APEXAIQ

Day3 of Internship

- **Introduction**

This document gives an overview of web scraping, web browsers, and Selenium. Web scraping is the process of automatically extracting data from websites. Web browsers like Chrome and Firefox are used to access these sites, while Selenium helps automate browser actions such as clicking, scrolling, and collecting data. Together, they make it easy to gather information from dynamic web pages efficiently.

- **Research Topics**

Web Scrapping

Web scraping is an automatic method to obtain large amounts of data from websites. Most of this data is unstructured data in an HTML format which is then converted into structured data in a spreadsheet or a database so that it can be used in various applications. There are many different ways to perform web scraping to obtain data from websites. These include using online services, particular API's or even creating your code for web scraping from scratch. Many large websites, like Google, Twitter, Facebook, StackOverflow, etc. have API's that allow you to access their data in a structured format. This is the best option, but there are other sites that don't allow users to access large amounts of data in a structured form or they are simply not that technologically advanced. In that situation, it's best to use Web Scraping to scrape the website for data.

How Web Scrapers Work?

Web Scrapers can extract all the data on particular sites or the specific data that a user wants. Ideally, it's best if you specify the data you want so that the web scraper only extracts that data quickly. For example, you might want to scrape an Amazon page for the types of juicers available, but you might only want the data about the models of different juicers and not the customer reviews. So, when a web scraper needs to scrape a site, first the URLs are provided. Then it loads all the HTML code for those sites and a more advanced scraper might even extract all the CSS and Javascript elements as well. Then the scraper obtains the required data from this HTML code and outputs this data in the format specified by the user. Mostly, this is in the form of an Excel spreadsheet or a CSV file, but the data can also be saved in other formats, such as a JSON file.

Why is Python a Popular Programming Language for Web Scraping?

Python seems to be in fashion these days! It is the most popular language for web scraping as it can handle most of the processes easily. It also has a variety of libraries that were created specifically for Web Scraping. Scrapy is a very popular open-source web crawling framework that is written in Python. It is ideal for web scraping as well as extracting data using APIs. Beautiful soup is another Python library that is highly suitable for Web Scraping. It creates a parse tree that can be used to extract data from HTML on a website. Beautiful soup also has multiple features for navigation, searching, and modifying these parse trees.

What is Web Scraping Used for?

1. Price Monitoring

2. Market Research

3. News Monitoring

4. Sentiment Analysis

.5. Email Marketing

Source: https://www.geeksforgeeks.org/blogs/what-is-web-scraping-and-how-to-use-it/

Tools for web scrapping

Selenium

Selenium is an open-source automation tool primarily used for testing web applications. It mimics the actions of a real user interacting with a website, making it an excellent choice for scraping dynamic pages that rely heavily on JavaScript.Unlike static HTML pages, where data can be easily retrieved using traditional scraping methods like BeautifulSoup or Scrapy,

dynamic pages require a more robust solution to render and interact with the content — this is Selenium's strength.

Why Use Selenium for Web Scraping?

Handling JavaScript

User Interaction Simulation

Headless Browsing

Setting Up Selenium

Before diving into examples, you need to set up Selenium in your Python environment. Here's a quick guide:

Install Selenium:

```
pip install selenium
```

Download a WebDriver:

Selenium requires a WebDriver to interact with browsers. WebDrivers are specific to each browser (e.g., ChromeDriver for Google Chrome, GeckoDriver for Firefox).

Setting Up the WebDriver:

After downloading, ensure that the WebDriver is accessible through your system's PATH. Alternatively, you can specify the WebDriver's path directly in your script.

Basic Web Scraping Example

Now, let's dive into a basic example where we'll scrape some data from a website using Selenium.

Step 1: Import the Required Libraries

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

Step 2: Set Up the WebDriver

```
# Make sure to replace 'path/to/chromedriver' with the actual path to your ChromeDriver
driver = webdriver.Chrome(executable_path='/path/to/chromedriver')
```

Step 3: Open the Web Page

```
driver.get("https://example.com")
```

Step 4: Interact with the Web Page

```
# Let's assume we want to scrape all article titles from a blog page
titles = driver.find_elements(By.CLASS_NAME, 'article-title')
for title in titles:
print(title.text)
```

Step 5: Close the Browser

```
driver.quit()
```

This simple script demonstrates how to open a web page, locate elements by their class name, and extract text from them.

