

How to Webscrape Using Python and Selenium

Lukas Fischer
SAFE Research Datacenter*

Last Update: November 26, 2024

This guide provides detailed instructions on how to set up and execute web scraping using Python and Selenium on the SAFE cluster where users do not have sudo rights. It walks through the installation of required Python packages, the configuration of ChromeDriver using `webdriver-manager`, and the use of advanced Chrome options to ensure compatibility with server environments.

Additionally, the guide includes methods to handle interrupted processes, save progress, and resume scraping tasks seamlessly. A section on SSH configuration, mandatory for cluster access, is also included.

Contents

1	Contact	1
2	Prerequisites and Setup	2
3	Installing Selenium and Required Packages	2
4	Setting Up ChromeDriver Without Sudo	2
5	Using Chrome Options	3
6	Handling Interrupted Processes and Saving Workspace	3

1 Contact

If you encounter any difficulties or just want general information, do not hesitate to contact us.

SAFE Research Datacenter: datacenter@safe-frankfurt.de

More information about the SAFE Research Datacenter, and further guides can be found [here](#), and [here](#).

*fischer@safe-frankfurt.de

2 Prerequisites and Setup

Before starting, ensure the following prerequisites are met:

- **Python Installation:** Python 3.8 or later is installed on the system.
- **Environment Access:** You have SSH access to an Ubuntu-based cluster.
- **Mandatory SSH Configuration:** To enable seamless cluster access, configure your SSH settings as follows:

1. Open your SSH config file:

```
nano ~/.ssh/config
```

2. Add the following configuration:

```
Host safe-gpu01.safe-frankfurt.de
  HostName safe-gpu01.safe-frankfurt.de
  User Your_HRZ_Username
  ForwardX11 no
  ForwardX11Trusted no
```

3. Save and exit. For Windows users, ensure the config file is located at:

```
C:\Users\YourName\.ssh\config
```

- **Understanding Headless Browsing:** Familiarize yourself with running web browsers in headless mode, which is crucial for server-based scraping.
- **Non-Sudo Environment:** This guide assumes no sudo rights.

3 Installing Selenium and Required Packages

Install Selenium and WebDriver Manager:

```
pip install selenium webdriver-manager
```

Key Packages Explained:

- **selenium:** Automates web browser actions, allowing interaction with web pages.
- **webdriver-manager:** Automatically downloads and configures ChromeDriver, removing the need for manual setup or sudo rights.

4 Setting Up ChromeDriver Without Sudo

Using `webdriver-manager`, ChromeDriver is managed entirely in user space. This is essential in environments where you lack root access.

Add the following imports to your script:

```
1 from selenium.webdriver.chrome.service import Service
2 from selenium.webdriver.chrome.options import Options
3 from selenium import webdriver
4 from webdriver_manager.chrome import ChromeDriverManager
```

5 Using Chrome Options

Configure your Selenium WebDriver with detailed options tailored for server environments:

```
1 options = Options()
2 options.add_argument("--headless") # Run in headless mode
3 options.add_argument("--disable-gpu") # Disable GPU usage
4 options.add_argument("--no-sandbox") # Bypass OS security model
5 options.add_argument("--disable-dev-shm-usage") # Overcome resource limits
6 options.add_argument("--remote-debugging-port=" + str(9222)) # Debugging
   port
7 options.add_argument("--disable-software-rasterizer") # Disable software
   rendering
8
9 driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()),
   options=options)
10 driver.get("https://example.com")
11 print(driver.title)
12 driver.quit()
```

Options Breakdown:

- `--headless`: Enables execution without a graphical user interface.
- `--disable-gpu`: Prevents GPU usage to improve compatibility and performance.
- `--no-sandbox`: Necessary for restricted server environments.
- `--disable-dev-shm-usage`: Addresses limited shared memory issues.
- `--remote-debugging-port`: Assigns a unique port for monitoring browser behavior.
- `--disable-software-rasterizer`: Avoids unnecessary software-based rendering, optimizing performance.

6 Handling Interrupted Processes and Saving Workspace

In environments where instances may be externally stopped or killed, it is critical to implement strategies to save progress and resume operations seamlessly.

Saving Progress

1. Log progress to a file:

```
1 import json
2
3 def save_progress(status_file, progress_data):
4     with open(status_file, 'w') as file:
5         json.dump(progress_data, file)
6
7 def load_progress(status_file):
8     try:
9         with open(status_file, 'r') as file:
10             return json.load(file)
11     except FileNotFoundError:
12         return {}
```

2. Example usage in scraping:

```

1 status_file = "progress.json"
2 progress_data = load_progress(status_file)
3 start_index = progress_data.get("last_index", 0)
4
5 urls_to_scrape = ["https://example1.com", "https://example2.com", "https
  ://example3.com"]
6
7 for i in range(start_index, len(urls_to_scrape)):
8     url = urls_to_scrape[i]
9     driver.get(url)
10    print(driver.title)
11
12    progress_data["last_index"] = i + 1
13    save_progress(status_file, progress_data)
14
15 driver.quit()
16

```

>>

Trade-Offs

While saving progress to a file allows resumption from the last processed task, it can slow down scraping, especially when large datasets need to be saved frequently. In such cases:

- Structure your scraping tasks to save intermediate results periodically.
- Avoid saving large raw data directly in progress files; instead, save summaries or checkpoints.