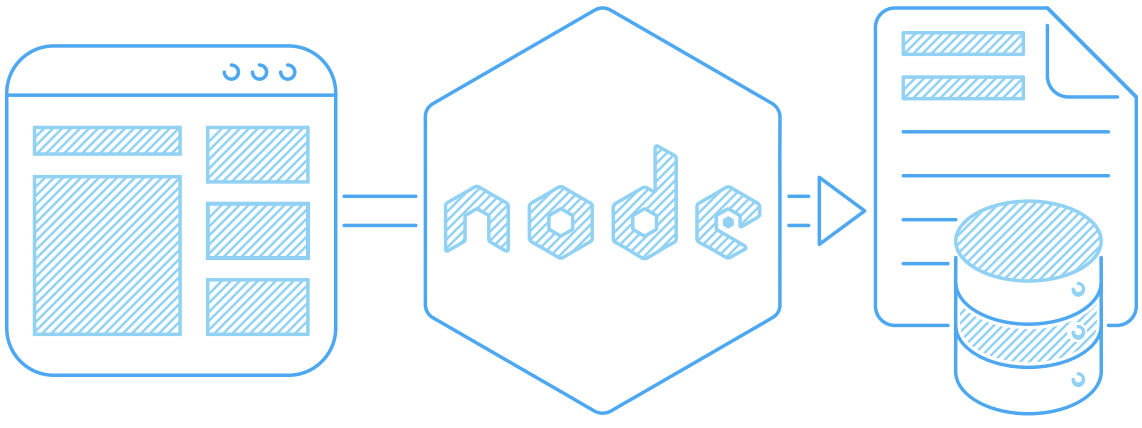
Web Scraping With NodeJS and Javascript

by Bernardas Ališauskas Aug 22, 2024 #NodeJS #HTTP #Data Parsing #Css Selectors #Scraping Introduction



Web scraping is mostly connection and data programming so using a web language for scraping seems like a natural fit, so can we scrape using javascript through nodejs runtime?

In this tutorial, we'll learn web scraping with NodeJS and Javascript. We'll cover an in-depth look at HTTP connections, HTML parsing, popular web scraping libraries and common challenges and web scraping idioms.

Finally, we'll finish everything off with an example web scraping project by scraping e-commerce products.

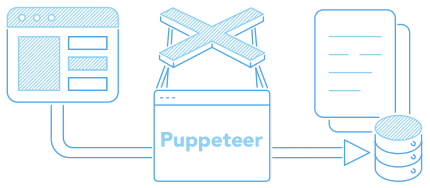
Overview and Setup

NodeJS in web scraping is mostly known because of Puppeteer browser automation toolkit. Using web browser automation for web scraping has a lot of benefits, though it's a complex and resource-heavy approach to JavaScript web scraping.

With a little reverse engineering and a few clever nodeJS libraries, we can achieve similar results without the entire overhead of a web browser!

Introduction to Puppeteer

For more on using browser automation with Puppeteer we have an entire introduction article that covers basic usage, best practices, tips and tricks and an example project!



In this article, we'll focus on scraping using HTTP clients and for this, we'll focus on two tools in particular:

- axios - HTTP client for retrieving web pages.
- cheerio - HTML parser for extracting data from HTML web pages.

Let's install them using these command line instructions:

```
$ mkdir nodejs-scraper
$ cd nodejs-scraper
$ npm install cheerio axios
```

Making Requests

Connection is a vital part of every web scraper and NodeJS has a big ecosystem of HTTP clients, though in this tutorial we'll be using the most popular one - axios.

HTTP in a Nutshell

To collect data from a public resource, we need to establish a connection with it first. Most of the web is served over HTTP. This protocol can be summarized as - client (our scraper) sends a request for a specific document and the server replies with the requested document or an error - a very straight-forward exchange:

