviors.
- Testing is a key activity in any software development, but particularly in agile development, where the test suites are a proxy for requirements documentation.
- *Test Driven Design* specifies that the tests should be written first, and the code designed specifically to pass those tests.
- Agile also relies heavily on test automation, so that every sprint or iteration can be checked against the existing test suite.

The V-model

- The V-model links types of tests to stages in the development process.
- We will focus on unit tests and system tests.



Types of test

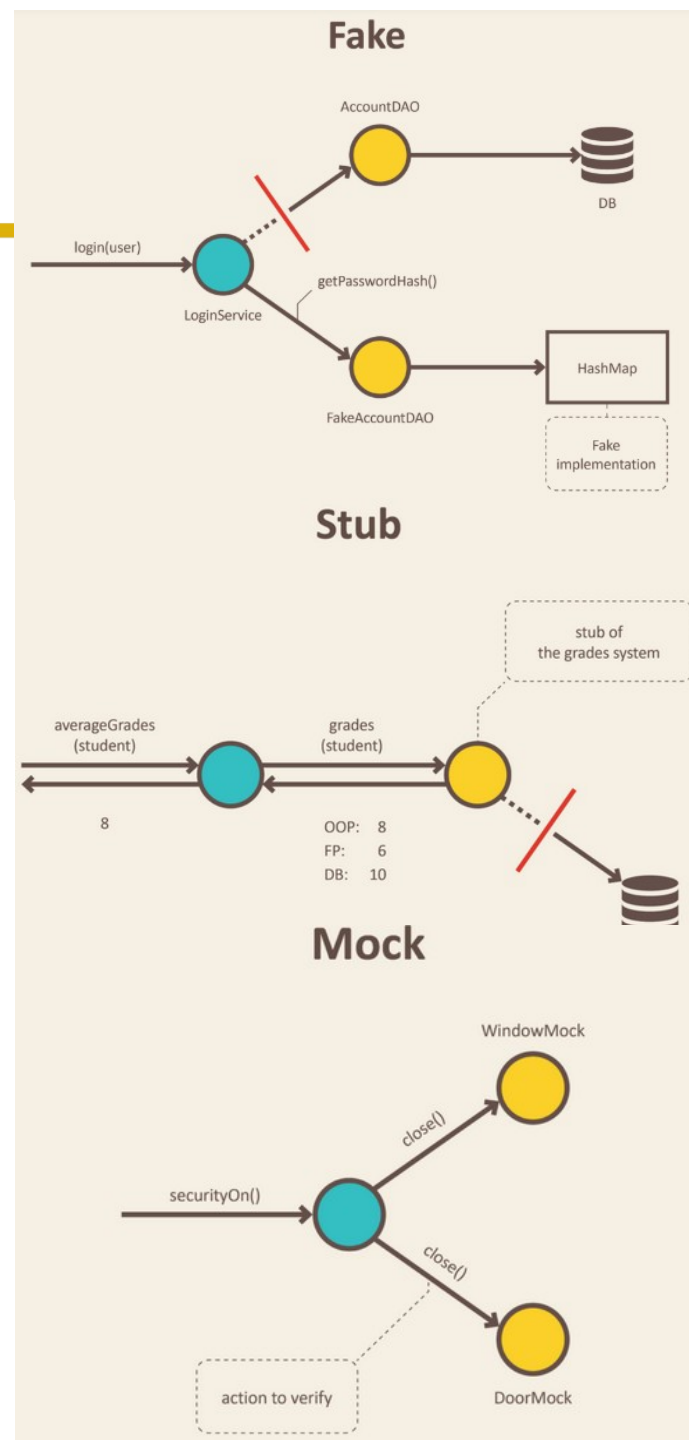
- Unit Tests: test each individual function for to ensure it behaves correctly (2-5 tests per function)
- Integration Test: Execute each scenario to make sure modules integrate correctly.
- System Test: Integrate real hardware platforms.
- Acceptance Test: Run through complete user scenarios via the user interface.

