Sep 13, 2019

10 simple rules for writing Selenium automated tests

Sebastian Bassi[1]

[1]Globant

In Development dx.doi.org/10.17504/protocols.io.7bbhiin

Sebastian Bassi
Globant

ABSTRACT

Some advices regarding writing Selenium tests. The examples are in Python but the rules are language agnostic, so it can be implemented in any supported language.

1- Use Page Object Model (POM) to structure your tests.

Page Object Model (POM): It is a design pattern where objects are used to represent web pages. The elements in the page are properties of the class and user interactions should be implemented as methods of the class.

Example:

A BasePage class for elements common to all pages, like titles, styles and basic actions such as click on buttons and wait for elements to be displayed.
All POM should inherited from BasePage class, you can have a Footer POM, a left menu POM, a Sign-up POM and more according to the site structure.

Properties:
Submit_button, log_in link, regular_text, title. code_display_box.

Methods:
register_a new user. login. add_item_to_cart. post_a_message.

2- Use UI Maps (aka Object Repository)

Since object location may change during an application life cycle, you could end up updating several parts of the code each time there is a change in the CSS. To avoid this problem and you should have an object repository to store all object locations. This can be implemented with a key/value database or with a hash table or Python dictionary. The key will be the name or alias of the object, while the value would be the location.

Example:

UImap.py

```
loginpagemap = {"userfield": "input[type=text][name='user[login]']",
                "passwordfield" : "input[type=password][name='user[password]']",
                "loginbutton" : "button[class='btn-mktg btn-primary-mktg']"}
```

To use it:

```
from UImap import loginpagemap

class LoginPage(BasePage):
```

```
def login(self):
   self.fill_form(loginpagemap["userfield"], username)
   self.fill_form(loginpagemap["passwordfield"], password)
   self.click_button(loginpagemap["loginbutton"])
```

3- Use CSS locator over XPATH when possible.

Xpath engines may be different on each browser so they may be inconsistent. Since Xpath tend to become complex, are not so easy to read as CSS locators. In some browsers (IE based mostly) Xpath selectors are slower. Performance may not be an issue for small projects, but when tests are ran several times per day, each second ads to the total time.

4- Use tools to find elements.

There are several browsers add-ons or extensions that can help to find elements in a web page. Eskry for Chrome and FirePath for Firefox. Both allows to select an item in the page and get the CSS and the XPATH location of it.

5- Write independent tests.

This is one of the basic principles of automation testing. It avoids code duplication and prevents a failure to mask another failure. How to write independent tests? One of the causes that produce dependency between tests is that some tests depends on a state generated by another test. To reach a specific state the application could support URIs (instead of using the UI) and by using flags.

6- Use tags.

Tags allows to apply metadata to your tests. This way you can set a filter at runtime to execute desired tests according to your needs. There are several criteria that can be used, like functionality, execution speed, priority and so on. Let's see some criteria.
Functionality: Registration, Login, Managing the profile, checkout and other functionality on your web site that may need to be tested in a separate way.
Execution speed: Slow, medium, fast. This is useful for sorting tests to be run in different continuous integration (CI) pipelines.
Priority: Lowest, medium, highest. This criteria may be related with business value of each functionality.
Story number: Group tests by user story related to the test

7- Wrap selenium calls.

Selenium calls like sending keys, waiting for an element, selecting an option from a drop down menu and more, all should be wrapped to avoid code duplication.

8- Don't use fixed timers.

When your code needs for an element to appear, instead of waiting for a fixed amount of time, always try to explicit wait for the object to be displayed or available. Fixed waits could be larger than the actual time needed for the element to appear, that would increase test runs without any benefit.

Example:

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as ec

driver = webdriver.Chrome()
driver.get(website)

def is_visible(locator, timeout = 50):
   try:
      WebDriverWait(driver, timeout).until(ec.visibility_of_element_located((By.XPATH, locator)))
      return True
   except TimeoutException:
      return False
```

9- Hide Selenium objects.

Your POM should not have Selenium properties nor methods. The code in the tests should be understandable by anybody who knows about the subject without any knowledge of Selenium

10- Document you tests

Each test should have the goal stated, with all steps needed to reach this goal. Comment on all methods and page objects. Don't forget to document also how to Install the test and its dependencies.