Table of contents

II. Objective

At the end of this article, you will be able to scrape any kind of JS-driven website with great speed while you avoid limitations.

You can download the project as zip or view it in GitHub.

III. Introduction

- **General Introduction**

As in 2017, about 94.5% of websites use Javascript and 35% of them require it. Even though web developers provide fallback to browsers that do not support Javascript, most websites today will not function unless Javascript is supported by the browser

- **Essential Difference Between Static Websites and JS-rendered ones**

Web scraping started off with making network requests to the server and parsing the returned html markup to get the required data. There are a lot of modules/libraries used in making requests, and we have the ones we use in parsing the returned html markup to get the data we want from a document.

A popular Node.js module for making HTTP requests isrequest-promise and a common module for parsing HTML markup is cheerio. With cheerio, you are able to use jQuery syntax to extract the data you want from an HTML document.

This style of scraping websites is straight-forward, direct, fast and very performant. But a web scraping professional will not rely on this style of making web scrapers because it is close to being archaic as a lot of websites today return a very minimal HTML markup so that the data you intend to extract are not in the markup.

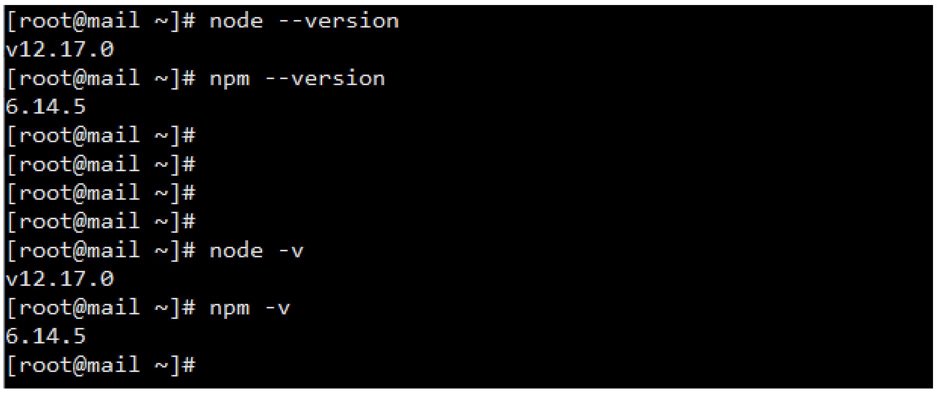
If you visit the same website with a Javascript-enabled web browser, the data is present. This is due to the fact that the data you want are being rendered with Javascript while the web scraper one makes cannot execute Javascript code. This article guides you through how to start building web crawlers that are able to execute Javascript code and expose an updated Document Object Model which contains data you want.

As a web bot developer, you need to up your game by learning to scrape Javascript-dependent websites effortlessly. Luckily, this guide teaches you all you need to be able to scrape any kind of website.

IV. Getting Started

- **Make sure Node.js and NPM are installed**

First check if your Node is installed by running*node -version* (shortcut: *node -v*) on the terminal (also called shell or command line but for the purpose of simplicity, let's stick to terminal throughout this article), and that npm (node package manager) is also installed by running *npm -version* (shortcut: *npm -v*). The two should output the current version of Node and npm you are running as shown below:



