Automating Django Functional Tests Using Selenium on Cloud

Jonghyun Park, Lablup Inc.

2016/08/14 (Sun) @PyCon APAC 2016

Contents and target audience

- How to functional tests on Django using Selenium.
- Running functional tests automatically on AWS EC2.
- And some tips on writing functional tests (if time permits).

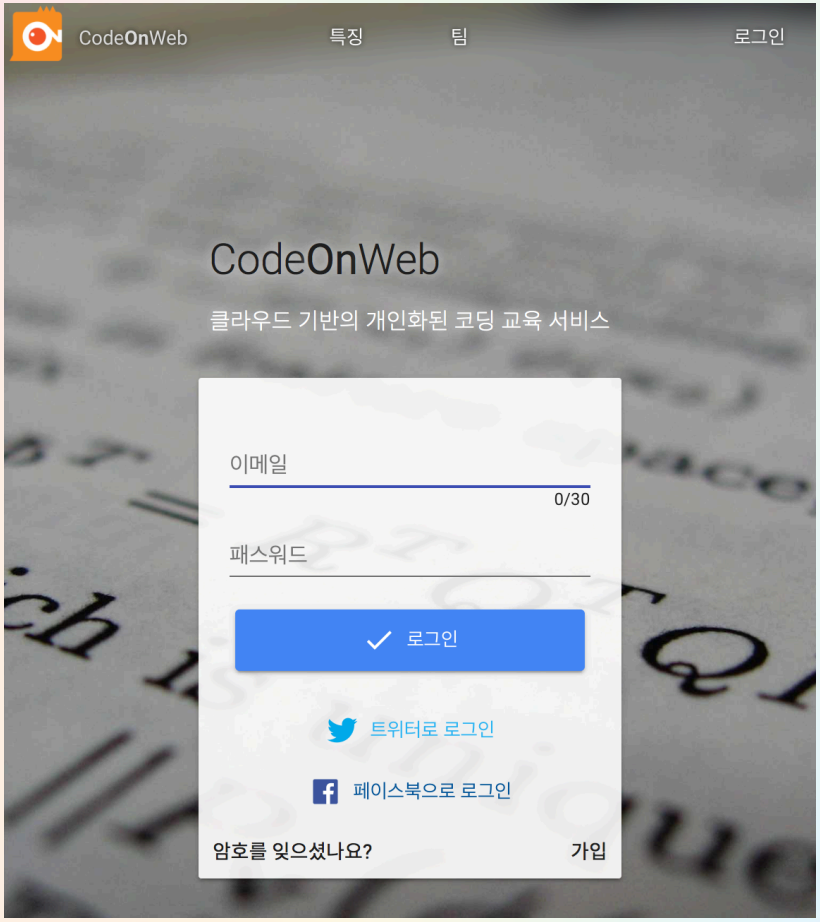
- Target audience
  - Python beginners - intermediates.
  - Who have written some tests in Python/Django
  - ... but not much.

About speaker

- Physics (~4 yr)
- Single-molecule Biophysics (~8 yr)
  - Optical microscope
  - DNA, protein

- IT newbie (~1.2 yr)
  - HTML, CSS, Javascript, Polymer (Google)
  - Python-Django

About speaker



[CodeOnWeb (codeonweb.com)]

Code Test

General remarks about code testing

Tests on CodeOnWeb

Test code

- Code that tests code.

login.py

```
def login(username, password, ...):
    ...
    if success: return True
    else: return False
```

test_login.py

```
def test_login():
    result = login('test', '1' , ...)
    assertTrue(result)
```

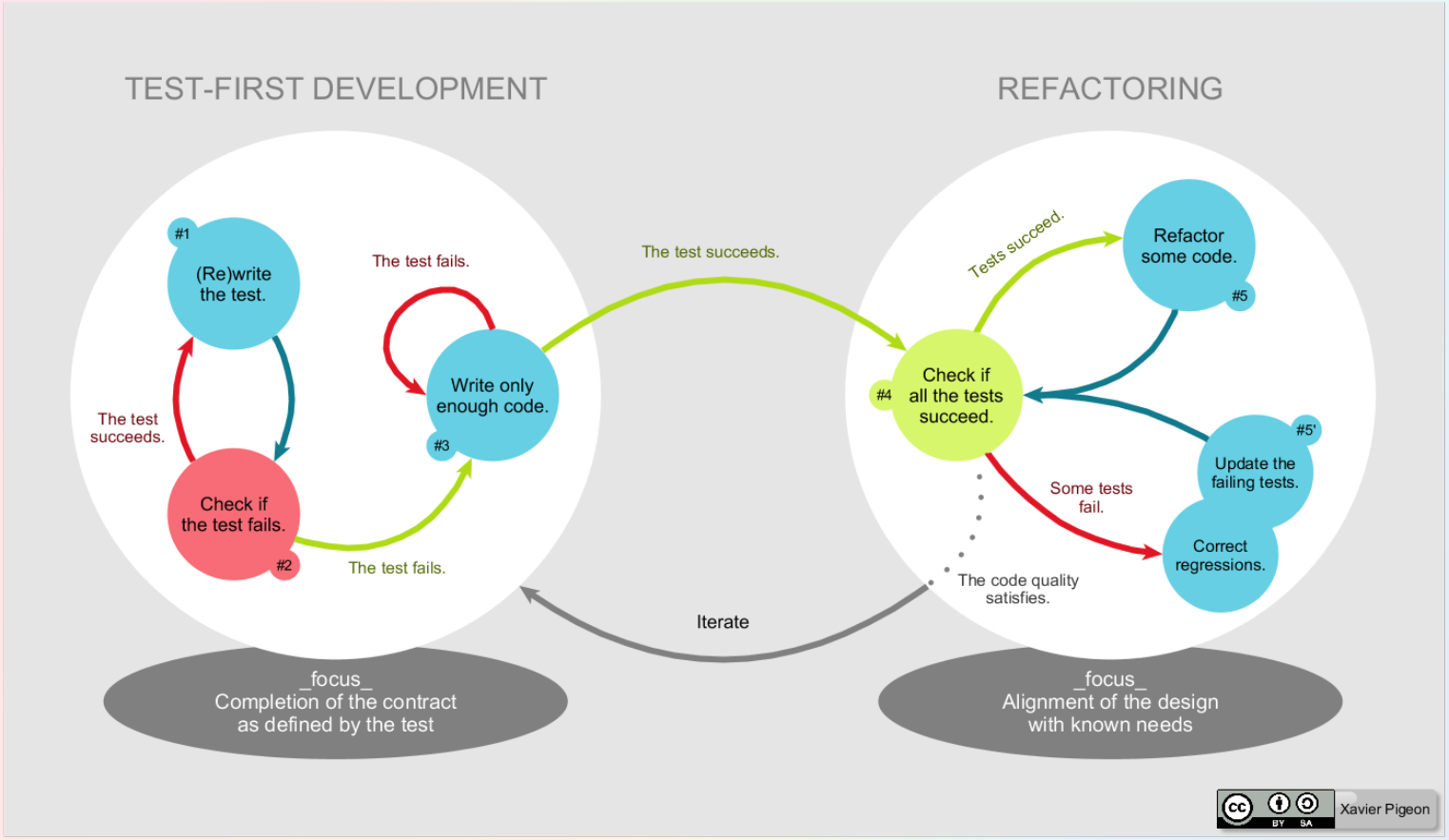
Why tests code?

