THE UNIVERSITY OF WESTERN AUSTRALIA

Topic 12: Testing

CITS3403 Agile Web Development

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

Semester 1, 2023

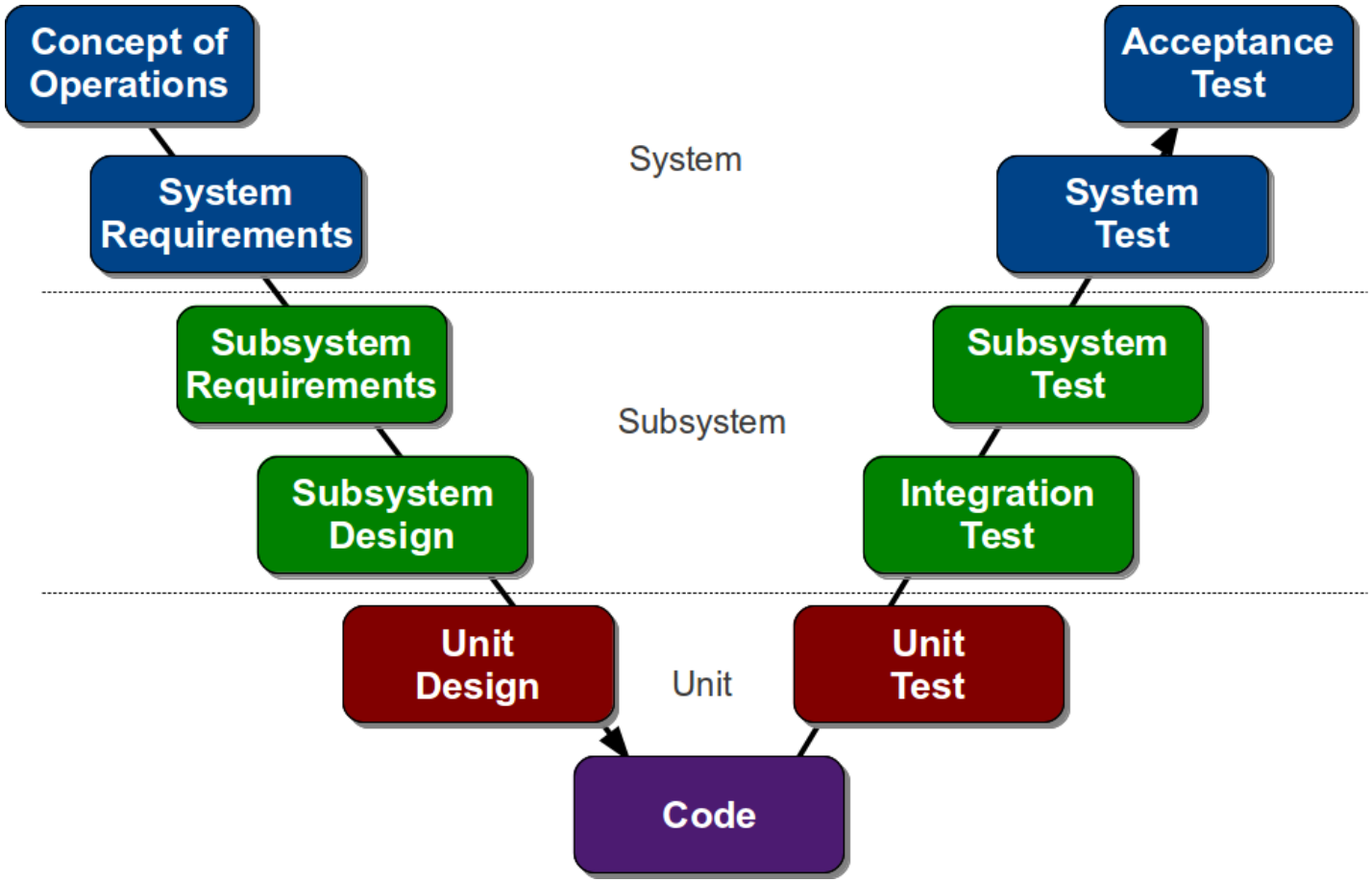
Verification and Validation

THE UNIVERSITY OF WESTERN AUSTRALIA

- Writing a bug free application is critical to to 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

