Headless Chrome: an answer to server-side rendering JS sites

By Eric Bidelman
Engineer @ Google working on web tooling: Headless Chrome, Puppeteer, Lighthouse

TL;DR

Headless Chrome can be a drop-in solution for turning dynamic JS sites into static HTML pages. Running it on a web server allows you to **prerender any modern JS features** so content **loads fast** and is **indexable by crawlers**.

The techniques in this article show how to use Puppeteer's APIs to add server-side rendering (SSR) capabilities to an Express web server. The best part is **the app itself requires almost no code changes**. Headless does all the heavy lifting. In a couple of lines of code you can SSR any page and get its final markup.

A taste of what's to come:

```
import puppeteer from 'puppeteer';

async function ssr(url) {
  const browser = await puppeteer.launch({headless: true});
  const page = await browser.newPage();
  await page.goto(url, {waitUntil: 'networkidle0'});
  const html = await page.content(); // serialized HTML of page DOM.
  await browser.close();
  return html;
}
```

Note: I'll be using ES modules (`import`) in this article, which require Node 8.5.0+ and running with the `--experimental-modules` flag. Feel free to use `require()` statements if they bother you. Read more about Node's ES Modules support

Introduction

If SEO has served me well, you landed on this article for one of two reasons. First, you've built a web app and it's not being indexed the search engines! Your app might be a SPA, PWA, using vanilla JS, or built with something more complex like a library or framework. To be honest, your tech stack doesn't matter. What matters is that you spent a lot of time building Awesome Web Thing and users are unable to discover it. The other reason you might be here is because some article out on The Webz mentioned that server-side rendering is good for performance. You're here for that quick win to reduce JavaScript startup cost and improve first meaningful paint.

Some frameworks like Preact ship with tools that address server-side rendering. If your framework has a prendering solution, please stick with that. There's no reason to bring another tool (headless Chrome / Puppeteer) into the mix.

Crawling the modern web

Search engine crawlers, social sharing platforms, even browsers have historically relied exclusively on static HTML markup to index the web and surface content. The modern web has evolved into something much different. JavaScript-based applications are here to stay, which means that in many cases, our content can be invisible to crawling tools.

Some crawlers like Google Search have gotten smarter! Google's crawler uses Chrome 41 to execute JavaScript and render the final page, but that process is still new and not perfect. For example, pages that use newer features like ES6 classes, Modules, and arrow functions will cause JS errors in this older browser and prevent the page from rendering correctly. As for other search engines,...who knows what they're doing!? ¯\(ツ)/¯

Prerendering pages using headless Chrome

All crawlers understand HTML. What we need to "solve" the indexing problem is a tool that produces HTML from executing JS. What if I told you there is such a tool?

1. The tool knows how to run *all* types of modern JavaScript and spit out static HTML.
2. The tool stays up to date as the web adds features.
3. You can quickly run the tool against an existing app with little to no code changes.

Sounds good right? **That tool is the browser**!

Headless Chrome doesn't care what library, framework, or tool chain you use. It eats JavaScript for breakfast and spits out static HTML before lunch. Well, hopefully a lot faster than that :) -Eric

If you're in Node, Puppeteer is an easy way to work with headless Chrome. Its APIs make it possible to take a client-side app and prerender (or "SSR") its markup. Below is an example of doing that.

1. Example JS app

Let's start with a dynamic page that generates its HTML via JavaScript:

public/index.html

```
<html>
<body>
  <div id="container">
    <!-- Populated by the JS below. -->
  </div>
</body>
<script>
function renderPosts(posts, container) {
  const html = posts.reduce((html, post) => {
    return `${html}
      <li class="post">
        <h2>${post.title}</h2>
        <div class="summary">${post.summary}</div>
        <p>${post.content}</p>
      </li>`;
  }, '');

  // CAREFUL: assumes html is sanitized.
  container.innerHTML = `<ul id="posts">${html}</ul>`;
}

(async() => {
  const container = document.querySelector('#container');
  const posts = await fetch('/posts').then(resp => resp.json());
  renderPosts(posts, container);
})();
</script>
</html>
```