- How do we know our code is okay even if we are messing around all the parts of the code?

- Ensure **minimal code integrity** when refactoring.
- Man cannot test all features by hand.
- Pre-define requirements for a feature.

Test-Driven Development (TDD)

Some people actually do!



https://goo.gl/IPwERJ

Writing tests is good for newbie

like me

- Can understand the logic of each function/module
- … **w/o destroying real service** whatever you do.
- Defeat bugs & improves codes, if you are lucky.

https://goo.gl/sU2D2M

There are stupid codes, but not stupid test codes!

Kinds of tests for CodeOnWeb

- Unit tests
  - Test of *independent unit of behavior.*
    - e.g., function, class method.
  - Test layer: model / form / view.

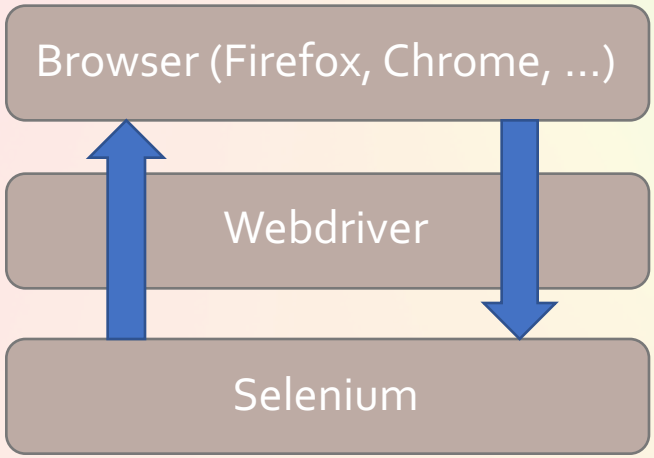
- **Functional tests**
  - Test of *independent piece of functionality*.
  - May test many functions or methods.
  - Test of actual user interaction **on the web browser**.

