**Report: Craigslist Car Scraper Code Analysis**

**Gear Trybe Inc. - Python Developer Intern**

**Muhammad Haroon**

## Overview

The provided code implements a web scraper for Craigslist, targeting the San Francisco Bay Area (sfbay.craigslist.org). It’s designed to extract contact information for car sellers based on a given search query and seller type. The implementation uses Fast API for the web framework and Selenium for web scraping.

## Code Structure

The code is split into two main files:

1. main.py: Contains the Fast API application and endpoint definitions.
2. modules.py: Contains the core scraping functionality.

### main.py

This file sets up the Fast API application and defines the search endpoint.

Key components:

1. **Fast API Setup**: The code initializes a Fast API application.
2. **Seller Type Enum**: Defines an enumeration for seller types (all, owner, dealer).
3. **Search Endpoint**:
   * Path: /search
   * Method: GET
   * Parameters:
     + query: The search query string
     + seller type: The type of seller (default is “all”)
   * Functionality: Calls perform search function and returns the result or raises an HTTP Exception if an error occurs.

### modules.py

This file contains the core scraping logic using Selenium WebDriver.

Key components:

1. **Extract\_link function**:
   * Extracts the href attribute from the search result’s anchor tag.
   * Uses WebDriverWait to handle dynamic content loading.
   * Handles potential exceptions (NoSuchElementException, StaleElementReferenceException, TimeoutException).
2. **perform\_search function**:
   * Sets up Chrome WebDriver with specific options for headless browsing and performance optimization.
   * Constructs the search URL based on the provided query and seller type.
   * Performs the search and waits for results to load.
   * Uses ThreadPoolExecutor for parallel processing of search results.
   * Returns a dictionary containing the query, seller type, and list of result links (or an error message).

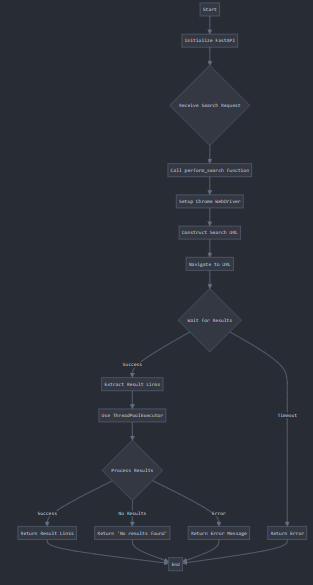
## Key Features and Techniques

1. **Headless Browsing**: The scraper uses Chrome in headless mode, which allows it to run without a visible browser window, improving performance and reducing resource usage.
2. **Dynamic Content Handling**: WebDriverWait is used to handle dynamically loaded content, ensuring that elements are present before attempting to interact with them.
3. **Parallel Processing**: ThreadPoolExecutor is employed to extract links from search results concurrently, significantly improving the scraper’s efficiency.
4. **Error Handling**: The code includes comprehensive error handling, catching and reporting various exceptions that may occur during the scraping process.
5. **Customizable Search**: The scraper allows for customization of the search query and seller type, making it flexible for different use cases.

## Next Steps

1. Fetching URLs of more than one result pages.
2. Further process of each result to extract information for the CRM.
3. Fetching contact information of the seller.

## Code Workflow



## Code

| # modules.py  from selenium import webdriver from selenium.webdriver.chrome.service import Service from selenium.webdriver.common.by import By from selenium.webdriver.chrome.options import Options from selenium.webdriver.support.ui import WebDriverWait from selenium.webdriver.support import expected\_conditions as EC from selenium.common.exceptions import TimeoutException, NoSuchElementException, StaleElementReferenceException from concurrent.futures import ThreadPoolExecutor, as\_completed  def extract\_link(result):  try:  return WebDriverWait(result, 2).until(  EC.presence\_of\_element\_located((By.CSS\_SELECTOR, "a.cl-app-anchor"))  ).get\_attribute('href')  except (NoSuchElementException, StaleElementReferenceException, TimeoutException):  return None  def perform\_search(query: str, seller\_type: str = 'all'):  chrome\_options = Options()  chrome\_options.add\_argument("--headless")  chrome\_options.add\_argument("--no-sandbox")  chrome\_options.add\_argument("--disable-dev-shm-usage")  chrome\_options.add\_experimental\_option("prefs", {  "profile.managed\_default\_content\_settings.images": 2,  })   try:  driver\_path = r"D:\Haroon\Career\Internships\Gear Trybe Inc\Day 2\chromedriver-win64\chromedriver.exe"  service = Service(driver\_path)  driver = webdriver.Chrome(service=service, options=chrome\_options)  except Exception as e:  return {"error": f"Error setting up ChromeDriver: {str(e)}"}    try:  base\_url = "https://sfbay.craigslist.org/search/sss?excats=5-2-13-22-26-1-26-1-1-1-3-6-11-1-5-8-1-1-1-1-1-4-1-7-1-10-2-2-2-1-1-1-1-1-1-2-3-1-1-2-2-1-1-2-1-2-1-1-1-1-1-1-3-1-1-1-1-1-4-1"    url = f"{base\_url}&{'purveyor=' + seller\_type + '&' if seller\_type in ['owner', 'dealer'] else ''}query={query}"    driver.get(url)    try:  WebDriverWait(driver, 5).until(  EC.presence\_of\_element\_located((By.CSS\_SELECTOR, "li.cl-search-result"))  )    results = WebDriverWait(driver, 5).until(  EC.presence\_of\_all\_elements\_located((By.CSS\_SELECTOR, "li.cl-search-result"))  )    with ThreadPoolExecutor(max\_workers=10) as executor:  future\_to\_link = {executor.submit(extract\_link, result): result for result in results}  result\_links = [future.result() for future in as\_completed(future\_to\_link) if future.result()]    if not result\_links:  return {"query": query, "seller\_type": seller\_type, "results": "No results found"}    return {"query": query, "seller\_type": seller\_type, "results": result\_links}    except TimeoutException:  return {"query": query, "seller\_type": seller\_type, "error": "Page took too long to load"}  except Exception as e:  return {"query": query, "seller\_type": seller\_type, "error": f"Error extracting results: {str(e)}"}    finally:  driver.quit() |
| --- |

| # main.py  from fastapi import FastAPI, Query, HTTPException from modules import perform\_search from enum import Enum  app = FastAPI()  class SellerType(str, Enum):  all = "all"  owner = "owner"  dealer = "dealer"  @app.get("/search") async def search(  query: str = Query(..., description="Search query"),  seller\_type: SellerType = Query(SellerType.all, description="Type of seller (all, owner, or dealer)") ):  result = perform\_search(query, seller\_type.value)  if "error" in result:  raise HTTPException(status\_code=500, detail=result["error"])  return result |
| --- |