

---

## The maintained successor of se-scrapers is the general purpose crawling infrastructure

### Search Engine Scraper - se-scrapers



This node module allows you to scrape search engines concurrently with different proxies.

If you don't have extensive technical experience or don't want to purchase proxies, you can use my scraping service.

#### Table of Contents

- Installation
- Docker
- Minimal Example
- Quickstart
- Contribute
- Using Proxies
- Custom Scrapers
- Examples
- Scraping Model
- Technical Notes
- Advanced Usage
- Special Query String Parameters for Search Engines

Se-scrapers supports the following search engines: \* Google \* Google News \* Google News App version (<https://news.google.com>) \* Google Image \* Bing \* Bing News \* Infospace \* Duckduckgo \* Yandex \* Webcrawler

This module uses puppeteer and a modified version of puppeteer-cluster. It was created by the Developer of GoogleScraper, a module with 1800 Stars on Github.

#### Installation

You need a working installation of **node** and the **npm** package manager.

For example, if you are using Ubuntu 18.04, you can install node and npm with the following commands:

---

```
1 sudo apt update;
2
3 sudo apt install nodejs;
4
5 # recent version of npm
6 curl -sL https://deb.nodesource.com/setup_10.x -o nodesource_setup.sh;
7 sudo bash nodesource_setup.sh;
8 sudo apt install npm;
```

Chrome and puppeteer need some additional libraries to run on ubuntu.

This command will install dependencies:

```
1 # install all that is needed by chromium browser. Maybe not everything
   needed
2 sudo apt-get install gconf-service libasound2 libatk1.0-0 libc6
   libcairo2 libcups2 libdbus-1-3 libexpat1 libfontconfig1 libgcc1
   libgconf-2-4 libgdk-pixbuf2.0-0 libgl1.0-0 libgtk-3-0 libnspr4
   libpango-1.0-0 libpangocairo-1.0-0 libstdc++6 libx11-6 libx11-xcb1
   libxcb1 libxcomposite1 libxcursor1 libxdamage1 libxext6 libxfixes3
   libxi6 libxrandr2 libxrender1 libxss1 libxtst6 ca-certificates fonts-
   -liberation libappindicator1 libnss3 lsb-release xdg-utils wget;
```

Install **se-scraper** by entering the following command in your terminal

```
1 npm install se-scraper
```

If you **don't** want puppeteer to download a complete chromium browser, add this variable to your environment. Then this module is not guaranteed to run out of the box.

```
1 export PUPPETEER_SKIP_CHROMIUM_DOWNLOAD=1
```

## Docker Support

I will maintain a public docker image of se-scraper. Pull the docker image with the command:

```
1 docker pull tschachn/se-scraper
```

Confirm that the docker image was correctly pulled:

```
1 docker image ls
```

Should show something like that:

1	tschachn/se-scraper	latest	897e1aeeba78	21
	minutes ago	1.29GB		

---

You can check the latest tag here. In the example below, the latest tag is **latest**. This will most likely remain **latest** in the future.

Run the docker image and map the internal port 3000 to the external port 3000:

```
1 $ docker run -p 3000:3000 tschachn/se-scrafer:latest
2
3 Running on http://0.0.0.0:3000
```

When the image is running, you may start scrape jobs via HTTP API:

```
1 curl -XPOST http://0.0.0.0:3000 -H 'Content-Type: application/json' \
2 -d '{
3   "browser_config": {
4     "random_user_agent": true
5   },
6   "scrape_config": {
7     "search_engine": "google",
8     "keywords": ["test"],
9     "num_pages": 1
10  }
11 }'
```

Many thanks goes to slotix for his tremendous help in setting up a docker image.