2. SSR function

Next, we'll take the `ssr()` function from earlier and beef it up a bit:

ssr.mjs

```
import puppeteer from 'puppeteer';

// In-memory cache of rendered pages. Note: this will be cleared whenever the
// server process stops. If you need true persistence, use something like
// Google Cloud Storage (https://firebase.google.com/docs/storage/web/start).
const RENDER_CACHE = new Map();

async function ssr(url) {
  if (RENDER_CACHE.has(url)) {
    return {html: RENDER_CACHE.get(url), ttRenderMs: 0};
  }

  const start = Date.now();

  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  try {
    // networkidle0 waits for the network to be idle (no requests for 500ms).
    // The page's JS has likely produced markup by this point, but wait longer
    // if your site lazy loads, etc.
    await page.goto(url, {waitUntil: 'networkidle0'});
    await page.waitForSelector('#posts'); // ensure #posts exists in the DOM.
  } catch (err) {
    console.error(err);
    throw new Error('page.goto/waitForSelector timed out.');
  }

  const html = await page.content(); // serialized HTML of page DOM.
  await browser.close();

  const ttRenderMs = Date.now() - start;
  console.info(`Headless rendered page in: ${ttRenderMs}ms`);

  RENDER_CACHE.set(url, html); // cache rendered page.

  return {html, ttRenderMs};
}

export {ssr as default};
```

The major changes:

1. Added caching. Caching the rendered HTML is the biggest win to speed up the response times. When the page gets re-requested, you avoid running headless Chrome altogether. I discuss other optimizations later on.
2. Add basic error handling if loading the page times out.
3. Add a call to `page.waitForSelector('#posts')`. This ensures that the posts exist in the DOM before we dump the serialized page.
4. Add science. Log how long headless takes to render the page and return the rendering time along with the HTML.
5. Stick the code in a module named `ssr.mjs`.

3. Example web server

Finally, here's the small express server that brings it all together. The main handler prerenders the URL `http://localhost/index.html` (e.g. the main page) and serves the result as its response. Users will immediately see posts when they hit the page because the static markup is now part of the response.

server.mjs

```
import express from 'express';
import ssr from './ssr.mjs';

const app = express();

app.get('/', async (req, res, next) => {
  const {html, ttRenderMs} = await ssr(`${req.protocol}://${req.get('host')}/inde
  // Add Server-Timing! See https://w3c.github.io/server-timing/.
  res.set('Server-Timing', `Prerender;dur=${ttRenderMs};desc="Headless render tim
  return res.status(200).send(html); // Serve prerendered page as response.
});

app.listen(8080, () => console.log('Server started. Press Ctrl+C to quit'));
```

To run this example, install the dependencies (`npm i --save puppeteer express`) and run the server using Node 8.5.0+ and the `--experimental-modules` flag:

Example of the response sent back by this server:

```
<html>
<body>
  <div id="container">
```

```
    <ul id="posts">
      <li class="post">
        <h2>Title 1</h2>
        <div class="summary">Summary 1</div>
        <p>post content 1</p>
      </li>
      <li class="post">
        <h2>Title 2</h2>
        <div class="summary">Summary 2</div>
        <p>post content 2</p>
      </li>
      ...
    </ul>
  </div>
</body>
<script>
...
</script>
</html>
```

A perfect use case for the new Server-Timing API

The Server-Timing API allows you to communicate server performance metrics (e.g. request/response times, db lookups) back to the browser. Client code can use this information to track overall performance of a web app.

A perfect use case for Server-Timing is to report how long it takes for headless Chrome to prerender a page! To do that, just add the **`Server-Timing`** header to the server response:

```
res.set('Server-Timing', `Prerender;dur=1000;desc="Headless render time (ms)`
```

On the client, the Performance Timeline API and/or PerformanceObserver can be used to access these metrics:

```
const entry = performance.getEntriesByType('navigation').find(
    e => e.name === location.href);
console.log(entry.serverTiming[0].toJSON());
```

```
{
  "name": "Prerender",
  "duration": 3808,
  "description": "Headless render time (ms)"
}
```

Performance results

Note: these numbers incorporate most of the performance optimizations I discuss later.