The tests should be repeatable, and should have a clear scope. Any changes to anything outside that scope should not affect whether the test passes.

To isolate the *system under test* (SUT) from external systems, we use test doubles: fakes, stubs and mocks.

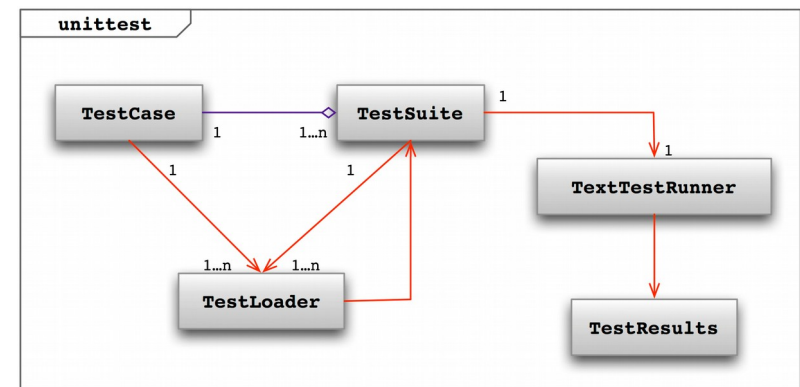
Test Doubles

- Fakes are objects with working implementations, but not the same as the production environment. In the diagram, the full database has been replaced by an object wrapping a hashmap.
- A stub is an object that holds predefined data to respond to specific requests. For example, To test the login GUI, we could provide a stub that accepts only the password 'pw' regardless of the user.
- Mocks work like stubs but they register the calls they receive, so we can assert that the correct action was performed, or the correct message was sent. In the example, a door mock is used to verify that the close() method was called, without interacting with hardware.



Unit Test Structure

- In python, unit testing is most commonly done with the module unittest.
- This provides a number of classes and functions;
 - Test fixtures: These are the methods to prepare for a test case, called `setUp` and `tearDown`.
 - TestCase: This is the standard class for running a test. It specifies the `setUp`, `tearDown`, and a number of functions to execute.
 - TestSuite: Running comprehensive tests is expensive, so often you don't want to run every test case. Test suites allow test cases to be grouped together to be run at once.
 - Test Runners: These run the tests and report the results
- Typically you only have to write the test cases, and the rest is automatic.



Writing some simple tests:

- To write some basic unit tests, we should import unittest, and the modules/classes under test.
- We then subclass `TestCase` for each unit we want to test.
- We specify the `setUp` for each test (e.g. populating a dummy database, or creating instances), and the `tearDown` after each test (e.g. resetting the database).
- Flask has a method `test_client()` to run a sandboxed version of the app.
- We then specify a set of tests. These must begin with 'test', and use the assert methods to define whether the test passes

`python unittest <filename>`

```
1 import unittest, os
2 from app import app, db
3 from app.models import Student, Project, Lab
4
5 class StudentModelCase(unittest.TestCase):
6
7     def setUp(self):
8         basedir = os.path.abspath(os.path.dirname(__file__))
9         app.config['SQLALCHEMY_DATABASE_URI'] = \
10             'sqlite:///'+os.path.join(basedir, 'test.db')
11         self.app = app.test_client() # creates a virtual test environment
12         db.create_all()
13         s1 = Student(id='00000000', first_name='Test', surname='Case', cits3403=True)
14         s2 = Student(id='11111111', first_name='Unit', surname='Test', cits3403=True)
15         lab = Lab(lab='test-lab', time='now')
16         db.session.add(s1)
17         db.session.add(s2)
18         db.session.add(lab)
19         db.session.commit()
20
21     def tearDown(self):
22         db.session.remove()
23         db.drop_all()
24
25     def test_password_hashing(self):
26         s = Student.query.get('00000000')
27         s.set_password('test')
28         self.assertFalse(s.check_password('case'))
29         self.assertTrue(s.check_password('test'))
30
31     def test_is_committed(self):
32         s = Student.query.get('00000000')
33         self.assertFalse(s.is_committed())
34         s2 = Student.query.get('11111111')
35         lab = Lab.query.first()
36         p = Project(description='test', lab_id=lab.lab_id)
37         db.session.add(p)
38         db.session.flush()
39         s.project_id = p.project_id
40         s2.project_id = p.project_id
41         db.session.commit()
42         self.assertTrue(s.is_committed())
43
44 if __name__ == '__main__':
45     unittest.main()
```

