Getting Started with Headless Chrome

By Eric Bidelman

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

TL;DR

Headless Chrome is shipping in Chrome 59. It's a way to run the Chrome browser in a headless environment. Essentially, running Chrome without chrome! It brings **all modern web platform features** provided by Chromium and the Blink rendering engine to the command line.

Why is that useful?

A headless browser is a great tool for automated testing and server environments where you don't need a visible UI shell. For example, you may want to run some tests against a real web page, create a PDF of it, or just inspect how the browser renders an URL.

Note: Headless mode has been available on Mac and Linux since **Chrome 59**. Windows support came in Chrome 60.

Starting Headless (CLI)

The easiest way to get started with headless mode is to open the Chrome binary from the command line. If you've got Chrome 59+ installed, start Chrome with the `--headless` flag:

```
chrome \
  --headless \                    # Runs Chrome in headless mode.
  --disable-gpu \                 # Temporarily needed if running on Windows.
  --remote-debugging-port=9222 \
  https://www.chromestatus.com    # URL to open. Defaults to about:blank.
```

Note: Right now, you'll also want to include the `--disable-gpu` flag if you're running on Windows. See crbug.com/737678.

`chrome` should point to your installation of Chrome. The exact location will vary from platform to platform. Since I'm on Mac, I created convenient aliases for each version of Chrome that I have installed.

If you're on the stable channel of Chrome and cannot get the Beta, I recommend using `chrome-canary`:

```
alias chrome="/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrom
alias chrome-canary="/Applications/Google\ Chrome\ Canary.app/Contents/MacOS/Goog
alias chromium="/Applications/Chromium.app/Contents/MacOS/Chromium"
```

Download Chrome Canary here.

Command line features

In some cases, you may not need to programmatically script Headless Chrome. There are some useful command line flags to perform common tasks.

Printing the DOM

The `--dump-dom` flag prints `document.body.innerHTML` to stdout:

```
chrome --headless --disable-gpu --dump-dom https://www.chromestatus.com/
```

Create a PDF

The `--print-to-pdf` flag creates a PDF of the page:

```
chrome --headless --disable-gpu --print-to-pdf https://www.chromestatus.com
```

Taking screenshots

To capture a screenshot of a page, use the `--screenshot` flag:

```
chrome --headless --disable-gpu --screenshot https://www.chromestatus.com/

# Size of a standard letterhead.
chrome --headless --disable-gpu --screenshot --window-size=1280,1696 https://www.
```

```
# Nexus 5x
chrome --headless --disable-gpu --screenshot --window-size=412,732 https://www.ch
```

Running with **`--screenshot`** will produce a file named **`screenshot.png`** in the current working directory. If you're looking for full page screenshots, things are a tad more involved. There's a great blog post from David Schnurr that has you covered. Check out Using headless Chrome as an automated screenshot tool .

REPL mode (read-eval-print loop)

The **`--repl`** flag runs Headless in a mode where you can evaluate JS expressions in the browser, right from the command line:

