scrape.do

Home Blog Web Scraping With Playwright

Category: Headless Browser

Web Scraping With Playwright

30 mins read Created Date: October 14, 2024 Updated Date: October 14, 2024

Table of Contents

Web scraping has become an essential skill for developers dealing with data extraction from web applications. While a lot of ink has been spilled in the Playwright vs Puppeteer web scraping debate, Playwright is a powerful solution for handling modern, dynamic websites that rely heavily on JavaScript.

This Playwright web scraping tutorial will explore advanced strategies for extracting data from JavaScript-heavy websites with Playwright, overcoming anti-scraping mechanisms, and handling dynamic content. It will provide step-by-step examples to ensure you can implement these techniques efficiently in your projects.

Whether you're dealing with CAPTCHA challenges, dynamic page
oing defenses, this guide will help you
ntial to meet your data extraction
et's dive right in!

Setup

Before diving into advanced Playwright web scraping techniques, let's first set up Playwright in a Node.js environment. If you already have Node.js and npm installed, you're ready to start by installing Playwright and its dependencies.

Playwright can be installed through npm, which simplifies the process. To install Playwright, run the following command in your project directory:

```
npm install playwright
```

This command installs Playwright along with browser binaries for Chromium, Firefox, and WebKit by default. These three browsers allow you to scrape a wide range of modern websites, giving you the flexibility to choose the right browser for each scraping task.

Playwright provides support for:

- **Chromium**: The core of Google Chrome and Microsoft Edge. Ideal for scraping websites optimized for Chrome-based browsers.
- **Firefox**: Perfect for scenarios where Firefox-specific rendering or behavior is required.
- **WebKit**: Used by Safari, enabling you to scrape websites as they appear in Apple's browser.

By default, when you install Playwright, all three browser binaries

lly. However, if you only need specific your install footprint, you can choose need. Here's how to install specific


```
# Install only Chromium
```

```
npm install playwright-chromium

# Install only Firefox
npm install playwright-firefox

# Install only WebKit
npm install playwright-webkit
```

Once Playwright is installed, launching a browser instance is straightforward. Here's an example of how to launch a Chromium browser, navigate to a website, and take a screenshot using Playwright:

```
const { chromium } = require('playwright');

(async () => {
// Launch a browser instance
const browser = await chromium.launch({
headless: false // Set to true if you want a headless browser (withou
});

// Create a new browser context
const context = await browser.newContext();

// Open a new page
const page = await context.newPage();

// Navigate to a website
await page.goto('https://www.scrapingcourse.com/ecommerce/');

// Take a screenshot of the page
await page.screenshot({ path: 'example.png' });

// Close the browser
await browser.close();
})();
```

Before launching the instance, you'll have to do this first:

from your terminal:

Launching Browsers and Navigating Pages

Like we said earlier, Playwright supports launching three types of browsers: Chromium, Firefox, and WebKit, giving you flexibility based on the specific requirements of your scraping tasks.

Here's how top launch the browser for Chromium:

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true }); // Headles
const page = await browser.newPage();
await page.goto('https://example.com');
console.log(await page.title());
await browser.close();
})();
```

For Firefox:

```
const { firefox } = require('playwright');

(async () => {
const browser = await firefox.launch({ headless: true }); // Headless
const page = await browser.newPage();
await page.goto('https://example.com');
console.log(await page.title());
await browser.close();
})();
```



```
playwright');