THE UNIVERSITY OF WESTERN AUSTRALIA

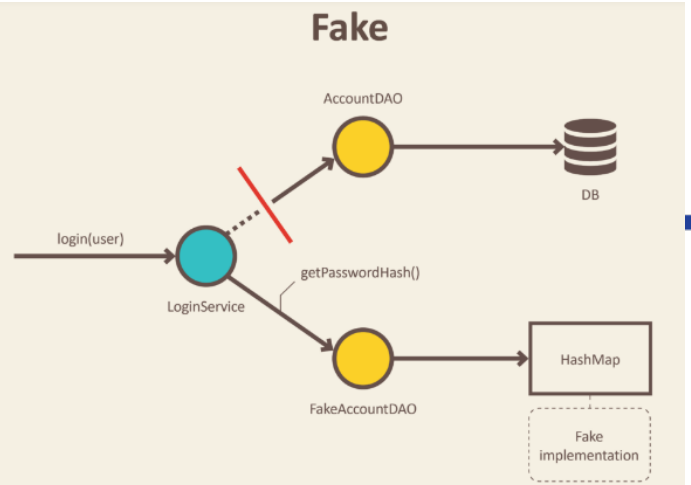
- Unit Tests: test each individual function for to ensure it behaves correctly (2-5 tests per function)
- Integration Test: Execute each scenario to make sure modules integrate correctly.
- System Test: Integrate real hardware platforms.
- Acceptance Test: Run through complete user scenarios via the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

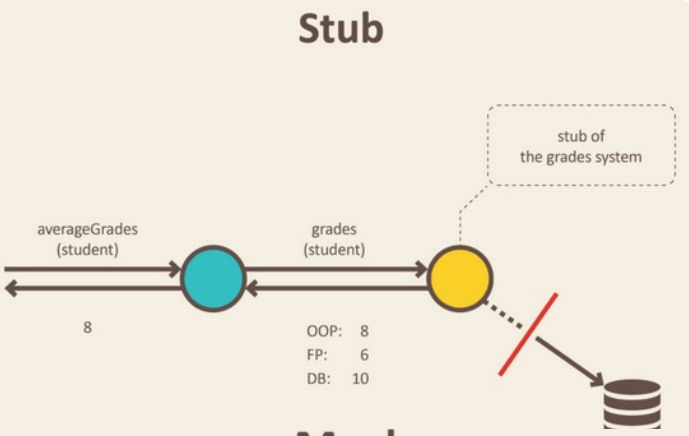
The tests should be repeatable, and should have a clear scope. Any changes to anything outside that scope should not affect whether the test passes.

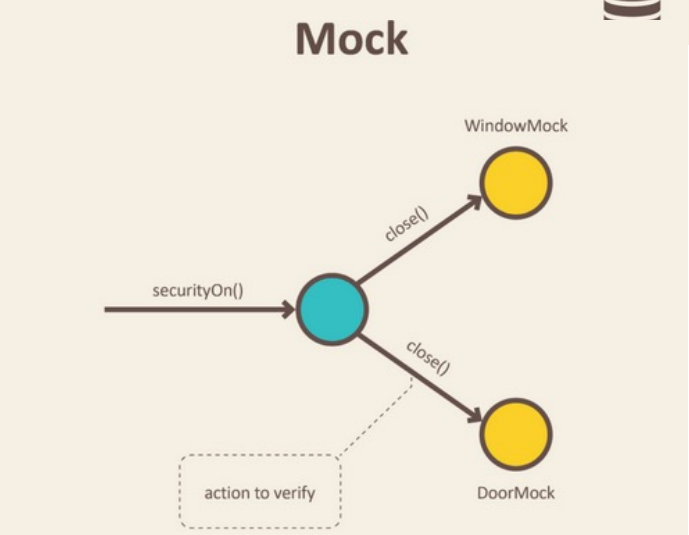
To isolate the *system under test* (SUT) from external systems, we use test doubles: fakes, stubs and mocks.

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() methoid was called, without interacting with hardware.

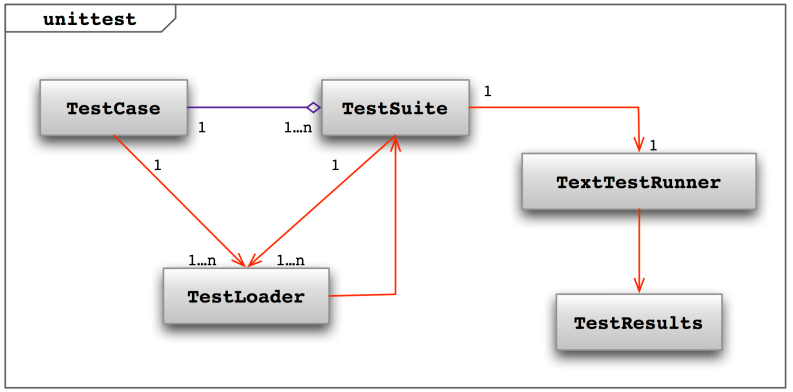






Unit Test Structure

- 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 reuslts
- 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__':
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,

