



## TUTORIAL

# How To Scrape a Website Using Node.js and Puppeteer

Node.js Open Source JavaScript Data Analysis

By [Gbadebo Bello](#)

Published on August 13, 2020 63.7k

English

*The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.*

## Introduction

Web scraping is the process of automating data collection from the web. The process typically deploys a “crawler” that automatically surfs the web and scrapes data from selected pages. There are many reasons why you might want to scrape data. Primarily, it makes data collection much faster by eliminating the manual data-gathering process. Scraping is also a solution when data collection is desired or needed but the website does not provide an API.

In this tutorial, you will build a web scraping application using Node.js and Puppeteer. Your app will grow in complexity as you progress. First, you will code your app to open Chromium and load a special website designed as a web-scraping sandbox: books.toscrape.com. In the next two steps, you will scrape all the books on a single page of books.toscrape and then all the books across multiple pages. In the remaining steps, you will filter your scraping by book category and then save your data as a JSON file.

**Warning:** The ethics and legality of web scraping are very complex and constantly evolving. They also differ based on your location, the data’s location, and the website in

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.

[Sign Up](#)

- Node.js installed on your development machine. This tutorial was tested on Node.js version 12.18.3 and npm version 6.14.6. [You can follow this guide to install Node.js on macOS or Ubuntu 18.04](#), or you can [follow this guide to install Node.js on Ubuntu 18.04 using a PPA](#).

## Step 1 — Setting Up the Web Scraper

With Node.js installed, you can begin setting up your web scraper. First, you will create a project root directory and then install the required dependencies. This tutorial requires just one dependency, and you will install it using Node.js's default package manager [npm](#). npm comes preinstalled with Node.js, so you don't need to install it.

Create a folder for this project and then move inside:

```
$ mkdir book-scraper
$ cd book-scraper
```

You will run all subsequent commands from this directory.


We need to install one package using npm, or the node package manager. First initialize npm in order to create a `package.json` file, which will manage your project's dependencies and metadata.

Initialize npm for your project:

```
$ npm init
```

npm will present a sequence of prompts. You can press `ENTER` to every prompt, or you can add personalized descriptions. Make sure to press `ENTER` and leave the default values in place when prompted for `entry point:` and `test command:`. Alternately, you can pass the `y` flag to npm — `npm init -y` — and it will submit all the default values for you.

Your output will look something like this:

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics. 

Sign Up

```
"main": "index.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "sammy the shark",
"license": "ISC "
}
```

Is this OK? (yes) yes

Type `yes` and press `ENTER`. `npm` will save this output as your `package.json` file.

Now use `npm` to install Puppeteer:

```
$ npm install --save puppeteer
```

This command installs both Puppeteer and a version of Chromium that the Puppeteer team knows will work with their API.

On Linux machines, Puppeteer might require some additional dependencies.

If you are using Ubuntu 18.04, check the 'Debian Dependencies' dropdown inside the 'Chrome headless doesn't launch on UNIX' section of Puppeteer's troubleshooting docs. You can use the following command to help find any missing dependencies:

```
$ ldd chrome | grep not
```

With `npm`, Puppeteer, and any additional dependencies installed, your `package.json` file requires one last configuration before you start coding. In this tutorial, you will launch your app from the command line with `npm run start`. You must add some information about this `start` script to `package.json`. Specifically, you must add one line under the `scripts` directive regarding your `start` command.

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

Find the `scripts:` section and add the following configurations. Remember to place a comma at the end of the `test` script line, or your file will not parse correctly.

Output

```
{
  . . .
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
  . . .
  "dependencies": {
    "puppeteer": "^5.2.1"
  }
}
```

You will also notice that `puppeteer` now appears under `dependencies` near the end of the file. Your `package.json` file will not require any more revisions. Save your changes and close your editor.

You are now ready to start coding your scraper. In the next step, you will set up a browser instance and test your scraper's basic functionality.

## Step 2 — Setting Up the Browser Instance

When you open a traditional browser, you can do things like click buttons, navigate with your mouse, type, open the dev tools, and more. A headless browser like Chromium allows you to do these same things, but programmatically and without a user interface. In this step, you will set up your scraper's browser instance. When you launch your application, it will automatically open Chromium and navigate to `books.toscrape.com`. These initial actions will form the basis of your program.

