CMS Clients - Automated Testing Tutorial

Objectives: Provide students with instructions on how to use the Selenium WebDriver software to automate user interface functional testing using Python and the Python unit test library.

Background: Selenium web-drivers are widely used in software testing with a number of frameworks. We will be using the native Selenium commands and Pycharm to help us build our automated test scripts. Pycharm has built in support for Selenium and Python unit testing. The goal is to create software automation scripts which can be used as a part of continuous integration / continuous delivery of software.

Directions:

1. We will be using the CMS Clients application created in a previous assignment. Open this project in Pycharm. It is important that you make this application a Pycharm project if you did not already do so in order to use Selenium to automate your testing.

2. Open your Pycharm Project for the CMS_Clients assignment and open the Terminal window for the project if it is not already open.

3. Activate your virtual environment if it is not already activated.
   PC: The command to activate the virtual environment in the Terminal window is (assuming you named your virtual environment venv):
   venv\Scripts\activate
   Mac:
   source venv/Scripts/activate

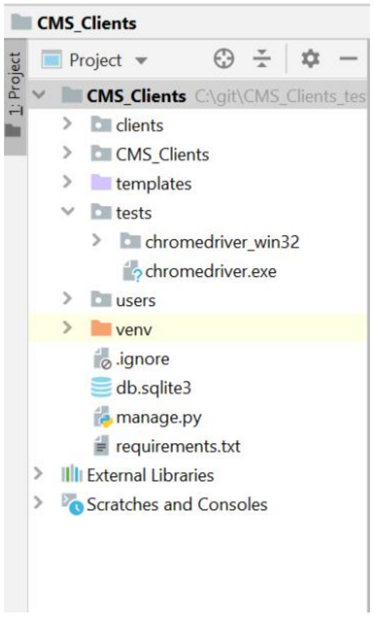
4. Download and install the Selenium Python package. In the Terminal window, enter the following command:
   pip install -U Selenium
   (Be sure your Windows or Mac user account has admin rights to do this)
   For your reference the site is: https://pypi.Python.org/pypi/Selenium.

5. Download the Selenium WebDriver for your choice of browser:
   a. **Add a new folder/directory to the CMS_Clients folder (highest level folder/directory) for your application named 'tests'. This folder should be at the same level as the templates folder.**
   b. **Mac**:
      i. No need to install the driver: "Safari now provides native support for the WebDriver API. Starting with Safari 10 on OS X El Capitan and macOS Sierra, Safari comes bundled with a new driver implementation that's maintained by the Web Developer Experience team at Apple"
      ii. Before Safari will work with automated testing, you need to configure Safari to allow automation.
      iii. As a feature intended for developers, Safari's WebDriver support is turned off by default.
      iv. To turn on WebDriver support, do the following:
         1. Ensure that the Develop menu is available in Safari. It can be turned on by opening Safari preferences (*Safari* > *Preferences* in the menu bar), going to the *Advanced* tab, and ensuring that the *Show Develop menu in menu bar* checkbox is checked.
         2. Enable Remote Automation in the Develop menu. This is toggled via *Develop* > *Allow Remote Automation* in the menu bar.
         https://webkit.org/blog/6900/webdriver-support-in-safari-10/

   c. **Windows**:
      i. Go to the Python language bindings for Selenium WebDriver page: https://pypi.org/project/selenium/
      ii. Scroll down to the 'Drivers' table on this page and select the driver for the browser you prefer. Chrome is used in the demo below. For the Mac, use Safari or Chrome.
         1. The newest version of the chromedriver.exe requires version 80 of Chrome. You may need to update your browser before running tests.
      iii. Select the download for your system, saving the file to the folder you created in step 3a above.
      iv. Extract the files

v. Move the driver application (executable file) to the ‘tests’ folder.

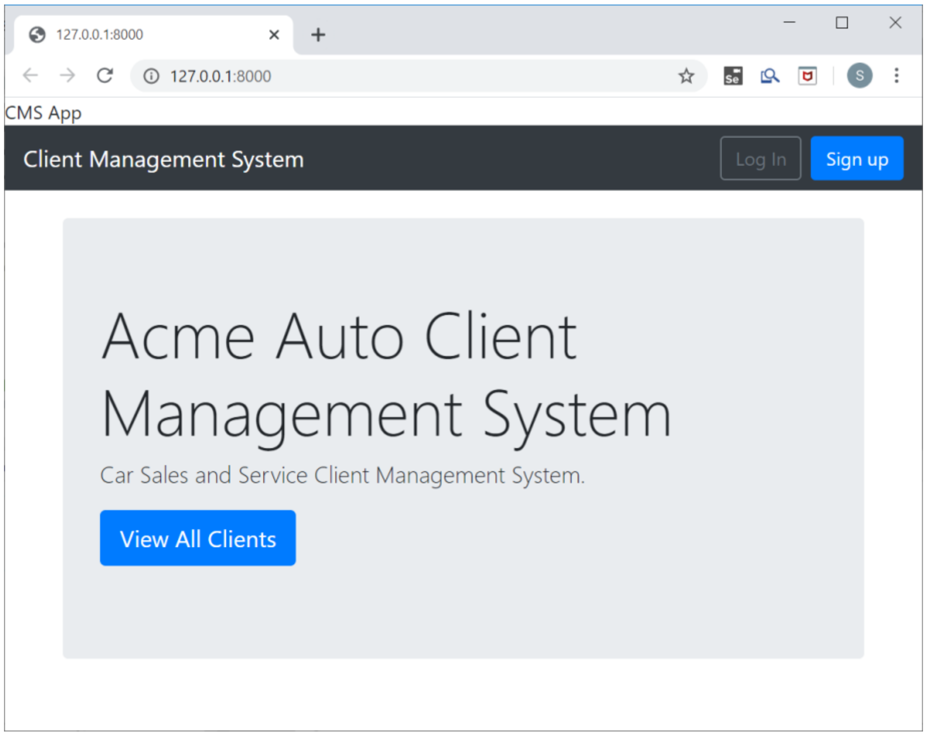
d. If you downloaded the chromedriver, your folders will now look like this in Pycharm (it will look similar for the other drivers as well with the exception of Safari on the Mac):