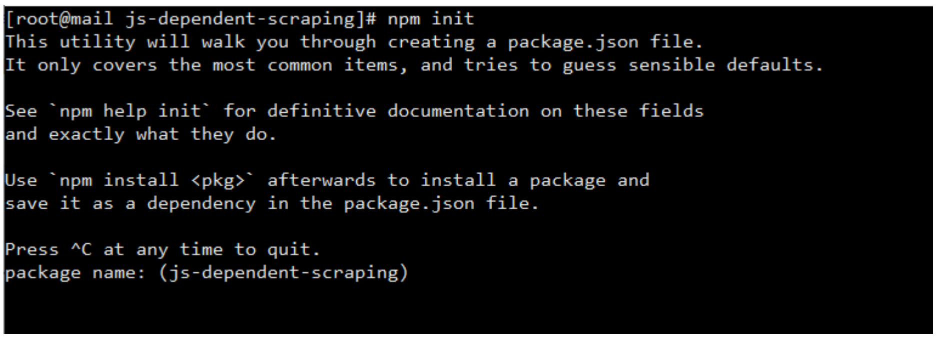
```
[root@mail ~]# node --version
v12.17.0
[root@mail ~]# npm --version
6.14.5
[root@mail ~]#
[root@mail ~]#
[root@mail ~]#
[root@mail ~]#
[root@mail ~]# node -v
v12.17.0
[root@mail ~]# npm -v
6.14.5
[root@mail ~]#
```

unless Node is not well installed on your version where you will now have to install it. Make sure you are running Node.js v12 or a later version. It is recommended to be running a stable release.

- **Create a folder and set it up for your web scraping project**

A good practice is to have a folder where all your web scraping projects are stored and you should name each folder the name of the project or rather follow a naming convention for your projects. The first step is to create a folder and navigate to it on the command line (also called *terminal* or *shell*). The folder I created for this project is *js-dependent-scrape*.

Then run ***npm init*** in your project directory and you will have an interactive screen waiting for your inputs like below:



```
[root@mail js-dependent-scraping]# npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (js-dependent-scraping)
```

As you can see, the package name is having*js-dependent-*

1. Launching a Headless Browser

We will be creating a function in the funcs/browser.js file, one launches a browser in headless mode and opens new pages. We will then export these functions so that other files can access it.

A simple function to launch the browser is:

```
const launchChrome = async () => {
  const puppeteer = require("puppeteer");

  const args = [
    "--disable-dev-shm-usage",
    "--no-sandbox",
    "--disable-setuid-sandbox",
    "--disable-accelerated-2d-canvas",
    "--disable-gpu",
    "--lang=en-US,en"
  ];  let chrome;
  try {
    chrome = await puppeteer.launch({
      headless: true, // run in headless mode
      devtools: false, // disable dev tools
      ignoreHTTPSErrors: true, // ignore https error
      args,
      ignoreDefaultArgs: ["--disable-extensions"],
    });
    return chrome;
  } catch(e) {
    console.error("Unable to launch chrome", e);
    return false;
  }
};
```

All the above does is launch Chrome, we need to create 2 functions inside this function such that our function returns an array of the 2 functions. The first function creates a new page and the second function exits the launched browser.

It is better to use the ***async/await*** *syntax (like you see above) when the next statement or expressions or functions depend on the current one.*

The **newPage** function:

VII. Numerical Results

Scrape speed comparison table

	Chromium headless instance*	HTTP requests
Setup time, ms	45000	5
Log-in time, ms	105000	13
1 page load time, ms	6	10

**based on TripAdvisor scrape*

Regular scraping without launching a Chromium instance seems slower to scrape the pages because it mostly involves making HTTP requests that download the whole HTML document. While headless scraping could make just AJAX (XmlHttp) requests to the server to get just the needed data (hotels in this context) without having to download other unnecessary data.

The scraper ran for approx. 3 mins. To scrape 30 hotel listings from the first page of TripAdvisor London hotels listings, it took approx. 3.55 mins and **193MB** of RAM on my local computer with (4.00GB of RAM and 1.30GHz CPU).

While for scraping the same amount of data, it ran for approx. 2.36 mins and **225MB** of RAM on a remote server (4.00GB of RAM and 2.40GHz CPU). Particular results depend on the available CPU, memory, and network speed.

Avg. time to launch the headless Chrome instance and sign in is **2.50 mins,** and the average scraping speed to scrape the 30 hotel listings off each page is approx. **1 page / 6 ms**.

While a regular scraping without launching a headless browser would take an average of **18 ms** to sign in, and the average speed would be **1 page / 10 ms**.