Scraping Data After Scrolling

```
from selenium.webdriver.common.keys import Keys
# Scroll down the page
driver.find_element(By.TAG_NAME, 'body').send_keys(Keys.END)
# Wait for content to load
import time
time.sleep(2) # Adjust the sleep time based on the website's loading speed
# Scrape the newly loaded content
new_content = driver.find_elements(By.CLASS_NAME, 'new-content-class')
for item in new_content:
print(item.text)
```

Handling Form Submissions and Button Clicks

```
# Locate the input fields and submit button
username = driver.find_element(By.NAME, 'username')
password = driver.find_element(By.NAME, 'password')
submit_button = driver.find_element(By.ID, 'submit')
# Enter data into the form fields
username.send_keys("myUsername")
password.send_keys("myPassword")
# Click the submit button
submit_button.click()
# Wait for the next page to load
time.sleep(3)
# Scrape data from the next page
result = driver.find_element(By.ID, 'result')
print(result.text)
```

Headless Browsing for Faster Scraping

```
from selenium.webdriver.chrome.options import Options
options = Options()
options.headless = True
driver = webdriver.Chrome(executable_path='/path/to/chromedriver', options=options)
```

Source: https://medium.com/@datajournal/web-scraping-with-selenium-955fbaae3421

WebDriver

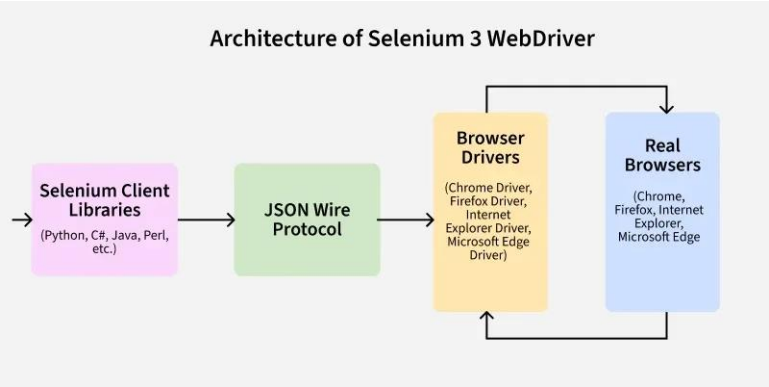
- ChromeDriver is used to automate the Google Chrome browser. It communicates with Chrome through the Chrome DevTools Protocol and must match the installed browser version.
- GeckoDriver is used for automating Mozilla Firefox. It connects Selenium with Firefox's Gecko engine and follows the W3C WebDriver standard.
- EdgeDriver is used to control Microsoft Edge, which is based on the Chromium engine, making it similar to ChromeDriver. The driver version should match the Edge browser version.
- SafariDriver is used to automate Apple's Safari browser. It comes pre-installed on macOS, and the "Allow Remote Automation" option must be enabled in Safari's Develop menu.

Multibrowser

Multibrowser refers to the capability of running a web application or automation script on multiple web browsers to ensure it works correctly on all of them. Different browsers (like Chrome, Firefox, Edge, Safari) may render websites differently, so multibrowser testing helps verify compatibility, functionality, and appearance across all browsers.

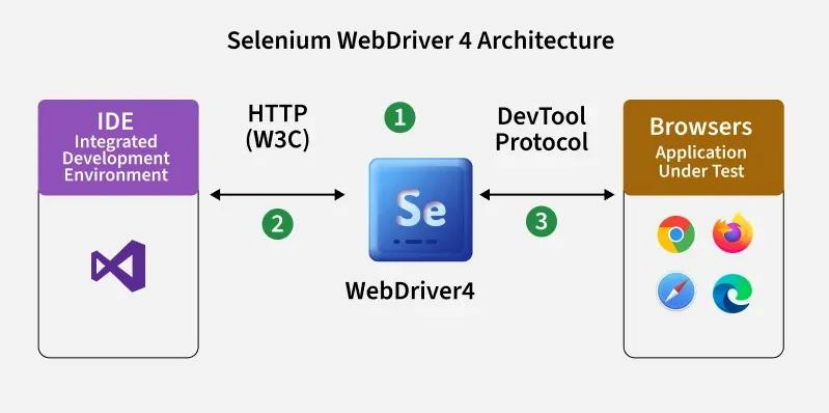
In automation tools like Selenium, multibrowser testing means using different WebDrivers (ChromeDriver, GeckoDriver, EdgeDriver, SafariDriver) to execute the same test scripts on multiple browsers without rewriting the code for each one.