Note the Chromedriver.exe is in the ‘tests’ folder not in the chromedriver_win32 folder.

6. **The application must be running for the tests to execute successfully.** Use the standard command to run the server from the Terminal window: **python manage.py runserver**

   Do not sign in as a user at this point. We will be testing the login process. Note the “+ New” is not visible in the screen shot shown below:



7. Now we will begin to create our automated test script. Create the test Python files in the CMS_Clients/tests folder.

- We will use the built-in unittest library in Python which is similar to Junit in Java. This will allow us run one or more test scripts and provide feedback on the success or failure of the tests. ‘Time’ will be used to pause but not stop the script so you can view what is occurring as the tests execute. This can be removed or changed to 0 when using for real testing. Do not remove these before running your tests. The Time may need to be increased in order for the process to communicate with the browser.
- The files in this demonstration use the Chrome driver, but you may use the driver of your choice. Change the appropriate lines in the test cases to reflect your choice of driver.

8. In the CMS_Clients->tests folder, create a new Python file -> select ‘Python unit test’ as the file type, and name the file **unitTest_Login.py**

9. Replace the text in the file with the code shown below – IF NOT USING THE CHROME DRIVER, CHANGE THE DRIVER TO REFLECT YOUR DRIVER OF CHOICE.

```
import unittest
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import NoSuchElementException


class CMS_ATS(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Chrome()

    def test_cms(self):
        user = "ID"
        pwd = "PW"

        driver = self.driver
        driver.maximize_window()
        driver.get("http://127.0.0.1:8000/admin")
        elem = driver.find_element_by_id("id_username")
        elem.send_keys(user)
        elem = driver.find_element_by_id("id_password")
        elem.send_keys(pwd)
        time.sleep(3)
        elem.send_keys(Keys.RETURN)
        driver.get("http://127.0.0.1:8000")
        time.sleep(3)
        #assert "Logged in"
        try:
            # attempt to find the '+ New' - if found, logged in
           elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a")
           assert True

        except NoSuchElementException:
            self.fail("Login Failed - user may not exist")
            assert False

        time.sleep(3)


def tearDown(self):
    self.driver.close()
```

```python


if __name__ == "__main__":
    unittest.main()
```

REPLACE THE VALUES 'ID' AND 'PW' IN THE ABOVE CODE WITH A VALID USERNAME AND PASSWORD FOR YOUR APPICATION. You will need to use a username and password you created for your application in the above file.

Note: All the commands are defined in the Selenium WebDriver documentation which provide the same commands in several popular languages. [Click here for detail Selenium Commands](). This site also lists pros and cons of each of the browser drivers.

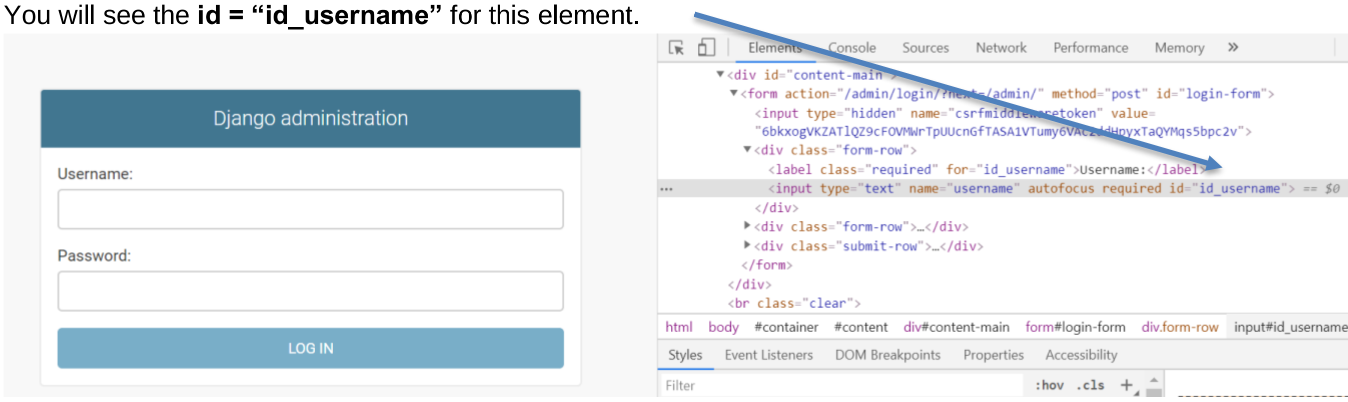
Let's walk through the code shown above. The test_cms method is defined and provides a place for a user ID and password we need to log onto our application.