Your web scraper will require four `.js` files: `browser.js`, `index.js`, `pageController.js`, and `pageScraper.js`. In this step, you will create all four files and then continually update them as your program grows in sophistication. Start with `browser.js`; this file will contain

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

First, you will require Puppeteer and then create an `async` function called `startBrowser()`. This function will start the browser and return an instance of it. Add the following code:

`./book-scraper/browser.js`

```
const puppeteer = require('puppeteer');

async function startBrowser(){
  let browser;
  try {
    console.log("Opening the browser.....");
    browser = await puppeteer.launch({
      headless: false,
      args: ["--disable-setuid-sandbox"],
      'ignoreHTTPSErrors': true
    });
  } catch (err) {
    console.log("Could not create a browser instance => ", err);
  }
  return browser;
}

module.exports = {
  startBrowser
};
```

Puppeteer has a `.launch()` method that launches an instance of a browser. This method returns a Promise, so you have to make sure the Promise resolves by using a `.then` or `await` block.

You are using `await` to make sure the Promise resolves, wrapping this instance around a try-catch code block, and then returning an instance of the browser.

Notice that the `.launch()` method takes a JSON parameter with several values:

- **headless** - `false` means the browser will run with an Interface so you can watch your script execute, while `true` means the browser will run in headless mode. Note well, however, that if you want to deploy your scraper to the cloud, set `headless` back to `true`. Most virtual machines are headless and do not include a user interface and hence

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

Save and close the file.

Now create your second `.js` file, `index.js`:

```
$ nano index.js
```

Here you will `require` `browser.js` and `pageController.js`. You will then call the `startBrowser()` function and pass the created browser instance to our page controller, which will direct its actions. Add the following code:

```
./book-scraper/index.js
```

```
const browserObject = require('./browser');
const scraperController = require('./pageController');

//Start the browser and create a browser instance
let browserInstance = browserObject.startBrowser();

// Pass the browser instance to the scraper controller
scraperController(browserInstance)
```

Save and close the file.

Create your third `.js` file, `pageController.js`:

```
$ nano pageController.js
```

`pageController.js` controls your scraping process. It uses the browser instance to control the `pageScraper.js` file, which is where all the scraping scripts execute. Eventually, you will use it to specify what book category you want to scrape. For now, however, you just want to make sure that you can open Chromium and navigate to a web page:

```
./book-scraper/pageController.js
```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

```

    catch(err){
      console.log("Could not resolve the browser instance => ", err);
    }
  }

  module.exports = (browserInstance) => scrapeAll(browserInstance)

```

This code exports a function that takes in the browser instance and passes it to a function called `scrapeAll()`. This function, in turn, passes this instance to `pageScraper.scrape()` as an argument which uses it to scrape pages.

Save and close the file.

Finally, create your last `.js` file, `pageScraper.js`:

```
$ nano pageScraper.js
```

Here you will create an object literal with a `url` property and a `scrape()` method. The `url` is the web URL of the web page you want to scrape, while the `scrape()` method contains the code that will perform your actual scraping, although at this stage it merely navigates to a URL. Add the following code:

```
./book-scraper/pageScraper.js
```

```

const scraperObject = {
  url: 'http://books.toscrape.com',
  async scrape(browser){
    let page = await browser.newPage();
    console.log(`Navigating to ${this.url}...`);
    await page.goto(this.url);
  }
}

module.exports = scraperObject;

```

Puppeteer has a `newPage()` method that creates a new page instance in the browser, and these page instances can do quite a few things. In our `scrape()` method, you created a

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

Your program's file-structure is now complete. The first level of your project's directory tree will look like this:

Output

```
.
├─ browser.js
├─ index.js
├─ node_modules
├─ package-lock.json
├─ package.json
├─ pageController.js
└─ pageScraper.js
```

Now run the command `npm run start` and watch your scraper application execute:

```
$ npm run start
```

It will automatically open a Chromium browser instance, open a new page in the browser, and navigate to `books.toscrape.com`.

In this step, you created a Puppeteer application that opened Chromium and loaded the homepage for a dummy online bookstore—`books.toscrape.com`. In the next step, you will scrape the data for every book on that homepage.