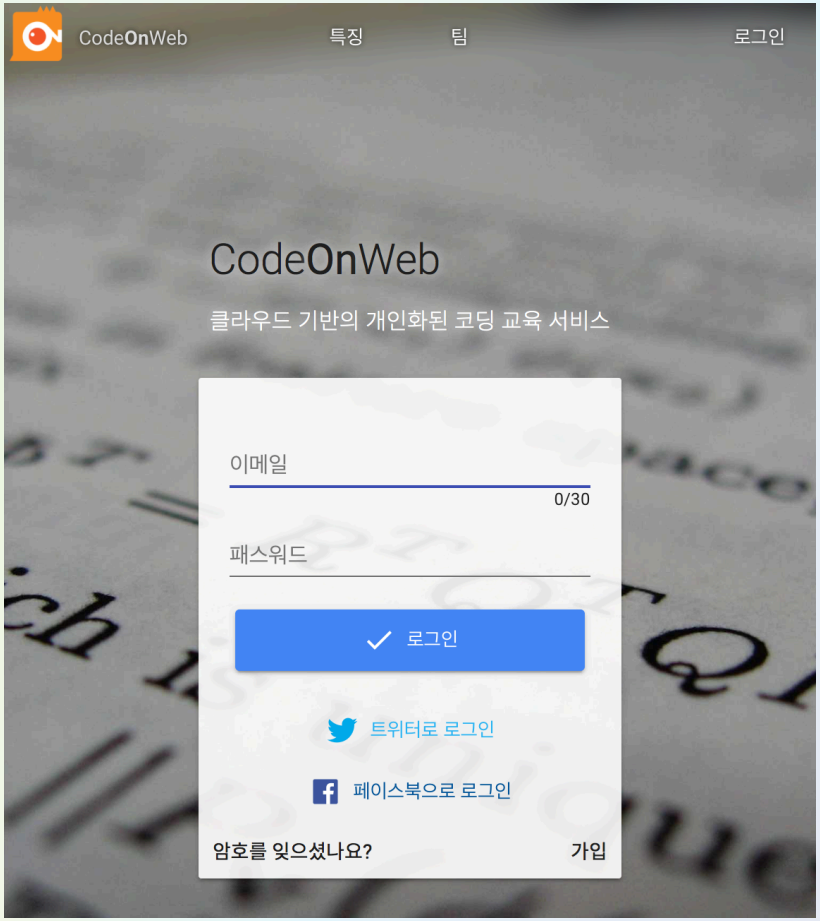
Using Selenium

Functional testing using selenium with Django

Selenium

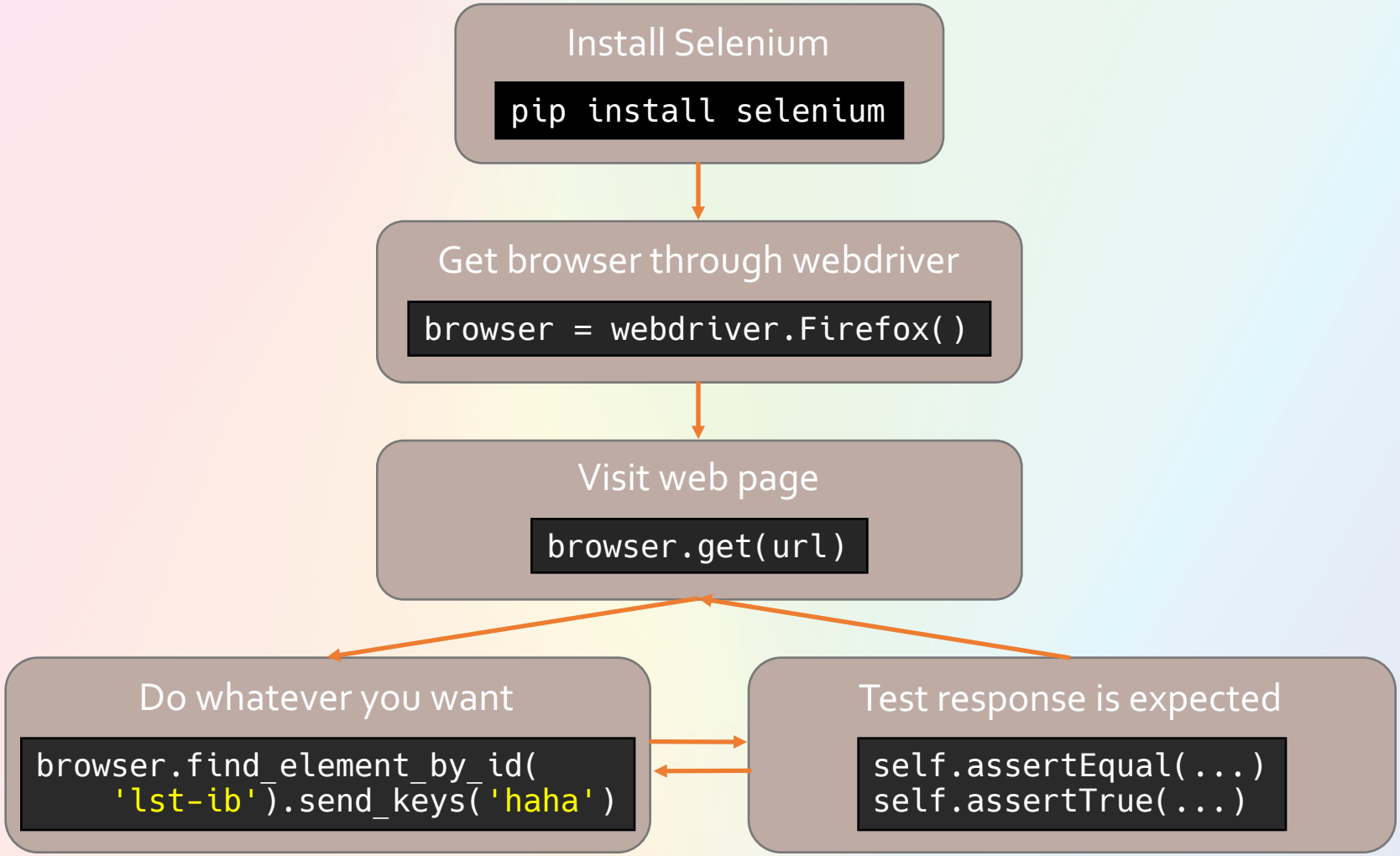
- Browser emulator.
- Automate user interaction on the browser.
- Can be utilized for testing
  … and other things.






```
python manage.py test
functional_tests.test_login_registration
.LoginTest
```

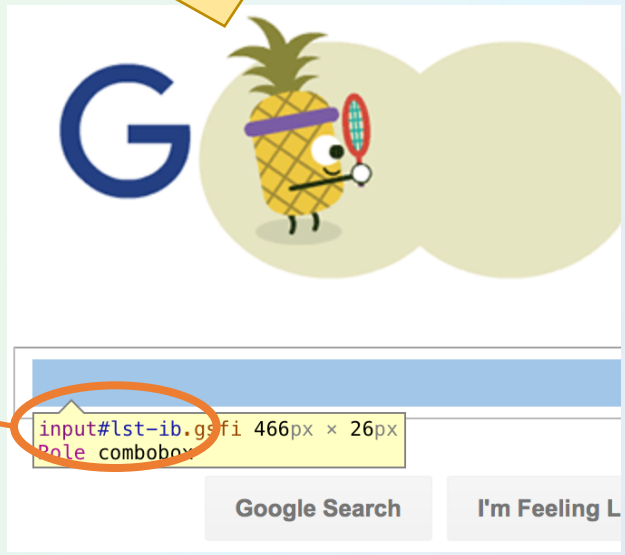
Test scheme using Selenium



Example: automating googling

```python
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

# Get browser through webdriver
browser = webdriver.Firefox()

# Visit Google!
browser.get('https://google.com/')

# Enter keyword and enter
inp = browser.find_element_by_id('lst-ib')
inp.send_keys('PyCon 2016')
inp.send_keys(Keys.RETURN)

# Just wait a little
time.sleep(3)

# Close browser window
browser.quit()
```

This is not a test case!



python functional_tests/test_pycon_google1.py

Convert to Django's test case

```python
import time
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys


class GoogleTest(StaticLiveServerTestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.set_window_size(1024, 768)

    def tearDown(self):
        time.sleep(3)
        self.browser.quit()

    def test_google_search(self):
        self.browser.get('https://google.com/')
        inp = self.browser.find_element_by_id('lst-ib')
        inp.send_keys('PyCon 2016')
        inp.send_keys(Keys.RETURN)
```

live django server
+ collect static files

```
(X)  python functional_tests/test_pycon_google2.py
(O)  python manage.py test functional_tests.test_pycon_google2
```

Finding elements

```
<html>
    <body>
        <form id="loginForm" class="login">
            <input name="username" type="text" />
            <input name="password" type="password" />
            <input name="continue" type="submit" value="Login" />
        </form>
        <a href=”https://google.com/">Go to Google</a>
    </body>
<html>
```

```
browser.find_element_by_id('loginForm')
       .find_element_by_css_selector('input[name="username"]')
       .find_element_by_name('username')
       .find_element_by_class_name('login')
       .find_element_by_tag_name('input')
       .find_element_by_tag_link_text('Go to Google')
       .find_element_by_tag_partial_link_text('Go to')
       .find_element_by_xpath('//form[@id=" loginForm "]')
```

```
browser.find_elements_by_id('loginForm')
...
```

Interacting with elements

```
item = browser.find_element_by_id('entry-item')

item.click()
ui_type = item.get_attribute('ui-type')
browser.execute_script(
    'arguments[0].setAttribute("disabled", "true")',
    item,
)
browser.execute_script(
    'arguments[0].querySelector("paper-menu").selected = arguments[1]',
    item, '2'
)
```

Not that difficult if you know Javascript.