Selenium Architecture



1. **Selenium Client Library**: This component provides language-specific bindings or APIs that allow users to write test scripts and interact with the WebDriver.
2. **JSON Wire Protocol over HTTP**: The JSON Wire Protocol is a standardized protocol used for communication between the Selenium Client Library and the Browser Drivers. It defines a set of commands and responses in JSON format exchanged over HTTP requests.

3. **Browser Drivers**: These are executable files that establish a communication channel between the WebDriver and the actual web browsers, such as Chrome, Firefox, Safari, etc. Each browser requires its specific driver, like ChromeDriver, GeckoDriver, to enable WebDriver to control and automate browser actions.
4. **Real Browsers**: These are web browsers like Google Chrome, Mozilla Firefox, Microsoft Edge, etc., where the actual testing and automation take place. The WebDriver interacts with these browsers through their respective browser drivers to perform actions like clicking elements, filling forms, navigating pages, and validating content.



1. **Selenium Client Library:** This component provides language-specific bindings or APIs (e.g., Java, Python, Ruby) that allow users to write test scripts and interact with the WebDriver.
2. **WebDriver W3C Protocol:** WebDriver is a protocol that provides a standard way for web browsers to communicate with an automation script. In Selenium 4, it focuses on W3C WebDriver Protocol, for better consistency and compatibility across different browsers.
3. **Browser Drivers:** These are executable files that establish a communication channel between the WebDriver and the actual web browsers such as Chrome, Firefox, Safari, etc. Each browser requires its specific driver (e.g., ChromeDriver, GeckoDriver, etc.) to enable WebDriver to control and automate browser actions.
4. **Real Browsers:** These are web browsers like Google Chrome, Mozilla Firefox, Microsoft Edge, etc., where the actual testing and automation take place. The WebDriver interacts with these browsers through their respective browser drivers to perform actions like clicking elements, filling forms, navigating pages, and validating content.

Xpath

XPath (XML Path Language) is a language used to locate elements on a web page in XML or HTML documents. In web automation with Selenium, XPath helps identify elements like buttons, text boxes, links, or images so that you can interact with them (click, type, read text, etc.).

XPath can select elements based on:

- Tag name (e.g., //input)
- Attributes (e.g., //input[@id='username'])
- Text content (e.g., //button[text()='Submit'])
- Hierarchy or position (e.g., //div[2]/span)

There are two main types of XPath:

1. Absolute XPath – starts from the root of the document (e.g., /html/body/div[1]/input)
2. Relative XPath – starts from anywhere in the document and is more flexible (e.g., //input[@name='email'])

```
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("https://example.com/login")

# Using XPath to find the username input box

username_box = driver.find_element("xpath", "//input[@id='username']")

username_box.send_keys("myusername")

# Using XPath to find the login button

login_button = driver.find_element("xpath", "//button[text()='Login']")

login_button.click()

driver.quit()
```

Regex

Regex (short for Regular Expression) is a pattern-matching tool used to search, match, or manipulate text. It helps find specific strings, validate inputs, or extract information from text in a flexible way.

For example, you can use regex to:

- Check if an email is valid

- Find all phone numbers in a document
- Extract numbers from a string

CSV

CSV stands for Comma Separated Values. It is a plain text file format for storing data in a table-like structure. Each line in a CSV file represents a data record, and a delimiter, typically a comma, separates each field in that record. CSV files have a .csv extension and are widely used to transfer data between different programs, especially those that require a structured data format, like databases or spreadsheets.

Example of CSV Data

Here is an example of how data is represented in a CSV file:

```
Name, Score, Department
Alex, 528, IT
Mallika, 650, Commerce
Joy, 670, Humanities
Yash, 679, IT
```