tests/unittest.py [+] 10,9 Top

```
(virtual-environment) drtnf@drtnf-ThinkPad:~$ python3 -W ignore -m tests.unittest
test_is_committed (__main__.StudentModelCase) ... ok
test_password_hashing (__main__.StudentModelCase) ... ok
.....
Ran 2 tests in 0.581s
OK
```

Assertions:

- Assertions describe the checks the test performs. They can be supplemented with messages to give diagnostic information about the failing cases.
- Each test can have multiple assertions, and the test only passes if every assertion is true.
- We can also assert that an exception or a warning is raised. If the exception is raised, then the test passes.
- There are many other assertion libraries that can be imported and produce more readable test cases,

```
from assertpy import assert_that

def test_something():
    assert_that(1 + 2).is_equal_to(3)
    assert_that('foobar').is_length(6).starts_with('foo').ends_with('bar')
    assert_that(['a', 'b', 'c']).contains('a').does_not_contain('x')
```

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Method	Checks that	New in
<code>assertRaises(exc, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <code>exc</code>	
<code>assertRaisesRegex(exc, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <code>exc</code> and the message matches regex <code>r</code>	3.1
<code>assertWarns(warn, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <code>warn</code>	3.2
<code>assertWarnsRegex(warn, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <code>warn</code> and the message matches regex <code>r</code>	3.2
<code>assertLogs(logger, level)</code>	The <code>with</code> block logs on <code>logger</code> with minimum <code>level</code>	3.4

Code Coverage:

- Testing is essential for reliable software, and we would like to have a set of test cases, where any code that passes the test “works”
- This means that any line of code that does not feature in at least one test case is redundant to your notion of “works”.
- There are different ways of measuring coverage: statement coverage, branch coverage, logic coverage, path coverage. Statement coverage is sufficient for our purposes, but you should always consider the ways your tests may be deficient.
- Coverage can be automatically measured by such tools as Coverage.py, and HtmlTestRunner can be used to give visual feedback on a test run.

Coverage report: 37.59%

Module \	statements	missing	excluded	branches	partial	coverage
cogapp/__init__.py	2	0	0	0	0	100.00%
cogapp/__main__.py	3	3	0	0	0	0.00%
cogapp/backward.py	19	8	0	2	1	57.14%
cogapp/cogapp.py	427	197	4	176	26	47.10%
cogapp/makefiles.py	28	20	3	14	0	19.05%
cogapp/test_cogapp.py	704	486	6	6	0	30.99%
cogapp/test_makefiles.py	55	55	0	6	0	0.00%
cogapp/test_whiteutils.py	69	69	0	0	0	0.00%
cogapp/whiteutils.py	45	3	0	32	3	92.21%
Total	1352	841	13	236	30	37.59%

coverage.py v4.4.2, created at 2017-11-05 07:56

Test Result

Start Time: 2018-08-19 19:57:03

Duration: 0:00:00

Status: Pass: 1, Fail: 1

MyTestExample.MyTestExample

Status

test_function_two (MyTestExample.MyTestExample)

Pass

test_function_one (MyTestExample.MyTestExample)

