Coles Scraper: Stealth & Environment Documentation

Overview

This Coles.com.au scraper is built using Selenium Wire, Smartproxy, BeautifulSoup, and MongoDB. It includes advanced anti-bot evasion techniques, such as rotating proxies, spoofing user agents, JavaScript-based stealth evasion (selenium-stealth), and realistic human-like interactions (e.g., delay, scrolls, click events).

Required Libraries and their Purpose

Library	Description	Purpose for this Project
selenium-wire	Extension of selenium	Captures requests and allows proxy configuration per session
selenium	Web automation library	Controls browser behavior programmatically
selenium-stealth	Anti-bot stealth utility	Masks WebDriver fingerprints and properties
fake-useragent	User-Agent generator	Randomizes headers to appear as real users
requests	HTTP Client	Handles cookies and requests outside Selenium
beautifulsoup4	HTML parser	Parses web content and extracts structured data
pymongo	MongoDB client	Saves collected data into MongoDB.
configparser	INI file handler	Loads configuration values from configuration.ini

json,os,random,time,datetime	Core Python	Utilities for handling delays, paths, data writing

chromedriver – WebDriver for Chrome

chromedriver is an executable required to automate and control the Chrome browser via Selenium.

How it fits in the project:

```
from seleniumwire.webdriver import Chrome
driver = Chrome(seleniumwire_options=proxy_options, options=options)
```

This command requires chromedriver to be installed and accessible via system PATH, or passed directly to the executable_path parameter.

Installation Instructions:

- Go to: https://chromedriver.chromium.org/downloads
- Download the version that matches your installed Chrome browser.
- Extract and place chromedriver in the same directory or somewhere on your system PATH.

Stealth and Anti-Bot Evasion Techniques

SmartProxy Proxies
Datacenter proxies are implemented from SmartProxy with location set to Australia. The current subscription gives us 100 IPs to use with 50 GB bandwidth.

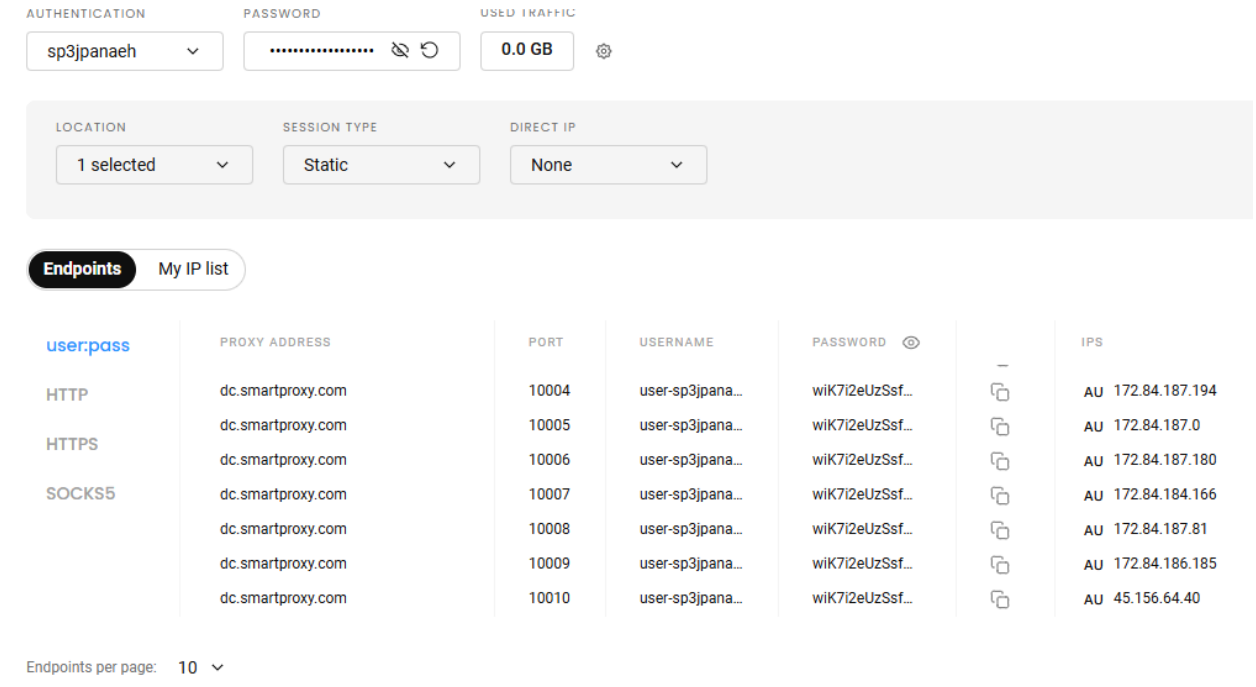
Proxy set up code snippet:

```
proxy_user = f"{proxy_user_base}-session-{session_id}"
proxy_options = {
    'proxy': {
        'http':
f"http://{proxy_user}:{proxy_pass}@{proxy_host}:{proxy_port}",
        'https':
f"http://{proxy_user}:{proxy_pass}@{proxy_host}:{proxy_port}",
        'no_proxy': 'localhost,127.0.0.1'
    },
    'verify_ssl': False
}
```

Smartproxy details are obtained from the configuration.ini file.

```
[Smartproxy]
Username = enter_username_here
Password = enter_password_here
Host = dc.smartproxy.com
Port = 10001,10002,10003,10004,10005,10006,10007
```

SmartProxy Setup Dashboard with Location set to Australia



A random `session_id` is used to **rotate proxy identity**. Switches between proxy ports and rotates users to simulate different sessions.