Let's write a login test

```
def test_user_login(self):
    # Store login count of the user
    user = get_user_model().objects.get(username='einstein')
    logincount = user.profile.logincount

    # visit root url ('/')
    self.browser.get(self.live_server_url)

    # Try to login with email and password
    self.browser.find_element_by_name('email').send_keys('einstein@lablup.com')
    self.browser.find_element_by_name('password').send_keys('1')
    self.browser.find_element_by_css_selector('fieldset paper-button').click()

    # Confirm login count is increased by 1
    user = get_user_model().objects.get(username='einstein')
    self.assertEqual(user.profile.logincount, logincount + 1)

    # Check visit correct url
    target_url = self.live_server_url + '/accounts/profile/usertype/'
    self.assertEqual(self.browser.current_url, target_url)
```

```
python manage.py test functional_tests.test_pycon_login
```

Waits in Selenium

Waiting is very important

Selenium, the beauty of waiting

- Wait what?
  - Page loading
  - Elements loading
  - Elements visibility
  - Ajax response
  - DB operation
  - …

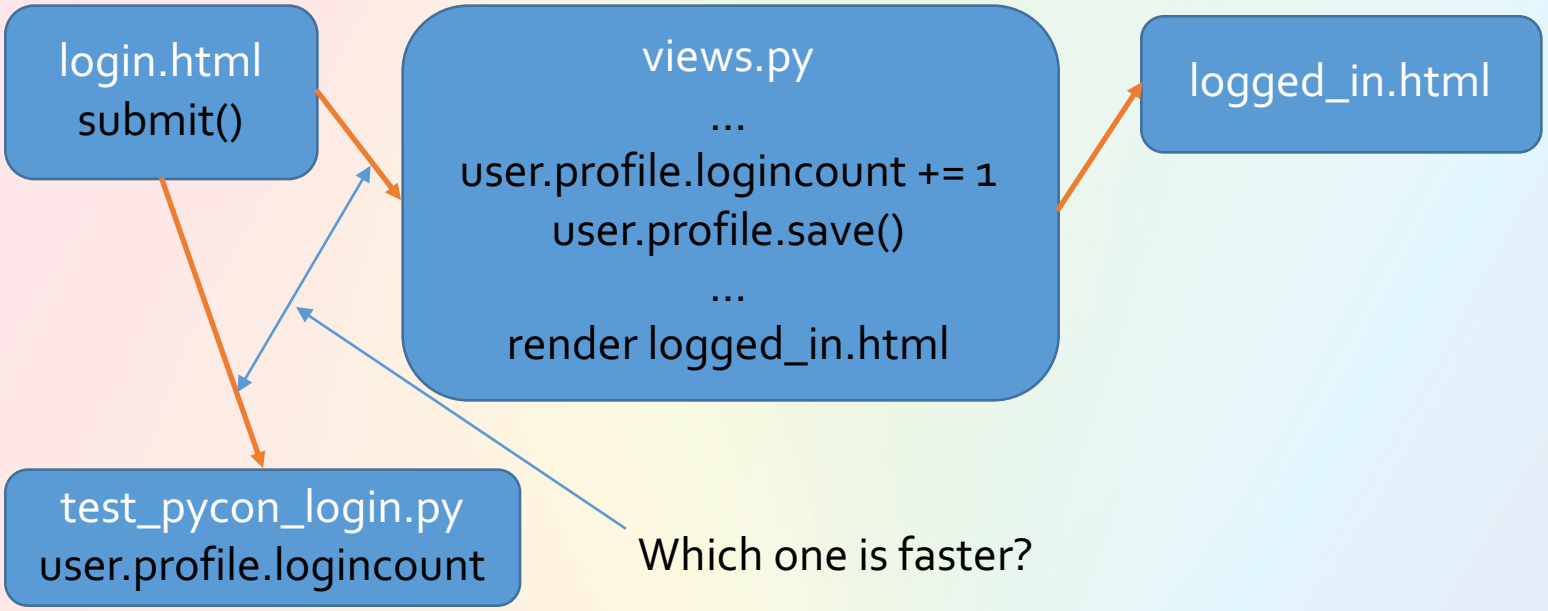
https://goo.gl/5ykwc4

Broken login example

```
def test_user_login(self):
    # Store login count of the user
    user = get_user_model().objects.get(username='einstein')
    logincount = user.profile.logincount

    # visit root url ('/')
    self.browser.get(self.live_server_url)

    # Try to login with email and password
    self.browser.find_element_by_name('email').send_keys('einstein@lablup.com')
    self.browser.find_element_by_name('password').send_keys('1')
    self.browser.find_element_by_css_selector('fieldset paper-button').click()

    # Confirm login count is increased by 1
    user = get_user_model().objects.get(username='einstein')
    self.assertEqual(user.profile.logincount, logincount + 1)

    # Check visit correct url
    target_url = self.live_server_url + '/accounts/profile/usertype/'
    self.assertEqual(self.browser.current_url, target_url)
```

```
python manage.py test functional_tests.test_pycon_login
```

Selenium do not wait automatically



Selenium do not wait automatically

```
browser.get("http://www.google.com")
```

Dependent on several factors, including the OS/Browser combination, **WebDriver may or may not wait for the page to load**.

Waiting methods in Selenium

- Implicit wait
- Explicit wait

} Provided by Selenium

- Stupid wait
  - It is highly **un-recommended** to wait for specific amounts of time.
  - If target is loaded in 5 s, we waste the remaining time.
  - If target is **not** loaded in 5 s, the test fails (slow system).

```
import time
time.sleep(5)
```

Implicit wait

- Official DOC:

  An implicit wait is to **tell WebDriver to poll the DOM for a certain amount of time** when **trying to find an element** or elements if they are not immediately available. The default setting is 0. Once set, the implicit wait is set for the life of the WebDriver object instance.

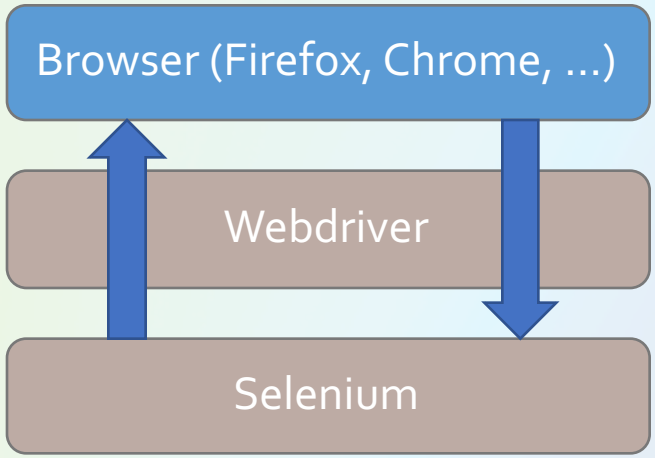
- Easy to use, global effect.

```
browser = webdriver.Firefox()
browser.implicitly_wait(10)  # in seconds
browser.get("http://somedomain/url_that_delays_loading")
myDynamicElement = browser.find_element_by_id("myDynamicElement")
```

- Applies only for element waiting.