### Step 3 — Scraping Data from a Single Page

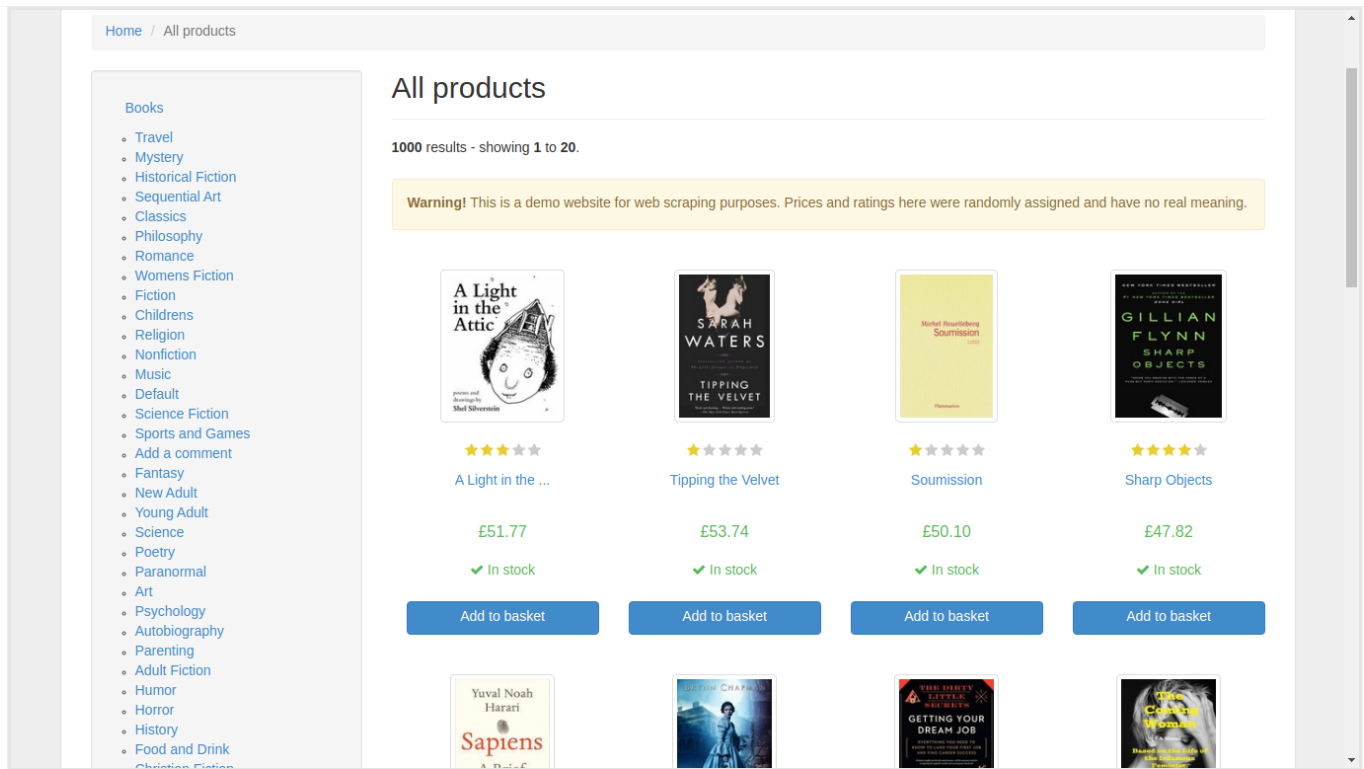
Before adding more functionality to your scraper application, open your preferred web browser and manually navigate to the [books to scrape homepage](https://books.toscrape.com/). Browse the site and get a sense of how data is structured.