.launch({ headless: true }); // Headless
ewPage();
mple.com');
());
```

```
})();
```

In these examples, we launch the specified browser in headless mode, open a new page, navigate to an example website, print the page title, and then close the browser.

Speaking of headless mode, Playwright supports both headless (no UI) and headed (with UI) modes. By default, Playwright launches in headless mode, but you can toggle it by setting headless: **false** for non-headless mode.

Both headless and non-headless modes have advantages, so you'll have to weigh both when deciding which mode to use.

Advantages of Headless Mode

- **Speed**: Headless browsers are faster because they don't need to render a user interface.
- **Resource Efficiency**: They consume less CPU and memory, making them ideal for large-scale scraping tasks.
- **Automation**: Suitable for scenarios where browser interaction is not needed (e.g., data scraping and API testing).

Advantages of Non-Headless Mode(Headed mode)

- **Debugging**: Headed mode allows you to see what the browser
  or debugging web scraping issues
  unexpected behaviors).

manual interaction is needed during
e.g., filling out forms with captchas or
ts).

When to Use Headless vs. Non-Headless Modes

Headless mode is ideal for production scraping environments where performance and speed are critical. By running the browser without a visible UI, headless mode reduces CPU and memory consumption, allowing for faster interaction with web pages.

This makes it perfect for large-scale scraping tasks or automated testing, where you don't need to visually monitor the browser's behavior. Additionally, its efficiency makes it well-suited for cloud-based or virtual environments, where resources are often limited, and running multiple scraping tasks in parallel is necessary.

On the other hand, non-headless mode is better suited for development and debugging. When scraping or automating, seeing the browser's actions visually helps identify and resolve issues such as failed navigation, missing elements, or unexpected errors. Non-headless mode provides direct feedback, allowing developers to monitor every step, which can be crucial in diagnosing problems with scraping workflows.

It's also particularly useful when dealing with websites that employ anti-scraping measures—some sites treat headless browsers differently, and running in non-headless mode can sometimes bypass these restrictions.

ic Content ered Pages)

erful features is its ability to handle y on JavaScript-heavy websites. Many

modern web applications rely on client-side rendering, meaning key elements of the page load only after JavaScript execution. Playwright scraping dynamic content allows you to efficiently scrape such pages by waiting for JavaScript to fully execute, ensuring that the content you need is rendered before extraction begins.

When scraping websites that rely on JavaScript, simply navigating to the page and extracting the DOM may result in incomplete data, as many elements appear only after the browser executes JavaScript. Here's an example of how Playwright can scrape such content by waiting for elements to appear:

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the e-commerce page
await page.goto('https://www.scrapingcourse.com/ecommerce/');

// Wait for the product names to load (using the class selector for "
await page.waitForSelector('.product-name'); // Wait until the produc

// Extract the content of all elements with the class "product-name"
const productNames = await page.$$eval('.product-name', elements =>
 elements.map(el => el.textContent.trim()) // Map over all product na
);

console.log('Extracted Product Names:', productNames); // Log the ext

await browser.close();
})();
```


its for an element with the #dynamic-
uring that JavaScript has rendered the
ction. This approach ensures you're
age instead of partial data.

Scenarios Where Waiting is Crucial

- **Single-Page Applications (SPA)**: SPAs load content dynamically without refreshing the page, making it necessary to wait for specific elements or network activity to complete before scraping. In SPAs, data often arrives through API calls after the initial page load, and Playwright can handle this by waiting for the right conditions before interacting with the page.
- **Infinite Scrolling**: Websites that implement infinite scrolling continuously load new content as the user scrolls down. Playwright can automate scrolling behavior and wait for new content to load before scraping. Here's an example of how to handle infinite scrolling:

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to a page with infinite scroll
await page.goto('https://example.com/infinite-scroll');

// Scroll down to load more content
let previousHeight;
while (true) {
previousHeight = await page.evaluate('document.body.scrollHeight');
await page.evaluate('window.scrollTo(0, document.body.scrollHeight)')
await page.waitForTimeout(1000); // Adjust timeout as needed

let newHeight = await page.evaluate('document.body.scrollHeight');
if (newHeight === previousHeight) {
break; // Stop scrolling if no new content loads
}
```



```
nt
val('.item-selector', elements => element
:', items);
```

Another useful Playwright feature is the ability to wait for network activity to finish before proceeding. This is particularly important on pages where resources (like images, scripts, or data) load asynchronously. You can wait for the network to become idle before attempting to scrape the content:

```
await page.goto('https://example.com', { waitUntil: 'networkidle' });
```

When you implement this, Playwright waits for the network to be idle (no more than two network connections open) before continuing, ensuring that all dynamic content, including JavaScript resources, is fully loaded.

Form Submission and Authentication

Scraping data from websites often involves interacting with forms, especially on pages requiring authentication, such as login forms.

Playwright makes it easy to automate form submissions by programmatically filling in form fields and submitting them. This is particularly useful for scraping content from websites that require a user account or authentication to access certain pages.

Let's look at a typical login automation example using Playwright. Suppose you're scraping a website that requires login credentials

ord). Here's how to automate this


```
('playwright');

um.launch({ headless: true });
ewPage();
```

```
// Navigate to the login page
await page.goto('https://example.com/login');

// Fill in the login form
await page.fill('input[name="username"]', 'your-username'); // Fill i
await page.fill('input[name="password"]', 'your-password'); // Fill i

// Submit the form
await page.click('button[type="submit"]'); // Click the login button

// Wait for navigation after login
await page.waitForNavigation();

// Check if login was successful by checking for an element only visi
const loggedIn = await page.$('selector-for-logged-in-element');
if (loggedIn) {
console.log('Login successful!');
} else {
console.log('Login failed.');
}

await browser.close();
})();
```

In this example, the script first navigates to the login page and then fills in the username and password fields using the await page.fill() method. After the credentials are entered, the form is submitted by clicking the submit button via await page.click(). The script then waits for the navigation to complete after form submission to ensure that the login process is handled. Finally, it checks for the presence of a specific element that only appears after a successful login, verifying whether the login attempt was successful.

Handling CAPTCHA Challenges

Many websites that require authentication may include CAPTCHA ated logins. These CAPTCHAs can but Playwright provides solutions to ke Playwright-reCAPTCHA to A challenges. These plugins integrate

with your Playwright workflow and help bypass some CAPTCHA challenges.