Fail

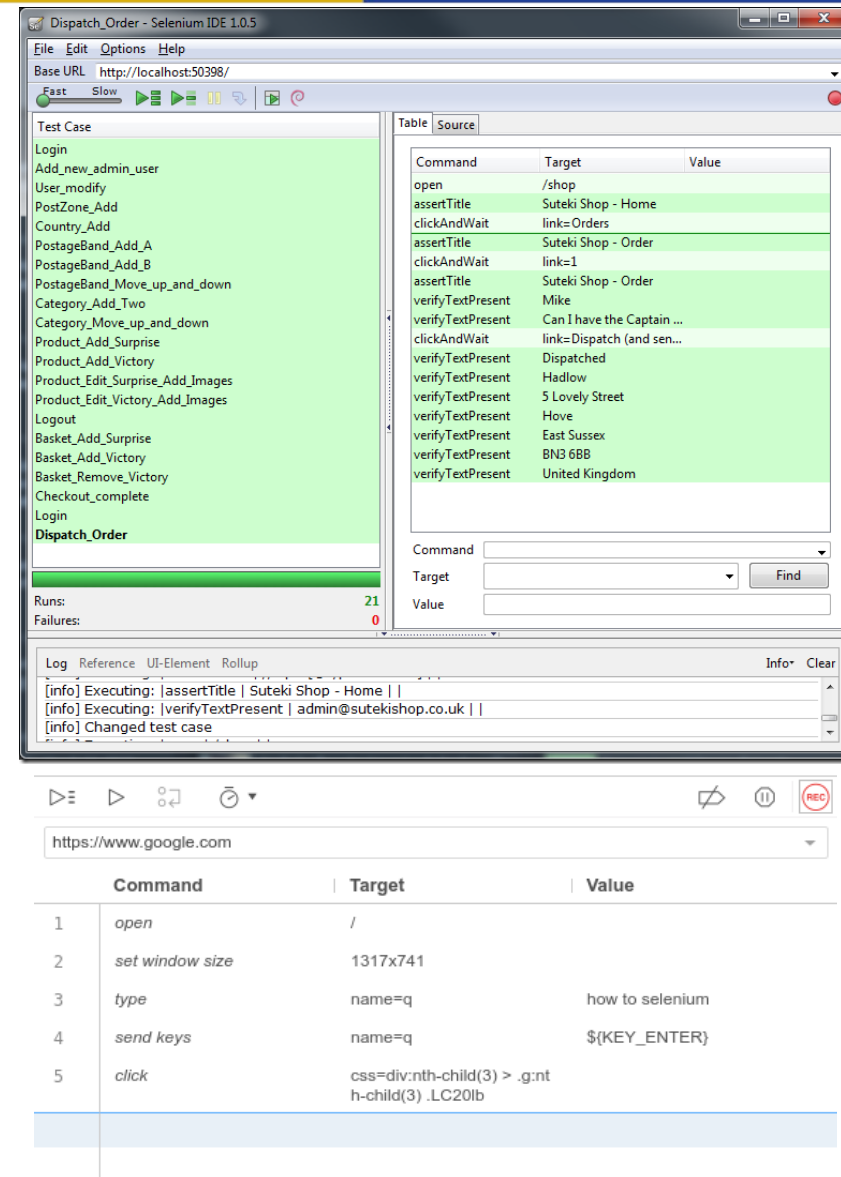
View

Total Test Runned:

Pass: 1, Fail: 1

System/User Tests

- User testing is more challenging since it depends on the end user environment.
- Selenium can be used to automate browsers to run test cases.
- PhantomJS is a headless browser that can be used for testing with Selenium, without the overhead of running a GUI.
- Selenium has two variations:
SeleniumIDE is a browser plugin that can record interactions with a web-site and run them back to confirm the outcome remains the same.
- Selenium WebDriver provides a set of tools for scripting User tests.



Dispatch_Order - Selenium IDE 1.0.5

Base URL: <http://localhost:50398/>

Test Case

- Login
- Add_new_admin_user
- User_modify
- PostZone_Add
- Country_Add
- PostageBand_Add_A
- PostageBand_Add_B
- PostageBand_Move_up_and_down
- Category_Add_Two
- Category_Move_up_and_down
- Product_Add_Surprise
- Product_Add_Victory
- Product_Edit_Surprise_Add_Images
- Product_Edit_Victory_Add_Images
- Logout
- Basket_Add_Surprise
- Basket_Add_Victory
- Basket_Remove_Victory
- Checkout_complete
- Login
- Dispatch_Order**

