SET-221
Software Testing Technologies

LAB # 11

LAB Title

Handling Waits and Alerts in Selenium Dealing with dynamic content and browser alerts

Assessment of CLO: 03, PLO: 05

Student Name:			
Roll No.			
Semester		Session	

S. No.	Perf. Level / Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
Total Marks Obtained Out of 10						

Experiment evaluated by

Instructor's Name	Engr.Bushra Aziz		
Date		Signature	

Lab Experiment 11: Handling Waits and Alerts in Selenium

Objective: To understand and implement various waiting strategies and effectively handle JavaScript alerts using Selenium with Python.

1. Understanding and Implementing Waits

In web automation, elements on a webpage may take varying amounts of time to load. Directly interacting with an element before it's fully loaded can lead to NoSuchElementException or other unexpected errors. Selenium provides mechanisms to handle these dynamic loading scenarios using waits.

1.1. Implicit Waits:

An implicit wait tells the WebDriver to wait for a certain amount of time globally when trying to find an element. If the element is found within that time, the script proceeds; otherwise, a NoSuchElementException is raised.

Procedure:

```
from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.chrome.service import Service as ChromeService

from webdriver_manager.chrome import ChromeDriverManager

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC

from selenium.common.exceptions import NoSuchElementException

from selenium.webdriver.common.keys import Keys

import time

driver=webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()))
```

1. Initialize your WebDriver (e.g., for Chrome):python

   ```
   driver = webdriver.Chrome() # Ensure chromedriver is in your PATH
   ```

2. Set the implicit wait duration (in seconds):python

   ```
   driver.implicitly_wait(10) # Wait up to 10 seconds for elements to become available
   ```

3. Navigate to a webpage with dynamically loading elements (you can use a practice website for this).
4. Attempt to locate an element that might take some time to load.
5. Observe the behavior of the script when the element loads within the implicit wait time and when it doesn't (you can intentionally use an incorrect locator to simulate this).

1.2. Explicit Waits:

An explicit wait allows you to wait for a specific condition to be met before proceeding with the script. You can define the condition you are waiting for (e.g., an element to be clickable, visible, present).

Lab Experiment 11: Handling Waits and Alerts in Selenium
Procedure:

1. Ensure you have the necessary imports (as shown in the Implicit Waits section).
2. Initialize your WebDriver.
3. Create a WebDriverWait instance, specifying the driver and the maximum wait time:

```
Python

wait = WebDriverWait(driver, 15) # Wait up to 15 seconds
```

4. Use the until() method of the WebDriverWait object along with expected_conditions to wait for a specific condition:

- **Element to be clickable:**

```
element = wait.until(EC.element_to_be_clickable((By.ID, "submit_button")))
element.click()
```

- **Element to be visible:**

```
element = wait.until(EC.visibility_of_element_located((By.XPATH, "//div[@class='loaded-content']")))
print(element.text)
```

- **Presence of an element:**

```
element = wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, ".dynamic-element")))
```

- **Text to be present in an element:**

```
wait.until(EC.text_to_be_present_in_element((By.ID, "status_message"), "Success!"))
```

Explore other expected_conditions in the selenium.webdriver.support.expected_conditions module.

5. Navigate to a webpage with dynamic elements and implement explicit waits for different conditions.
6. Experiment with setting a shorter timeout than necessary to observe the TimeoutException.

3. Fluent Waits:

Fluent waits provide more flexibility in defining the waiting conditions. You can specify the polling interval (how frequently Selenium checks for the condition) and ignore specific exception types while waiting.

Procedure:

1. Import the NoSuchElementException and TimeoutException from selenium.common.exceptions.
2. Initialize your WebDriver.
3. Create a WebDriverWait instance and configure it as a fluent wait:

```
Python

from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import NoSuchElementException, TimeoutException
import time

driver = webdriver.Chrome()
wait = WebDriverWait(driver, 30, poll_frequency=0.5,
ignored_exceptions=[NoSuchElementException])


try:
```

```
        element = wait.until(EC.visibility_of_element_located((By.ID, "delayed_element")))
        print("Element found:", element.text)
    except TimeoutException as e:
        print("Element not found after waiting.")
    finally:
        driver.quit()
```

4. Experiment with different poll_frequency values and ignored_exceptions.

2. Handling JavaScript Alerts

JavaScript alerts are small pop-up boxes that appear in the browser to display messages or request user input. Selenium provides mechanisms to interact with these alerts.

2.1.1. Understanding Alert Types:

- **Simple Alert:** Displays a message with an "OK" button.
- **Confirmation Alert:** Displays a message with "OK" and "Cancel" buttons.
- **Prompt Alert:** Displays a message with a text input field, "OK," and "Cancel" buttons.

2.1.2. Interacting with Alerts:

- **Procedure:**

1. Navigate to a webpage that triggers JavaScript alerts (you can find practice websites for this).
2. Locate the element that triggers the alert and interact with it (e.g., click a button).
3. Switch the driver's focus to the alert using driver.switch_to.alert:

Python

```
alert = driver.switch_to.alert
```

4. Perform actions on the alert:

   - **Get the alert message:**

```
message = alert.text
print("Alert message:", message)
```

   - **Accept the alert (click "OK"):**

```
alert.accept()
```

   - **Dismiss the confirmation or prompt alert (click "Cancel"):**

```
alert.dismiss()
```

   - **Enter text into a prompt alert:**

```
alert.send_keys("Your Input")

alert.accept()
```

5. After handling the alert, the driver's focus remains on the alert. You might need to switch back to the main content (though in most cases, Selenium automatically handles this after alert interaction).

- **Example (Handling a Simple Alert):**

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

driver = webdriver.Chrome()
driver.get("YOUR_WEBSITE_WITH_ALERTS") # Replace with a suitable URL
driver.implicitly_wait(5)

trigger_button = driver.find_element(By.ID, "simple_alert_button")
trigger_button.click()

alert = driver.switch_to.alert
print("Alert message:", alert.text)
alert.accept()

driver.quit()
```

- **Example (Handling a Confirmation Alert):**

```
# ... (setup driver) ...
trigger_button = driver.find_element(By.ID, "confirm_alert_button")
trigger_button.click()

alert = driver.switch_to.alert
print("Confirmation message:", alert.text)
alert.dismiss() # Or alert.accept() to click OK
# ... (rest of the code) ...
```

- **Example (Handling a Prompt Alert):**

```
# ... (setup driver) ...
trigger_button = driver.find_element(By.ID, "prompt_alert_button")
trigger_button.click()

alert = driver.switch_to.alert
print("Prompt message:", alert.text)
alert.send_keys("Selenium Input")
alert.accept()
# ... (rest of the code) ...
```

Task:

1. Using Selenium WebDriver in Python, to Trigger a JavaScript prompt alert then Send the text "Test User" to the alert then Accept the alert finally Print the alert message before entering text.
2. Write a Selenium script to Wait explicitly until an alert is present (up to 10 seconds).Then Print the text of the alert. Accept it.

- **Note:** Use below given link to perform actions in Task 1 and 2.
  "https://www.selenium.dev/documentation/webdriver/interactions/alerts/

3. Research and implement a **Fluent Wait** strategy using Selenium and WebDriverWait with polling intervals to wait for a changing element text.
4. Write a Python Selenium script that sets an implicit wait, opens the Selenium dynamic elements demo page ("https://www.selenium.dev/selenium/web/dynamic.html"), clicks the "adder" button

to add a new element, then locates and prints the tag names of both the newly added element (box0) and the "adder" button.