nullshot

Platform

Browser Rendering

Enable your agents to scrape and extract web content in real-time. Cloudflare Browser Rendering provides headless browser automation capabilities that your agents can use to capture screenshots, extract HTML content, scrape specific elements, and gather structured data from any webpage.

Key Features

- **Puppeteer Integration**: Full Puppeteer API support with Cloudflare's optimized fork
- **Session Management**: Create, reuse, and manage browser sessions with keep-alive functionality
- **Content Extraction**: Extract HTML, text, and structured data from web pages
- **Screenshot Capture**: Take screenshots of web pages for visual analysis
- **PDF Generation**: Convert web pages to PDF documents
- **Element Scraping**: Target specific HTML elements for data extraction
- **Global Network**: Browser instances run on Cloudflare's edge network worldwide
- **Rate Limiting**: Built-in limits and monitoring for production usage

Getting Started

Wrangler Configuration

First, add browser rendering in your `wrangler.json`:

```
{
  "bindings": [
    {
      "binding": "BROWSER",
      "type": "browser"
    }
  ]
}
```

Agent Example

TYPESCRIPT Copy

```
// src/agents/WebScrapingAgent.ts
import { AiAgentSDK } from '@typescript-agent-framework/agent';

export class WebScrapingAgent extends AiAgentSDK {
  constructor(env: Env) {
    super(env);
  }

  async processMessage(sessionId: string, messages: AIUISDKMessage):
Promise<AgentResponse> {
    // This is an unrealistic example, but showcases how you can use browser
rendering with agents
    const enrichedMessage = await this.enrichMessageWithWebContent(message);

    const result = await this.streamText(sessionId, {
      model: this.model,
      system: 'You are a helpful assistant with access to real-time web
content.',
      messages: enrichedMessage,
      maxSteps: 10,
    });

    return result.toDataStreamResponse();
  }

  // This is an unrealistic example, but showcases how you can use browser
rendering with agents
```

```
    enrichedContext: any[];
  }> {
    try {
      // Extract URLs from the message for content enrichment
      const urls = this.extractUrlsFromMessage(message.content);

      if (urls.length === 0) {
        return {
          content: message.content,
          enrichedContext: []
        };
      }

      // Create a new page for web scraping
      const page = await this.browser.newPage();

      // Scrape content from each URL
      const enrichedContext = [];

      for (const url of urls.slice(0, 3)) {
        try {
          // Set viewport for consistent screenshots
          await page.setViewport({ width: 1200, height: 800 });

          // Navigate to the URL
          await page.goto(url, { waitUntil: 'networkidle0' });

          // Extract main content
          const content = await page.evaluate(() => {
            const selectors = ['main', 'article', '.content', '#content',
'.post'];
            for (const selector of selectors) {
              const element = document.querySelector(selector);
              if (element) {
                return {
                  title: document.title,
                  text: element.textContent?.substring(0, 500) || ''
                };
              }
            }
            return {
              title: document.title,
              text: document.body.textContent?.substring(0, 500) || ''
```

```
            // Take screenshot for visual context
            const screenshot = await page.screenshot({
              type: 'png',
              fullPage: false
            });

            enrichedContext.push({
              url,
              title: content.title || 'Unknown Title',
              text: content.text,
              screenshot: screenshot,
              extractedAt: Date.now()
            });
          } catch (error) {
            console.error(`Failed to scrape ${url}:`, error);
          }
        }

        // Close the page when done
        await page.close();

        // Add web content to the message
        let enrichedContent = message.content;
        if (enrichedContext.length > 0) {
          const contextSummary = enrichedContext
            .map(ctx => `Web Content from ${ctx.url}:\n${ctx.text}`)
            .join('\n\n');

          enrichedContent = `${message.content}\n\n[Web Content
Added]:\n${contextSummary}`;
        }

        return {
          content: enrichedContent,
          enrichedContext
        };
      } catch (error) {
        console.error('Failed to enrich message with web content:', error);
        return {
          content: message.content,
          enrichedContext: []
        };
      }
```

nullshot

MCP Tool Integration

Integrate browser rendering capabilities with MCP tools:

TYPESCRIPT Copy

```typescript
// src/mcp/BrowserRenderingMCPServer.ts
import { MCPServerDO } from '@typescript-agent-framework/mcp';

export class BrowserRenderingMCPServer extends MCPServerDO<Env> {
  constructor(state: DurableObjectState, env: Env) {
    super(state, env);
  }

  protected configureServer() {
    return {
      'scrape-content': async ({ url, selector }: {
        url: string;
        selector?: string
      }) => {
        try {
          // Create a new page for scraping
          const page = await this.browser.newPage();

          // Navigate to the URL
          await page.goto(url, { waitUntil: 'networkidle0' });

          // Extract content using the specified selector
          const content = await page.evaluate((sel) => {
            const element = document.querySelector(sel);
            return element ? element.textContent : '';
          }, selector || 'body');

          // Close the page
          await page.close();

          return {
            success: true,
            url,
            content: content || '',
```

```
    } catch (error) {
      return {
        success: false,
        error: error.message,
        url
      };
    }
  },

  'take-screenshot': async ({ url, viewport }: {
    url: string;
    viewport?: { width: number; height: number }
  }) => {
    try {
      const page = await this.browser.newPage();

      // Set viewport if provided
      if (viewport) {
        await page.setViewport(viewport);
      }

      // Navigate and take screenshot
      await page.goto(url, { waitUntil: 'networkidle0' });
      const screenshot = await page.screenshot({
        type: 'png',
        fullPage: false
      });

      await page.close();

      return {
        success: true,
        url,
        screenshot: screenshot.toString('base64'),
        capturedAt: Date.now()
      };
    } catch (error) {
      return {
        success: false,
        error: error.message,
        url
      };
    }
```

```
}
```

TypeScript API Reference

You can access the browser rendering API via the `env` of the **Agent** or **MCP Tool**

TYPESCRIPT Copy

```
// Example function that resides within any Agent or MCP Tool
function example() {
  // BROWSER can be named to anything and equals the "binding" sent in
wrangler.json
  const { BROWSER } = this.env;
}
```

Browser Instance Methods

`this.browser.newPage(): Promise<Page>`
Create a new page within the browser instance for web operations.

`this.browser.isConnected(): boolean`
Check if the browser session is still active and connected.

`this.browser.close(): Promise<void>`
Permanently close the browser instance and terminate all sessions.

`this.browser.disconnect(): void`
Disconnect from browser session while keeping it alive for reuse.

Page Methods

`page.goto(url, options?): Promise<Response>`
Navigate to a specific URL and wait for the page to load.

`page.screenshot(options?): Promise<Buffer>`
Capture a screenshot of the current page with customizable options.

page.evaluate(fn, ...args): Promise<any>
Execute JavaScript code within the page context.

page.waitForSelector(selector): Promise<ElementHandle>
Wait for an element to appear on the page.

page.close(): Promise<void>
Close the current page while keeping the browser session alive.

Browser Options

TYPESCRIPT Copy

```typescript
interface ScreenshotOptions {
  url: string;
  viewport?: { width: number; height: number };
  fullPage?: boolean;
  format?: 'png' | 'jpeg';
  quality?: number; // 1-100 for JPEG
  waitFor?: number; // milliseconds to wait
}

interface PDFOptions {
  url: string;
  format?: 'A4' | 'Letter' | 'Legal';
  landscape?: boolean;
  margin?: {
    top?: string;
    right?: string;
    bottom?: string;
    left?: string;
  };
  printBackground?: boolean;
}

interface ScrapeOptions {
  url: string;
  elements: Array<{
    selector: string;
    attribute?: 'text' | 'innerHTML' | 'outerHTML' | string;
  }>;
```

```
interface StructuredDataOptions {
  url: string;
  schema: Record<string, string>; // CSS selectors for data extraction
  waitFor?: number;
}
```

Examples

Learn how to integrate browser rendering capabilities into your agents and MCP tools through practical, real-world implementations.

Basic Puppeteer Usage

Get started with Cloudflare's Puppeteer integration using the standard Puppeteer API. For complete API documentation, see Cloudflare's Puppeteer Platform Guide.