Runs: 21
Failures: 0

Command	Target	Value
open	/shop	
assertTitle	Suteki Shop - Home	
clickAndWait	link= Orders	
assertTitle	Suteki Shop - Order	
clickAndWait	link=1	
assertTitle	Suteki Shop - Order	
verifyTextPresent	Mike	
verifyTextPresent	Can I have the Captain ...	
clickAndWait	link= Dispatch (and sen...	
verifyTextPresent	Dispatched	
verifyTextPresent	Hadlow	
verifyTextPresent	5 Lovely Street	
verifyTextPresent	Hove	
verifyTextPresent	East Sussex	
verifyTextPresent	BN3 6BB	
verifyTextPresent	United Kingdom	

Log Reference UI-Element Rollup

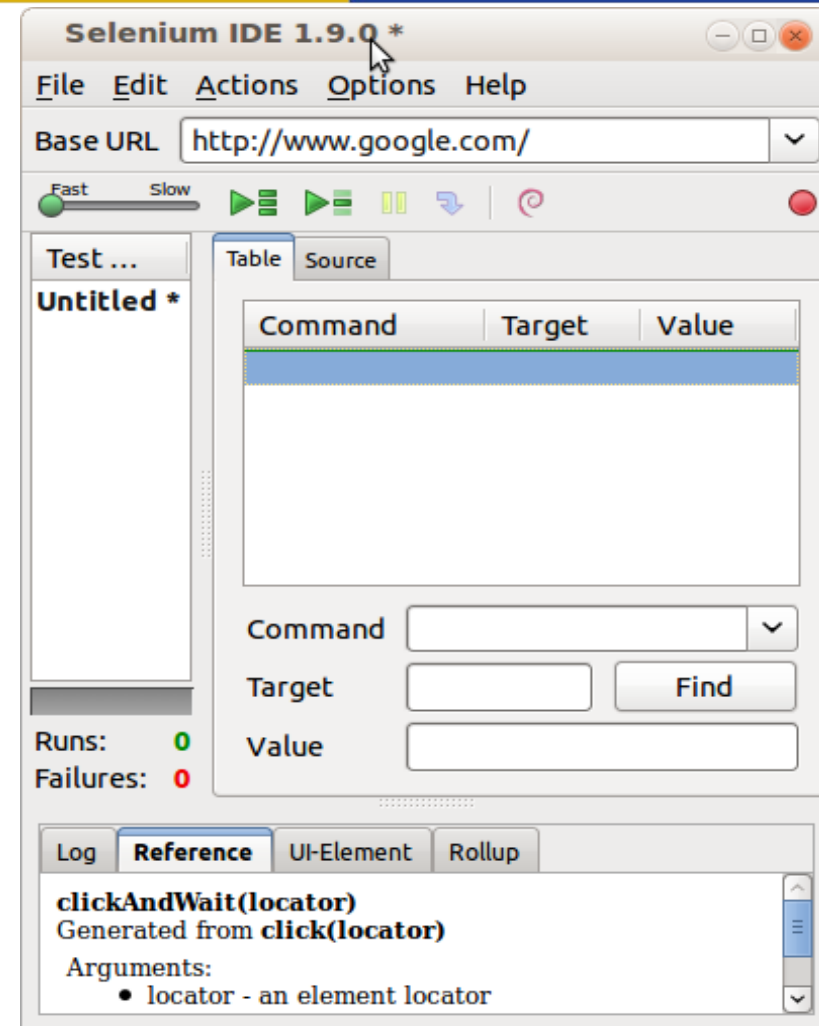
[info] Executing: |assertTitle | Suteki Shop - Home | |
[info] Executing: |verifyTextPresent | admin@sutekishop.co.uk | |
[info] Changed test case

https://www.google.com

	Command	Target	Value
1	open	/	
2	set window size	1317x741	
3	type	name=q	how to selenium
4	send keys	name=q	#{KEY_ENTER}
5	click	css=div:nth-child(3) > .g:nt h-child(3) .LC20lb	