## Minimal Example

Create a file named `minimal.js` with the following contents

```
1 const se_scrafer = require('se-scrafer');
2
3 (async () => {
4   let scrape_job = {
5     search_engine: 'google',
6     keywords: ['lets go boys'],
7     num_pages: 1,
8   };
9
10   var results = await se_scrafer.scrape({}, scrape_job);
11
12   console.dir(results, {depth: null, colors: true});
13 })();
```

Start scraping by firing up the command `node minimal.js`

## Quickstart

Create a file named `run.js` with the following contents

---

```
1 const se_scraper = require('se-scraper');
2
3 (async () => {
4     let browser_config = {
5         debug_level: 1,
6         output_file: 'examples/results/data.json',
7     };
8
9     let scrape_job = {
10         search_engine: 'google',
11         keywords: ['news', 'se-scraper'],
12         num_pages: 1,
13         // add some cool google search settings
14         google_settings: {
15             gl: 'us', // The gl parameter determines the Google country
16                       // to use for the query.
17             hl: 'en', // The hl parameter determines the Google UI
18                       // language to return results.
19             start: 0, // Determines the results offset to use, defaults
20                       // to 0.
21             num: 100, // Determines the number of results to show,
22                       // defaults to 10. Maximum is 100.
23         },
24     };
25
26     var scraper = new se_scraper.ScraperManager(browser_config);
27
28     await scraper.start();
29
30     var results = await scraper.scrape(scrape_job);
31
32     console.dir(results, {depth: null, colors: true});
33
34     await scraper.quit();
35 })();
```

Start scraping by firing up the command `node run.js`

## Contribute

I really help and love your help! However scraping is a dirty business and it often takes me a lot of time to find failing selectors or missing JS logic. So if any search engine does not yield the results of your liking, please create a **static test case** similar to this static test of google that fails. I will try to correct se-scraper then.

That's how you would proceed:

1. Copy the static google test case

- 
2. Remove all unnecessary testing code
  3. Save a search to file where se-scrapers does not work correctly.
  4. Implement the static test case using the saved search html where se-scrapers currently fails.
  5. Submit a new issue with the failing test case as pull request
  6. I will fix it! (or better: you submit a pull request directly)

## Proxies

**se-scrapers** will create one browser instance per proxy. So the maximal amount of concurrency is equivalent to the number of proxies plus one (your own IP).

```
1  const se_scrapers = require('se-scrapers');
2
3  (async () => {
4      let browser_config = {
5          debug_level: 1,
6          output_file: 'examples/results/proxyresults.json',
7          proxy_file: '/home/nikolai/.proxies', // one proxy per line
8          log_ip_address: true,
9      };
10
11     let scrape_job = {
12         search_engine: 'google',
13         keywords: ['news', 'scrapeulous.com', 'incolumitas.com', 'i
14             work too much', 'what to do?', 'javascript is hard'],
15         num_pages: 1,
16     };
17
18     var scraper = new se_scrapers.ScrapeManager(browser_config);
19     await scraper.start();
20
21     var results = await scraper.scrape(scrape_job);
22     console.dir(results, {depth: null, colors: true});
23     await scraper.quit();
24 })();
```

With a proxy file such as

```
1  socks5://53.34.23.55:55523
2  socks4://51.11.23.22:22222
```

This will scrape with **three** browser instance each having their own IP address. Unfortunately, it is currently not possible to scrape with different proxies per tab. Chromium does not support that.

---

## Custom Scrapers

You can define your own scraper class and use it within se-scrapers.

Check this example out that defines a custom scraper for Ecosia.

## Examples

- Reuse existing browser yields these results
- Simple example scraping google yields these results
- Scrape with one proxy per browser yields these results
- Scrape 100 keywords on Bing with multiple tabs in one browser produces this
- Inject your own scraping logic
- For the Lulz: Scraping google dorks for SQL injection vulnerabilities and confirming them.
- Scrape google maps/locations yields these results

## Scraping Model

**se-scrapers** scrapes search engines only. In order to introduce concurrency into this library, it is necessary to define the scraping model. Then we can decide how we divide and conquer.

**Scraping Resources** What are common scraping resources?

1. **Memory and CPU.** Necessary to launch multiple browser instances.
2. **Network Bandwidth.** Is not often the bottleneck.
3. **IP Addresses.** Websites often block IP addresses after a certain amount of requests from the same IP address. Can be circumvented by using proxies.
4. Spoofable identifiers such as browser fingerprint or user agents. Those will be handled by **se-scrapers**

**Concurrency Model** **se-scrapers** should be able to run without any concurrency at all. This is the default case. No concurrency means only one browser/tab is searching at the time.

For concurrent use, we will make use of a modified puppeteer-cluster library.