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

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

```
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')
```

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

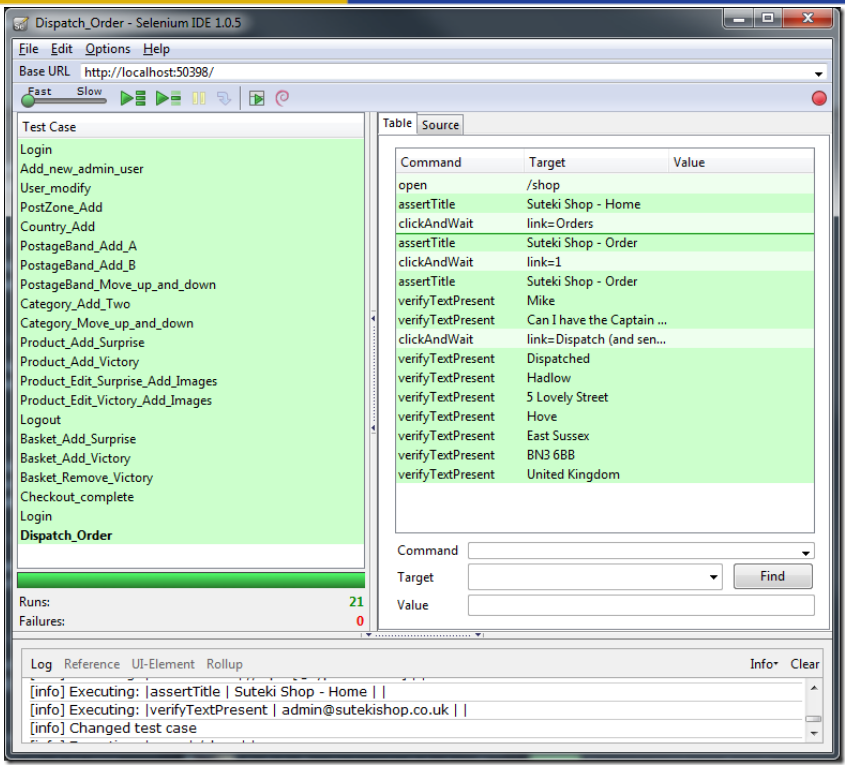
Duration: 0:00:00

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

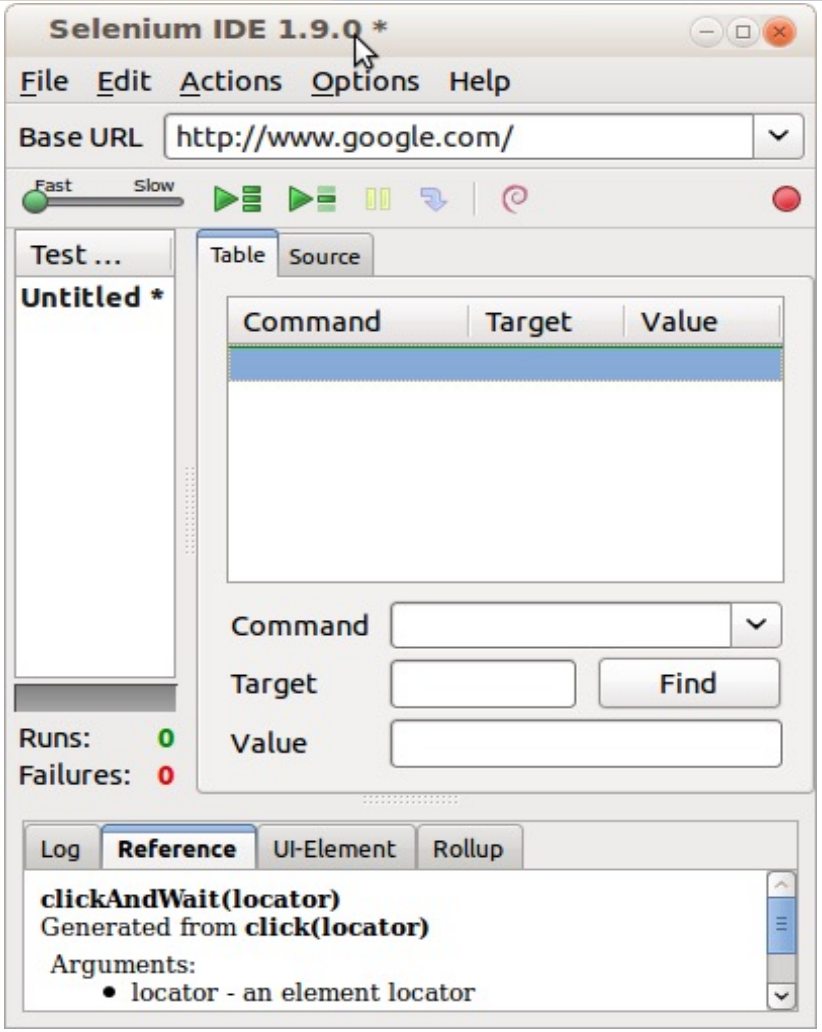
System/User Tests

- 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 Selenuim, 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.



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

THE UNIVERSITY OF WESTERN AUSTRALIA

1. Start recording in Selenium IDE

1. Execute scenario on running web application

1. Stop recording in Selenium IDE

1. Verify / Add assertions

1. 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

THE UNIVERSITY OF WESTERN AUSTRALIA

- 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 interactioning 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.

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'sshh!'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or '
sqlite:///'+os.path.join(basedir,'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

class ProductionConfig(Config):
    SECRET_KEY = os.environ.get('SECRET_KEY')
#   SQLALCHEMY_DATABASE_URI = Postgres remote

class DevelopmentConfig(Config):
    DEBUG=True

class TestingConfig(Config):
    SQLALCHEMY_DATABASE_URI = 'sqlite:///'+os.path.join(basedir,'
tests/test.db')
    #SQLALCHEMY_DATABASE_URI = 'sqlite:///:memory:' #in memory da
tabase
~
config.py                                   1,1           All
```

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_login import LoginManager

app = Flask(__name__)
app.config.from_object('config.TestingConfig')
db = SQLAlchemy(app)
migrate = Migrate(app, db)
login = LoginManager(app)
login.login_view = 'login'

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.

```python
import unittest, os, time
from app import app, db
from app.models import Student, Project, Lab
from selenium import webdriver

#To do, find simple way for switching from test context to development to production.


class SystemTest(unittest.TestCase):