```
const { chromium } = require('playwright');
const { getReCaptchaSolution } = require('playwright-recaptcha-plugin

(async () => {
const browser = await chromium.launch({ headless: false });
const page = await browser.newPage();

try {
// Navigate to the login page
await page.goto('https://example.com/login');

// Solve reCAPTCHA (requires a CAPTCHA-solving service)
const { solved, error } = await getReCaptchaSolution(page, {
provider: {
id: '2captcha',
token: 'YOUR_2CAPTCHA_API_KEY' // Replace with your 2Captcha API key
}
});

if (solved) {
console.log('CAPTCHA solved successfully!');
} else {
throw new Error(`CAPTCHA not solved: ${error}`);
}

// Fill the login form and submit
await page.fill('input[name="username"]', 'your-username');
await page.fill('input[name="password"]', 'your-password');
await page.click('button[type="submit"]');

// Wait for navigation to ensure login was successful
await page.waitForNavigation();

console.log('Login form submitted successfully!');
} catch (error) {
console.error('An error occurred:', error);
} finally {
await browser.close();
}
```


ıtes the login process on a website with
/right-recaptcha-plugin. It navigates
eCAPTCHA, fills in the username and
orm, and waits for the page to load.

However, to make sure this script works properly, Ensure you have the necessary dependencies installed:

```
npm install playwright playwright-recaptcha-plugin
```

For more complex CAPTCHA types (e.g., image-based CAPTCHAs), integrating external services like 2Captcha can help solve CAPTCHA challenges using human workers, returning the solution to your script. Here's a simplified example of how you could integrate 2Captcha:

```
const axios = require('axios');
const { chromium } = require('playwright');
const fs = require('fs');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

try {
// Navigate to the login page
await page.goto('https://example.com/login');

// Capture the CAPTCHA image
const captchaImage = await page.$('img#captcha'); // Example selector
await captchaImage.screenshot({ path: 'captcha.png' });

// Convert CAPTCHA image to base64
const captchaBase64 = fs.readFileSync('captcha.png', { encoding: 'bas

// Send the CAPTCHA to 2Captcha for solving
const apiKey = 'YOUR_2CAPTCHA_API_KEY';
const captchaResponse = await axios.post(`https://2captcha.com/in.php
params: {
key: apiKey,
method: 'base64'
```



```
ponse.data.request;

ution

get(`https://2captcha.com/res.php`, {
params: {
```

```
key: apiKey,
action: 'get',
id: captchaId,
json: 1
}
});

if (result.data.status === 1) {
captchaSolution = result.data.request;
break;
} else if (result.data.status === 0) {
console.log('Waiting for CAPTCHA solution...');
await new Promise(res => setTimeout(res, 5000)); // Wait for 5 second
} else {
throw new Error(`Captcha solving failed: ${result.data.request}`);
}
}

console.log('CAPTCHA solution received:', captchaSolution);

// Fill in the CAPTCHA solution and other form fields
await page.fill('input[name="captcha"]', captchaSolution);
await page.fill('input[name="username"]', 'your-username');
await page.fill('input[name="password"]', 'your-password');

// Submit the form
await page.click('button[type="submit"]');
await page.waitForNavigation();

console.log('Form submitted successfully!');

} catch (error) {
console.error('An error occurred:', error);
} finally {
await browser.close();
}
})();
```

In this approach, the CAPTCHA image is extracted from the webpage and sent to 2Captcha for human solving. Once the solution is returned, it's entered into the CAPTCHA input field on the website

s, iFrames, and

sites, you may encounter popups, hat can disrupt the process.

Playwright offers effective strategies for managing these elements, ensuring your scraping flows smoothly without interruptions.

Handling Popups

Popups can occur unexpectedly during web scraping, often requiring action before you can continue scraping the main content. Playwright allows you to intercept and manage these popups by listening for new page events.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Listen for popups and close them based on their URLs
page.on('popup', async popup => {
const popupUrl = popup.url();
console.log(`Popup detected with URL: ${popupUrl}`);

// Check if the popup URL contains specific keywords
if (popupUrl.includes('ads') || popupUrl.includes('marketing')) {
console.log('Ad popup detected, closing it...');
await popup.close();
} else {
console.log('Non-ad popup detected, closing it...');
await popup.close();
}
});

// Navigate to the page
await page.goto('https://example.com');

// Continue scraping...

await browser.close();
})();
```


opup window triggered by the page

› prevent interruptions. You can also

ded (e.g., filling in a form or

Handling iFrames

Many websites load content inside iFrames, making it necessary to switch contexts before interacting with or scraping the data inside the iFrame. Playwright allows you to access and manipulate iFrames seamlessly.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to a page with an iFrame
await page.goto('https://example.com/page-with-iframe');