Next the Chrome browser is opened and maximized.

Now we get to an important part of any screen reading tool and that is **LOCATING THE SCREEN ELEMENT.** In order for you to do anything in a browser screen you need to be able to identify to the program the element you want to manipulate.
The first element we want to find is the username field on the admin screen:

```python
elem = driver.find_element_by_id("id_username")
```

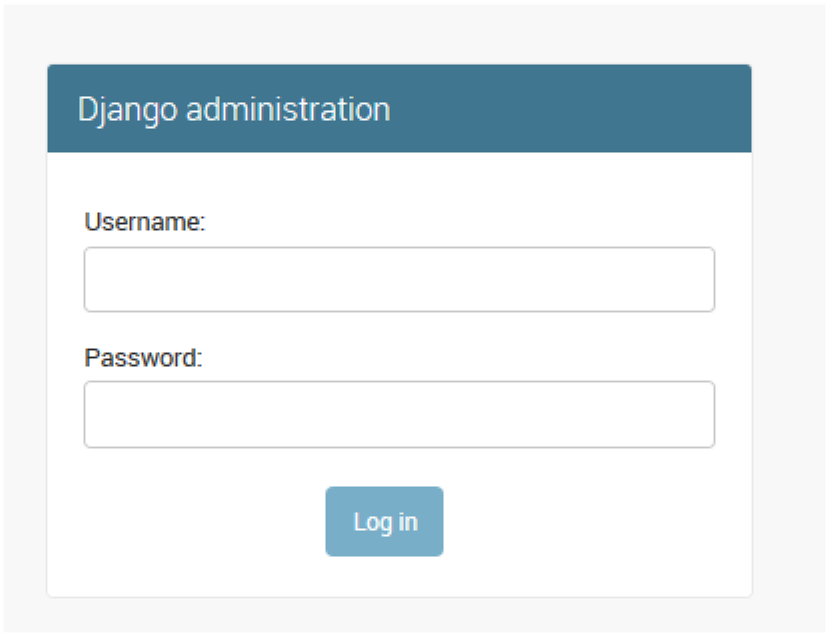
To find an element's ID, show the source html in a browser and find the ID of the element.
<u>In Chrome,</u> for example, right-click on element you want to use. In this case, right-click on the username text box and select 'Inspect'. The html for this element will be highlighted and will show the information about this element on the screen, including the id.
You will see the **id = "id_username"** for this element.



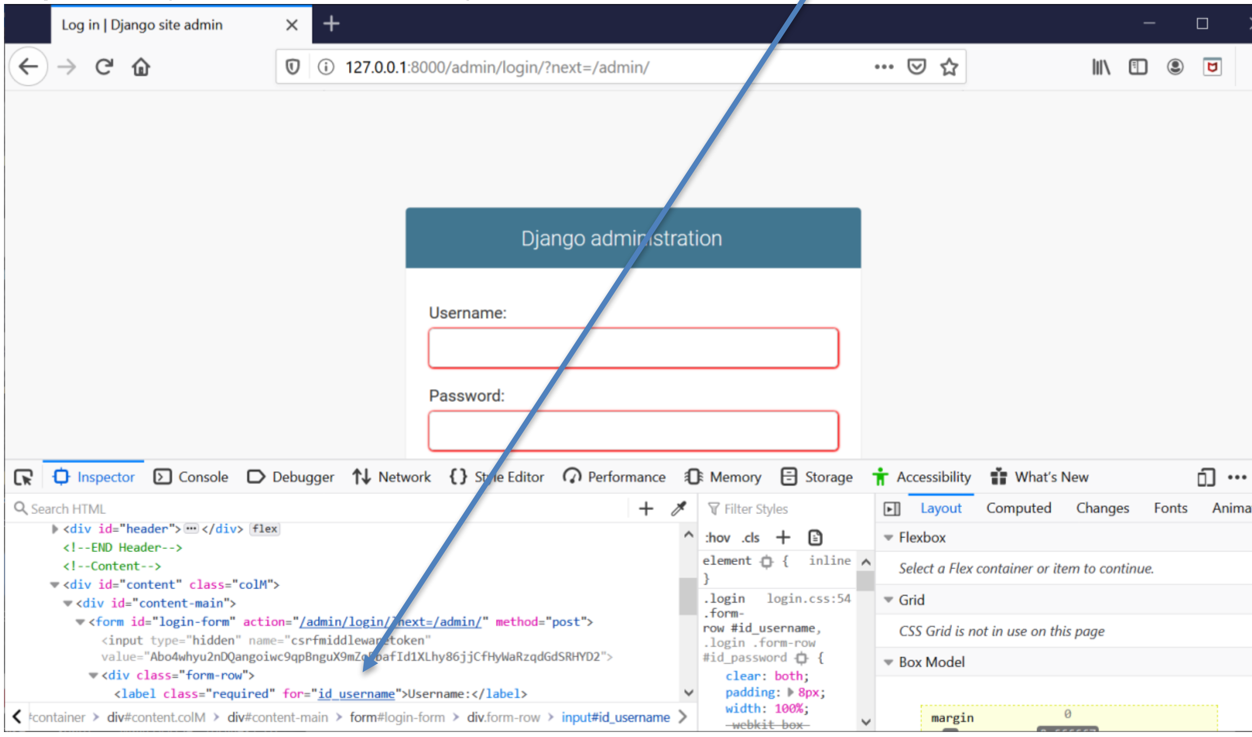
```html
▼<div id="content-main">
  ▼<form action="/admin/login/?next=/admin/" method="post" id="login-form">
     <input type="hidden" name="csrfmiddlewaretoken" value=
     "6bkxogVKZATlQZ9cFOVMWrTpUUcnGfTASA1VTumy6VAc2ddHpyxTaQYMqs5bpc2v">
   ▼<div class="form-row">
       <label class="required" for="id_username">Username:</label>
...    <input type="text" name="username" autofocus required id="id_username"> == $0
     </div>
   ▶<div class="form-row">…</div>
   ▶<div class="submit-row">…</div>
   </form>
 </div>
 <br class="clear">
```

