THE BROWSER HACKER’S GUIDE TO
INSTANTLY
LOADING
EVERYTHING

@addyosmani
LOADING IS A USER JOURNEY WITH MANY DISPARATE EXPECTATIONS

YOU'VE PROBABLY HEARD TO REDUCE DNS LOOKUPS, REDUCE ROUND-TRIP TIMES, MINIMIZE REDIRECTS, ELIMINATE UNNECESSARY RESOURCES.
LOADING IS SLOW BECAUSE OF...

calling the network, idling, JavaScript, css, parsing, compiling, third parties, parser blocking patterns, disk i/o, eviction, IPC jank, thermal throttling, RTTs, images, fonts, kitten GIFs...
Users look for **visual feedback** to reassure them everything is working as expected.
is it happening?
- Navigation begins
- Time to first byte
- First Paint
  - The first non-blank paint on screen

is it useful?
- First Contentful Paint
  - Navigation has successfully started
- First Meaningful Paint
  - Page's primary content is visible
- Visually ready
  - Page looks nearly done
- Fully Loaded
  - End of load lifecycle

is it usable?
- First Interactive
  - consistently Interactive
- Time to Interactive
  - Visually usable and engagable
Time to Interactive

<5s

on an average mobile device over 3G

*2s on repeat-load after Service Worker registered
The average web page on mobile in 2017:

- **16s** until interactive
- **19s** fully loaded
- **420KB** JavaScript

JavaScript Startup Performance, Double-Click Mobile Speed Matters report & the HTTP Archive
First request

INITIAL REQUEST

HTML
# JavaScript Start-up Performance

## V8 Runtime Call Stats

<table>
<thead>
<tr>
<th>Activity</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[V8 Runtime]</td>
<td>499.3ms</td>
<td>499.3ms</td>
<td>26.8%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>228.1ms</td>
<td>231.9ms</td>
<td>12.4%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>154.9ms</td>
<td>155.1ms</td>
<td>8.3%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>19.3ms</td>
<td>19.3ms</td>
<td>1.0%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>14.9ms</td>
<td>18.4ms</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

### Desktop

<table>
<thead>
<tr>
<th>Activity</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>228.1ms</td>
<td>231.9ms</td>
<td>12.4%</td>
</tr>
<tr>
<td>Compile</td>
<td>154.9ms</td>
<td>155.1ms</td>
<td>8.3%</td>
</tr>
<tr>
<td>setTimeout</td>
<td>19.3ms</td>
<td>19.3ms</td>
<td>1.0%</td>
</tr>
<tr>
<td>split</td>
<td>14.9ms</td>
<td>18.4ms</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

### Mobile (with slower CPU)

<table>
<thead>
<tr>
<th>Activity</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[V8 Runtime]</td>
<td>2483.2ms</td>
<td>2483.2ms</td>
<td>32.2%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>1020.7ms</td>
<td>1020.7ms</td>
<td>13.2%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>789.9ms</td>
<td>790.3ms</td>
<td>10.2%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>136.5ms</td>
<td>152.6ms</td>
<td>2.0%</td>
</tr>
<tr>
<td>[V8 Runtime]</td>
<td>88.9ms</td>
<td>88.9ms</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1020.7ms</td>
<td>1020.7ms</td>
<td>13.2%</td>
</tr>
<tr>
<td>Compile</td>
<td>789.9ms</td>
<td>790.3ms</td>
<td>10.2%</td>
</tr>
<tr>
<td>split</td>
<td>136.5ms</td>
<td>152.6ms</td>
<td>2.0%</td>
</tr>
<tr>
<td>setTimeout</td>
<td>88.9ms</td>
<td>88.9ms</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
Test on real phones & real networks

There's no substitute.
about:inspect in Chrome DevTools

Devices
- Discover USB devices
- Discover network targets

Open dedicated DevTools for Node

Nexus 5X #002E3F8F15989DD9

Chrome (58.0.3029.83)

React Hacker News https://react-hn.kristoferbaxter.com/
inspect focus tab reload close

Performance Console Sources

Network: Slow 3G

CPU: No throttling
Apple iPhone 7 Plus vs Motorola Moto G4 Play

**iPhone 7 Plus**
- **Clock Speed**: 2.34 GHz
- **RAM**: 3 GB
- **Size**: 5.5 inches
- **Resolution**: 400 ppi

**Moto G4 Play**
- **Clock Speed**: 1.2 GHz
- **RAM**: 2 GB
- **Size**: 5 inches
- **Resolution**: 293 ppi
Test a website's performance

https://react-hn.kristoferbaxter.com

Test Configuration: Mobile - Emerging Markets

Chrome Beta on a Motorola G (gen 4) tested from Dulles, Virginia on a 400 Kbps 3G connection with 400ms of latency.

Include Repeat View: ☑ (Loads the page, closes the browser and then loads the page again)

Run Lighthouse Audit: ☑ (Mobile devices only)

Moto G4 + 3G
ONLY LOAD WHAT YOU NEED
Code-splitting

Webpack 2+

```javascript
import('./UserProfile')
  .then(loadRoute(cb))
  .catch(errorLoading)
```

Webpack 1

```javascript
// Defines a “split-point” for a separate bundle
require.ensure([], () => {
  const profile = require('./UserProfile', cb);
});
```

Also see Splittable, Closure Compiler or Browserify
Do I need to split?