// Wait for the iFrame to load
const frame = await page.frame({ url: /iframe-url-pattern/ });

// Interact with elements inside the iFrame
const iframeContent = await frame.$eval('#iframe-element', el => el.t
console.log('iFrame Content:', iframeContent);

await browser.close();
})();
```

Here, Playwright waits for the iFrame to load, then accesses its content using the frame() method. You can now scrape data from the iFrame or interact with elements within it.

Handling Modal Dialogs

Modal dialogs often block interaction with the main page until they

em dialogs (like alerts or prompts) or

l by the website. Playwright provides

)th types, and can intercept and

)matically.

```
e('playwright');
```

```
(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Handle alert dialogs by accepting them automatically
page.on('dialog', async dialog => {
console.log(`Dialog message: ${dialog.message()}`);
await dialog.accept(); // You can also use dialog.dismiss() if needed
});

// Navigate to a page that triggers an alert
await page.goto('https://example.com');
// Continue with scraping tasks...

await browser.close();
})();
```

In this example, any alert or dialog that appears is intercepted and automatically accepted, preventing the dialog from halting the scraping process.

For custom modal dialogs that are part of the website's UI, you can interact with them like any other element, either dismissing them or extracting data.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to a page that shows a modal dialog
await page.goto('https://example.com');

// Wait for the modal dialog and close it
await page.waitForSelector('.modal-dialog');
await page.click('.modal-dialog .close-button'); // Close the modal b
sks...
```


or the modal dialog to appear, then dismisses the dialog before

proceeding with further scraping tasks.

Popups and modal dialogs can disrupt scraping flows if they are not handled correctly. By using event listeners and conditional logic in Playwright, you can automatically manage these elements to avoid interruptions.

Extracting and Manipulating Data

One of the primary objectives in web scraping is to extract structured data from the Document Object Model (DOM). Playwright allows you to interact with web pages programmatically and retrieve the data you need, and as you can see in the section where we handled dynamic content, you can use Playwright to extract text.

However, Playwright allows you do much more than that. For instance, you may need to extract an element's attribute (e.g., an image's src attribute or a link's href)

```
const { chromium } = require('playwright');
(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();
// Navigate to the page with products ordered by price
await page.goto('https://www.scrapingcourse.com/ecommerce/?orderby=pr
// Extract the 'src' attribute from all product images
const imageUrls = await page.$$eval('.products img', imgs => imgs.map
// Log the extracted image URLs
                          imageUrls);
```


oles is another common task web

vright can extract data from each cell

and organize it into a structured format like arrays or objects for further processing.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to a page with a table
await page.goto('https://example.com/table-page');

// Extract table data
const tableData = await page.$$eval('table tr', rows => {
return rows.map(row => {
const cells = Array.from(row.querySelectorAll('td'));
return cells.map(cell => cell.textContent.trim());
});
});

console.log('Table Data:', tableData);

await browser.close();
})();
```

Cleaning and Formatting Extracted Data

Once you've extracted raw data from the DOM, it may require cleaning and formatting to make it usable. Techniques like trimming whitespace, normalizing strings, or converting data types can help structure the data for further analysis. Let's look at a couple of these in more detail:

- **Trimming whitespace**: Use .trim() to remove unnecessary spaces


```
trim();
```

f you extract numeric data stored as
arseFloat() to convert them.

```
const price = parseFloat(rawPrice.replace('$', '')); // Convert price
```

- **Replacing unwanted characters**: Use .replace() to remove or replace unwanted characters like currency symbols or special characters.

```
const sanitizedText = rawText.replace(/[^\w\s]/gi, ''); // Remove spe
```

- **Formatting dates**: Convert date strings into a consistent format using JavaScript's Date object or libraries like moment.js for more complex formatting.