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

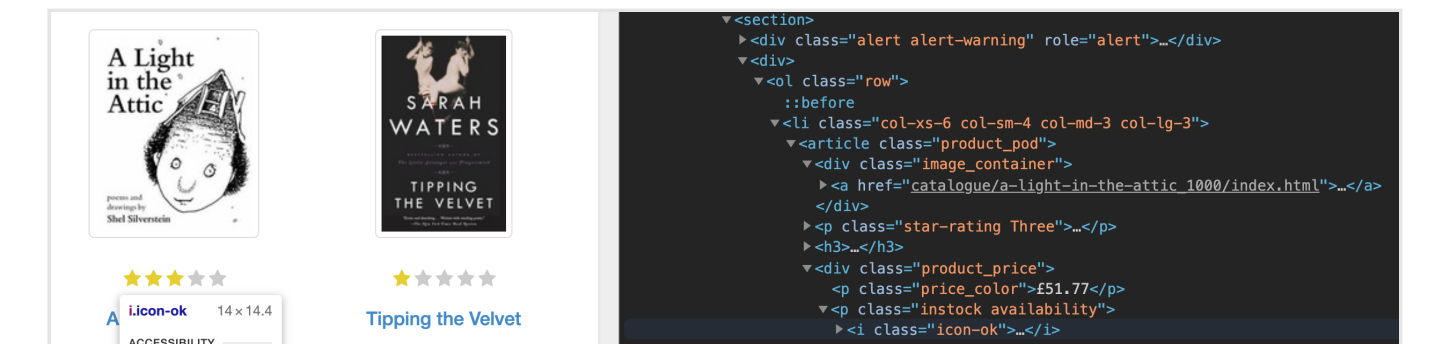




You will find a category section on the left and books displayed on the right. When you click on a book, the browser navigates to a new URL that displays relevant information regarding that particular book.

In this step, you will replicate this behavior, but with code; you will automate the business of navigating the website and consuming its data.

First, if you inspect the source code for the homepage using the Dev Tools inside your browser, you will notice that the page lists each book's data under a `section` tag. Inside the `section` tag every book is under a `li` tag, and it is here that you find the link to the book's dedicated page, the price, and the in-stock availability.



**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics. ✕

Enter your email address

**Sign Up**

You'll be scraping these book URLs, filtering for books that are in-stock, navigating to each individual book page, and scraping that book's data.

Reopen your `pageScraper.js` file:

```
$ nano pageScraper.js
```

Add the following highlighted content. You will nest another `await` block inside `await page.goto(this.url);`:

`./book-scraper/pageScraper.js`

```
const scraperObject = {
  url: 'http://books.toscrape.com',
  async scraper(browser){
    let page = await browser.newPage();
    console.log(`Navigating to ${this.url}...`);
    // Navigate to the selected page
    await page.goto(this.url);
    // Wait for the required DOM to be rendered
    await page.waitForSelector('.page_inner');
    // Get the link to all the required books
    let urls = await page.$$eval('section ol > li', links => {
      // Make sure the book to be scraped is in stock
      links = links.filter(link => link.querySelector('.instock.availability > i'))
      // Extract the links from the data
      links = links.map(el => el.querySelector('h3 > a').href)
      return links;
    });
    console.log(urls);
  }
}

module.exports = scraperObject;
```

In this code block, you called the `page.waitForSelector()` method. This waited for the div that contains all the book-related information to be rendered in the DOM, and then you called the `page.$$eval()` method. This method gets the URL element with the

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics. ✕

Sign Up

Save and close the file.

Re-run your application:

```
$ npm run start
```

## Output

Opening the browser.....

Navigating to <http://books.toscrape.com>...

```
[
  'http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html',
  'http://books.toscrape.com/catalogue/tipping-the-velvet_999/index.html',
  'http://books.toscrape.com/catalogue/soumission_998/index.html',
  'http://books.toscrape.com/catalogue/sharp-objects_997/index.html',
  'http://books.toscrape.com/catalogue/sapiens-a-brief-history-of-humankind_996/index.html',
  'http://books.toscrape.com/catalogue/the-requiem-red_995/index.html',
  'http://books.toscrape.com/catalogue/the-dirty-little-secrets-of-getting-your-dream-job_994/index.html',
  'http://books.toscrape.com/catalogue/the-coming-woman-a-novel-based-on-the-life-of-the-irish-writer-sarah-jane-lane_993/index.html',
  'http://books.toscrape.com/catalogue/the-boys-in-the-boat-nine-americans-and-their-epic-cruise_992/index.html',
  'http://books.toscrape.com/catalogue/the-black-maria_991/index.html',
  'http://books.toscrape.com/catalogue/starving-hearts-triangular-trade-trilogy-1_990/index.html',
  'http://books.toscrape.com/catalogue/shakespeares-sonnets_989/index.html',
  'http://books.toscrape.com/catalogue/set-me-free_988/index.html',
  'http://books.toscrape.com/catalogue/scott-pilgrims-precious-little-life-scott-pilgrim-1_987/index.html',
  'http://books.toscrape.com/catalogue/rip-it-up-and-start-again_986/index.html',