What about performance numbers? On one of my apps (code), headless Chrome takes about 1s to render the page on the server. Once the page is cached, DevTools **3G Slow emulation** puts FCP at **8.37s faster** than the client-side version.

	First Paint (FP)	First Contentful Paint (FCP)
Client-side app	4s	11s
SSR version	2.3s	~2.3s

These results are promising. Users see meaningful content much quicker because the server-side rendered page **no longer relies on JavaScript to load + shows posts**.

Preventing re-hydration

Remember when I said "we didn't make any code changes to the client-side app"? That was a lie.

Our Express app takes a request, uses Puppeteer to load the page into headless, and serves the result as a response. But this setup has a problem.

The **same JS that executes in headless Chrome** on the server **runs again** when the user's browser loads the page on the frontend. We have two places generating markup. #doublerender!

Let's fix it. We need to tell the page its HTML is already in place. The solution I found was to have the page JS check if `<ul id="posts">` is already in the DOM at load time. If it is, we know the page was SSR'd and can avoid re-adding posts again. 👌

public/index.html

```
<html>
<body>
  <div id="container">
    <!-- Populated by JS (below) or by prerendering (server). Either way,
         #container gets populated with the posts markup:
      <ul id="posts">...</ul>
    -->
```

```
  </div>
</body>
<script>
...
(async() => {
  const container = document.querySelector('#container');

  // Posts markup is already in DOM if we're seeing a SSR'd.
  // Don't re-hydrate the posts here on the client.
  const PRE_RENDERED = container.querySelector('#posts');
  if (!PRE_RENDERED) {
    const posts = await fetch('/posts').then(resp => resp.json());
    renderPosts(posts, container);
  }
})();
</script>
</html>
```

Optimizations

Apart from caching the rendered results, there are plenty of interesting optimizations we can make to `ssr()`. Some are quick wins while others may be more speculative. The performance benefits you see may ultimately depend on the types of pages you prerender and the complexity of the app.

Aborting non-essential requests

Right now, the entire page (and all of the resources it requests) is loaded unconditionally into headless Chrome. However, we're only interested in two things:

1. The rendered markup.
2. The JS requests that produced that markup.

Network requests that don't construct DOM are wasteful. Resources like images, fonts, stylesheets, and media don't participate in building the HTML of a page. They style and supplement the structure of a page but they don't explicitly create it. We should tell the browser to ignore these resources! Doing so would reduce the workload headless Chrome has to chew through, **save bandwidth** and potentially **speed up prerendering time** for larger pages.

The Devtools Protocol supports a powerful feature called Network interception which can be used to **modify requests before they're issued by the browser**. Puppeteer supports network

interception by turning on `page.setRequestInterception(true)` and listening for the page's `request` event. That allows us to abort requests for certain resources and let others continue through.

ssr.mjs

```
async function ssr(url) {
  ...
  const page = await browser.newPage();

  // 1. Intercept network requests.
  await page.setRequestInterception(true);

  page.on('request', req => {
    // 2. Ignore requests for resources that don't produce DOM
    // (images, stylesheets, media).
    const whitelist = ['document', 'script', 'xhr', 'fetch'];
    if (!whitelist.includes(req.resourceType())) {
      return req.abort();
    }

    // 3. Pass through all other requests.
    req.continue();
  });

  await page.goto(url, {waitUntil: 'networkidle0'});
  const html = await page.content(); // serialized HTML of page DOM.
  await browser.close();

  return {html};
}
```

Note: I'm using a whitelist to play it safe and allowing all other types of requests to continue. This may preemptively avoid any gotchas that arise from aborting more resources than necessary.