Fake User-Agent Spoofing
Fake User-Agent library is used to randomly obtain and select a user-agent on Windows OS. It helps to prevent detection based on consistent or headless-like user agents.

Code Snippet:

```
ua = UserAgent(browsers=['chrome', 'safari'], os='win',
fallback=fallback_ua)
user_agent = ua.random
options.add_argument(f"user-agent={user_agent}")
```

Browser fingerprint evasion via selenium-stealth
Code Snippet

```
stealth(driver,
    languages=["en-US", "en"],
    vendor="Google Inc.",
    platform="Win32",
    webgl_vendor="Intel Inc.",
    renderer="Intel Iris OpenGL Engine",
    fix_hairline=True,
    run_on_insecure_origins=True
)
```

Overrides key JavaScript-detectable values like navigator.language, navigator.vendor, navigator.platform, WebGLRenderingContext.vendor, and renderer

fix_hairline=True corrects rendering issues that bots typically cause.

Removes the navigator.webdriver flag via:

```
driver.execute_script("delete navigator.__proto__.webdriver")
```

Simulated Human Interaction
Code Snippet:

```
def human_delay(min_sec=1.5, max_sec=3.5, scroll=False):
    if random.random() < 0.8:
        time.sleep(random.uniform(min_sec, max_sec))
    if scroll and random.random() < 0.4:
        scroll_px = random.choice([random.randint(100, 400),
random.randint(-150, -50), 0])
        try:
            driver.execute_script(f"window.scrollBy(0, {scroll_px})")
        except:
            pass
```

Adds randomized delays and scrolls to simulate a real user browsing.

CAPTCHA Handling
The captcha is manually handled now. Current logic detects if there is a captcha challenge and waits until the captcha is solved before continuing with the execution.

Code Snippet:

```
def is_captcha_present(driver):
    return "_incapsula_" in driver.current_url.lower() or ...
```

```
def wait_for_captcha(driver, timeout=120):
    try:
        print("CAPTCHA iframe detected. Waiting for it to disappear...")
        WebDriverWait(driver, timeout).until_not(
            EC.presence_of_element_located((By.XPATH,
"//iframe[contains(@src, '_Incapsula_Resource')]"))
        ) ...
```

It checks for _Incapsula_Resource iframes (used by Imperva). If captcha is found, it waits until we manually solve it.

Random Window Size

```
driver.set_window_size(random.randint(1000, 1400), random.randint(700,
1000))
```

The code above generates random window size every time a fresh session is created which mimics different devices.

Ethical Web Scraping

Respecting Robot.txt file

The scraper now reads and respects robots.txt from https://www.coles.com.au/robots.txt.
The function can_fetch(url) checks if the scraper is allowed to crawl that URL based on user-agent *.

Code Snippet

```
def can_fetch(url, user_agent='*'):
    rp = RobotFileParser()
    parsed_url = urlparse(url)
    robots_url = f'{parsed_url.scheme}://{parsed_url.netloc}/robots.txt'
    try:
        rp.set_url(robots_url)
        rp.read()
        return rp.can_fetch(user_agent, parsed_url.path)
    except Exception as e:
        print(f"Warning: Failed to parse robots.txt ({robots_url}):",
str(e))
        return False
```

Humanized Delays

As mentioned above under stealth measures, humanized delays are added so that we do not place too much stress on the server.

Realistic User-Agents with rotation

Realistic user-agents are used and rotated as mentioned above in stealth measures.

Proxy Implementation and rotation

Proxy is implemented which can be rotated each session.

Current Limitations

Although we have implemented advanced anti-bot evasion techniques, the current implementation can only scrape limited data from limited pages for all the categories. We are using a single session and IP to go through all categories as this is just a proof of concept for implementing stealth measures to bypass bot blocking which we need to extend in the next iterations. We need to rotate proxies as we go across different pages of all the categories. However, this should be done with more research and testing along with the use of residential/mobile IP pools as just randomly rotating proxies between pages will be instantly blocked.

Coles uses **Imperva Incapsula** to protect its content from being scraped, which is a big company with an estimated annual revenue of around $56.3M per year, backed with investment from **Thales**, and one of their **main business model** is to **block scrapers** and protect the content of their customers, especially prevent price comparisons in the context of supermarkets as the supermarkets use competitive pricing to their advantage - Incapsula invest heavily in the development of advanced techniques that detect web scraping patterns and block scrapers.

So, for an academic project with limited resources, scraping data on a large scale bypassing such security mechanisms may not be feasible. Although it is not impossible to scrape data from Coles, it will need significant research and development effort and budgeting. For example, subscription for residential/mobile IP pool rotation is more expensive than the datacenter proxy we are using, integration of paid CAPTCHA bypass APIs comes with their own costs, and to develop an architecture where the scraping logic can be distributed across multiple servers will come up with additional costs.

Future Work Areas

The current limitation of our scraper presents a lot of areas to research and work in the future. For example, we can perform extensive research around the behavioral patterns used by modern web application firewalls to protect the content and block bots. Based on that, we can enhance our script to work on our patterns while scraping data. Likewise, areas like implementation of residential/mobile proxies can be explored and analysed. Furthermore, integrations of APIs or machine learning to automatically solve CAPTCHAs instead of manual intervention can be explored. Moreover, additional stealth measures can be explored and the current techniques can be enhanced.