WebScraper Report

Overview

The WebScraper class is designed to automate the process of searching for flights on the website "cheapflights.ca" using Selenium WebDriver and subsequently scrape flight data. The scraped data includes flight times, prices, and companies, which are then saved into a CSV file. The class also parses the webpage content to generate a vocabulary file and performs some basic autocomplete suggestions using a Red-Black Binary Search Tree (BST).

Website parsed

https://www.cheapflights.ca/

Code Explanation

Imports

The class imports several libraries necessary for web scraping, browser automation, HTML parsing, and data manipulation:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;

import java.io.*;
import java.time.Duration;
import java.util.*;
```

Main Method

The main method orchestrates the entire scraping process:

1. **Setup WebDriver**: Initializes the Chrome WebDriver and opens the "cheapflights.ca" website.
2. **Flight Search**: Uses the SearchPage class to input flight details and perform a search.
3. **Scraping Flight Data**: Uses the ScrapeFlights class to scrape the resulting flight data.
4. **Data Parsing and Storage**: Parses the webpage content, writes data to a CSV file, and updates a vocabulary file.
5. **Autocomplete Suggestions**: Uses a Red-Black BST to provide top word suggestions based on a given prefix.

Setup WebDriver

```
System.setProperty("webdriver.chrome.driver","C:\\Users\\macon\\Downloads\\
chromedriver-win64\\chromedriver-win64\\chromedriver.exe");

// Initialize ChromeDriver
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();

// Navigate to the website
driver.get("https://www.cheapflights.ca/");
```

This code sets up the ChromeDriver path, initializes the WebDriver, maximizes the browser window, and navigates to the target website.

Search Flights

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

// Get flight origin input field
WebElement inputField1 =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector("[a
ria-label='Flight origin input']")));

// Get the page source
String pageSource = driver.getPageSource();

// Parse the page source and update vocabulary file
parsePage(pageSource, "vocabulary1.txt");
parseFile("vocabulary1.txt");

// Initialize SearchPage object and perform search
SearchPage searchPage = new SearchPage(driver);
searchPage.searchFlights();
```

This segment waits for the flight origin input field to be visible, retrieves the page source, parses it, and uses the SearchPage class to perform a flight search.

Switch to New Tab

```
// Switch to the new tab for search results
List<String> windowHandlesList = new ArrayList<>(allWindowHandles);
String secondWindowHandle = null;
if (windowHandlesList.size() > 1) {
    secondWindowHandle = windowHandlesList.get(1);
    driver.switchTo().window(secondWindowHandle);
}
```

This code switches to the new tab that opens after the search button is clicked.

Scrape Flight Data

```
ScrapeFlights scrapeFlights = new ScrapeFlights(driver);
scrapeFlights.scrapeFlights();

// Write flight data to CSV file
writeListsToCSV("flight_data.csv", scrapeFlights.getArrivingFlightTimes(),
scrapeFlights.getDepartingFlightTimes(), scrapeFlights.getFlightPrices(),
scrapeFlights.getFlightCompanies());
```

The ScrapeFlights class is used to scrape flight data, which is then written to a CSV file.

Update Vocabulary and Autocomplete

```
// Get the updated page source
pageSource = driver.getPageSource();

// Parse the updated page source and update vocabulary file
parsePage(pageSource, "vocabulary2.txt");
parseFile("vocabulary2.txt");

// Initialize Red-Black BST
RedBlackBST<String, Integer> redBlackBstTree = new RedBlackBST<>();

// Populate Red-Black BST with unique words and their counts
for (Map.Entry<String, Integer> entry : uniqueWordsMap1.entrySet()) {
    redBlackBstTree.put(entry.getKey(), entry.getValue());
}

// Get top suggestions for a given prefix
String prefix = "ai";
System.out.println("-----Top suggested words for prefix " + prefix + "---
");
for (String key : getTopSuggestions(redBlackBstTree.autoComplete(prefix)))
{
    System.out.println(key + ":" + redBlackBstTree.get(key));
}
```

The webpage is parsed again, the vocabulary file is updated, and a Red-Black BST is used to provide autocomplete suggestions based on a given prefix.

Helper Methods

- **writeListsToCSV**: Writes multiple lists to a CSV file.
- **parseWholeWebPage**: Parses the whole webpage and counts word occurrences.
- **getTopSuggestions**: Gets top suggestions based on frequency.
- **parsePage**: Parses the page source and writes text content to a file.
- **parseFile**: Parses a file and updates the unique words map.