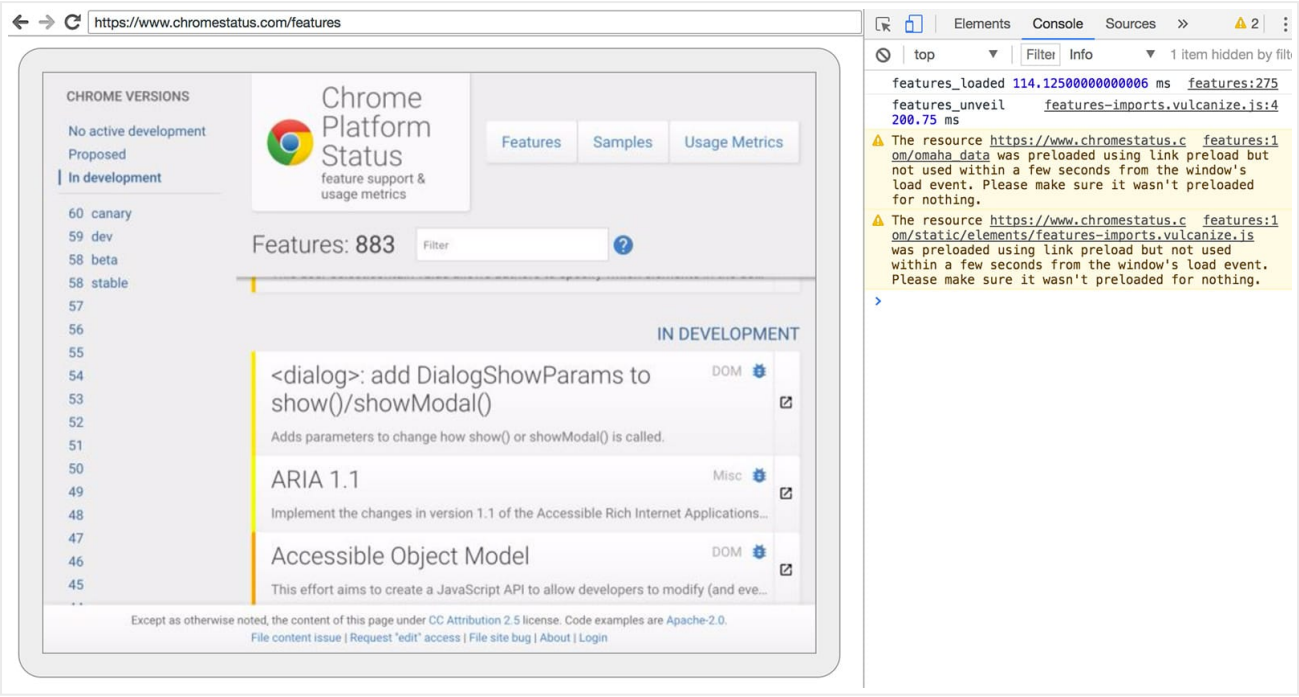
```
$ chrome --headless --disable-gpu --repl --crash-dumps-dir=./tmp https://ww
[0608/112805.245285:INFO:headless_shell.cc(278)] Type a Javascript expression to
>>> location.href
{"result":{"type":"string","value":"https://www.chromestatus.com/features"}}
>>> quit
$
```

Note: the addition of the --crash-dumps-dir flag when using repl mode.

Debugging Chrome without a browser UI?

When you run Chrome with **`--remote-debugging-port=9222`**, it starts an instance with the DevTools protocol enabled. The protocol is used to communicate with Chrome and drive the headless browser instance. It's also what tools like Sublime, VS Code, and Node use for remote debugging an application. #synergy

Since you don't have browser UI to see the page, navigate to **`http://localhost:9222`** in another browser to check that everything is working. You'll see a list of inspectable pages where you can click through and see what Headless is rendering:



DevTools remote debugging UI

From here, you can use the familiar DevTools features to inspect, debug, and tweak the page as you normally would. If you're using Headless programmatically, this page is also a powerful debugging tool for seeing all the raw DevTools protocol commands going across the wire, communicating with the browser.

Using programmatically (Node)

Puppeteer

Puppeteer is a Node library developed by the Chrome team. It provides a high-level API to control headless (or full) Chrome. It's similar to other automated testing libraries like Phantom and NightmareJS, but it only works with the latest versions of Chrome.

Among other things, Puppeteer can be used to easily take screenshots, create PDFs, navigate pages, and fetch information about those pages. I recommend the library if you want to quickly automate browser testing. It hides away the complexities of the DevTools protocol and takes care of redundant tasks like launching a debug instance of Chrome.

Install it:

```
npm i --save puppeteer
```

Example - print the user agent

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

(async() => {
  const browser = await puppeteer.launch();
  console.log(await browser.version());
  await browser.close();
})();
```

Example - taking a screenshot of the page

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

(async() => {
const browser = await puppeteer.launch();
const page = await browser.newPage();
await page.goto('https://www.chromestatus.com', {waitUntil: 'networkidle2'});
await page.pdf({path: 'page.pdf', format: 'A4'});

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

Check out Puppeteer's documentation to learn more about the full API.

The CRI library

chrome-remote-interface is a lower-level library than Puppeteer's API. I recommend it if you want to be close to the metal and use the DevTools protocol directly.

Launching Chrome

chrome-remote-interface doesn't launch Chrome for you, so you'll have to take care of that yourself.

In the CLI section, we started Chrome manually using `--headless --remote-debugging-port=9222`. However, to fully automate tests, you'll probably want to spawn Chrome *from* your application.

One way is to use `child_process`:

```
const execFile = require('child_process').execFile;

function launchHeadlessChrome(url, callback) {
  // Assuming MacOSx.
  const CHROME = '/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome'
```