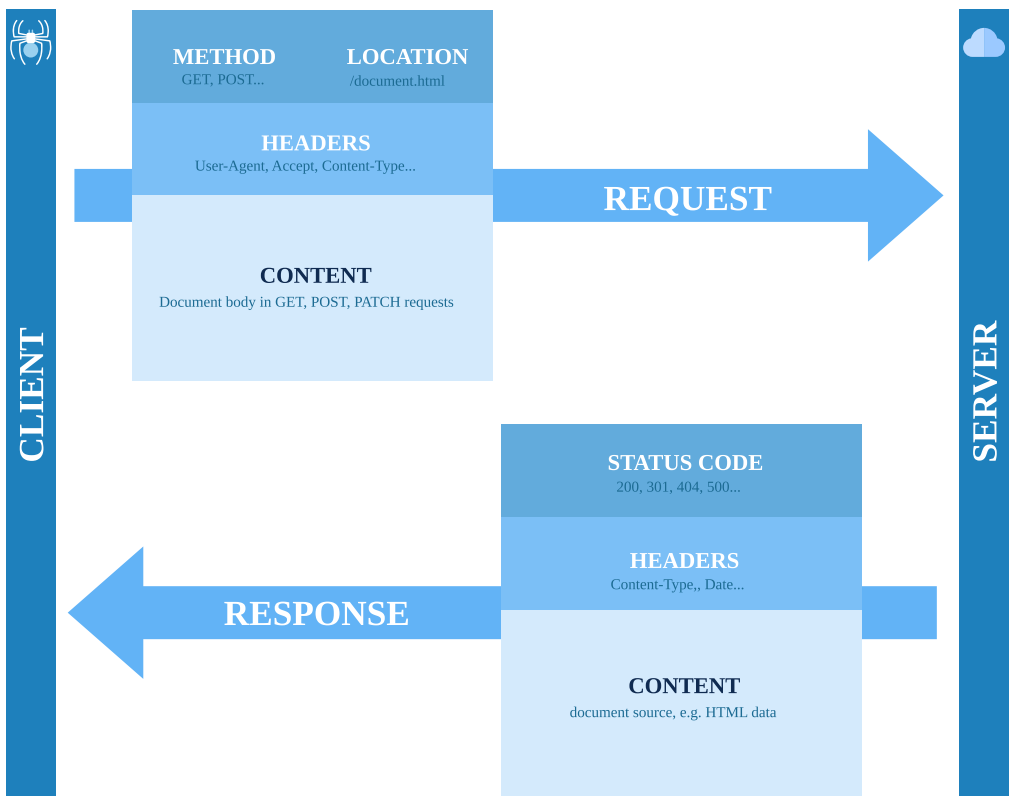


illustration of a standard http exchange

As you can see in this illustration: we send a request object which consists of method (aka type), location and headers. In turn, we receive a response object that consists of the status code, headers and document content itself.

In our axios example, this looks something like this:

```
import axios from 'axios';

// send request
response = await axios.get('https://httpbin.org/get');
// print response
console.log(response.data);
```

Though, for node js web scraping we need to know a few key details about requests and responses: method types, headers, cookies... Let's take a quick overview.

Request Methods

HTTP requests are conveniently divided into a few types that perform a distinct function. Most commonly in web scraping we use:

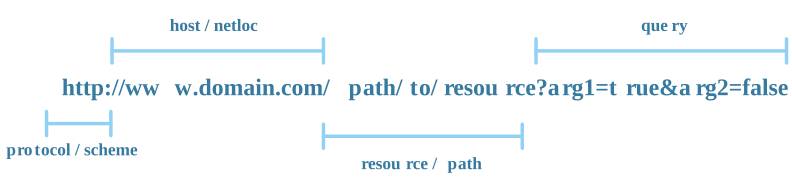
- `GET` requests a document - most commonly used method in scraping.
- `POST` sends a document to receive one. For example, this is used in form submissions like login, search etc.
- `HEAD` checks the state of a resources. This is mostly used to check whether a web page has updated it's contents as these type of requests are super fast.

Other methods aren't as commonly encountered but it's good to be aware of them nevertheless:

- `PATCH` requests are intended to update a document.
- `PUT` requests are intended to either create a new document or update it.
- `DELETE` requests are intended to delete a document.

Request Location - The URL

URL (Universal Resource Location) is the most important part of our request - it tells where our nodejs scraper should look for the resources. Though URLs can be quite complicated, let's take a look at how they are structured:



Example of a URL structure

Here, we can visualize each part of a URL:

- protocol - is either `http` or `https`.
- host - is the address/domain of the server.
- location - is the location of the resource we are requesting.
- parameters - allows customizing of a resource. For example `language=en` would give us the English version of the resource.

If you're ever unsure of a URL's structure, you can always fire up Node's interactive shell (`node` in the terminal) and let it figure it out for you:

```
$ node
> new URL("http://www.domain.com/path/to/resource?arg1=true&arg2=false")
URL {
  href: 'http://www.domain.com/path/to/resource?arg1=true&arg2=false',
  origin: 'http://www.domain.com',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'www.domain.com',
  hostname: 'www.domain.com',
  port: '',
  pathname: '/path/to/resource',
  search: '?arg1=true&arg2=false',
  searchParams: URLSearchParams { 'arg1' => 'true', 'arg2' => 'false' },
  hash: ''
}
```

Request Headers

Request headers indicate meta information about our request. While it might appear like request headers are just minor metadata details in web scraping, they are extremely important.

Headers contain essential details about the request, like who's requesting the data? What type of data they are expecting? Getting these wrong might result in a scraping error.

Let's take a look at some of the most important headers and what they mean:

User-Agent is an identity header that tells the server who's requesting the document.

```
# example user agent for Chrome browser on Windows operating system:
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.110 Safari/537.36
```

Whenever you visit a web page in your web browser identifies itself with a User-Agent string that looks something like "Browser Name, Operating System, Some version numbers".
User-agent helps the server to determine whether to serve or deny the client. When scraping we want to blend in to prevent being blocked so it's best to set user agent to look like that one of a browser.

> There are many online databases that contain latest user-agent strings of various platforms, like this Chrome user agent list bye whatismybrowser.com

Cookie is used to store persistent data. This is a vital feature for websites to keep track of user state: user logins, configuration preferences etc.