Implicit wait

- Waiting behavior is determined on the "remote" side of the webdriver.
- Different waiting behavior on different OS, browser, versions, etc.
  - Return element immediately
  - Wait until timeout
  - …




```
def implicitly_wait(self, time_to_wait):
    self.execute(Command.IMPLICIT_WAIT, {'ms': float(time_to_wait) * 1000})
```

WebDriver

RemoteConnection.execute()

Explicit wait

- Official DOC:

  An explicit wait is **code you define to wait for a certain condition to occur** before proceeding further in the code. ... WebDriverWait in combination with ExpectedCondition is one way this can be accomplished.

- A little messy, local effect.

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
import selenium.webdriver.support.expected_conditions as EC

ff = webdriver.Firefox()
ff.get("http://somedomain/url_that_delays_loading")
try:
    element = WebDriverWait(ff, 10).until(
        EC.presence_of_element_located((By.ID, "myDynamicElement"))
    )
finally:
    ff.quit()
```

Explicit wait

- Highly flexible (variety of expected conditions)

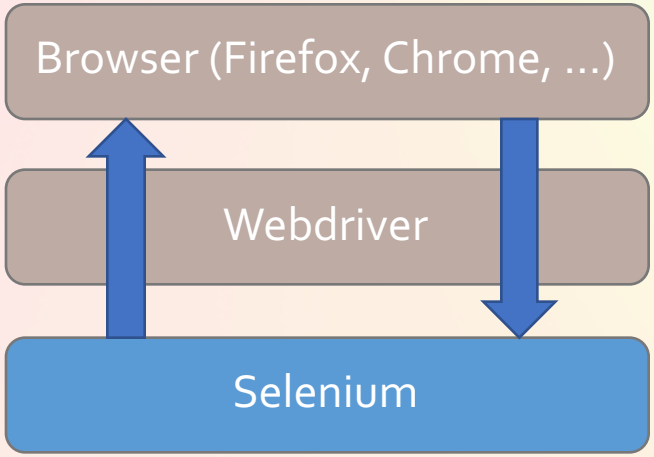
```
EC.title_is(...)
  .title_contains(...)
  .presence_of_element_located(...)
  .visibility_of_element_located(...)
  .visibility_of(...)
  .presence_of_all_elements_located(...)
  .text_to_be_present_in_element(...)
  .text_to_be_present_in_element_value(...)
  .frame_to_be_available_and_switch_to_it(...)
  .invisibility_of_element_located(...)
  .element_to_be_clickable(...)
  .staleness_of(...)
  .element_to_be_selected(...)
  .element_located_to_be_selected(...)
  .element_selection_state_to_be(...)
  .element_located_selection_state_to_be(...)
  .alert_is_present(...)
```

Explicit wait

- Highly flexible (not limited for waiting element)

```
condition = lambda _: self.browser.execute_script('return document.readyState') == 'complete'
WebDriverWait(self.browser, 30).until(condition)
```

- Waiting is controlled on the “local” side.
- Defined behavior.




```
def until(self, method, message=''):
    end_time = time.time() + self._timeout
    while(True):
        try:
            value = method(self._driver)
            if value:
                return value
        except self._ignored_exceptions:
            pass
        time.sleep(self._poll)
        if(time.time() > end_time):
            break
    raise TimeoutException(message)
```

Use explicit wait

- Use explicit wait!

- In good conditions (OS, browser, version), implicit wait can simplify test codes.
  - Be aware that the condition may change in the future.
  - I think implicit wait is not very stable way to go.

- Never use stupid wait.
  - Except in a very limited cases (like presentation?).

Running Tests on Cloud (AWS EC2)

Functional tests: automatically once a day

Run selenium on AWS

- No screen
  - Virtual display
- Slow machine (low cost)
  - Give timeout generously
- That’s all!

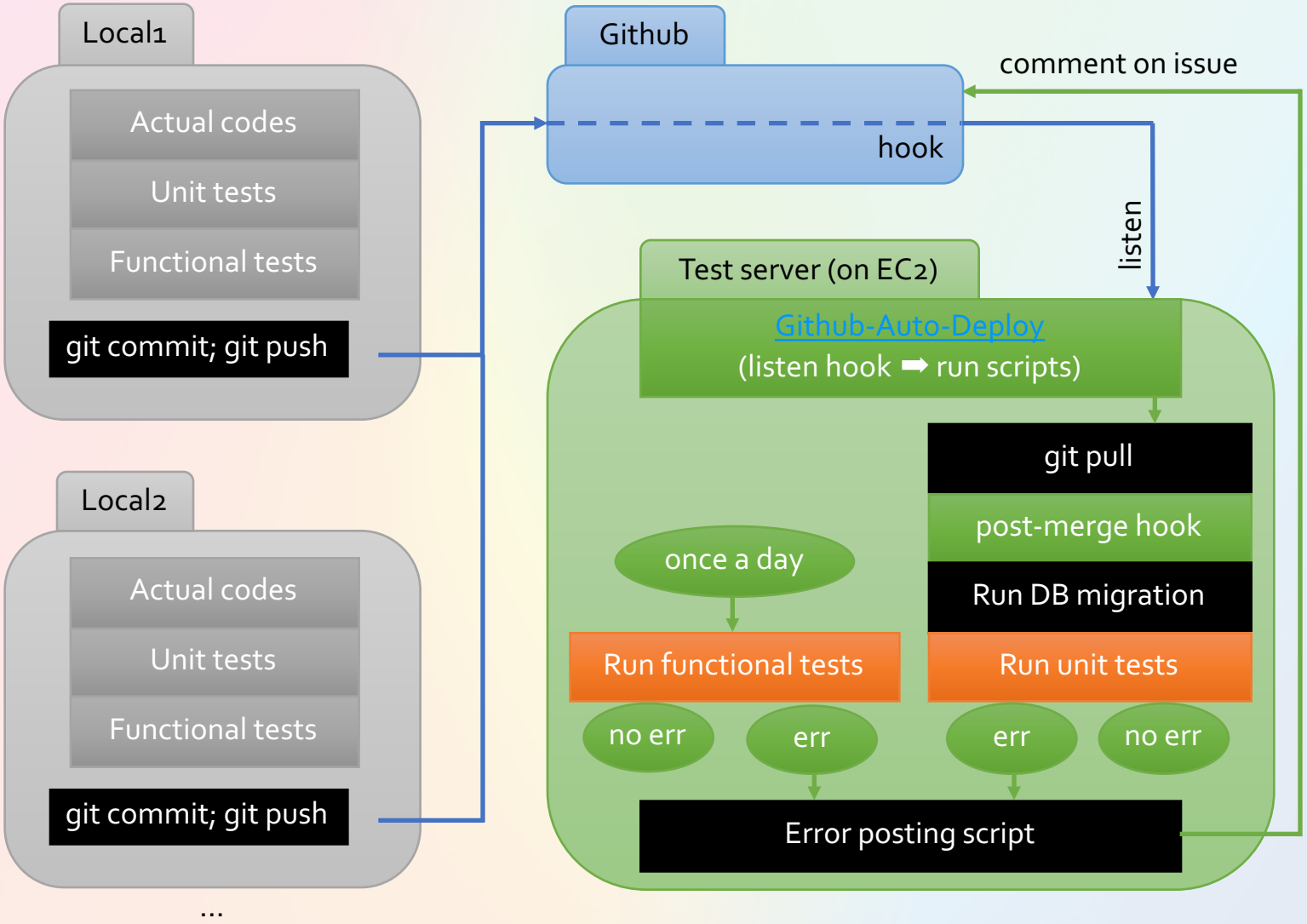
Run selenium on AWS