```
const formattedDate = new Date(rawDate).toISOString(); // Convert dat
```

By applying these cleaning and formatting techniques, you can transform raw data into structured, usable information, ready for further analysis or integration into a database.

Handling Pagination and Infinite Scrolling

When scraping data from websites, handling pagination and infinite scrolling is crucial for collecting data that spans multiple pages. Playwright pagination scraping provides the flexibility to s and infinite scroll scenarios, ensuring content efficiently.

o handle infinite scrolling, so let's look

Many websites use pagination to

divide large datasets across multiple pages. Playwright allows you to navigate through each page and scrape data sequentially.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the first page
await page.goto('https://www.scrapingcourse.com/ecommerce/');

let hasNextPage = true;

while (hasNextPage) {
// Scrape product names and prices on the current page
const products = await page.$$eval('.products .product', items =>
 items.map(item => {
const name = item.querySelector('.woocommerce-loop-product__title').t
const price = item.querySelector('.price').textContent.trim();
return { name, price };
})
);

console.log('Scraped Products:', products);

// Check if the "Next" button exists
const nextButton = await page.$('a.next.page-numbers');

if (nextButton) {
// Click the "Next" button and wait for the next page to load
try {
await Promise.all([
page.click('a.next.page-numbers'),
// Wait for the product grid to appear on the next page, increasing t
page.waitForSelector('.products', { timeout: 60000 })
]);
} catch (error) {
console.error('Navigation failed, retrying...', error);
// Optionally, implement retry logic here
}
} else {
```

In this example, the script navigates to a paginated website and scrapes data from the current page. It then checks for the presence of a “Next” button, clicks it, and waits for the next page to load before continuing the scraping process. The loop continues until no more pages are available.

Dealing with Anti-Scraping Mechanisms

Many websites employ anti-scraping measures to detect and block automated bots. These techniques can range from rate limiting and user-agent detection to more sophisticated methods like browser fingerprinting. To ensure your scraping operations remain undetected and effective, it’s pertinent to implement the following strategies that mimic human browsing behavior and overcome these protections.

Rotating User Agents

Websites often use user-agent strings to identify and block bots that repeatedly send requests with the same user-agent. To avoid detection, you can rotate user-agent strings with each request to simulate traffic coming from different browsers or devices.

```
const { chromium } = require('playwright');
```



```
um.launch({ headless: true });
ewPage();

tate

0; Win64; x64) AppleWebKit/537.36 (KHTML,
el Mac OS X 10_15_7) AppleWebKit/537.36 (
hone OS 14_6 like Mac OS X) AppleWebKit/6
```

```
// Rotate the user agent
```

```
const userAgent = userAgents[Math.floor(Math.random() * userAgents.le
await page.setUserAgent(userAgent);

// Navigate to the website
await page.goto('https://example.com');

// Continue scraping...

await browser.close();
})();
```

In this example, a random user-agent is selected from a predefined list and applied to the page request. This rotation helps make your requests appear more legitimate and harder to detect as bots.

Adding Delays Between Requests

Many websites implement rate-limiting measures, blocking bots that send too many requests too quickly. Adding delays between your requests can help you avoid triggering these limits and prevent your IP address from being banned.

```
function randomDelay(min, max) {
return Math.floor(Math.random() * (max - min + 1)) + min;
}

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the first page
await page.goto('https://example.com');

// Scrape data
// ...
```

```
before the next request
ndomDelay(1000, 3000)); // Wait between 1

 or repeat the process...
mple.com/page2');
```

Here, a random delay between 1 to 3 seconds is introduced between requests. Varying the delay prevents bots from making requests in a predictable pattern, making detection more difficult.

Rotating Proxies

Using a single IP address for multiple requests can quickly get you blocked. To avoid this, rotating proxies allow you to send requests from different IP addresses, making it appear as though they are coming from different users. Playwright supports rotating proxies by specifying different proxy servers for each request.

```
const { chromium } = require('playwright');

(async () => {
// Launch browser with a proxy
const browser = await chromium.launch({
headless: true,
proxy: {
server: 'http://your-proxy-server.com:8000', // Replace with your pro
}
});

const page = await browser.newPage();

// Navigate and scrape content through the proxy
await page.goto('https://example.com');

await browser.close();
})();
```

Using Stealth Mode to Bypass Bot

techniques such as browser

which includes checking for headless

size, or monitoring browser behaviors

ction. Playwright's stealth mode helps

you bypass these basic bot detection mechanisms by making the browser behave more like a real user.

Stealth mode can be implemented with the help of libraries like playwright-extra and the stealth plugin. First, you have to install these packages:

```
npm install playwright-extra playwright-extra-plugin-stealth
```

Next, you can run stealth mode like this:

```
const { chromium } = require('playwright-extra');
const stealth = require('playwright-extra-plugin-stealth');

// Use stealth plugin
chromium.use(stealth());

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the website
await page.goto('https://example.com');

// Scrape the page or perform actions here...
console.log('Stealth mode enabled, scraping...');

await browser.close();
})();
```

The stealth mode plugin helps to hide Playwright's automation characteristics by masking common attributes that websites use to detect bots, such as the navigator.webdriver property or certain

ave differently in headless mode. This

ces of being detected by basic bot-

ON, CSV, Database)

Once you have successfully scraped data, it's essential to save it in a format that allows further analysis or processing. Playwright integrates smoothly with various data storage formats like JSON and CSV, as well as databases such as MongoDB and PostgreSQL, enabling you to store the data in a structured, reusable way.