TYPESCRIPT Copy

```
// src/agents/PuppeteerAgent.ts
import { AiAgentSDK } from '@typescript-agent-framework/agent';
import puppeteer from '@cloudflare/puppeteer';

export class PuppeteerAgent extends AiAgentSDK {
  constructor(env: Env) {
    super(env);
  }

  async processMessage(sessionId: string, messages: AIUISDKMessage):
Promise<AgentResponse> {
    const { BROWSER } = this.env;

    if (message.content.includes('analyze page')) {
      const urls = this.extractUrlsFromMessage(message.content);
      if (urls.length > 0) {
        const analysis = await this.analyzePage(urls[0]);
        return {
          message: `Page analysis complete for ${urls[0]}`,
          actions: [{
```

```
        };
      }
    }

    return { message: "Send me a URL to analyze!" };
  }

  private async analyzePage(url: string) {
    const { BROWSER } = this.env;

    // Launch browser with 10-minute keep-alive
    const browser = await puppeteer.launch(BROWSER, {
      keep_alive: 600000 // 10 minutes
    });

    try {
      const page = await browser.newPage();

      // Set mobile viewport
      await page.setViewport({ width: 375, height: 667 });

      // Navigate and wait for network to be idle
      await page.goto(url, { waitUntil: 'networkidle0' });

      // Get page metrics and content
      const [metrics, title, screenshot] = await Promise.all([
        page.metrics(),
        page.title(),
        page.screenshot({ type: 'png', fullPage: false })
      ]);

      // Extract structured data
      const analysis = await page.evaluate(() => {
        return {
          title: document.title,
          description:
document.querySelector('meta[name="description"]')?.getAttribute('content'),
          headings: Array.from(document.querySelectorAll('h1, h2,
h3')).map(h => ({
            tag: h.tagName.toLowerCase(),
            text: h.textContent?.trim()
          })),
          links: Array.from(document.querySelectorAll('a[href]')).length,
```

```
      await page.close();

      return {
        url,
        metrics,
        analysis,
        screenshot: screenshot.toString('base64'),
        analyzedAt: Date.now()
      };
    } finally {
      // Browser will stay alive for 10 minutes due to keep_alive setting
      await browser.disconnect(); // Don't close, just disconnect
    }
  }

  private extractUrlsFromMessage(content: string): string[] {
    const urlRegex = /https?:\/\/[^\s]+/g;
    return content.match(urlRegex) || [];
  }
}
```

Session Reuse Example

Optimize performance by reusing browser sessions with Durable Objects. This advanced pattern keeps browser instances alive for 60 seconds, reducing startup time and improving response times for multiple requests.

TYPESCRIPT Copy

```
// src/agents/BrowserSessionAgent.ts
import { AiAgentSDK } from '@typescript-agent-framework/agent';
import puppeteer from '@cloudflare/puppeteer';

export class BrowserSessionAgent extends AiAgentSDK {
  private browser: any = null;
  private keptAliveInSeconds = 0;
  private readonly KEEP_BROWSER_ALIVE_IN_SECONDS = 60;

  constructor(env: Env) {
```

```
  async processMessage(sessionId: string, messages: AIUISDKMessage):
Promise<AgentResponse> {
    if (message.content.includes('screenshot')) {
      const urls = this.extractUrlsFromMessage(message.content);
      if (urls.length > 0) {
        const screenshots = await this.takeScreenshots(urls);
        return {
          message: `Captured ${screenshots.length} screenshots`,
          actions: [{
            type: 'screenshots_taken',
            data: { screenshots }
          }]
        };
      }
    }

    return { message: "Send me URLs to take screenshots!" };
  }

  private async takeScreenshots(urls: string[]) {
    // Reuse existing browser session or create new one
    if (!this.browser || !this.browser.isConnected()) {
      console.log('Starting new browser session');
      this.browser = await puppeteer.launch(this.env.BROWSER);
    }

    // Reset keep-alive timer
    this.keptAliveInSeconds = 0;

    const screenshots = [];
    const page = await this.browser.newPage();

    // Take screenshots of different screen sizes
    const viewports = [
      { width: 1920, height: 1080 },
      { width: 1366, height: 768 },
      { width: 414, height: 896 }
    ];

    for (const url of urls.slice(0, 3)) {
      for (const viewport of viewports) {
        try {
          await page.setViewport(viewport);
```

```
            type: 'jpeg',
            quality: 80
          });

          screenshots.push({
            url,
            viewport,
            screenshot: screenshot.toString('base64'),
            capturedAt: Date.now()
          });
        } catch (error) {
          console.error(`Failed to screenshot ${url}:`, error);
        }
      }
    }

    // Close page but keep browser session alive
    await page.close();

    // Start keep-alive timer if not already running
    this.startKeepAliveTimer();

    return screenshots;
  }

  private startKeepAliveTimer() {
    if (this.keptAliveInSeconds === 0) {
      console.log('Starting browser keep-alive timer');
      this.scheduleKeepAlive();
    }
  }

  private scheduleKeepAlive() {
    setTimeout(async () => {
      this.keptAliveInSeconds += 10;

      if (this.keptAliveInSeconds < this.KEEP_BROWSER_ALIVE_IN_SECONDS) {
        console.log(`Browser kept alive for ${this.keptAliveInSeconds}s`);
        this.scheduleKeepAlive();
      } else {
        console.log(`Browser session timeout after
${this.KEEP_BROWSER_ALIVE_IN_SECONDS}s`);
        if (this.browser) {
```

```
      }
    }, 10000); // Check every 10 seconds
  }

  private extractUrlsFromMessage(content: string): string[] {
    const urlRegex = /https?:\/\/[^\s]+/g;
    return content.match(urlRegex) || [];
  }
}
```

Best Practices

Essential patterns and techniques for building robust, performant browser automation workflows. These practices help you avoid common pitfalls and optimize resource usage.