```
# Install headless java
sudo apt-get install openjdk-6-jre-headless

# Fonts
sudo apt-get install xfonts-100dpi xfonts-75dpi xfonts-scalable xfonts-cyrillic

# Headless X11 magic is here
sudo apt-get install xvfb

# We still demand X11 core
sudo apt-get install xserver-xorg-core

# Firefox installation
sudo apt-get install firefox

# Download Selenium server
wget http://selenium.googlecode.com/files/selenium-server-standalone-2.31.0.jar

# Run Selenium server
Xvfb :0 -screen 0 1440x900x16 2>&1 >/dev/null &
export DISPLAY=:0
nohup xvfb-run java -jar selenium-server-standalone-2.53.0.jar > selenium.log &
```

+ crontab

http://goo.gl/GRbztO

Test automation for CodeOnWeb



Slack integration is also possible

githubinion BOT 13:43

[lablup/neumann] New comment by testion on issue #132: Version tagging / deploy (assigned to installion)

> UNIT_TEST_ERROR: Ran 384 tests, FAILED (failures=1)

githubinion BOT 03:25

[lablup/neumann] New comment by testion on issue #466: Test error reports by testion (assigned to testion)

> FUNCTIONAL_TEST_ERROR: Ran 69 test, FAILED (errors=1)

Some Tips

Most of them is on the web.

Faster password hasher

- Use fastest password hasher (MD5) in testing.
  - Django's default: PBKDF2 algorithm with SHA256 hash.
  - It's more secure, but quite slow.

settings.py

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.MD5PasswordHasher',
]
```

- In our case, this reduces the test running time by
  - half (for unit tests).
  - ~10% (for functional tests).
- **Never use this in production server.**

Shorter login

```
from django.conf import settings
from django.contrib.auth import get_user_model, BACKEND_SESSION_KEY,
     SESSION_KEY, HASH_SESSION_KEY
from django.contrib.sessions.backends.db import SessionStore

def login_browser(self, username='einstein'):
    # Set fake session
    user = get_user_model().objects.get(username=username)
    session = SessionStore()
    session[SESSION_KEY] = user.pk
    session[BACKEND_SESSION_KEY] = 'django.contrib.auth.backends.ModelBackend'
    session[HASH_SESSION_KEY] = user.get_session_auth_hash()
    session.save()

    # To set cookies, the browser should visit a page (due to same origin policy).
    # It is faster to visit dummy page than login page itself.
    self.browser.get(self.live_server_url + '/selenium_dummy')
    self.browser.add_cookie({
        'name': settings.SESSION_COOKIE_NAME,
        'value': session.session_key,
        'secure': False,
        'path': '/'
    })
```

Tests for logged-in user is requred.

Fluently wait

- Wait for page load.

```
self.browser.get(self.live_server_url + '/dashboard/entry/')
```

?

```
class CommonMethods:
    @contextmanager
    def wait_for_page_load(self, timeout=TIMEOUT):
        old_element = self.browser.find_element_by_tag_name('html')
        yield
        WebDriverWait(self.browser, timeout).until(EC.staleness_of(old_element))

        # Wait until ready state is complete
        self.wait_until(
            lambda _: self.browser.execute_script('return document.readyState') == 'complete'
        )
    ...
```

```
with self.wait_for_page_load():
    self.browser.get(self.live_server_url + '/dashboard/entry/'
```

Fluently wait

- Simpler explicit wait.

```
element = WebDriverWait(ff, 10).until(
    EC.presence_of_element_located((By.ID, "myDynamicElement"))
)
```

```
class CommonMethods:
    ...
    def wait_until(self, condition):
        WebDriverWait(self.browser, TIMEOUT).until(condition)
```

```
element = self.wait_until(
    EC.element_to_be_clickable((By.ID, 'myDynamicElement'))
)
```

Make your own useful scripts

```
class CommonMethods:
    ...
    def clear_input(self, element):
        self.browser.execute_script('arguments[0].value = ""', element)

    def click_button(self, element):
        self.browser.execute_script('arguments[0].click()', element)
```