Saving Scraped Data to a JSON File

JSON is a common format for saving structured data. It's easy to read, write, and work with in most programming environments.

```
const { chromium } = require('playwright');
const fs = require('fs');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the website and scrape data
await page.goto('https://example.com');
const scrapedData = await page.$$eval('.data-item', items =>
items.map(item => ({
title: item.querySelector('.title').textContent.trim(),
price: item.querySelector('.price').textContent.trim()
}))
);

// Save the data as JSON
fs.writeFileSync('scrapedData.json', JSON.stringify(scrapedData, null
console.log('Data saved to scrapedData.json');

await browser.close();
})();
```

Saving Scraped Data to a CSV File

…t for tabular data. It's especially useful …like Excel or for further analysis in …

```
…e('playwright');
```

```
const fs = require('fs');
const { parse } = require('json2csv');

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Navigate to the website and scrape data
await page.goto('https://example.com');
const scrapedData = await page.$$eval('.data-item', items =>
items.map(item => ({
title: item.querySelector('.title').textContent.trim(),
price: item.querySelector('.price').textContent.trim()
}))
);

// Convert data to CSV format
const csv = parse(scrapedData);

// Save the data as CSV
fs.writeFileSync('scrapedData.csv', csv);
console.log('Data saved to scrapedData.csv');

await browser.close();
})();
```

Saving Scraped Data to a Database

For more complex use cases, storing scraped data in a database is ideal for querying and managing large datasets. Playwright's Node.js drivers make it easy to integrate with databases like MongoDB or PostgreSQL. Let's see a MongoDB example:

```
const { chromium } = require('playwright');
const { MongoClient } = require('mongodb');

(async () => {
// Launch the Chromium browser in headless mode (without GUI)
const browser = await chromium.launch({ headless: true });
                          ewPage(); // Open a new page/tab



                          bsite
                          mple.com');

                          : Select elements with the class '.data-i
                          d price for each product
                          ge.$$eval('.data-item', items =>

                          title').textContent.trim(), // Get the ti
price: item.querySelector( .price').textContent.trim() // Get the pri
```

```
}))
);

// Log the scraped data for verification
console.log('Scraped Data:', scrapedData);

// MongoDB connection string (local MongoDB instance)
const uri = 'mongodb://localhost:27017';
const client = new MongoClient(uri); // Initialize the MongoDB client

try {
// Connect to MongoDB server
await client.connect();

// Access the 'scrapingDB' database and the 'products' collection
const database = client.db('scrapingDB');
const collection = database.collection('products');

// Insert the scraped data into the 'products' collection
const result = await collection.insertMany(scrapedData);

// Log the number of inserted records for verification
console.log(`${result.insertedCount} records inserted into MongoDB`);
} catch (dbError) {
// Catch any errors related to MongoDB connection or insertion
console.error('Error inserting data into MongoDB:', dbError);
} finally {
// Ensure MongoDB client is closed after the operations are complete
await client.close();
}

} catch (scrapeError) {
// Catch any errors that occur during the scraping process
console.error('Error during scraping:', scrapeError);
} finally {
// Ensure the browser is closed after scraping is complete
await browser.close();
}
})();
```

Here, each scraped data item is inserted into a PostgreSQL products table. The pg Node.js client establishes a connection to the PostgreSQL database and executes SQL INSERT queries. Saving

ats like JSON and CSV, or directly into

ensures that your data is ready for

or integration into other systems.

nsiderations

Optimizing performance is essential when scraping data at scale with Playwright to speed up the process and reduce resource consumption. Several strategies can help improve the efficiency of your scrapers, such as running browsers in headless mode, controlling browser behavior to reduce unnecessary loading, and running scrapers concurrently with multiple browser instances.

Running Browsers in Headless Mode

As we said earlier, Playwright launches browsers in headless mode, which significantly improves performance because the browser doesn't need to render a user interface. Using Playwright headless browser reduces the amount of CPU and memory used, making headless mode ideal for scraping large volumes of data or running multiple scrapers in parallel.

Running in headless mode is especially beneficial in environments where graphical user interfaces are unnecessary, such as server-based scraping jobs or cloud environments.