One scrape job is properly defined by

- 1 search engine such as [google](#)
- [M](#) pages

- 
- $N$  keywords/queries
  - $K$  proxies and  $K+1$  browser instances (because when we have no proxies available, we will scrape with our dedicated IP)

Then **se-scrafer** will create  $K+1$  dedicated browser instances with a unique ip address. Each browser will get  $N / (K+1)$  keywords and will issue  $N / (K+1) * M$  total requests to the search engine.

The problem is that puppeteer-cluster library does only allow identical options for subsequent new browser instances. Therefore, it is not trivial to launch a cluster of browsers with distinct proxy settings. Right now, every browser has the same options. It's not possible to set options on a per browser basis.

Solution:

1. Create a upstream proxy router.
2. Modify puppeteer-cluster library to accept a list of proxy strings and then pop() from this list at every new call to `workerInstance()` in <https://github.com/thomasdondorf/puppeteer-cluster/blob/master/src/Cluster.ts> I wrote an issue here. **I ended up doing this.**

## Technical Notes

Scraping is done with a headless chromium browser using the automation library puppeteer. Puppeteer is a Node library which provides a high-level API to control headless Chrome or Chromium over the DevTools Protocol.

If you need to deploy scraping to the cloud (AWS or Azure), you can contact me at [hire@incolumitas.com](mailto:hire@incolumitas.com)

The chromium browser is started with the following flags to prevent scraping detection.

```
1 var ADDITIONAL_CHROME_FLAGS = [  
2   '--disable-infobars',  
3   '--window-position=0,0',  
4   '--ignore-certifcate-errors',  
5   '--ignore-certifcate-errors-spki-list',  
6   '--no-sandbox',  
7   '--disable-setuid-sandbox',  
8   '--disable-dev-shm-usage',  
9   '--disable-accelerated-2d-canvas',  
10  '--disable-gpu',  
11  '--window-size=1920x1080',  
12  '--hide-scrollbar',  
13  '--disable-notifications',  
14 ];
```

Furthermore, to avoid loading unnecessary ressources and to speed up scraping a great deal, we instruct chrome to not load images and css and media:

---

```
1 await page.setRequestInterception(true);
2 page.on('request', (req) => {
3   let type = req.resourceType();
4   const block = ['stylesheet', 'font', 'image', 'media'];
5   if (block.includes(type)) {
6     req.abort();
7   } else {
8     req.continue();
9   }
10 });
```

**Making puppeteer and headless chrome undetectable** Consider the following resources:

- <https://antoinevastel.com/bot%20detection/2019/07/19/detecting-chrome-headless-v3.html>
- <https://intoli.com/blog/making-chrome-headless-undetectable/>
- <https://intoli.com/blog/not-possible-to-block-chrome-headless/>
- <https://news.ycombinator.com/item?id=16179602>

**se-scrafer** implements the countermeasures against headless chrome detection proposed on those sites.

Most recent detection counter measures can be found here:

- <https://github.com/paulirish/headless-cat-n-mouse/blob/master/apply-evasions.js>

**se-scrafer** makes use of those anti detection techniques.

To check whether evasion works, you can test it by passing `test_evasion` flag to the config:

```
1 let config = {
2   // check if headless chrome escapes common detection techniques
3   test_evasion: true
4 };
```

It will create a screenshot named `headless-test-result.png` in the directory where the scraper was started that shows whether all test have passed.

## Advanced Usage

Use **se-scrafer** by calling it with a script such as the one below.

```
1 const se_scraper = require('se-scrafer');
2
3 // those options need to be provided on startup
4 // and cannot give to se-scrafer on scrape() calls
```



```
5 let browser_config = {
6   // the user agent to scrape with
7   user_agent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
8               /537.36 (KHTML, like Gecko) Chrome/77.0.3835.0 Safari/537.36',
9   // if random_user_agent is set to True, a random user agent is
   chosen
10  random_user_agent: false,
11  // whether to select manual settings in visible mode
12  set_manual_settings: false,
13  // log ip address data
14  log_ip_address: false,
15  // log http headers
16  log_http_headers: false,
17  // how long to sleep between requests. a random sleep interval
   within the range [a,b]
18  // is drawn before every request. empty string for no sleeping.
19  sleep_range: '',
20  // which search engine to scrape
21  search_engine: 'google',
22  compress: false, // compress
23  // whether debug information should be printed
24  // level 0: print nothing
25  // level 1: print most important info
26  // ...
27  // level 4: print all shit nobody wants to know
28  debug_level: 1,
29  keywords: ['nodejs rocks'],
30  // whether to start the browser in headless mode
31  headless: true,
32  // specify flags passed to chrome here
33  chrome_flags: [],
34  // the number of pages to scrape for each keyword
35  num_pages: 1,
36  // path to output file, data will be stored in JSON
37  output_file: '',
38  // whether to also passthru all the html output of the serp pages
39  html_output: false,
40  // whether to return a screenshot of serp pages as b64 data
41  screen_output: false,
42  // whether to prevent images, css, fonts and media from being
   loaded
43  // will speed up scraping a great deal
44  block_assets: true,
45  // path to js module that extends functionality
46  // this module should export the functions:
47  // get_browser, handle_metadata, close_browser
48  // custom_func: resolve('examples/pluggable.js'),
49  custom_func: '',
50  throw_on_detection: false,
51  // use a proxy for all connections
   // example: 'socks5://78.94.172.42:1080'
```