Inline critical resources

It's common to use separate build tools (e.g. `gulp`) to process an app and inline critical CSS/JS into the page at build-time. Doing so can speed up first meaningful paint because the browser makes fewer requests during initial page load.

Instead of a separate build tool, **use the browser as your build tool**! We can use Puppeteer to manipulate the page's DOM, inlining styles, JavaScript, or whatever else you want to stick in the page before prerendering it.

This example shows how to intercept responses for local stylesheets and inline those resources into the page as `<style>` tags:

ssr.mjs

```
import urlModule from 'url';
const URL = urlModule.URL;

async function ssr(url) {
  ...
  const stylesheetContents = {};

  // 1. Stash the responses of local stylesheets.
  page.on('response', async resp => {
    const responseUrl = resp.url();
    const sameOrigin = new URL(responseUrl).origin === new URL(url).origin;
    const isStylesheet = resp.request().resourceType() === 'stylesheet';
    if (sameOrigin && isStylesheet) {
      stylesheetContents[responseUrl] = await resp.text();
    }
  });

  // 2. Load page as normal, waiting for network requests to be idle.
  await page.goto(url, {waitUntil: 'networkidle0'});

  // 3. Inline the CSS.
  // Replace stylesheets in the page with their equivalent <style>.
  await page.$$eval('link[rel="stylesheet"]', (links, content) => {
    links.forEach(link => {
      const cssText = content[link.href];
      if (cssText) {
        const style = document.createElement('style');
        style.textContent = cssText;
        link.replaceWith(style);
      }
    });
  }, stylesheetContents);

  // 4. Get updated serialized HTML of page.
  const html = await page.content();
  await browser.close();

  return {html};
}
```

This code:

1. Use a `page.on('response')` handler to listen for network responses.
2. Stashes the responses of local stylesheets.
3. Finds all `<link rel="stylesheet">` in the DOM and replaces them with an equivalent `<style>`. See `page.$$eval` API docs. The `style.textContent` is set to the stylesheet response.

Auto-minify resources

Another trick you can do with network interception is to **modify the responses returned by a request**.

As an example, say you want to minify the CSS in your app but also want to keep the convenience having it unminified when developing. Assuming you've setup another tool to pre-minify `styles.css`, one can use `Request.respond()` to rewrite the response of `styles.css` to be the content of `styles.min.css`.

ssr.mjs

```
import fs from 'fs';

async function ssr(url) {
  ...

  // 1. Intercept network requests.
  await page.setRequestInterception(true);

  page.on('request', req => {
    // 2. If request is for styles.css, respond with the minified version.
    if (req.url().endsWith('styles.css')) {
      return req.respond({
        status: 200,
        contentType: 'text/css',
        body: fs.readFileSync('./public/styles.min.css', 'utf-8')
      });
    }
    ...

    req.continue();
  });
  ...

  const html = await page.content();
  await browser.close();
```

```
  return {html};
}
```

Reusing a single Chrome instance across renders

Launching a new browser for every prerender creates a lot of overhead. Instead, you may want to launch a single instance and reuse it for rendering multiple pages.

Puppeteer can reconnect to an existing instance of Chrome by calling `puppeteer.connect()` and passing it the instance's remote debugging URL. To keep a long-running browser instance, we can move the code that launches Chrome from the `ssr()` function and into the Express server:

server.mjs

```
import express from 'express';
import puppeteer from 'puppeteer';
import ssr from './ssr.mjs';

let browserWSEndpoint = null;
const app = express();

app.get('/', async (req, res, next) => {
  if (!browserWSEndpoint) {
    const browser = await puppeteer.launch();
    browserWSEndpoint = await browser.wsEndpoint();
  }

  const url = `${req.protocol}://${req.get('host')}/index.html`;
  const {html} = await ssr(url, browserWSEndpoint);

  return res.status(200).send(html);
});
```

ssr.mjs

```
import puppeteer from 'puppeteer';

/**
 * @param {string} url URL to prerender.
 * @param {string} browserWSEndpoint Optional remote debugging URL. If
 *     provided, Puppeteer's reconnects to the browser instance. Otherwise,
 *     a new browser instance is launched.
 */
async function ssr(url, browserWSEndpoint) {
```

```
  ...
  console.info('Connecting to existing Chrome instance.');
  const browser = await puppeteer.connect({browserWSEndpoint});

  const page = await browser.newPage();
  ...
  await page.close(); // Close the page we opened here (not the browser).

  return {html};
}
```