Accept headers (also Accept-Encoding, Accept-Language etc.) contain information about what sort of content we're expecting. Generally when web-scraping we want to mimic this of one of the popular web browsers, like Chrome browser use:

```
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
```

> For more, see list of default Accept header values

X- prefixed headers are special custom headers. These are important to keep an eye on when web scraping, as they might configure important functionality of the scraped website/webapp.

These are a few of the most important observations, for more see the extensive full documentation page: MDN HTTP Headers

Response Status Code

Once we send our request we'll eventually receive a response and the first thing we'll notice is the status code. Status codes indicate whether the request succeeded, failed or needs something more (like authentication/login).

Let's take a quick look at the status codes that are most relevant to web scraping:

- 200 range codes generally mean success!
- 300 range codes tend to mean redirection. In other words, if we request page at `/product1.html` it might be moved to a new location like `/products/1.html`.
- 400 range codes mean the request is malformed or denied. Our node web scraper could be missing some headers, cookies or authentication details.
- 500 range codes typically mean server issues. The website might be unavailable right now or is purposefully disabling access to our web scraper.

For more on http status codes, see documentation: HTTP Status definitions by MDN

Response Headers

The next thing we notice about our response is the metadata - also known as headers. When it comes to web scraping, response headers provide some important information for connection functionality and efficiency.

For example, `Set-Cookie` header requests our client to save some cookies for future requests, which might be vital for website functionality. Other headers such as `Etag`, `Last-Modified` are intended to help the client with caching to optimize resource usage.

For the entire list of all HTTP headers, see MDN HTTP Headers

Finally, just like with request headers, headers prefixed with an `X-` are custom web functionality headers that we might need to integrate to our scraper.

We took a brief overlook of core HTTP components, and now it's time we give it a go and see how HTTP works in practical Node!

Making GET Requests

Now that we're familiar with the HTTP protocol and how it's used in javascript scraping let's send some requests!

Let's start with a basic GET request:

```
import axios from 'axios';

const response = await axios.get('https://httpbin.org/get');
console.log(response.data);
```

Here we're using http://httpbin.org HTTP testing service to retrieve a simple HTML page. When run, this script should print basic details about our made request:

```
{
  args: {},
  headers: {
    Accept: 'application/json, text/plain, */*',
    Host: 'httpbin.org',
    'User-Agent': 'axios/0.25.0',
  },
  origin: '180.111.222.223',
  url: 'https://httpbin.org/get'
}
```

Making POST requests

POST type requests are used to interact with the website through its interactive features like login, search functionality or result filtering.

For these requests our scraper needs to send something to receive the response. That something is usually a JSON document:

```
import axios from 'axios';

const response = await axios.post('https://httpbin.org/post', {'query': 'cats',
'page': 1});
console.log(response.data);
```

Another document type we can POST is **form data** type. For this we need to do a bit more work and use form-data package:

```
import axios from 'axios';
import FormData from 'form-data';

function makeForm(data){
    var bodyFormData = new FormData();
    for (let key in data){
        bodyFormData.append(key, data[key]);
    }
    return bodyFormData;
}

const resposne = await axios.post('https://httpbin.org/post', makeForm({'query':
'cats', 'page': 1}));
console.log(response.data);
```

Axios is smart enough to fill in the required header details (like `content-type` and `content-length`) based on the data argument. So, if we're sending an object it'll set `Content-Type` header to `application/json` and form data to `application/x-www-form-urlencoded` - pretty convenient!

Setting Headers

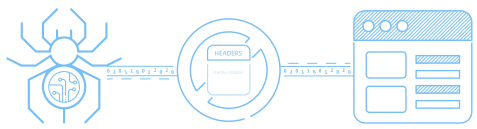
As we've covered before our requests must provide some metadata which helps the server to determine what content to return or whether to cooperate with us at all. Often, this metadata can be used to identify web scrapers and block them, so when scraping we should avoid standing out and mimic a modern web browser.

To start all browsers set `User-Agent` and `Accept` headers. To set them in our `axios` scraper we should create a `Client` and copy the values from a Chrome web browser:

```
import axios from 'axios';

const response = await axios.get(
    'https://httpbin.org/get',
    {headers: {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36',
        'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
',
    }}
);
console.log(response.data);
```

This will ensure that every request the HTTP client is making includes expected default headers.

How request headers are used to block scrapers



For a complete guide on headers in web scraper blocking see our full introduction article.

Tip: Set Default Settings

When scraping we typically want to apply the same configuration to multiple requests like setting those User-Agent headers for every request our scraper is making to avoid being blocked.

Axios comes with a great shortcut that allows to configure default values for all connections:

```
import axios from 'axios';

const session = axios.create({
    headers: {'User-Agent': 'tutorial program'},
    timeout: 5000,
    proxy: {
            host: 'proxy-url',
            port: 80,
            auth: {username: 'my-user', password: 'my-password'}
        }
    }
)

const response1 = await session.get('http://httpbin.org/get');
console.log(response1.data);
const response2 = await session.get('http://httpbin.org/get');
console.log(response2.data);
```

Here we created an instance of `axios` that will apply custom headers, timeout and proxy settings to every request!