```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Enter your email address

## Sign Up

This is a great start, but you want to scrape all the relevant data for a particular book and not only its URL. You will now use these URLs to open each page and scrape the book's title, author, price, availability, UPC, description, and image URL.

Reopen `pageScraper.js`:

```
$ nano pageScraper.js
```

Add the following code, which will loop through each scraped link, open a new page instance, and then retrieve the relevant data:

```
./book-scraper/pageScraper.js
```

```
const scraperObject = {
  url: 'http://books.toscrape.com',
  async scraper(browser){
    let page = await browser.newPage();
    console.log(`Navigating to ${this.url}...`);
    // Navigate to the selected page
    await page.goto(this.url);
    // Wait for the required DOM to be rendered
    await page.waitForSelector('.page_inner');
    // Get the link to all the required books
    let urls = await page.$$eval('section ol > li', links => {
      // Make sure the book to be scraped is in stock
      links = links.filter(link => link.querySelector('.instock.availability > i'));
      // Extract the links from the data
      links = links.map(el => el.querySelector('h3 > a').href)
      return links;
    });

    // Loop through each of those links, open a new page instance and get the relevant data
    let pagePromise = (link) => new Promise(async(resolve, reject) => {
      let dataObj = {};
      let newPage = await browser.newPage();
      await newPage.goto(link);
      dataObj['bookTitle'] = await newPage.$eval('.product_main > h1', text => text.textContent);
      dataObj['bookPrice'] = await newPage.$eval('.price_color', text => text.textContent);
      dataObj['noAvailable'] = await newPage.$eval('.instock.availability', text => text.textContent);
    });
  }
};
```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

```

    dataObj['imageUrl'] = await newPage.$eval('#product_gallery img', img => img
    dataObj['bookDescription'] = await newPage.$eval('#product_description', div
    dataObj['upc'] = await newPage.$eval('.table.table-striped > tbody > tr > td
    resolve(dataObj);
    await newPage.close();
  });

  for(link in urls){
    let currentPageData = await pagePromise(urls[link]);
    // scrapedData.push(currentPageData);
    console.log(currentPageData);
  }
}

module.exports = scraperObject;

```

You have an array of all URLs. You want to loop through this array, open up the URL in a new page, scrape data on that page, close that page, and open a new page for the next URL in the array. Notice that you wrapped this code in a Promise. This is because you want to be able to wait for each action in your loop to complete. Therefore each Promise opens a new URL and won't resolve until the program has scraped all the data on the URL, and then that page instance has closed.

**Warning:** note well that you waited for the Promise using a `for-in` loop. Any other loop will be sufficient but avoid iterating over your URL arrays using an array-iteration method like `forEach`, or any other method that uses a callback function. This is because the callback function will have to go through the callback queue and event loop first, hence, multiple page instances will open all at once. This will place a much larger strain on your memory.

Take a closer look at your `pagePromise` function. Your scraper first created a new page for each URL, and then you used the `page.$eval()` function to target selectors for relevant details that you wanted to scrape on the new page. Some of the texts contain whitespaces, tabs, newlines, and other non-alphanumeric characters, which you stripped

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

```
$ npm run start
```

The browser opens the homepage and then opens each book page and logs the scraped data from each of those pages. This output will print to your console:

#### Output

Opening the browser.....