html body #container #content div#content-main form#login-form div.form-row input#id_username
Styles
Event Listeners
DOM Breakpoints
Properties
Accessibility
Filter
:hov
.cls
+

<u>The Firefox browser</u> also provides an easy way to view the element ids:



We need to navigate to the Username box. Below shows the Firefox developer tool called **Inspector**. Right-click the Username input box to view the name.



```
<div id="header"> ⋯ </div> flex
<!--END Header-->
<!--Content-->
<div id="content" class="colM">
  <div id="content-main">
    <form id="login-form" action="/admin/login/?next=/admin/" method="post">
      <input type="hidden" name="csrfmiddlewaretoken"
      value="Abo4whyu2nDQangoiwc9qpBnguX9mZ[illegible]bafId1XLhy86jjCfHyWaRzqdGdSRHYD2">
      <div class="form-row">
        <label class="required" for="id_username">Username:</label>
```

```
container > div#content.colM > div#content-main > form#login-form > div.form-row > input#id_username
```

```
element { inline
}
.login .form-row #id_username,   login.css:54
.login .form-row #id_password {
    clear: both;
    padding: 8px;
    width: 100%;
    ~~-webkit-box-~~
```

Both the User ID and Password boxes are clearly labeled with ID labels.

Next: we send the username box the 'keys' or text for the user name:

```
 elem.send_keys(user)
```

Then we repeat these steps for the password box.

Next, we send the RETURN command which is like pressing enter on many keyboards, and if the username and password are correct, we are logged in.

```
  elem.send_keys(Keys.RETURN)
```

Next, we return to the application home page to prepare for the next action with:

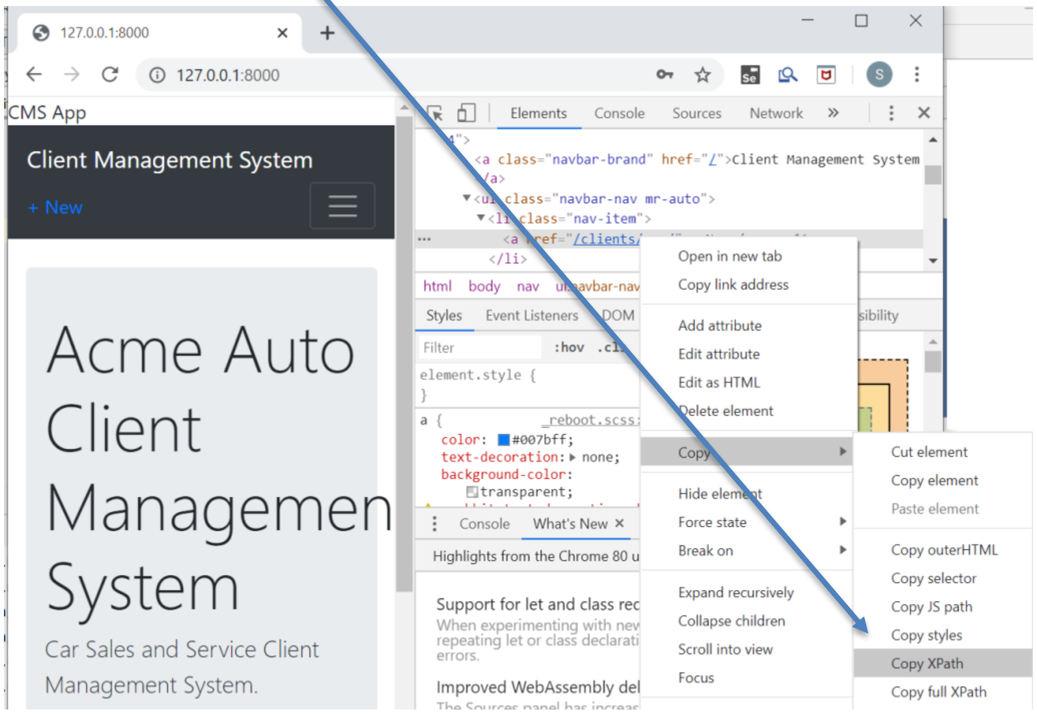
```
driver.get("http://127.0.0.1:8000")
```