Selenium IDE

- Firefox/Chrome extension
- Easy record and replay
- Debug and set breakpoints
- Save tests in HTML, WebDriver and other Formats.
- Selenium saves all information in an HTML table format
- Each record consists of:
 - **Command** – tells Selenium what to do (e.g. “open”, “type”, “click”, “verifyText”)
 - **Target** – tells Selenium which HTML element a command refers to (e.g. textbox, header, table)
 - **Value** – used for any command that might need a value of some kind (e.g. type something into a textbox)



How to record/replay with Selenium IDE

1. Start recording in Selenium IDE
2. Execute scenario on running web application
3. Stop recording in Selenium IDE
4. Verify / Add assertions
5. Replay the test.



... or using webdriver you can integrate selenium with any unit testing scripting language.

You can test functionality, responsiveness and general usability.

Selenium WebDriver

- Selenium IDE is good for quickly prototyping tests, but is not very good for maintaining tests.
- You can't apply test fixtures easily and, you need a running instance of the application.
- WebDriver provides a set of python classes for interaction with a browser.
- We require a driver executable for each browser we wish to test (Firefox, Chrome, Edge, PhantomJS).
- The executable needs to be in the path, or the current directory
- We also need to set up our doubles. We want a clean database for testing, so we really need flask to be running in testing configuration.

```
1 import os
2 basedir = os.path.abspath(os.path.dirname(__file__))
3
4 class Config(object):
5     SECRET_KEY = os.environ.get('SECRET_KEY') or 'ssh!'
6     SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or '
    sqlite:///'+os.path.join(basedir, 'app.db')
7     SQLALCHEMY_TRACK_MODIFICATIONS = False
8
9 class ProductionConfig(Config):
10     SECRET_KEY = os.environ.get('SECRET_KEY')
11     # SQLALCHEMY_DATABASE_URI = Postgres remote
12
13 class DevelopmentConfig(Config):
14     DEBUG=True
15
16 class TestingConfig(Config):
17     SQLALCHEMY_DATABASE_URI = 'sqlite:///'+os.path.join(basedir, '
    tests/test.db')
18     #SQLALCHEMY_DATABASE_URI = 'sqlite:///memory:' #in memory da
    tabase
```

config.py

1,1

All

```
1 from flask import Flask
2 from config import Config
3 from flask_sqlalchemy import SQLAlchemy
4 from flask_migrate import Migrate
5 from flask_login import LoginManager
6
7 app = Flask(__name__)
8 app.config.from_object('config.TestingConfig')
9 db = SQLAlchemy(app)
10 migrate = Migrate(app, db)
11 login = LoginManager(app)
12 login.login_view = 'login'
13
14 from app import routes, models
```