Efficient Scraping

Use targeted CSS selectors and avoid scraping entire page content to minimize bandwidth and processing time.

TYPESCRIPT Copy

```
// ✅ Good: Use specific selectors for targeted content
const efficientScraping = async (browser: any, url: string) => {
  return browser.fetch('/scrape', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      url,
      elements: [
        { selector: 'h1', attribute: 'text' },
        { selector: '.article-content', attribute: 'text' },
        { selector: 'meta[name="description"]', attribute: 'content' }
      ],
      waitFor: 2000
    })
  });
```

```
const inefficientScraping = async (browser: any, url: string) => {
  return browser.fetch('/scrape', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      url,
      elements: [{ selector: 'body', attribute: 'innerHTML' }]
    })
  });
};
```

Session Management

Implement browser session reuse patterns to reduce startup overhead and improve performance for multiple operations.

TYPESCRIPT Copy

```
// ✅ Good: Reuse browser sessions for related tasks
class BrowserSession {
  private sessionId: string | null = null;

  async getSession(browser: any): Promise<string> {
    if (!this.sessionId) {
      const response = await browser.fetch('/session', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ keepAlive: 300000 }) // 5 minutes
      });
      const data = await response.json();
      this.sessionId = data.sessionId;
    }
    return this.sessionId;
  }

  async scrapeWithSession(browser: any, url: string, selectors: string[]) {
    const sessionId = await this.getSession(browser);

    return browser.fetch('/scrape', {
      method: 'POST',
```

```
        sessionId,
        elements: selectors.map(selector => ({ selector, attribute: 'text'
}))
      })
    });
  }
}
```

Error Handling & Retries

Implement robust error handling with exponential backoff and retry logic for network failures and rate limits.

TYPESCRIPT Copy

```
// ✅ Good: Implement retry logic for failed requests
async function robustBrowserFetch(
  browser: any,
  endpoint: string,
  options: any,
  maxRetries = 3
): Promise<any> {
  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      const response = await browser.fetch(endpoint, options);

      if (response.ok) {
        return response;
      }

      if (response.status === 429) {
        // Rate limited, wait before retry
        await new Promise(resolve => setTimeout(resolve, 1000 * attempt));
        continue;
      }

      throw new Error(`HTTP ${response.status}: ${response.statusText}`);
    } catch (error) {
      console.error(`Attempt ${attempt} failed:`, error);
```

```
      // Exponential backoff
      await new Promise(resolve => setTimeout(resolve, 1000 * Math.pow(2,
attempt - 1)));
    }
  }
}
```

Performance Optimization

Batch operations and use parallel processing to maximize throughput while respecting rate limits and resource constraints.

TYPESCRIPT Copy

```
// ✅ Good: Batch multiple operations
async function batchBrowserOperations(browser: any, urls: string[]) {
  const promises = urls.map(async (url) => {
    try {
      const [screenshot, content, links] = await Promise.all([
        browser.fetch('/screenshot', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({
            url,
            viewport: { width: 800, height: 600 },
            format: 'jpeg',
            quality: 80
          })
        }),
        browser.fetch('/content', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ url })
        }),
        browser.fetch('/links', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ url })
        })
```

```
        url,
        screenshot: await screenshot.arrayBuffer(),
        content: await content.json(),
        links: await links.json()
      };
    } catch (error) {
      console.error(`Failed to process ${url}:`, error);
      return { url, error: error.message };
    }
  });

  return Promise.all(promises);
}
```

Limitations & Considerations

Important constraints and pricing considerations when using browser rendering at scale. Understanding these limits helps you design efficient workflows and manage costs effectively.

- **Rate Limits**: 1000 requests per minute per account
- **Session Timeout**: Browser sessions timeout after 5 minutes of inactivity
- **Memory Limits**: Complex pages may exceed memory allocation
- **JavaScript Execution**: Pages with heavy JavaScript may require longer wait times
- **Geographic Restrictions**: Some websites may block requests from certain regions
- **Cost**: Each browser operation incurs usage charges based on execution time

Related Services

- **Durable Objects** - Maintain browser sessions across requests
- **Queues** - Queue browser rendering tasks for batch processing
- **Storage** - Store screenshots, PDFs, and scraped content
- **Memory Store** - Cache frequently accessed web content

- Browser Rendering API Documentation
- REST API Reference
- Workers Bindings Guide

Previous
Storage (R2/S3)

Next
Workflows