Next, we try to find the 'New +' symbol to determine if the login was successful. If the 'New +' is found, the logon was successful, and we set the assert status to true. If it was not found, logon failed, and we set the self.fail message and set the assert status to false.

```python
try:
    # attempt to find the '+ New' - if found, logged in
   elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a")

   assert True

except NoSuchElementException:
    self.fail("Login Failed - user may not exist")
    assert False
```

Note that for this element, we will locate the item by 'XPath'.
To find the 'XPath' of the element, log on using the http://127.0.0.1:8000/admin page, go back to the main page, http://127.0.0.1:8000,

In Chrome: Right-click on the 'New +' and select 'Inspect' to view that element, then right-click on the high-lighted text in the Inspect pane, and select
'Copy –>Copy XPath'.



```
4">
  <a class="navbar-brand" href="/">Client Management System
  </a>
▼<ul class="navbar-nav mr-auto">
  ▼<li class="nav-item">
      <a href="/clients/
    </li>
```

```
element.style {
}
a {                        _reboot.scss:
   color: #007bff;
   text-decoration: ▸ none;
   background-color:
       transparent;
```

Highlights from the Chrome 80 u

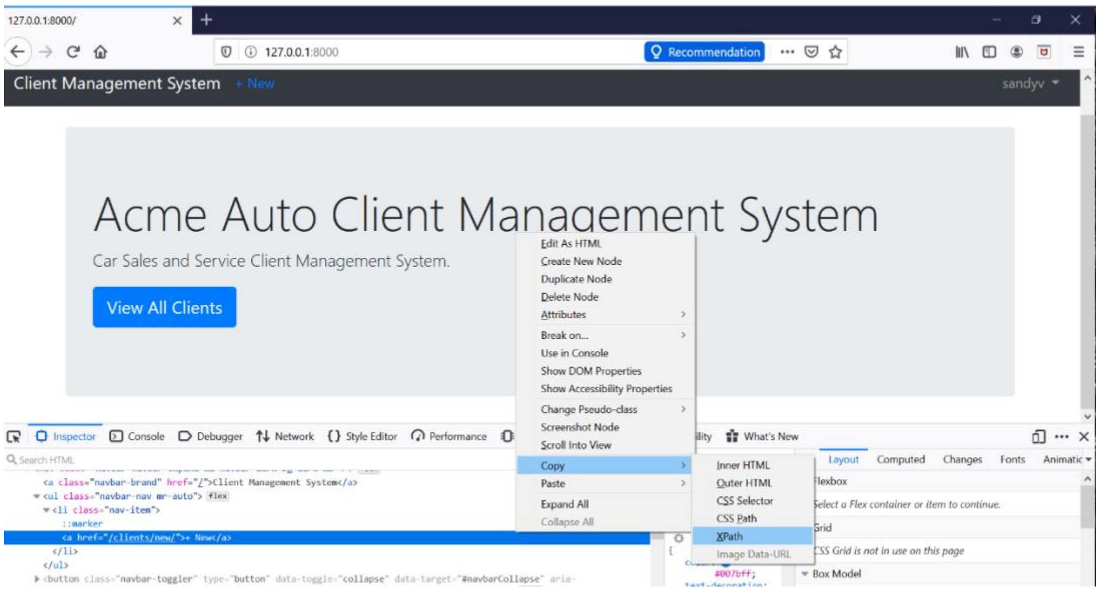
Support for let and class rec

When experimenting with nev repeating let or class declarati errors.

Improved WebAssembly del

The Sources panel has increas

In Firefox: Right-click on the + and select 'Inspect Element'. Then right-click on the highlighted text and select 'Copy - > XPath'

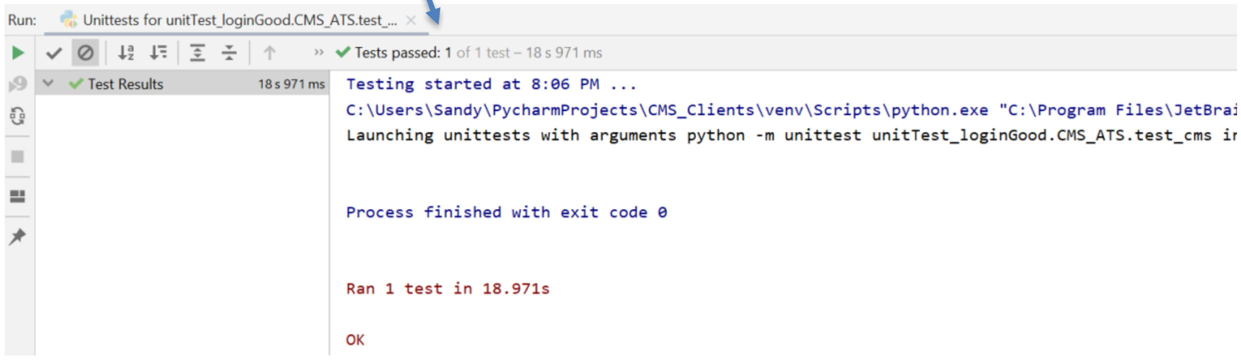


```
<a class="navbar-brand" href="/">Client Management System</a>
<ul class="navbar-nav mr-auto"> flex
  <li class="nav-item">
    ::marker
    <a href="/clients/new/">+ New</a>
  </li>
</ul>
<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarCollapse" aria-
```

This XPath is then pasted into the Python file for the driver.find_element_by_xpath method.