Navigating to <http://books.toscrape.com>...

```
{
  bookTitle: 'A Light in the Attic',
  bookPrice: '£51.77',
  noAvailable: '22',
  imageUrl: 'http://books.toscrape.com/media/cache/fe/72/fe72f0532301ec28892ae79a629a293c.j
  bookDescription: "It's hard to imagine a world without A Light in the Attic. [...]",
  upc: 'a897fe39b1053632'
}
{
  bookTitle: 'Tipping the Velvet',
  bookPrice: '£53.74',
  noAvailable: '20',
  imageUrl: 'http://books.toscrape.com/media/cache/08/e9/08e94f3731d7d6b760dfbfbcb02ca5c62.j
  bookDescription: `"Erotic and absorbing...Written with starling power."--"The New York Ti
  upc: '90fa61229261140a'
}
{
  bookTitle: 'Soumission',
  bookPrice: '£50.10',
  noAvailable: '20',
  imageUrl: 'http://books.toscrape.com/media/cache/ee/cf/eecfe998905e455df12064dba399c075.j
  bookDescription: 'Dans une France assez proche de la nôtre, [...]',
  upc: '6957f44c3847a760'
}
...
```

In this step, you scraped relevant data for every book on the homepage of [books.toscrape.com](http://books.toscrape.com). but you could add much more functionality. Each page of books. for

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics. ✕

**Sign Up**

Pages on [books.toscrape.com](https://books.toscrape.com) that are paginated have a `next` button beneath their content, while pages that are not paginated do not.

You will use the presence of this button to determine if the page is paginated or not. Since the data on each page is of the same structure and has the same markup, you won't be writing a scraper for every possible page. Rather, you will use the practice of recursion.

First, you need to change the structure of your code a bit to accommodate recursively navigating to several pages.

Reopen `pagescraper.js`:

```
$ nano pagescraper.js
```

You will add a new function called `scrapeCurrentPage()` to your `scraper()` method. This function will contain all the code that scrapes data from a particular page and then click the next button if it exists. Add the following highlighted code:

```
./book-scraper/pageScraper.js scraper()
```

```
const scraperObject = {
  url: 'http://books.toscrape.com',
  async scraper(browser){
    let page = await browser.newPage();
    console.log(`Navigating to ${this.url}...`);
    // Navigate to the selected page
    await page.goto(this.url);
    let scrapedData = [];
    // Wait for the required DOM to be rendered
    async function scrapeCurrentPage(){
      await page.waitForSelector('.page_inner');
      // Get the link to all the required books
      let urls = await page.$$eval('section ol > li', links => {
        // Make sure the book to be scraped is in stock
        links = links.filter(link => link.querySelector('.instock.availability >
        // Extract the links from the data
        links = links.map(e1 => e1.querySelector('h3 > a').href)
      })
    }
  }
}
```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

```

dataObj['bookPrice'] = await newPage.$eval('.price_color', text => text.textContent);
dataObj['noAvailable'] = await newPage.$eval('.instock.availability', text => text.textContent);
// Strip new line and tab spaces
text = text.textContent.replace(/(\r\n\t|\n|\r|\t)/gm, "");
// Get the number of stock available
let regexp = /^.*\((.*)\).*/i;
let stockAvailable = regexp.exec(text)[1].split(' ')[0];
return stockAvailable;
});
dataObj['imageUrl'] = await newPage.$eval('#product_gallery img', img => img.src);
dataObj['bookDescription'] = await newPage.$eval('#product_description', text => text.textContent);
dataObj['upc'] = await newPage.$eval('.table.table-striped > tbody > tr > td:nth-child(2)', text => text.textContent);
resolve(dataObj);
await newPage.close();
});

for(link in urls){
  let currentPageData = await pagePromise(urls[link]);
  scrapedData.push(currentPageData);
  // console.log(currentPageData);
}
// When all the data on this page is done, click the next button and start the next page
// You are going to check if this button exist first, so you know if there are more pages
let nextButtonExist = false;
try{
  const nextButton = await page.$eval('.next > a', a => a.textContent);
  nextButtonExist = true;
}
catch(err){
  nextButtonExist = false;
}
if(nextButtonExist){
  await page.click('.next > a');
  return scrapeCurrentPage(); // Call this function recursively
}
await page.close();
return scrapedData;
}
let data = await scrapeCurrentPage();
console.log(data);
return data;
}
}

module.exports = scraperObject;

```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up



If `nextButtonExists` is false, it returns the `scrapedData` array as usual.

Save and close the file.

Run your script again:

```
$ npm run start
```

This might take a while to complete; your application, after all, is now scraping the data from over 800 books. Feel free to either close the browser or press `CTRL + C` to cancel the process.