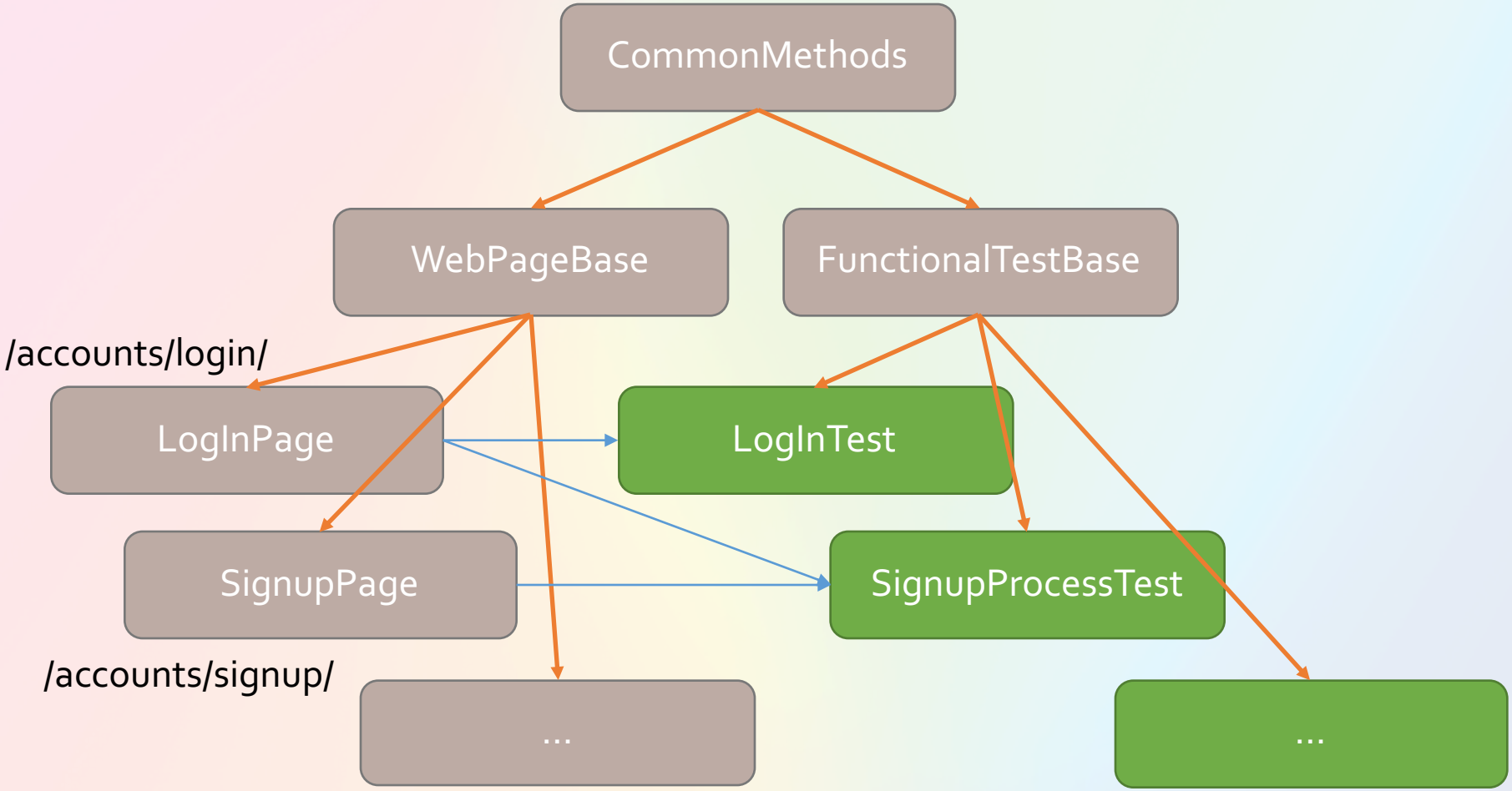
Using send_keys(str) appends str, not replacing

```
self.clear_input(self.elm['email_input'])
```

Using selenium's click() sometimes not working (animated buttons, ...)

```
self.click_button(self.elm['submit_button'])
```

Separate page implementation from test codes



Web page classes

```
class LogInPage(WebPageBase):
    def __init__(self, browser):
        super(LogInPage, self).__init__(browser)
        self.elm.update({
            'email_input': browser.find_element_by_name('email'),
            'pwd_input': browser.find_element_by_name('password'),
            'submit_button': browser.find_element_by_css_selector('fieldset paper-button'),
            ...
        })

    def login(self, email, password):
        self.elm['email_input'].send_keys(email)
        self.elm['pwd_input'].send_keys(password)
        with self.wait_for_page_load():
            self.click_button(self.elm['submit_button'])

    def click_signup_button(self):
        with self.wait_for_page_load():
            self.click_button(self.elm['signup_link'])

    def click_forgot_password_button(self):
        with self.wait_for_page_load():
            self.click_button(self.elm['forgot_pwd'])


class SignUpPage(WebPageBase):
    ...
```

pre-define static elements

modularize representative features

Test classes using web page classes

```
class LoginTest(FunctionalTestBase):
    def test_user_login(self):
        # Store login count of the user
        user = get_user_model().objects.get(username='einstein')
        logincount = user.profile.logincount

        # Visit login page and login
        with self.wait_for_page_load():
            self.browser.get(self.live_server_url)
        login_page = LogInPage(self.browser)
        login_page.login('einstein@lablup.com', '0000')

        # Check login count is increased by 1
        user = get_user_model().objects.get(username='einstein')
        self.assertEqual(user.profile.logincount, logincount + 1)

        # Check expected url
        target_url = self.live_server_url + '/dashboard/'
        self.assertEqual(self.browser.current_url, target_url)


class SignupProcessTest(FunctionalTestBase):
    def test_user_signup(self):
        # User clicks signup button at login page
        with self.wait_for_page_load():
            self.browser.get(self.live_server_url)
        login_page = LogInPage(self.browser)
        login_page.click_signup_button()
        ...
```

use web page classes / methods

Summary

- Using Selenium for Django functional tests.
- Wait is important in Selenium.
  - **Use explicit wait.**
- Automate tests on WS EC2.
- Give structure to your tests.
  - Make your useful scripts.