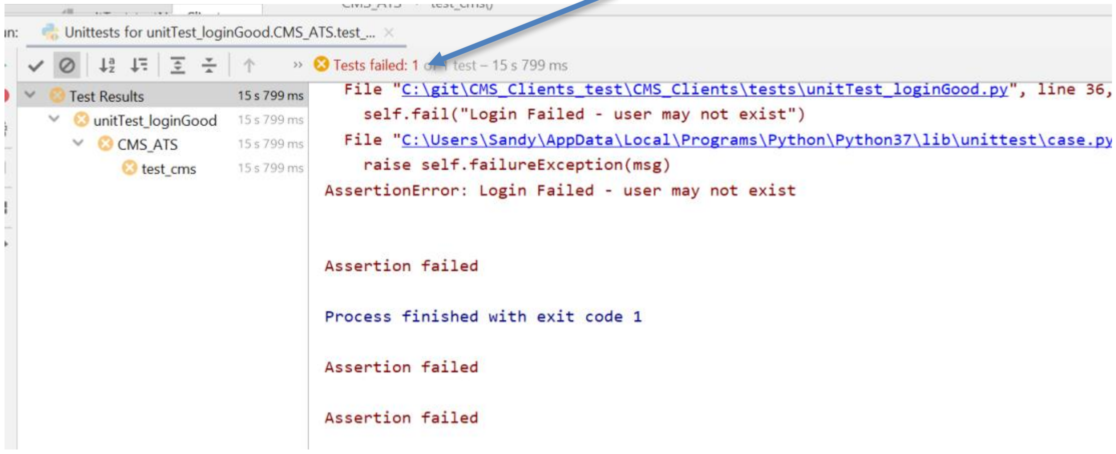
- **In order for this test to pass successfully,** you will need to create an account with the username of ‘ID’ and a password of ‘PW’. Alternatively, you can change the values in the test file to use an account you previously created.

10. Log out of the application if you were logged in but leave it running locally.
11. Run the unitTest_loginGood.py in Python by right-clicking the file 'unitTest_loginGood.py' file or selecting Run from the menu bar. You should see the application page open and the login process execute automatically as defined in the unit test file. The test is successful when you see something similar the following in the Pycharm 'Run' pane – specifically notice 'Tests passed:


Run:
Unittests for unitTest_loginGood.CMS_ATS.test_...
Tests passed: 1 of 1 test – 18 s 971 ms
Test Results
18 s 971 ms

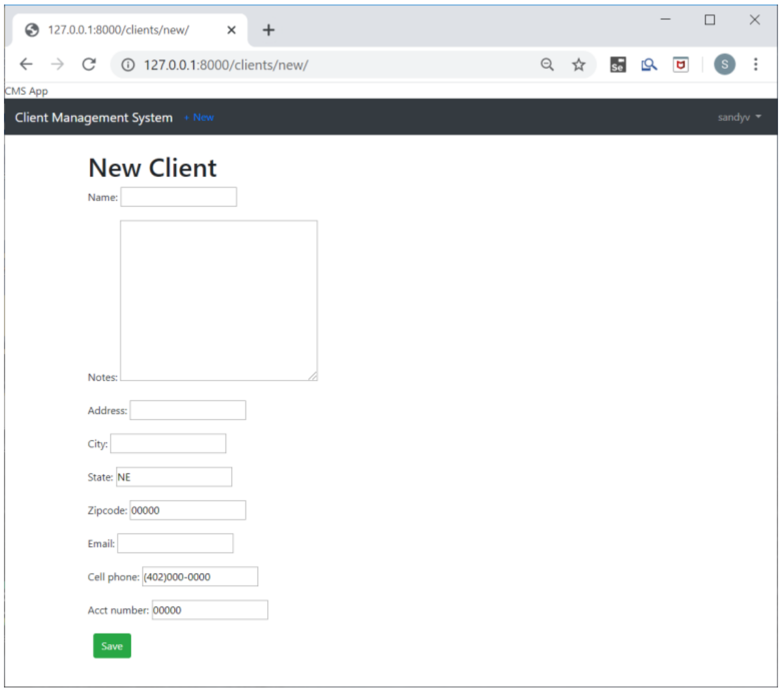
```
Testing started at 8:06 PM ...
C:\Users\Sandy\PycharmProjects\CMS_Clients\venv\Scripts\python.exe "C:\Program Files\JetBrai
Launching unittests with arguments python -m unittest unitTest_loginGood.CMS_ATS.test_cms ir


Process finished with exit code 0


Ran 1 test in 18.971s

OK
```

12. If the test fails, you will see something like the following in the Run pane (you will need to scroll through the Run pane to view this output - specifically notice 'Tests failed:



```
  File "C:\git\CMS_Clients_test\CMS_Clients\tests\unitTest_loginGood.py", line 36,
    self.fail("Login Failed - user may not exist")
  File "C:\Users\Sandy\AppData\Local\Programs\Python\Python37\lib\unittest\case.py
    raise self.failureException(msg)
AssertionError: Login Failed - user may not exist


Assertion failed

Process finished with exit code 1

Assertion failed

Assertion failed
```

13. Next, we want to test the add of a new Client. To do this we need to determine what actions happen to bring up the add New Client entry screen below.