Tip: Automatic Cookie Tracking

Sometimes when web-scraping we care about persistent connection state. For websites where we need to login or configure preferences like currency or language - cookies are used to do all of that!

Unfortunately, by default axios doesn't support cookie tracking, however, it can be enabled via axios-cookiejar-support extension package:

```
import axios from 'axios';
import { CookieJar } from 'tough-cookie';
import { wrapper } from 'axios-cookiejar-support';

const jar = new CookieJar();
const session = wrapper(axios.create({ jar }));

async function setLocale(){
    // set cookies:
    let respSetCookies = await
session.get('http://httpbin.org/cookies/set/locale/usa');
    // retrieve existing cookies:
    let respGetCookies = await session.get('http://httpbin.org/cookies');
    console.log(respGetCookies.data);
}

setLocale();
```

In the example above, we're configuring axios instance with a cookie jar object which allows us to have persistent cookies in our web scraping session. If we run this script we should see:

```
{ cookies: { locale: 'usa' } }
```

Now that we're familiar HTTP connection and how can we use it in axios HTTP client package let's take a look at the other half of the web scraping process: parsing HTML data!

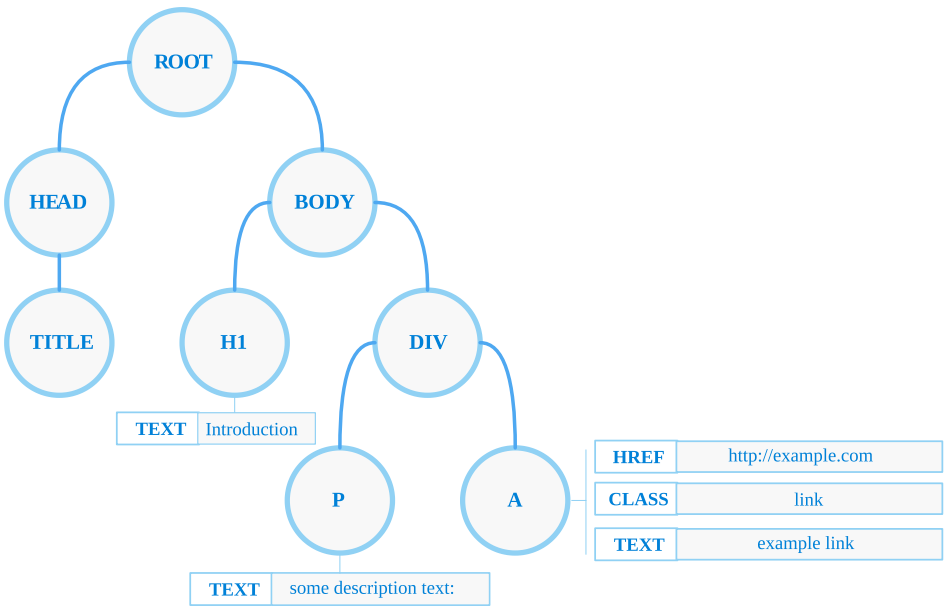
Parsing HTML

HTML (HyperText Markup Language) is a text data structure that powers the web. The great thing about it is that it's intended to be machine-readable text content, which is great news for web scraping as we can easily parse the relevant data with a bit of javascript code!

HTML is a tree-type structure that lends easily to parsing. For example, let's take this simple HTML content:

```
<head>
  <title>
  </title>
</head>
<body>
  <h1>Introduction</h1>
  <div>
    <p>some description text: </p>
    <a class="link" href="http://example.com">example link</a>
  </div>
</body>
```

This is a basic HTML document that a simple website might serve. You can already see the tree-like structure just by indentation of the text, but we can even go further and illustrate it:



example of a HTML node tree. Note that branches are ordered (left-to-right)

This tree structure is brilliant for web scraping as we can easily navigate the whole document and extract specific parts that we want.

For example, to find the title of the website, we can see that it's under `<body>` and under `<h1>` element. In other words - if we wanted to extract 1000 titles for 1000 different pages, we would write a rule to find `body->h1->text` rule, but how do we execute this rule?

When it comes to HTML parsing, there are two standard ways to write these rules: CSS selectors and XPATH selectors. Let's see how can we use them in NodeJS and Cheerio next.

CSS Selectors with Cheerio

Cheerio is the most popular HTML parsing package in NodeJS which allows us to use CSS selectors to select specific nodes of an HTML tree.