Example: cron job to periodically prerender

In my App Engine dashboard app, I setup a cron handler to periodically re-render the top pages on the site. This helps visitors always see fast, fresh content and avoid and helps them from seeing the "startup cost" of a new prerender. Spawning several instances of Chrome would be wasteful for this case. Instead, I'm using a shared browser instance to render several pages all at once:

```
import puppeteer from 'puppeteer';
import * as prerender from './ssr.mjs';
import urlModule from 'url';
const URL = urlModule.URL;

app.get('/cron/update_cache', async (req, res) => {
  if (!req.get('X-Appengine-Cron')) {
    return res.status(403).send('Sorry, cron handler can only be run as admin.');
  }

  const browser = await puppeteer.launch();
  const homepage = new URL(`${req.protocol}://${req.get('host')}`);

  // Re-render main page and a few pages back.
  prerender.clearCache();
  await prerender.ssr(homepage.href, await browser.wsEndpoint());
  await prerender.ssr(`${homepage}?year=2018`);
  await prerender.ssr(`${homepage}?year=2017`);
  await prerender.ssr(`${homepage}?year=2016`);
  await browser.close();

  res.status(200).send('Render cache updated!');
});
```

I also added a `clearCache()` export to **ssr.js**:

```
...
function clearCache() {
  RENDER_CACHE.clear();
}

export {ssr, clearCache};
```

Other considerations

Create a signal for the page: "You're being rendered in headless"

When your page is being rendered by headless Chrome on the server, it may be helpful for the page's client-side logic to know that. In my app, I used this hook to "turn off" portions of my page that don't play a part in rendering the posts markup. For example, I disabled code that lazy-loads firebase-auth.js. There's no user to sign in!

Adding a **`?headless`** parameter to the render URL is a simple way to give the page a hook:

ssr.mjs

```
import urlModule from 'url';
const URL = urlModule.URL;

async function ssr(url) {
  ...
  // Add ?headless to the URL so the page has a signal
  // it's being loaded by headless Chrome.
  const renderUrl = new URL(url);
  renderUrl.searchParams.set('headless', '');
  await page.goto(renderUrl, {waitUntil: 'networkidle0'});
  ...

  return {html};
}
```

And in the page, we can look for that parameter:

public/index.html

```
<html>
<body>
  <div id="container">
    <!-- Populated by the JS below. -->
  </div>
```

```
</body>
<script>
...

(async() => {
  const params = new URL(location.href).searchParams;

  const RENDERING_IN_HEADLESS = params.has('headless');
  if (RENDERING_IN_HEADLESS) {
    // Being rendered by headless Chrome on the server.
    // e.g. shut off features, don't lazy load non-essential resources, etc.
  }

  const container = document.querySelector('#container');
  const posts = await fetch('/posts').then(resp => resp.json());
  renderPosts(posts, container);
})();
</script>
</html>
```

Tip: Another handy method to look at is `Page.evaluateOnNewDocument()`. It allows you to inject code into the page and have Puppeteer run that code before the rest of the page's JavaScript executes.

Avoid inflating Analytics pageviews

Be careful if you're using Analytics on your site. Prerendering pages will may result in inflated pageviews. Specifically, **you'll see 2x the number of hits**, one hit when headless Chrome renders the page and another when the user's browser renders it.