SearchPage Report

Overview

The SearchPage class is responsible for interacting with the search functionality on the flight booking website. It inputs the flight origin and destination details and performs the search operation.

Code Explanation

Imports

The class imports libraries necessary for web interaction and waiting for elements:

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;
```

Class Variables

```
// WebDriver instance to control the browser
private WebDriver driver;

// WebDriverWait instance to wait for elements to be visible
private WebDriverWait wait;
```

These variables hold the WebDriver instance for controlling the browser and WebDriverWait instance for waiting until elements are visible.

Constructor

```
public SearchPage(WebDriver driver){
    this.driver = driver;
    this.wait = new WebDriverWait(driver, Duration.ofSeconds(10));
}
```

The constructor initializes the WebDriver and WebDriverWait instances.

searchFlights Method

```
public void searchFlights() throws InterruptedException {
    // Wait for the flight origin input field to be visible
    WebElement inputField1 =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector("[a
ria-label='Flight origin input']")));

    // Enter the flight origin as "DXB" (Dubai), navigate through the
suggestions, and select the first one
    inputField1.sendKeys("DXB");
    inputField1.sendKeys(Keys.ARROW_DOWN);
    Thread.sleep(500);
    inputField1.sendKeys(Keys.RETURN);

    // Wait for the flight destination input field to be visible
    WebElement inputField2 =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector("[a
ria-label='Flight destination input']")));

    // Enter the flight destination as "AMD" (Ahmedabad), navigate through
the suggestions, and select the first one
    inputField2.sendKeys("AMD");
    inputField2.sendKeys(Keys.ARROW_DOWN);
    Thread.sleep(500);
    inputField2.sendKeys(Keys.RETURN);
```

```
    // Find the search button and click it to initiate the search
    WebElement search =
driver.findElement(By.xpath("//button[@type='submit']"));
    search.click();

    // Wait for a second to ensure the new tab has opened
    Thread.sleep(1000);
}
```

This method performs the following steps:

1. Waits for the flight origin input field to be visible.
2. Enters "DXB" for Dubai, navigates through suggestions, and selects the first one.
3. Waits for the flight destination input field to be visible.
4. Enters "AMD" for Ahmedabad, navigates through suggestions, and selects the first one.
5. Finds the search button and clicks it.
6. Waits for a second to ensure the new tab has opened.

ScrapeFlights Report

Overview

The ScrapeFlights class scrapes flight data such as flight times, prices, and companies from the search results page.

Code Explanation

Imports

The class imports libraries necessary for web interaction and waiting for elements:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
```

These variables hold the WebDriver instance, WebDriverWait instance, and lists to store flight data.

Constructor

```
public ScrapeFlights(WebDriver driver){
    this.driver = driver;
```

```
    this.wait = new WebDriverWait(driver, Duration.ofSeconds(10));
}
```

The constructor initializes the WebDriver and WebDriverWait instances.

scrapeFlights Method

```
public void scrapeFlights(){
    // Get all window handles
    Set<String> allWindowHandles = driver.getWindowHandles();
    List<String> windowHandlesList = new ArrayList<>(allWindowHandles);

    // Switch to the second window if it exists
    if (windowHandlesList.size() > 1) {
        String secondWindowHandle = windowHandlesList.get(1);
        driver.switchTo().window(secondWindowHandle);
    }

    // Wait for the flight data table to be visible
    WebElement flightDataTable =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector(".c
-OCp")));

    // Fetch corresponding departing flight elements
    List<WebElement> departingFlightTimes =
flightDataTable.findElements(By.cssSelector("div.vmXl.vmXl-mod-variant-
large > span:nth-child(1)"));

    // Fetch corresponding arriving flight elements
    List<WebElement> arrivingFlightTimes =
flightDataTable.findElements(By.cssSelector("div.vmXl.vmXl-mod-variant-
large > span:nth-child(3)"));

    // Fetch corresponding flight prices elements
    List<WebElement> flightPrices =
flightDataTable.findElements(By.cssSelector(".f8F1-price-text"));

    // Fetch corresponding flight company elements
    List<WebElement> flightCompanies =
flightDataTable.findElements(By.cssSelector(".J0g6-operator-text"));

    // Lists to store the text values of the elements
    List<String> departingFlightTimesList = new ArrayList<>();
    List<String> arrivingFlightTimesList = new ArrayList<>();
    List<String> flightPricesList = new ArrayList<>();
    List<String> flightCompaniesList = new ArrayList<>();

    // Test data elements for flight times
    List<WebElement> testData =
flightDataTable.findElements(By.cssSelector(" .hJSA-list > li.hJSA-
item:nth-child(1) "));

    // Print and store departing flight times
    System.out.println("\n\n-----DEPARTING FLIGHTS TIMINGS------");
    for (WebElement element : testData) {
        WebElement departingEle =
element.findElement(By.cssSelector("div.vmXl.vmXl-mod-variant-large >
span:nth-child(1) "));
        System.out.println(departingEle.getText());
```

```
            departingFlightTimesList.add(departingEle.getText());
        }
```