Parsing HTML with CSS Selectors

For more on CSS selectors see our complete introduction tutorial which covers basic usage, tips and tricks and common web scraping idioms

To use Cheerio we have to create a tree parser object from an HTML string and then we can use a combination of CSS selectors and element functions to extract specific data:

```
import cheerio from 'cheerio';

const tree = cheerio.load(`
    <head>
        <title>My Website</title>
    </head>
    <body>
        <div class="content">
            <h1>First blog post</h1>
            <p>Just started this blog!</p>
            <a href="http://scrapfly.io/blog">Checkout My Blog</a>
        </div>
    </body>
`);

console.log({
    // we can extract text of the node:
    title: tree('.content h1').text(),
    // or a specific attribute value:
    url: tree('.content a').attr('href')
});
```

In the example above, we're loading Cheerio with our example HTML document and highlighting two ways of selecting relevant data. To select the text of an HTML element we're using `text()` method and to select a specific attribute we're using the `attr()` method.

XPath Selectors with Xmldom

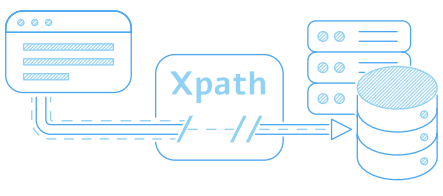
While CSS selectors are short, robust and easy to read sometimes when dealing with complex web pages we might need something more powerful. For that nodeJS also supports XPATH selectors via libraries like xpath and @xmldom/xmldom:

```
import xpath from 'xpath';
import { DOMParser } from '@xmldom/xmldom'

const tree = new DOMParser().parseFromString(`
    <head>
        <title>My Website</title>
    </head>
    <body>
        <div class="content">
            <h1>First blog post</h1>
            <p>Just started this blog!</p>
            <a href="http://scrapfly.io/blog">Checkout My Blog</a>
        </div>
    </body>
`);

console.log({
    // we can extract text of the node, which returns `Text` object:
    title: xpath.select('//div[@class="content"]/h1/text()', tree)[0].data,
    // or a specific attribute value, which return `Attr` object:
    url: xpath.select('//div[@class="content"]/a/@href', tree)[0].value,
});
```

Here, we're replicating our Cheerio example in `xmldom + xpath` setup selecting title text and the URL's `href` attribute.

Parsing HTML with Xpath

For more on XPATH selectors see our complete introduction tutorial which covers basic usage, tips and tricks and common web scraping idioms



We looked into two methods of parsing HTML content with NodeJS: using CSS selectors with Cheerio and using Xpath selectors with xmldom + xpath. Generally, it's best to stick with Cheerio as it complies with HTML standard better and CSS selectors are easier to work with.

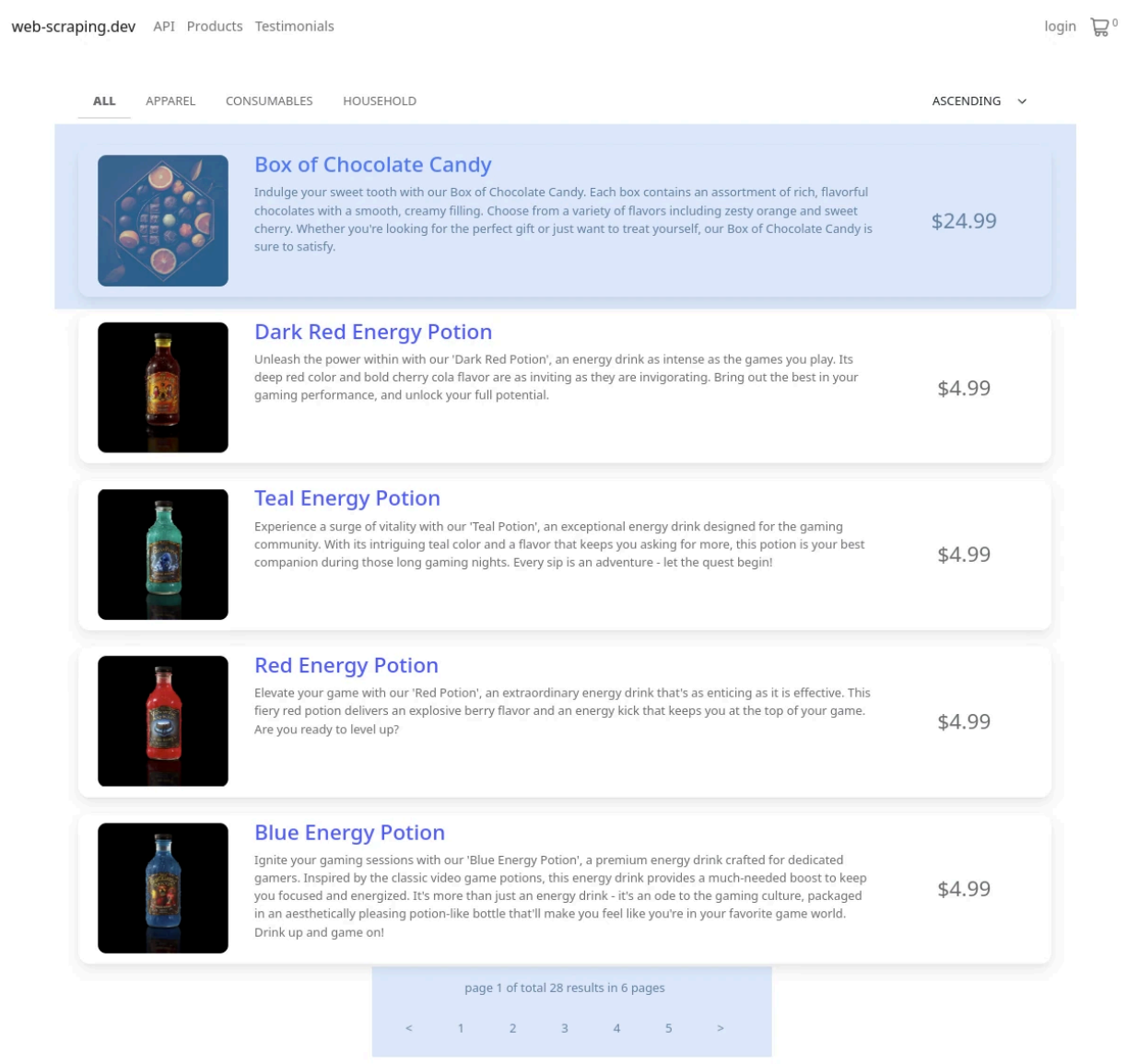
Let's put everything we've learned by exploring an example project next!