You have now maximized your scraper's capabilities, but you've created a new problem in the process. Now the issue is not too little data but too much data. In the next step, you will fine-tune your application to filter your scraping by book category.

## Step 5 — Scraping Data by Category

To scrape data by category, you will need to modify both your `pageScraper.js` file and your `pageController.js` file.

Open `pageController.js` in a text editor:

```
nano pageController.js
```

Call the scraper so that it only scrapes travel books. Add the following code:

```
./book-scraper/pageController.js
```

```
const pageScraper = require('./pageScraper');
async function scrapeAll(browserInstance){
  let browser;
  try{
    browser = await browserInstance;
    let scrapedData = {};
    // Call the scraper for different set of books to be scraped
```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

```
}
module.exports = (browserInstance) => scrapeAll(browserInstance)
```

You are now passing two parameters into your `pageScraper.scrape()` method, with the second parameter being the category of books you want to scrape, which in this example is `Travel`. But your `pageScraper.js` file does not recognize this parameter yet. You will need to adjust this file, too.

Save and close the file.

Open `pageScraper.js`:

```
$ nano pageScraper.js
```

Add the following code, which will add your category parameter, navigate to that category page, and then begin scraping through the paginated results:

```
./book-scraper/pageScraper.js
```

```
const scraperObject = {
  url: 'http://books.toscrape.com',
  async scraper(browser, category){
    let page = await browser.newPage();
    console.log(`Navigating to ${this.url}...`);
    // Navigate to the selected page
    await page.goto(this.url);
    // Select the category of book to be displayed
    let selectedCategory = await page.$$eval('.side_categories > ul > li > ul > li >

    // Search for the element that has the matching text
    links = links.map(a => a.textContent.replace(/(\r\n\t|\n|\r|\t|^\s|\s$|\B\s|

    let link = links.filter(tx => tx !== null)[0];
    return link.href;
  }, category);
  // Navigate to the selected category
  await page.goto(selectedCategory);
  let scrapedData = [];
  // Wait for the required DOM to be rendered
  async function scrapeCurrentPage(){
```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

```

});
// Loop through each of those links, open a new page instance and get the re
let pagePromise = (link) => new Promise(async(resolve, reject) => {
  let dataObj = {};
  let newPage = await browser.newPage();
  await newPage.goto(link);
  dataObj['bookTitle'] = await newPage.$eval('.product_main > h1', text =>
  dataObj['bookPrice'] = await newPage.$eval('.price_color', text => text.t
  dataObj['noAvailable'] = await newPage.$eval('.instock.availability', te
    // Strip new line and tab spaces
    text = text.textContent.replace(/(\r\n\t|\n|\r|\t)/gm, "");
    // Get the number of stock available
    let regexp = /^.*\((.*)\).*$$/i;
    let stockAvailable = regexp.exec(text)[1].split(' ')[0];
    return stockAvailable;
  });
  dataObj['imageUrl'] = await newPage.$eval('#product_gallery img', img =>
  dataObj['bookDescription'] = await newPage.$eval('#product_description',
  dataObj['upc'] = await newPage.$eval('.table.table-striped > tbody > tr :
  resolve(dataObj);
  await newPage.close();
});

for(link in urls){
  let currentPageData = await pagePromise(urls[link]);
  scrapedData.push(currentPageData);
  // console.log(currentPageData);
}
// When all the data on this page is done, click the next button and start th
// You are going to check if this button exist first, so you know if there re
let nextButtonExist = false;
try{
  const nextButton = await page.$eval('.next > a', a => a.textContent);
  nextButtonExist = true;
}
catch(err){
  nextButtonExist = false;
}
if(nextButtonExist){
  await page.click('.next > a');
  return scrapeCurrentPage(); // Call this function recursively
}
await page.close();
return scrapedData;
}
let data = await scrapeCurrentPage();
console.log(data);

```

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

This code block uses the category that you passed in to get the URL where the books of that category reside.

The `page.$$eval()` can take in arguments by passing the argument as a third parameter to the `$$eval()` method, and defining it as the third parameter in the callback as such:

example `page.$$eval()` function

```
page.$$eval('selector', function(elem, args){  
  // .....  
}, args)
```

This was what you did in your code; you passed the category of books you wanted to scrape, mapped through all the categories to check which one matches, and then returned the URL of this category.