Disabling Images, Stylesheets, and Other Resources

For many scraping tasks, loading images, stylesheets, and fonts is unnecessary and adds overhead. By disabling these resource types, you can significantly speed up page load times and reduce


```
('playwright');


um.launch({ headless: true });
ewPage();

.in types of requests
await page.route('**/*', (route) => {
```

```
const request = route.request();
const blockResourceTypes = ['image', 'stylesheet', 'font'];
if (blockResourceTypes.includes(request.resourceType())) {
route.abort();
} else {
route.continue();
}
});

await page.goto('https://example.com');
await browser.close();
})();
```

In this example, Playwright intercepts and blocks image, stylesheet, and font requests, allowing the page to load faster by focusing only on essential resources like HTML and JavaScript. This is particularly useful when scraping data-focused content, such as text or structured data.

Running Scrapers Concurrently

To improve scraping efficiency and handle large-scale scraping tasks, running multiple browser instances concurrently can drastically reduce the time it takes to gather data. Playwright allows you to launch multiple instances of browsers or pages, enabling parallel processing of web scraping tasks.

```
const { chromium } = require('playwright');

(async () => {
const browser = await chromium.launch({ headless: true });

// Define a function to scrape a single page
const scrapePage = async (url) => {
                            ewPage();


                             with a timeout
                            ut: 30000 }); // 30 seconds timeout for p

                             element
                            l('h1', el => el.textContent.trim());


                            om ${url}: ${data}`);
} catch (error) {
```

```
// Handle errors such as timeout or element not found
console.error(`Error scraping ${url}:`, error);
} finally {
// Close the page after scraping is complete
await page.close();
}
};

// List of URLs to scrape concurrently
const urls = [
'https://example.com/page1',
'https://example.com/page2',
'https://example.com/page3'
];

try {
// Run the scrapers concurrently and wait for all of them to finish
await Promise.all(urls.map(url => scrapePage(url)));
} catch (error) {
console.error('Error during the scraping process:', error);
} finally {
// Close the browser after all pages have been scraped
await browser.close();
}
})();
```

Here, the script launches a new browser page for each URL and runs the scrapers concurrently using Promise.all(). This allows multiple scraping tasks to run in parallel, significantly improving the speed and efficiency of the process. You can launch multiple browser instances in parallel for even greater scalability.

Error Handling and Debugging

Web scraping is inherently prone to errors, such as network timeouts, unexpected element changes, or site blocks. Handling

es that your scraper remains resilient.

gging can help you quickly resolve

ur scraping tasks run smoothly.

rror-handling strategies and

try mechanisms and the ability to

es. Let's look at these in more detail.

Implementing a Retry Mechanism

Web scraping may encounter temporary issues like network errors or server overloads. Implementing a retry mechanism allows your scraper to attempt the task again before failing entirely. This reduces the chance of incomplete data extraction due to transient issues.

```
const { chromium } = require('playwright');
const retry = async (fn, retries = 3, delay = 1000) => {
for (let i = 0; i < retries; i++) {
try {
return await fn();
} catch (error) {
console.log(`Attempt ${i + 1} failed: ${error.message}`);
if (i < retries - 1) {
await new Promise(resolve => setTimeout(resolve, delay));
} else {
throw error; // Throw error after exhausting all retries
}
}
}
};

(async () => {
const browser = await chromium.launch({ headless: true });
const page = await browser.newPage();

// Retry scraping the page up to 3 times in case of failures
await retry(async () => {
await page.goto('https://example.com', { timeout: 5000 });
const data = await page.$eval('h1', el => el.textContent.trim());
console.log('Scraped data:', data);
});

await browser.close();
})();
```


scraping process up to three times with attempts, ensuring that transient uts, don't cause the script to fail

Errors

You can also handle specific errors, such as timeouts or missing elements, and take appropriate action based on the type of error.

```
try {
await page.goto('https://example.com', { timeout: 5000 });
const content = await page.$eval('.content', el => el.textContent.tri
console.log('Page content:', content);
} catch (error) {
if (error.name === 'TimeoutError') {
console.log('Navigation timed out.');
} else if (error.message.includes('No node found for selector')) {
console.log('Element not found.');
} else {
console.log('An unexpected error occurred:', error.message);
}
}
```

In this case, the script catches and handles errors based on the type, ensuring more informative error reporting and tailored responses to specific issues.

Debugging with Playwright's Built-in Tools

Playwright provides several built-in tools to aid debugging, including taking screenshots, recording traces, and viewing browser interactions visually.

- **Taking Screenshots :** Screenshots are a great way to visually confirm the state of the page at a given time, especially if your scraper encounters issues with missing or misidentified

g Tool: Playwright's tracing tool allows
tivities and review exactly what
aping session, making it easier to
nt wrong.

```
await page.tracing.start({ screenshots: true, snapshots: true });
await page.goto('https://example.com');

// Perform your scraping tasks...

// Save the trace for analysis
await page.tracing.stop({ path: 'trace.zip' });
console.log('Trace saved: trace.zip');
```

With tracing enabled, Playwright captures screenshots and browser interactions during the session. The trace file (trace.zip) can be opened in Playwright's trace viewer, which provides a detailed view of the interactions, including DOM snapshots, network requests, and timing information.