Example Project: Web-Scraping.dev

We've learned about HTTP connections using `axios` and HTML parsing using `cheerio` and now it's time to put everything together and solidify our knowledge and write a real javascript web scraper!

In this section, we'll use NodeJS to scrape web-scraping.dev/products. This is a special mock website used for testing web scrapers and it implements many common patterns encountered in real-world web scraping.

We'll write a NodeJS scraper that scrapes all products listed on the `/products` pagination endpoint:



Note the highlighted areas - we'll collect each product preview and navigate to other pagination pages. Our process will look something like this:

1. Scrape 1st products page.
2. Parse product URLs and other pagination URLs.
3. Scrape other pagination URLs to collect all product URLs.
4. Scrape all product URLs for product data.

Let's start with pagination scraping. We'll scrape the whole paging section to find all individual product URLs:

```
import axios from 'axios';
import cheerio from 'cheerio';

const firstPageResponse = await axios.get("https://web-scraping.dev/products");
let selector = cheerio.load(firstPageResponse.data);
// find all product URLs
let productUrls = selector(".product h3 a").map((i, el) => el.attribs['href']).get();
console.log(`found ${productUrls.length} products`);
// find all other page URLs
let otherPages = selector(".paging>a").map((i, el) => el.attribs['href']).get();
console.log(`found ${otherPages.length} pages`);

// to scrape concurrently we can use Promise.all()
const allResponses = await Promise.all(otherPages.map(url => axios.get(url)));

for (let resp of allResponses) {
    selector = cheerio.load(resp.data);
    productUrls.push(...selector(".product h3 a").map((i, el) =>
el.attribs['href']).get());
}
console.log(productUrls);
```

▶ Output

Here we're using `axios` to make a request to the first page and then using `cheerio` to parse the HTML content. We're using CSS selectors to find all product URLs and all other page URLs.

Finally, we're using `Promise.all()` to scrape all other pages concurrently and then merging all product URLs into a single list. We can see concurrent requests at work as the output is not ordered.

Next, let's scrape each product page to collect product details. Let's extend our scraper with:

```
productUrls = productUrls.slice(0, 5);  // for testing, only scrape 5 products
const productResponses = await Promise.all(
    productUrls.map(url => axios.get(url))
);
let products = [];
for (let resp of productResponses) {
    selector = cheerio.load(resp.data);
    products.push({
        "name": selector(".product-title").text(),
        "description": selector(".product-description").text(),
        "price_full": selector(".product-price-full").text(),
        "price": selector(".product-price").text(),
    });
}
console.log(products)
```

▶ Output

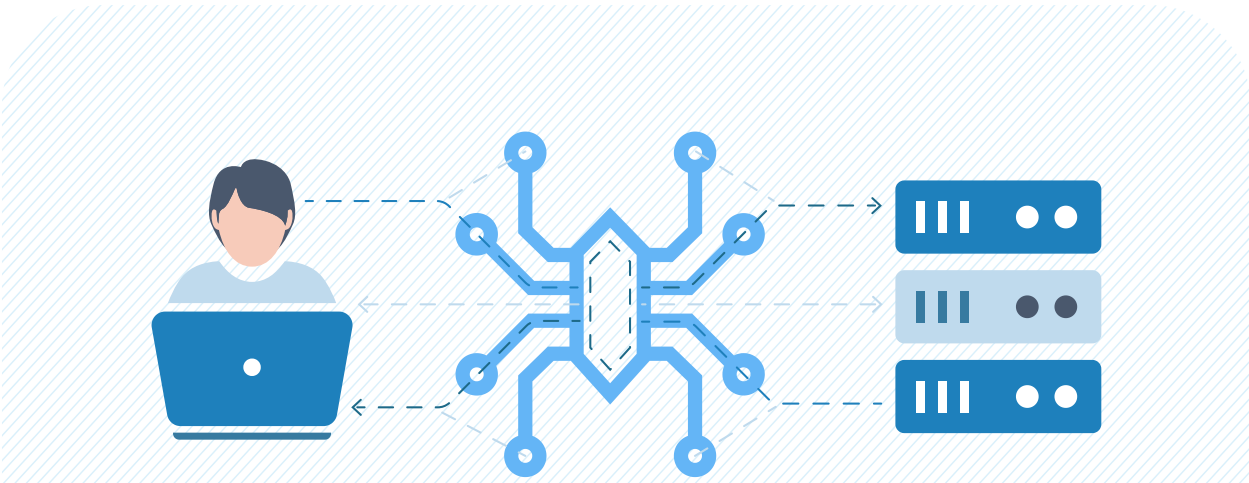
Here we're using `Promise.all()` to scrape all product pages concurrently and then using CSS selectors to extract product details like name, description and price.