This URL is then used to navigate to the page that displays the category of books you want to scrape using the `page.goto(selectedCategory)` method.

Save and close the file.

Run your application again. You will notice that it navigates to the `Travel` category, recursively opens books in that category page by page, and logs the results:

```
$ npm run start
```

In this step, you scraped data across multiple pages and then scraped data across multiple pages from one particular category. In the final step, you will modify your script to scrape data across multiple categories and then save this scraped data to a stringified JSON file.

## Step 6 — Scraping Data from Multiple Categories and Saving the Data as JSON

In this final step, you will make your script scrape data off of as many categories as you

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



**Sign Up**

Open `pageController.js`:

```
$ nano pageController.js
```

Adjust your code to include additional categories. The example below adds `HistoricalFiction` and `Mystery` to our existing `Travel` category:

`./book-scraper/pageController.js`

```
const pageScraper = require('./pageScraper');
async function scrapeAll(browserInstance){
  let browser;
  try{
    browser = await browserInstance;
    let scrapedData = {};
    // Call the scraper for different set of books to be scraped
    scrapedData['Travel'] = await pageScraper.scraper(browser, 'Travel');
    scrapedData['HistoricalFiction'] = await pageScraper.scraper(browser, 'HistoricalFiction');
    scrapedData['Mystery'] = await pageScraper.scraper(browser, 'Mystery');
    await browser.close();
    console.log(scrapedData)
  }
  catch(err){
    console.log("Could not resolve the browser instance => ", err);
  }
}

module.exports = (browserInstance) => scrapeAll(browserInstance)
```

Save and close the file.

Run the script again and watch it scrape data for all three categories:

```
$ npm run start
```

With the scraper fully-functional, your final step involves saving your data in a more useful format. You will now store it in a JSON file using the `fs` module in Node.js.

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Sign Up

Add the following highlighted code:

./book-scraper/pageController.js

```
const pageScraper = require('./pageScraper');
const fs = require('fs');
async function scrapeAll(browserInstance){
  let browser;
  try{
    browser = await browserInstance;
    let scrapedData = {};
    // Call the scraper for different set of books to be scraped
    scrapedData['Travel'] = await pageScraper.scraper(browser, 'Travel');
    scrapedData['HistoricalFiction'] = await pageScraper.scraper(browser, 'HistoricalFiction');
    scrapedData['Mystery'] = await pageScraper.scraper(browser, 'Mystery');
    await browser.close();
    fs.writeFile("data.json", JSON.stringify(scrapedData), 'utf8', function(err) {
      if(err) {
        return console.log(err);
      }
      console.log("The data has been scraped and saved successfully! View it at 'data.json'");
    });
  }
  catch(err){
    console.log("Could not resolve the browser instance => ", err);
  }
}

module.exports = (browserInstance) => scrapeAll(browserInstance)
```

First, you are requiring Node.js's `fs` module in `pageController.js`. This ensures that you can save your data as a JSON file. Then you are adding code so that when the scraping completes and the browser closes, the program will create a new file called `data.json`. Note that the contents of `data.json` are stringified JSON. Therefore, when reading the content of `data.json`, always parse it as JSON before reusing the data.

Save and close the file.

You have now built a web-scraping application that scrapes books across multiple categories and then stores your scraped data in a JSON file. As your application grows in

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.




Sign Up

In this tutorial, you built a web crawler that scraped data across multiple pages recursively and then saved it in a JSON file. In short, you learned a new way to automate data-gathering from websites.

Puppeteer has quite a lot of features that were not within the scope of this tutorial. To learn more, check out [Using Puppeteer for Easy Control Over Headless Chrome](#). You can also visit [Puppeteer's official documentation](#).

Was this helpful?

Yes

No



[Report an issue](#)

## About the authors



### [Gbadebo Bello](#)

Gbadebo is a software engineer that is extremely passionate about JavaScript technologies, Open Source Development and community advocacy.



### [Matt Abrams](#)

Editor

## Still looking for an answer?



Ask a question



Search for more help

**Sign up for our newsletter** Get the latest tutorials on SysAdmin and open source topics.



Enter your email address

Sign Up