---

```

52     // example: 'http://118.174.233.10:48400'
53     proxy: '',
54     // a file with one proxy per line. Example:
55     // socks5://78.94.172.42:1080
56     // http://118.174.233.10:48400
57     proxy_file: '',
58     // whether to use proxies only
59     // when this is set to true, se-scrafer will not use
60     // your default IP address
61     use_proxies_only: false,
62     // check if headless chrome escapes common detection techniques
63     // this is a quick test and should be used for debugging
64     test_evasion: false,
65     apply_evasion_techniques: true,
66     // settings for puppeteer-cluster
67     puppeteer_cluster_config: {
68         timeout: 30 * 60 * 1000, // max timeout set to 30 minutes
69         monitor: false,
70         concurrency: Cluster.CONCURRENCY_BROWSER,
71         maxConcurrency: 1,
72     }
73 };
74
75 (async () => {
76     // scrape config can change on each scrape() call
77     let scrape_config = {
78         // which search engine to scrape
79         search_engine: 'google',
80         // an array of keywords to scrape
81         keywords: ['cat', 'mouse'],
82         // the number of pages to scrape for each keyword
83         num_pages: 2,
84
85         // OPTIONAL PARAMS BELOW:
86         google_settings: {
87             gl: 'us', // The gl parameter determines the Google country
88                     // to use for the query.
89             hl: 'fr', // The hl parameter determines the Google UI
90                     // language to return results.
91             start: 0, // Determines the results offset to use, defaults
92                     // to 0.
93             num: 100, // Determines the number of results to show,
94                     // defaults to 10. Maximum is 100.
95         },
96         // instead of keywords you can specify a keyword_file. this
97         // overwrites the keywords array
98         keyword_file: '',
99         // how long to sleep between requests. a random sleep interval
100        // within the range [a,b]
101        // is drawn before every request. empty string for no sleeping.
102        sleep_range: '',

```

---

```
97      // path to output file, data will be stored in JSON
98      output_file: 'output.json',
99      // whether to prevent images, css, fonts from being loaded
100     // will speed up scraping a great deal
101     block_assets: false,
102     // check if headless chrome escapes common detection techniques
103     // this is a quick test and should be used for debugging
104     test_evasion: false,
105     apply_evasion_techniques: true,
106     // log ip address data
107     log_ip_address: false,
108     // log http headers
109     log_http_headers: false,
110   };
111
112   let results = await se_scraper.scrape(browser_config, scrape_config
113   );
114   console.dir(results, {depth: null, colors: true});
115 }());
```

Output for the above script on my machine.

## Query String Parameters

You can add your custom query string parameters to the configuration object by specifying a `google_settings` key. In general: `{{search engine}}_settings`.

For example you can customize your google search with the following config:

```
1  let scrape_config = {
2    search_engine: 'google',
3    // use specific search engine parameters for various search engines
4    google_settings: {
5      google_domain: 'google.com',
6      gl: 'us', // The gl parameter determines the Google country to
7               // use for the query.
8      hl: 'us', // The hl parameter determines the Google UI language
9               // to return results.
10     start: 0, // Determines the results offset to use, defaults to
11              // 0.
12     num: 100, // Determines the number of results to show, defaults
13              // to 10. Maximum is 100.
14   },
15 }
```