New Client

Name:

Notes:

Address:

City:

State: NE

Zipcode: 00000

Email:

Cell phone: (402)000-0000

Acct number: 00000

Save

There is more than one way to bring this screen up. The easiest way is to click on the "+ New" in the Home screen after the user has logged in. We will mimic this process in the unit test file.

- We will treat the "+ New" like a button.
- We will need to find the XPath address of the object (+ New).
- You can find the XPath as demonstrated in the previous test above.

14. In the CMS_Clients -> tests folder, create a new Python file -> select 'Python unit test' as the file type, and name the file **unitTest_addNewClient.py**

Add the code shown below to this file – note where the XPath is referenced, and where the Client field values are entered. Also note that in this test, we confirm the Client was added successfully if the new screen displayed after clicking the 'Save' button includes the 'Edit' action i.e: Edit | Delete.

Reminder: As in the previous test, the username is set to '**ID'** and the password is set to '**PW'**. If you do not have a user with this id and password, replace those values with the values used by one of the admin accounts you created. IF NOT USING THE CHROME DRIVER, CHANGE THE DRIVER TO REFLECT YOUR DRIVER OF CHOICE.

This code continues on the next page – be sure to capture all the code.

```
# # automated unit test to ensure a window to add a new client appears
# when the "+ New" button is clicked
import unittest
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import NoSuchElementException

class CMS_ATS(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Chrome()

    def test_cms(self):
        #login from the admin pane
        user = "ID"
        pwd = "PW"
        driver = self.driver
        driver.maximize_window()
        driver.get("http://127.0.0.1:8000/admin")
        elem = driver.find_element_by_id("id_username")
        elem.send_keys(user)
        elem = driver.find_element_by_id("id_password")
        elem.send_keys(pwd)
        elem.send_keys(Keys.RETURN)
        driver.get("http://127.0.0.1:8000")
        assert "Logged In"
        time.sleep(5)

        # find the '+ New' and click it
        elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a").click()
        time.sleep(5)
        continue_test = False
        try:
          #verify New Client exists on new screen after clicking "+ New" button
            elem = driver.find_element_by_xpath("/html/body/div/h1").click()
            continue_test = True

        except NoSuchElementException:
            self.fail("Add new client does not appear = New Client button not present")
            assert False
            time.sleep(1)
        except:
            self.fail("Edit post NOT successful - error occurred: ")
            assert False
            time.sleep(1)
        time.sleep(2)
        #if test successful so far - set up the required inputs for a Client
        if continue_test:
            elem = driver.find_element_by_id("id_name")
            elem.send_keys("Test Name")
            elem = driver.find_element_by_id("id_notes")
            elem.send_keys("This is a test note via Selenium testing")
            elem = driver.find_element_by_id("id_address")
            elem.send_keys("1234 Main St")
            elem = driver.find_element_by_id("id_city")
            elem.send_keys("Omaha")
            elem = driver.find_element_by_id("id_state")
            elem.send_keys("") # use default value
            elem = driver.find_element_by_id("id_zipcode")
            elem.send_keys("") #use default
            elem = driver.find_element_by_id("id_email")
            elem.send_keys("testEmail@test.com")
            elem = driver.find_element_by_id("id_cell_phone")
            elem.send_keys("")  #use default value
            elem = driver.find_element_by_id("id_acct_number")
            elem.send_keys("1")
            time.sleep(6)
            #click the Save button
            elem = driver.find_element_by_xpath("/html/body/div/form/button").click()
            time.sleep(6)
            try:
                # find the 'Edit' button - if client added, Edit | Delete are displayed
                elem = driver.find_element_by_xpath("/html/body/div/p[1]/a[1]")
                assert True
            except NoSuchElementException:
                self.fail("Add Client NOT successful")
                assert False
            time.sleep(3)

    def tearDown(self):
        self.driver.close()


if __name__ == "__main__":
    unittest.main()
```

REPLACE THE VALUES 'ID' AND 'PW' WITH A VALID USERNAME AND PASSWORD FOR YOUR APPICATION.