So what's the fix? Use network interception to abort any request(s) that tries to load the Analytics library.

```
page.on('request', req => {
  // Don't load Google Analytics lib requests so pageviews aren't 2x.
  const blacklist = ['www.google-analytics.com', '/gtag/js', 'ga.js', 'analytics.
  if (blacklist.find(regex => req.url().match(regex))) {
    return req.abort();
  }
  ...
  req.continue();
});
```

Page hits never get recorded if the code never loads. Boom 💥.

Alternatively, continue to load your Analytics libraries to gain insight into how many prerenders your server is performing.

Conclusion

Puppeteer makes it easy to server-side render pages by running headless Chrome, as a companion, on your web server. My favorite "feature" of this approach is that you **improve loading performance** and the **indexability** of your app **without significant code** changes!

Note: If you're curious to see a working app that uses the techniques described in this article, check out this app and it's code.

Appendix

Discussion of prior art

Server-side rendering client-side apps is hard. How hard? Just look at how many npm packages people have written which are dedicated to the topic. There are countless patterns, tools, and services available to help with SSRing JS apps.

Isomorphic / Universal JavaScript

The concept of Universal JavaScript is simple: the same code that runs on server also runs on the client (the browser). You share code between server and client and everyone feels a moment of zen.

In practice, I've found universal JS difficult to pull off. A personal story...

> I recently started a a project and wanted to give lit-html a try. Lit is a great little library that lets you write HTML <template>s using JS template literals, then efficiently render those templates to DOM. The problem is that its core feature (using the `<template>` element) doesn't work outside of the browser. That means it won't work in a Node server. My hopes of sharing code to SSR between Node and the frontend were thrown out the window.
>
> I finally came to the realization that I could use headless Chrome to SSR render the app. It didn't matter if Chrome ran by the hands of a user or automated on a server. Chrome happily runs any JS you give it. No questions asked.

Headless Chrome enables "isomorphic JS" between server and client. It's a great option if your library doesn't work on the server (Node).

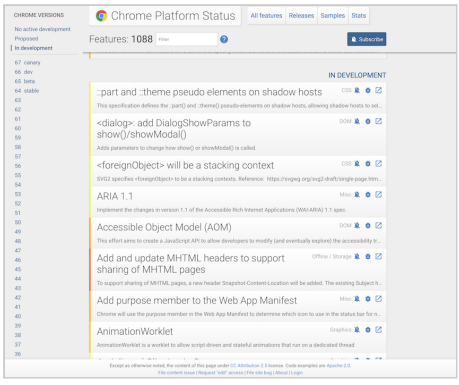
Prerender tools

The Node community has built tons of tools for dealing with SSR JS apps. No surprises there! Personally, I've found that YMMV with some of these tools, so definitely do your homework before committing to one. For example, some SSR tools are older and don't use headless Chrome (or any headless browser for that matter). Instead, they use PhantomJS (aka old Safari), which means your pages aren't going to render properly if they're using newer features.

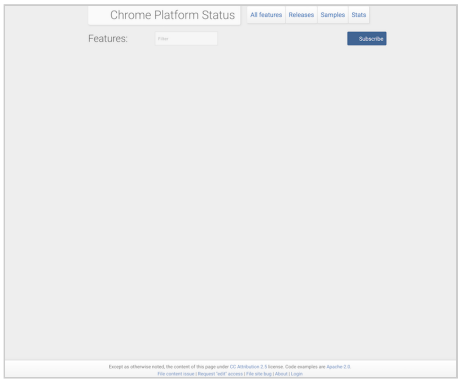
One of the notable exceptions is Prerender. Prerender is interesting in that it uses headless Chrome and comes with drop-in middleware for Express:

```
const prerender = require('prerender');
const server = prerender();
server.use(prerender.removeScriptTags());
server.use(prerender.blockResources());
server.start();
```

It's worth noting that Prerender leaves out the details of downloading and installing Chrome on different platforms. Oftentimes, that's fairly tricky to get right, which is one of the reasons why Puppeteer does for you. I've also had issues with the online service rendering some of my apps:



Site rendered in a browser



Same site rendered by prerender.io

Last updated July 2, 2018.