Taking advantage of NodeJS async concurrency with `Promise.all` allow us to create really fast web scrapers that can collect all product data in a matter of seconds!

In real life web scraping speed might be limited not by our code but by scraper blocking. Next, let's take a look at how to bypass many scraper blocks with the assistance of Scrapfly.

Avoiding Blocking with ScrapFly

Unfortunately identifying nodeJS-based web scrapers is really easy which can lead to web scraper blocking. This is where Scrapfly can lend a hand!



ScrapFly provides web scraping, screenshot, and extraction APIs for data collection at scale.

- Anti-bot protection bypass - scrape web pages without blocking!
- Rotating residential proxies - prevent IP address and geographic blocks.
- JavaScript rendering - scrape dynamic web pages through cloud browsers.
- Full browser automation - control browsers to scroll, input and click on objects.
- Format conversion - scrape as HTML, JSON, Text, or Markdown.
- Python and Typescript SDKs, as well as Scrapy and no-code tool integrations.

To use Scrapfly in NodeJS we can use Scrapfly Typescript SDK which is also accessible in Node:

```
npm install scrapfly-sdk
```

```
import { ScrapflyClient, ScrapeConfig, ScrapeResult, log } from "scrapfly-sdk";
// Optional: set log level to debug to see all details
log.setLevel("DEBUG");

// 1. Create a scrapfly client with your API key
const scrapfly = new ScrapflyClient({ key: "YOUR SCRAPFLY KEY" })
// 2. Start scraping!
const result = await scrapfly.scrape(new ScrapeConfig({
  url: "https://web-scraping.dev/product/1",
  // optional configuration:
  asp: true,  // enable scraper blocking bypass
  country: "US", // set proxy country
  render_js: true,  // enable headless web browser
  // ... and much more
}))
// 3. access scraped result data
console.log(result.result.content);
// 3.1 and even process it with CSS selectors:
console.log(result.selector("h3").text())
```

Scrapfly SDK integrates everything we've learned in this tutorial including HTML parsing with `cheerio` as well as many other scraper power ups like:

- Anti Scraping Protection Bypass - bypasses anti scraping services like Cloudflare, Perimeterx and others.
- Millions of residential proxies - for anonymous scraping at scale from over 50+ countries.
- Cloud Headless Browsers - for scraping dynamic websites that require a browser to render.
- And much more!

FAQ

To wrap up this tutorial let's take a look at frequently asked questions about web scraping in JS:

What's the difference between nodejs and puppeteer in web scraping?

Puppeteer is a popular browser automation library *for Nodejs*. It is frequently used for web scraping. However, we don't always need a web browser to web scrape. In this article, we've learned how can we use Nodejs with a simple HTTP client to scrape web pages. Browsers are very complicated and expensive to run and upkeep so HTTP client web scrapers are much faster and cheaper.

How to scrape concurrently in NodeJS?

Since NodeJS javascript code is naturally asynchronous we can perform concurrent requests to scrape multiple pages by wrapping a list of scrape promises in Promise.all or Promise.allSettled functions. These async await functions take a list of promise objects and executes them in parallel which can speed up web scraping process hundreds of times:

```
urls = [...]
async function scrape(url){
    ...
};
let scrape_promises = urls.map((url) => scrape(url));
await Promise.all(scrape_promises);
```

How to use proxy in NodeJS?

When scraping at scale we might need to use proxies to prevent blocking. Most NodeJS http client libraries implement proxy support through simple arguments. For example in `axios` library we can set proxy using sessions:

```
const session = axios.create({
    proxy: {
            host: 'http://111.22.33.44',  //proxy ip address with protocol
            port: 80,  // proxy port
            auth: {username: 'proxy-auth-username', password: 'proxy-auth-password'}
// proxy auth if needed
        }
    }
)
```

What is the best nodejs web scraping library?

Web Scraping with Cheerio and Nodejs is the most popular way to scrape without using browser automation (Puppeteer) and Axios is the most popular way to make HTTP requests. Though less popular alternatives like `xmldom` shouldn't be overlooked as they can help with scraping more complex web pages.

How to click buttons, input text do other browser actions in NodeJS?