***Pro tip**: Running Playwright in non-headless mode (with a visible browser) is often helpful for debugging. You can visually inspect interactions and ensure the scraper behaves as expected.*

Conclusion

While Playwright is a powerful tool for scraping dynamic, JavaScript-heavy websites, setting it up and implementing advanced strategies like handling CAPTCHAs, rotating proxies, and bypassing anti-scraping mechanisms can be time-consuming and complex.

If you're looking for a more straightforward solution that simplifies the scraping process, services like Scrape.do offer a more efficient

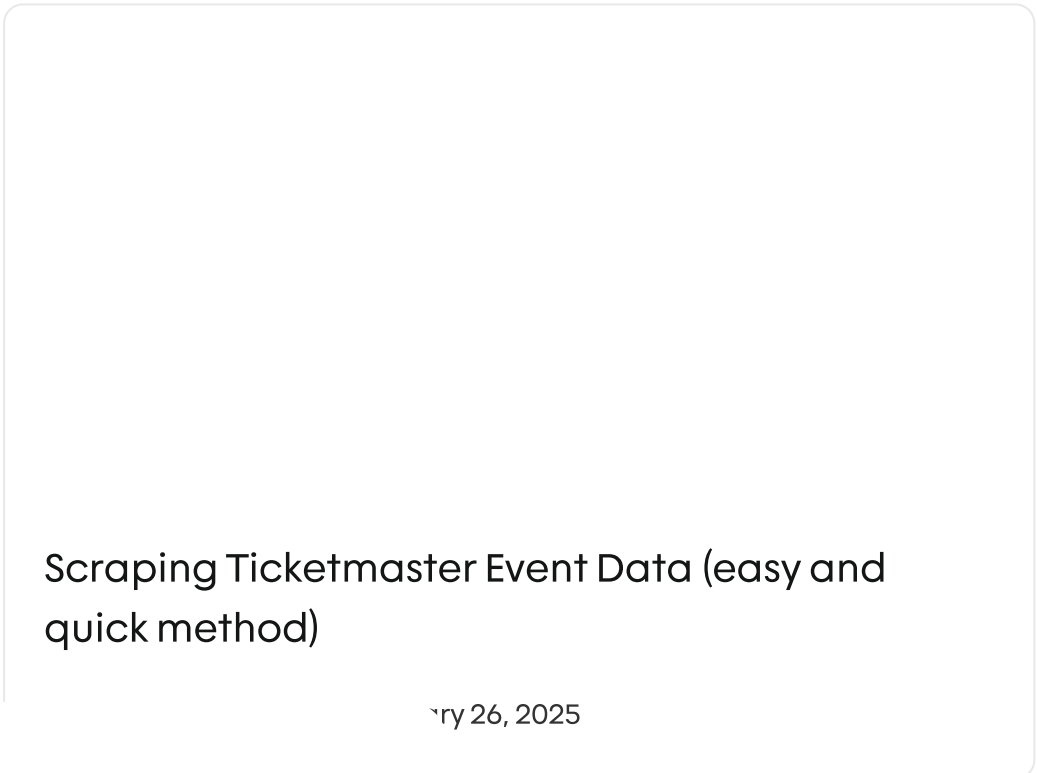
ıcts away much of the technical setup,
ecting data without worrying about
, or anti-bot measures.

form large-scale scraping with less
ecially when compared to setting up

and managing Playwright scripts. With built-in proxy rotation, CAPTCHA handling, and API-based access, Scrape.do is a robust, ready-made solution for extracting data efficiently and quickly, making it a strong choice for developers who want to avoid the complexities of browser automation and focus purely on gathering the data they need.

You can get started with Srape.do now for free!

Also You May Interest



Web Scraping in NodeJS: Advanced Techniques and Best Practices

Scraping Basics September 18, 2024

6 Effective Ways to Scrape & Improve Shopify Business

Scraping Use Cases September 18, 2024

Contact
Features
FAQ
Term of Use
Privacy Policy
Affiliate Program

Web Scraping is Legal?

Web Scraping Proxies

Rotating Proxy

Search Engine Scraping

Learn Web Scraping

Web Scraping in Python

Web Scraping in NodeJS

Web Scraping in Java

Web Scraping in PHP

Web Scraping in R

Web Scraping in Ruby

Web Scraping in Golang

Web Scraping in C#

Web Scraping in Rust

Web Scraping in C++

Web Scraping in C

Web Scraping In Perl

Scrapy Python Web Scraping

Selenium Web Scraping

Playwright Web Scraping

Puppeteer Web Scraping