  driver = None

  def setUp(self):
    self.driver = webdriver.Firefox(executable_path=r'/home/drtnf/Dropbox/Tim/teaching/2019/CITS3403/pair-up/geckodriver')

    if not self.driver:
      self.skipTest('Web browser not available')
    else:
      db.init_app(app)
      db.create_all()
      s1 = Student(id='22222222',first_name='Test',surname='Case',cits3403=True)
      s2 = Student(id='11111111',first_name='Unit',surname='Test',cits3403=True)
      lab = Lab(lab='test-lab',time='now')
      db.session.add(s1)
      db.session.add(s2)
      db.session.add(lab)
      db.session.commit()
      self.driver.maximize_window()
      self.driver.get('http://localhost:5000/')

  def tearDown(self):
    if self.driver:
      self.driver.close()
      db.session.query(Student).delete()
      db.session.query(Project).delete()
      db.session.query(Lab).delete()
      db.session.commit()
      db.session.remove()

  def test_register(self):
    s = Student.query.get('22222222')
    self.assertEqual(s.first_name,'Test',msg='student exists in db')
    self.driver.get('http://localhost:5000/register')
    self.driver.implicitly_wait(5)
    num_field = self.driver.find_element_by_id('student_number')
    num_field.send_keys('22222222')
    pref_name = self.driver.find_element_by_id('prefered_name')
    pref_name.send_keys('Testy')
    new_pin = self.driver.find_element_by_id('new_pin')
    new_pin.send_keys('0000')
    new_pin2 = self.driver.find_element_by_id('new_pin2')
    new_pin2.send_keys('0000')
    time.sleep(1)
    self.driver.implicitly_wait(5)
    submit = self.driver.find_element_by_id('submit')
    submit.click()
    #check login success
    self.driver.implicitly_wait(5)
    time.sleep(1)
    logout = self.driver.find_element_by_partial_link_text('Logout')
    self.assertEqual(logout.get_attribute('innerHTML'), 'Logout Testy', msg='Logged in')


if __name__=='__main__':
  unittest.main(verbosity=2)
```

tests/systemtest.py

Navigating with Selenium

THE UNIVERSITY OF WESTERN AUSTRALIA

- You need to design your web pge 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`

```
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
```