This method performs the following steps:

1. Switches to the new tab with flight search results.
2. Waits for the flight data table to be visible.
3. Scrapes departing times, arriving times, prices, and company names from the search results.
4. Stores the scraped data in the respective lists.

Getters

```
    // Getter methods to retrieve the flight data
    public List<String> getArrivingFlightTimes() {
        return arrivingFlightTimes;
    }

    public List<String> getFlightCompanies(){
        return flightCompanies;
    }

    public List<String> getFlightPrices() {
        return flightPrices;
    }

    public List<String> getDepartingFlightTimes() {
        return departingFlightTimes;
    }
}
```

These methods return the scraped data lists.

Helper Methods

The ScrapeFlights class doesn't include additional helper methods within the provided code snippet, focusing primarily on scraping and returning flight data.

Red-Black BST

Overview

The RedBlackBST class implements a Red-Black Binary Search Tree (BST) to manage the vocabulary and provide autocomplete suggestions.

Code Explanation

```
    public Map<String,Integer> autoComplete(String prefix) {
        return autoCompleteRec(root, prefix);
    }

    private Map<String,Integer> autoCompleteRec(Node x, String prefix) {
        Map<String,Integer> autoCompleteMap=new HashMap<>();
```

```
        if (x == null) {
            return autoCompleteMap;
        }

        if (x.key.toString().startsWith(prefix)) {
            autoCompleteMap.put(x.key.toString(), (Integer) x.val);
//              System.out.println(x.key);
        }

        if (prefix.compareTo(x.key.toString()) <= 0) {
            autoCompleteMap.putAll(autoCompleteRec(x.left, prefix));
//              autoCompleteList.addAll(autoCompleteRec(x.left, prefix));
        }

        autoCompleteMap.putAll(autoCompleteRec(x.right, prefix));

        return autoCompleteMap;
    }
```

This inner class defines a Node in the Red-Black BST, including its key, value, color, left and right children, and size.

BST Methods

The class implements various BST methods for insertion, deletion, search, and balancing, including:

- put: Inserts a key-value pair into the BST.
- get: Retrieves the value associated with a given key.
- size: Returns the size of the BST.
- isRed: Checks if a node is red.
- rotateLeft: Rotates a subtree to the left.
- rotateRight: Rotates a subtree to the right.
- flipColors: Flips the colors of a node and its children.
- balance: Balances the subtree after insertion or deletion.

Autocomplete Feature

The RedBlackBST class can also provide autocomplete suggestions by:

1. **Collecting Keys with a Given Prefix**: Uses methods to traverse the BST and collect keys that match a given prefix.
2. **Sorting and Returning Top Suggestions**: Sorts the collected keys by their frequency and returns the top suggestions.

Usage Example

```
RedBlackBST<String, Integer> redBlackBstTree = new RedBlackBST<>();

// Populate Red-Black BST with unique words and their counts
for (Map.Entry<String, Integer> entry : uniqueWordsMap1.entrySet()) {
    redBlackBstTree.put(entry.getKey(), entry.getValue());
}
```

```
// Get top suggestions for a given prefix
String prefix = "ai";
System.out.println("-----Top suggested words for prefix " + prefix + "---
");
for (String key : getTopSuggestions(redBlackBstTree.autoComplete(prefix)))
{
    System.out.println(key + ":" + redBlackBstTree.get(key));
}
```

This example shows how to populate the Red-Black BST with words and their frequencies and how to get top suggestions for a given prefix.

Summary

The provided code combines web scraping with a Red-Black BST-based autocomplete feature. The WebScraper class automates flight searches and scrapes relevant data, while the RedBlackBST class manages the vocabulary and provides autocomplete suggestions. This approach efficiently combines data extraction and user interaction enhancement.