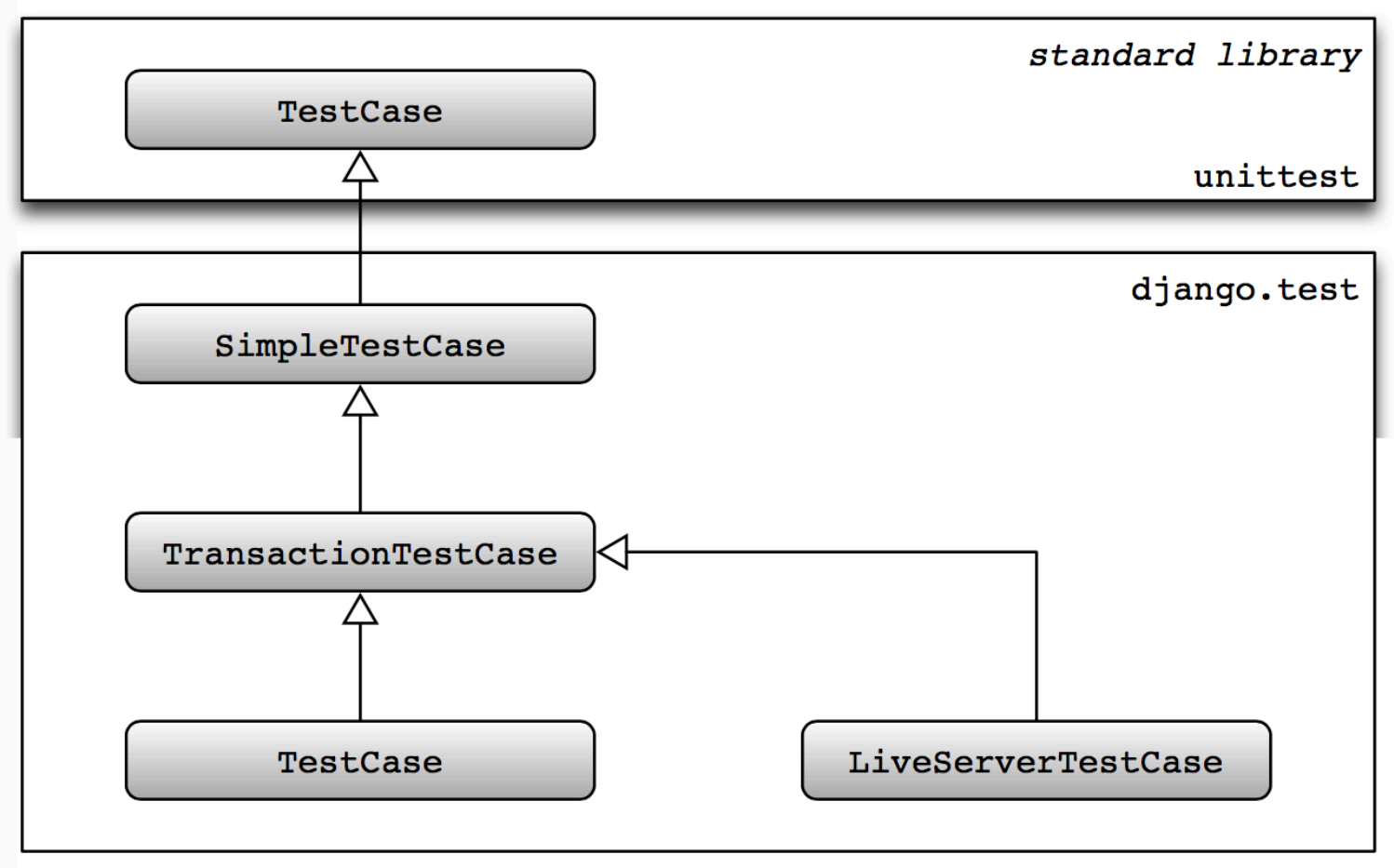
Thank you!

- Staffs of APAC PyCon 2016
- **… And ALL of YOU!**

http://www.lablup.com/
https://codeonweb.com/
https://kid.codeonweb.com/

Additional Information

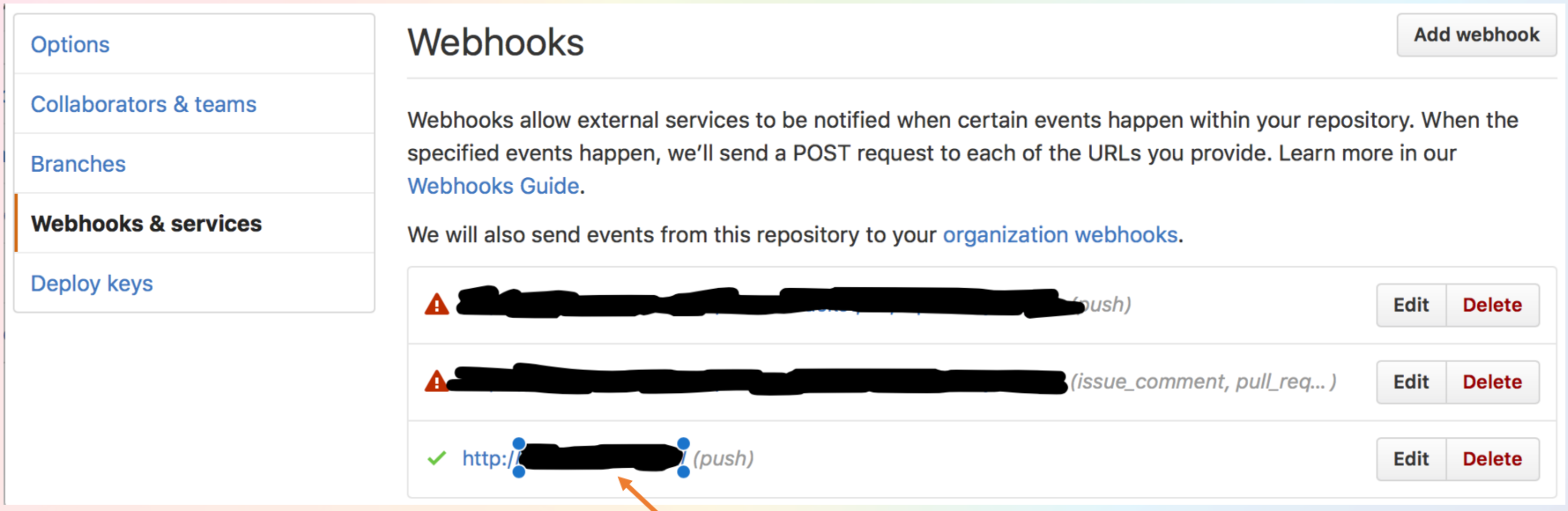
Hierarchy of Django unit testing classes



https://goo.gl/yDZlCw

Set Github webhook for unit test

- Repository settings - Webhooks & services.
- Add webhook for push.
  - When commit is pushed, send webhook to a test server.



Process webhook on AWS EC2

- Create an EC2 instance for testing server.
- Catch webhook and execute a command.
  - Github-Auto-Deploy (Python2).

GitAutoDeploy.conf.json

```
{
    "port": 9092,
    "repositories":
    [{
        "url": "https://github.com/lablup/neumann",
        "path": "/home/jpark/neumann",
        "deploy": "git pull"
    }]
}
```

```
python2 GitAutoDeploy.py
```

Set post-merge hook

.git/hooks/post-merge

```
#!/bin/bash
/home/jpark/.pyenv/versions/neumann/bin/python3 publish_unittest_errors.py
```

- Run tests and record errors.
- If error occurred, comment on a Github issue.
  - github3 package.