Since NodeJS engine is not fully browser compliant we cannot automatically click buttons or submit forms. For this something like Puppeteer needs to be used to automate a real web browser. For more see Web Scraping With a Headless Browser: Puppeteer

Summary

In this extensive introduction article, we've introduced ourselves with NodeJS web scraping ecosystem. We looked into using axios as our HTTP client to collect multiple pages and using cheerio/@xmldom/xmldom to parse information from this data using CSS/XPATH selectors.

Finally, we wrapped everything up with an example nodejs web scraper project which scrapes product information from web-scraping.dev/products and looked into ScrapFly's API solution which takes care of difficult web scraping challenges such as scaling and blocking!

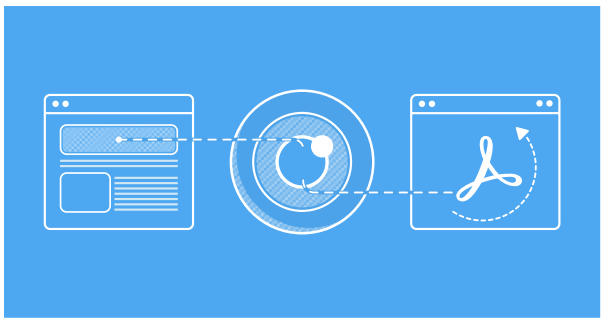
Related Questions

- How to take screenshots in NodeJS?
- How to use CSS selectors in NodeJS when web scraping?
- How to find HTML elements by text with Cheerio and NodeJS?
- How to use proxies with NodeJS axios?
- How to use XPath selectors in NodeJS when web scraping?

More >

NODEJS HTTP DATA PARSING CSS SELECTORS SCRAPING INTRODUCTION

Related Posts

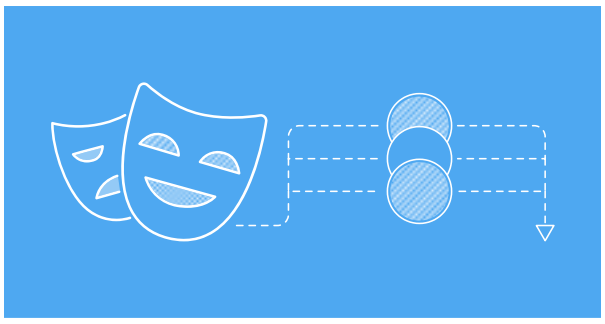


Jan 22, 2025

How to Capture and Convert a Screenshot to PDF

Quick guide on how to effectively capture web screenshots as PDF documents

SCREENSHOTS PYTHON NODEJS

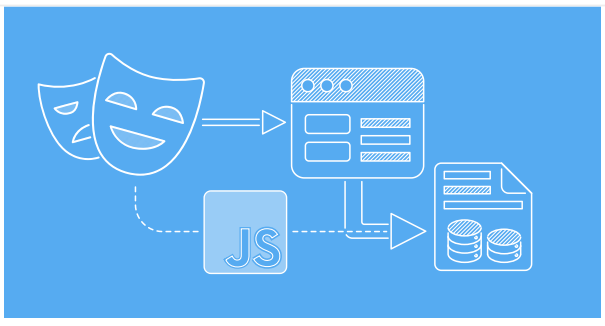


Jan 15, 2025

Playwright Examples for Web Scraping and Automation

Learn Playwright with Python and JavaScript examples for automating browsers like Chromium, WebKit, and Firefox.

PLAYWRIGHT PYTHON NODEJS



Jan 13, 2025

Web Scraping with Playwright and JavaScript

Learn about Playwright - a browser automation toolkit for server side Javascript like NodeJS, Deno or Bun.

PLAYWRIGHT HEADLESS BROWSERS NODEJS

Company

Careers
Terms of service
Privacy Policy
Data Processing Agreement
KYC Compliance
Status

Integrations

Zapier
Make
N8n
LlamaIndex
LangChain

Social

Tools

Convert cURL commands to Python code
JA3/TLS Fingerprint
HTTP2 Fingerprint
Xpath/CSS Selector Tester

Resources

API Documentation
Web Scraping Academy
Is Web Scraping Legal?
Web Scraping Tools
FAQ

Learn Web Scraping

Web Scraping with Python
Web Scraping with PHP
Web Scraping with Ruby
Web Scraping with R
Web Scraping with NodeJS
Web Scraping with Python Scrapy
How to Scrape without getting blocked tutorial
Web Scraping with Python and BeautifulSoup
Web Scraping with Nodejs and Puppeteer
How To Scrape Graphql
Best Proxies for Web Scraping
Top 5 Best Residential Proxies

Usage

What is Web Scraping used for?
Web Scraping for AI Training
Web Scraping for Compliance
Web Scraping for eCommerce
Web Scraping for Finance
Web Scraping for Fraud Detection
Web Scraping for Jobs
Web Scraping for Lead Generation
Web Scraping for News & Media
Web Scraping for Real Estate
Web Scraping for SERP & SEO
Web Scraping for Social Media
Web Scraping for Travel