POSSIBLE ISSUE WITH ABOVE TEST:

The above test to verify the new user screen appears when the + New is clicked after logon does not always work correctly, especially on the Mac. For some reason, the 'click' is not processed.
To fix this problem, replace this line:

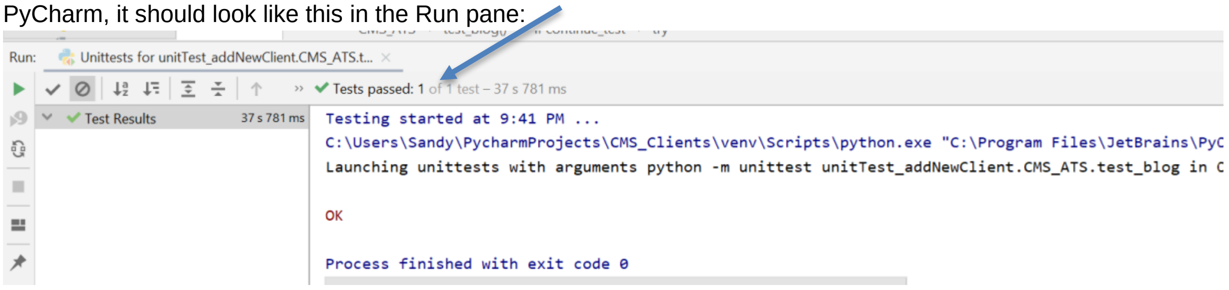
```
        elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a").click()
```

With these 3 lines:

```
        elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a")
        elem.send_keys(Keys.RETURN)
        time.sleep(5)
```

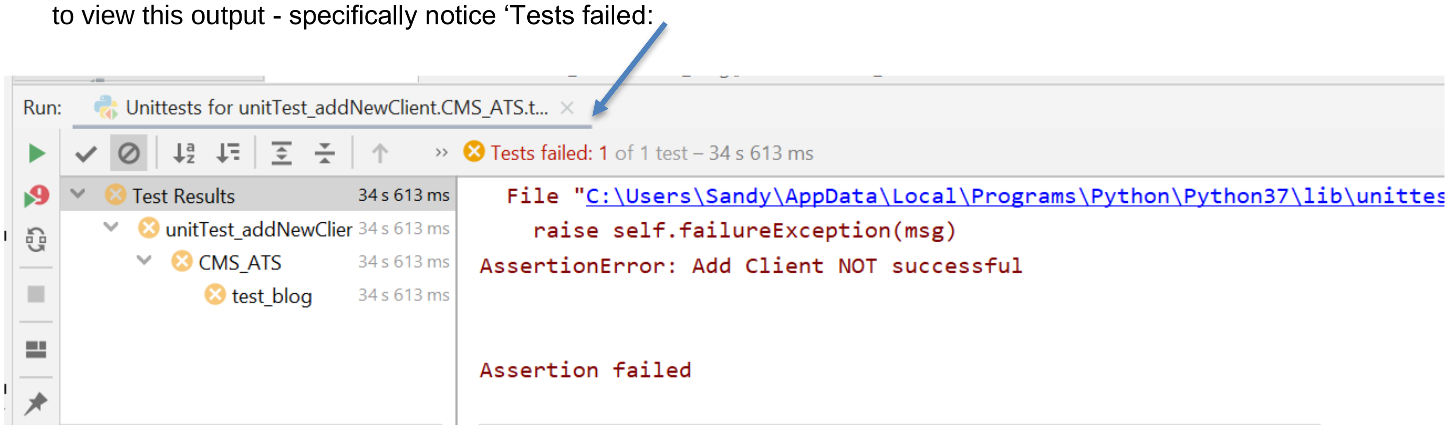
Note that you may also need to increase the 'sleep' times to allow the system to display the screen correctly.

Ensure your application is running and run this Python unit test program. You will see the application open, and the login, the entries for the New Client and the Save. This now completes the test and we can assert that we have created a new client. When you run this in PyCharm, it should look like this in the Run pane:


Run:
Unittests for unitTest_addNewClient.CMS_ATS.t...
Tests passed: 1 of 1 test – 37 s 781 ms
Test Results
37 s 781 ms

```
Testing started at 9:41 PM ...
C:\Users\Sandy\PycharmProjects\CMS_Clients\venv\Scripts\python.exe "C:\Program Files\JetBrains\PyC
Launching unittests with arguments python -m unittest unitTest_addNewClient.CMS_ATS.test_blog in C

OK

Process finished with exit code 0
```

If the test fails, you will see something like the following in the Run pane (you will need to scroll through the Run pane to view this output - specifically notice 'Tests failed:


Run:
Unittests for unitTest_addNewClient.CMS_ATS.t...
Tests failed: 1 of 1 test – 34 s 613 ms
Test Results 34 s 613 ms
unitTest_addNewClier 34 s 613 ms
CMS_ATS 34 s 613 ms
test_blog 34 s 613 ms

```
  File "C:\Users\Sandy\AppData\Local\Programs\Python\Python37\lib\unittes
    raise self.failureException(msg)
AssertionError: Add Client NOT successful


Assertion failed
```

Note that your test could also fail if the “Edit” does not appear on the screen after a new Client is added. If you do not have that text on your form, the test will fail – as it should.

Now it is your turn to create your own Python test scripts as described in the assignment. The Selenium commands that you have learned in this tutorial will help you to accomplish this. Most CRUD applications involve text boxes, tabs, and buttons. You need to identify these and take the appropriate action as you would when you are running a manual functional test.