```
  execFile(CHROME, ['--headless', '--disable-gpu', '--remote-debugging-port=9222'
}

launchHeadlessChrome('https://www.chromestatus.com', (err, stdout, stderr) => {
  ...
});
```

But things get tricky if you want a portable solution that works across multiple platforms. Just look at that hard-coded path to Chrome :(

Using ChromeLauncher

Lighthouse is a marvelous tool for testing the quality of your web apps. A robust module for launching Chrome was developed within Lighthouse and is now extracted for standalone use. The `chrome-launcher` NPM module will find where Chrome is installed, set up a debug instance, launch the browser, and kill it when your program is done. Best part is that it works cross-platform thanks to Node!

By default, **`chrome-launcher` will try to launch Chrome Canary** (if it's installed), but you can change that to manually select which Chrome to use. To use it, first install from npm:

```
npm i --save chrome-launcher
```

Example - using `chrome-launcher` to launch Headless

```
const chromeLauncher = require('chrome-launcher');

// Optional: set logging level of launcher to see its output.
// Install it using: npm i --save lighthouse-logger
// const log = require('lighthouse-logger');
// log.setLevel('info');

/**
 * Launches a debugging instance of Chrome.
 * @param {boolean=} headless True (default) launches Chrome in headless mode.
 *     False launches a full version of Chrome.
 * @return {Promise<ChromeLauncher>}
 */
function launchChrome(headless=true) {
  return chromeLauncher.launch({
    // port: 9222, // Uncomment to force a specific port of your choice.
    chromeFlags: [
      '--window-size=412,732',
      '--disable-gpu',
      headless ? '--headless' : ''
```

```
    ]
  });
}

launchChrome().then(chrome => {
  console.log(`Chrome debuggable on port: ${chrome.port}`);
  ...
  // chrome.kill();
});
```

Running this script doesn't do much, but you should see an instance of Chrome fire up in the task manager that loaded **`about:blank`**. Remember, there won't be any browser UI. We're headless.

To control the browser, we need the DevTools protocol!

Retrieving information about the page

Warning: The DevTools protocol can do a ton of interesting stuff, but it can be a bit daunting at first. I recommend spending a bit of time browsing the DevTools Protocol Viewer, first. Then, move on to the **`chrome-remote-interface`** API docs to see how it wraps the raw protocol.

Let's install the library:

```
npm i --save chrome-remote-interface
```

Examples

Example - print the user agent

```
const CDP = require('chrome-remote-interface');

...

launchChrome().then(async chrome => {
  const version = await CDP.Version({port: chrome.port});
  console.log(version['User-Agent']);
});
```

Results in something like: **`HeadlessChrome/60.0.3082.0`**

Example - check if the site has a web app manifest

```
const CDP = require('chrome-remote-interface');

...

(async function() {

const chrome = await launchChrome();
const protocol = await CDP({port: chrome.port});

// Extract the DevTools protocol domains we need and enable them.
// See API docs: https://chromedevtools.github.io/devtools-protocol/
const {Page} = protocol;
await Page.enable();

Page.navigate({url: 'https://www.chromestatus.com/'});

// Wait for window.onload before doing stuff.
Page.loadEventFired(async () => {
  const manifest = await Page.getAppManifest();

  if (manifest.url) {
    console.log('Manifest: ' + manifest.url);
    console.log(manifest.data);
  } else {
    console.log('Site has no app manifest');
  }

  protocol.close();
  chrome.kill(); // Kill Chrome.
});

})();
```

Example - extract the `<title>` of the page using DOM APIs.

```
const CDP = require('chrome-remote-interface');

...

(async function() {

const chrome = await launchChrome();
const protocol = await CDP({port: chrome.port});

// Extract the DevTools protocol domains we need and enable them.
// See API docs: https://chromedevtools.github.io/devtools-protocol/
const {Page, Runtime} = protocol;
```

```
await Promise.all([Page.enable(), Runtime.enable()]);

Page.navigate({url: 'https://www.chromestatus.com/'});

// Wait for window.onload before doing stuff.
Page.loadEventFired(async () => {
  const js = "document.querySelector('title').textContent";
  // Evaluate the JS expression in the page.
  const result = await Runtime.evaluate({expression: js});

  console.log('Title of page: ' + result.result.value);

  protocol.close();
  chrome.kill(); // Kill Chrome.
});

})();
```

Using Selenium, WebDriver, and ChromeDriver

Right now, Selenium opens a full instance of Chrome. In other words, it's an automated solution but not completely headless. However, Selenium can be configured to run headless Chrome with a little work. I recommend Running Selenium with Headless Chrome if you want the full instructions on how to set things up yourself, but I've dropped in some examples below to get you started.

Using ChromeDriver

ChromeDriver 2.32 uses Chrome 61 and works well with headless Chrome.

Install:

```
npm i --save-dev selenium-webdriver chromedriver
```

Example:

```
const fs = require('fs');
const webdriver = require('selenium-webdriver');
const chromedriver = require('chromedriver');

const chromeCapabilities = webdriver.Capabilities.chrome();
chromeCapabilities.set('chromeOptions', {args: ['--headless']});
```

```
const driver = new webdriver.Builder()
  .forBrowser('chrome')
  .withCapabilities(chromeCapabilities)
  .build();

// Navigate to google.com, enter a search.
driver.get('https://www.google.com/');
driver.findElement({name: 'q'}).sendKeys('webdriver');
driver.findElement({name: 'btnG'}).click();
driver.wait(webdriver.until.titleIs('webdriver - Google Search'), 1000);

// Take screenshot of results page. Save to disk.
driver.takeScreenshot().then(base64png => {
  fs.writeFileSync('screenshot.png', new Buffer(base64png, 'base64'));
});

driver.quit();
```

Using WebDriverIO

WebDriverIO is a higher level API on top of Selenium WebDriver.

Install:

```
npm i --save-dev webdriverio chromedriver
```

Example: filter CSS features on chromestatus.com

```
const webdriverio = require('webdriverio');
const chromedriver = require('chromedriver');

const PORT = 9515;

chromedriver.start([
  '--url-base=wd/hub',
  `--port=${PORT}`,
  '--verbose'
]);

(async () => {

const opts = {
  port: PORT,
  desiredCapabilities: {
    browserName: 'chrome',
    chromeOptions: {args: ['--headless']}
```

```
  }
};

const browser = webdriverio.remote(opts).init();

await browser.url('https://www.chromestatus.com/features');

const title = await browser.getTitle();
console.log(`Title: ${title}`);

await browser.waitForText('.num-features', 3000);
let numFeatures = await browser.getText('.num-features');
console.log(`Chrome has ${numFeatures} total features`);

await browser.setValue('input[type="search"]', 'CSS');
console.log('Filtering features...');
await browser.pause(1000);

numFeatures = await browser.getText('.num-features');
console.log(`Chrome has ${numFeatures} CSS features`);

const buffer = await browser.saveScreenshot('screenshot.png');
console.log('Saved screenshot...');

chromedriver.stop();
browser.end();

})();
```

Further resources

Here are some useful resources to get you started:

Docs

- DevTools Protocol Viewer - API reference docs

Tools

- chrome-remote-interface - node module that wraps the DevTools protocol
- Lighthouse - automated tool for testing web app quality; makes heavy use of the protocol
- chrome-launcher - node module for launching Chrome, ready for automation

Demos

- "The Headless Web" - Paul Kinlan's great blog post on using Headless with api.ai.

FAQ

Do I need the `--disable-gpu` flag?

Only on Windows. Other platforms no longer require it. The `--disable-gpu` flag is a temporary work around for a few bugs. You won't need this flag in future versions of Chrome. See crbug.com/737678 for more information.

So I still need Xvfb?

No. Headless Chrome doesn't use a window so a display server like Xvfb is no longer needed. You can happily run your automated tests without it.

What is Xvfb? Xvfb is an in-memory display server for Unix-like systems that enables you to run graphical applications (like Chrome) without an attached physical display. Many people use Xvfb to run earlier versions of Chrome to do "headless" testing.

How do I create a Docker container that runs Headless Chrome?

Check out lighthouse-ci. It has an example Dockerfile that uses `node:8-slim` as a base image, installs + runs Lighthouse on App Engine Flex.

Note: `--no-sandbox` is not needed if you properly setup a user in the container.

Can I use this with Selenium / WebDriver / ChromeDriver?

Yes. See Using Selenium, WebDrive, or ChromeDriver.

How is this related to PhantomJS?

Headless Chrome is similar to tools like PhantomJS. Both can be used for automated testing in a headless environment. The main difference between the two is that Phantom uses an older version of WebKit as its rendering engine while Headless Chrome uses the latest version of Blink.

At the moment, Phantom also provides a higher level API than the DevTools protocol.

Where do I report bugs?

For bugs against Headless Chrome, file them on crbug.com.

For bugs in the DevTools protocol, file them at github.com/ChromeDevTools/devtools-protocol.

Last updated July 2, 2018.