app/__init__.py

1,1

Running Selenium Tests

- In the TestingConfig, we have a new database, test.db, that always starts empty so our tests are repeatable.
- The tests can be run by unittest, even though they're not unit tests.
- In our setUp we create a selenium web driver for Firefox, enter dummy data to our databases, and navigate to the app's page.
- The test is executed by describing the interactions selenium has with the web page.
- This is specified via DOM elements, and selenium offers different methods for simulating events.

```
1 import unittest, os, time
2 from app import app, db
3 from app.models import Student, Project, Lab
4 from selenium import webdriver
5
6 #To do, find simple way for switching from test context to development to production.
7
8
9 class SystemTest(unittest.TestCase):
10     driver = None
11
12     def setUp(self):
13         self.driver = webdriver.Firefox(executable_path=r'/home/drtnf/Dropbox/Tim/teaching/2019/CITS3403/pair-up/geckodriver')
14
15         if not self.driver:
16             self.skipTest('Web browser not available')
17         else:
18             db.init_app(app)
19             db.create_all()
20             s1 = Student(id='22222222', first_name='Test', surname='Case', cits3403=True)
21             s2 = Student(id='11111111', first_name='Unit', surname='Test', cits3403=True)
22             lab = Lab(lab='test-lab', time='now')
23             db.session.add(s1)
24             db.session.add(s2)
25             db.session.add(lab)
26             db.session.commit()
27             self.driver.maximize_window()
28             self.driver.get('http://localhost:5000/')
29
30     def tearDown(self):
31         if self.driver:
32             self.driver.close()
33             db.session.query(Student).delete()
34             db.session.query(Project).delete()
35             db.session.query(Lab).delete()
36             db.session.commit()
37             db.session.remove()
38
39     def test_register(self):
40         s = Student.query.get('22222222')
41         self.assertEqual(s.first_name, 'Test', msg='student exists in db')
42         self.driver.get('http://localhost:5000/register')
43         self.driver.implicitly_wait(5)
44         num_field = self.driver.find_element_by_id('student_number')
45         num_field.send_keys('22222222')
46         pref_name = self.driver.find_element_by_id('preferred_name')
47         pref_name.send_keys('Testy')
48         new_pin = self.driver.find_element_by_id('new_pin')
49         new_pin.send_keys('0000')
50         new_pin2 = self.driver.find_element_by_id('new_pin2')
51         new_pin2.send_keys('0000')
52         time.sleep(1)
53         self.driver.implicitly_wait(5)
54         submit = self.driver.find_element_by_id('submit')
55         submit.click()
56         #check login success
57         self.driver.implicitly_wait(5)
58         time.sleep(1)
59         logout = self.driver.find_element_by_partial_link_text('Logout')
60         self.assertEqual(logout.get_attribute('innerHTML'), 'Logout Testy', msg='Logged in')
61
62
63 if __name__ == '__main__':
64     unittest.main(verbosity=2)
```

tests/systemtest.py

Navigating with Selenium

- You need to design your web page so that all elements are accessible. And have a fixed id, so the tests are robust if the page layout changes.
- Selenium can enter information in forms, click on elements and drag and drop etc
- You can extract information by searching for text or accessing the attributes of HTML elements.
- An standard assertion library can be used to confirm that the page behaved as expected.

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element_by_name('name'))
select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```

```
element = driver.find_element_by_name("source")
target = driver.find_element_by_name("target")
```

```
from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target).perform()
```

- *find_element_by_id*
- *find_element_by_name*
- *find_element_by_xpath*
- *find_element_by_link_text*
- *find_element_by_partial_link_text*
- *find_element_by_tag_name*
- *find_element_by_class_name*
- *find_element_by_css_selector*

```
continue_link = driver.find_element_by_link_text('Continue')
continue_link = driver.find_element_by_partial_link_text('Conti')
```

Running the Selenium Tests

- To run the Selenium tests, you need to have the flask app running in TestingConfig.
- You execute the tests as with unittest
`python -m tests.systemtest`
- You will see the browser launch and the GUI execute the set of specified steps, and you'll receive a report of the tests as you do for unittest

```
OK
(virtual-environment) drtnf@drtnf-ThinkPad:~$ python3 -W ignore -m tests.systemtest
test_register (__main__.SystemTest) ... 127.0.0.1 - - [01/May/2019 13:21:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [01/May/2019 13:21:59] "GET /static/bootstrap.min.css HTTP/1.1" 200 -
127.0.0.1 - - [01/May/2019 13:21:59] "GET /static/bootstrap.min.js HTTP/1.1" 200 -
127.0.0.1 - - [01/May/2019 13:21:59] "GET /static/bootstrap-theme.min.css HTTP/1.1" 200 -
127.0.0.1 - - [01/May/2019 13:22:00] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [01/May/2019 13:22:00] "GET /register HTTP/1.1" 200 -
127.0.0.1 - - [01/May/2019 13:22:01] "POST /register HTTP/1.1" 302 -
127.0.0.1 - - [01/May/2019 13:22:01] "GET /index HTTP/1.1" 200 -
ok
-----
Ran 1 test in 11.748s
OK
```

Pair Up!

CITS3403 group allocation tool, and I

Register

Student Number

Preferred Name

New Pin

Confirm Pin

Written by Tim, 201