Try Code Coverage in Chrome DevTools
Tree-shaking

// app.js
import { a } from './module.js';

// module.js
export function a () {}  

export function b () {}  

❌
Only transpile what you need with **Babel**

Use `babel-preset-env` to only transpile code for browsers that need it

```json
{
    "presets": [
        ["env", {
            "targets": {
                "browsers": ["last 2 versions"]
            }
        }]
    ]
}
```
Minify _everything_
Babelified ES5 w/Uglify
ES2015+ with Babili
css-loader + minimize: true

Code-splitting
Dynamic import()
Route-based chunking

Tree-shaking
Webpack 2+ with Uglify
RollUp
DCE w/ Closure Compiler

Optimize “Vendor” libs
NODE_ENV=production
CommonsChunk + HashedModuleIdsPlugin()

Transpile less code
babel-preset-env + modules:false
Browserlist
useBuiltIns: true

Scope Hoisting:
Webpack 3
RollUp

Strip unused Lodash modules
lodash-webpack-plugin
babel-plugin-lodash

Fewer Moment.js locales
ContextReplacementPlugin()
The bloat of your **baseline** defines how much headroom you have for **app code**. How much is taken by your framework?
Plenty of lightweight options for mobile

Lower total cost on size + parse times from the get-go
ATTACK OF THE THIRD PARTY SCRIPTS
Third-party Badging
Request Blocking

With Whole Foods purchase, Amazon is one step closer to the Everything Store by Jacob Kanevski and Lauren Goodie.

The Last Night was one of E3’s most dazzling games — and also its most frustrating by Ali Robertson.
Byte savings @ Google
Display Ads from Google now served using Brotli compression!

https://developers.googleblog.com/

Data-savings up to 40%

15% in aggregate over gzip
Brotli

Google Play
1.5 petabytes (million gigs) saved a day
bit.ly/playstore-brotli

LinkedIn
Improved load time by 7% in India & 4% U.S
bit.ly/linkedin-brotli

Dropbox
Decreased the size of static assets by 20%
bit.ly/dropbox-brotli

CertSimple
17% improvement for largest JS bundles
bit.ly/certsimple-brotli
WebP image format - UNOFF

Image format that supports lossy and lossless compression, as well as animation and alpha transparency.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Current aligned</th>
<th>Usage relative</th>
<th>Date relative</th>
<th>Show all</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>57</td>
</tr>
<tr>
<td>Edge</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>59</td>
</tr>
<tr>
<td>Firefox</td>
<td>49</td>
<td>56</td>
<td>58</td>
<td>10.1</td>
</tr>
<tr>
<td>Chrome</td>
<td>57</td>
<td>58</td>
<td>60</td>
<td>61</td>
</tr>
<tr>
<td>Safari</td>
<td>49</td>
<td>56</td>
<td>58</td>
<td>61</td>
</tr>
<tr>
<td>Opera</td>
<td>9.2</td>
<td>10.2</td>
<td>10.3</td>
<td>11</td>
</tr>
<tr>
<td>iOS Safari</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Opera Mini</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Android Browser</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
</tr>
<tr>
<td>Chrome for Android</td>
<td>56</td>
<td>59</td>
<td>59</td>
<td>59</td>
</tr>
</tbody>
</table>
25-30% savings for WebP on average (26% lossless)

Serving over 43B image requests a day

bit.ly/webp-format
XNConvert

Windows/Mac/Linux
Can convert in batch
Supports most formats

Alternatively:
imagemin
Pixelmator
ImageMagick
GIMP
Leptonica
<picture>
  <!-- Chrome: WebP -->
  <source srcset="photo.webp" type="image/webp">
  <!-- Edge: JPEG-XR -->
  <source srcset="photo.jxr" type="image/vnd.ms-photo">
  <!-- Safari: JPEG 2000 -->
  <source srcset="photo.jp2" type="image/jp2">
  <!-- Firefox: Fallback -->
  <img srcset="photo.jpg">
</picture>

Or use the Accept header + .htaccess to serve WebP if a browser supports it and it exists on disk.
10% improvement in Time-to-Interactive

Inbox by Gmail
CACHE AGGRESSIVELY
HTTP Caching Checklist

1. Use consistent URLs and minimize resource churn
2. Provide a validation token (ETag) to avoid transferring unchanged bytes
3. Identify resources that can be cached by intermediaries (like CDNs)
4. Determine the optimal cache lifetime of resources (max-age)
5. Consider a Service Worker for more control over your repeat visit caching
ORDER LOADING THOUGHTFULLY
ResourceLoadPriority TypeToPriority(Resource::Type type) {
    switch (type) {
    case Resource::kMainResource:
    case Resource::kStyleSheet:
    case Resource::kFont:
        // Also parser-blocking scripts (set explicitly in loadPriority)
        return kResourceLoadPriorityVeryHigh;
    case Resource::kXSLStyleSheet:
        DCHECK(RuntimeEnabledFeatures::xsltEnabled());
    case Resource::kRaw:
    case Resource::kImportResource:
    case Resource::kScript:
        // Also visible resources/images (set explicitly in loadPriority)
        return kResourceLoadPriorityLow;
    case Resource::kImage:
    case Resource::kTextTrack:
    case Resource::kMedia:
    case Resource::kSVGDocument:
        // Also async scripts (set explicitly in loadPriority)
        return kResourceLoadPriorityVeryHigh;
    }
<table>
<thead>
<tr>
<th></th>
<th>Layout-blocking</th>
<th>Load in layout-blocking phase</th>
<th>Load one-at-a-time in layout-blocking phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Net Priority</strong></td>
<td>Highest</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td><strong>Blink Priority</strong></td>
<td>VeryHigh</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VeryLow</td>
</tr>
<tr>
<td><strong>DevTools Priority</strong></td>
<td>Highest</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Low</td>
</tr>
<tr>
<td><strong>Main Resource</strong></td>
<td></td>
<td></td>
<td>Lowest</td>
</tr>
<tr>
<td><strong>CSS (match)</strong></td>
<td>Script (early**)</td>
<td>Script (late**)</td>
<td>Script (async)</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>Font (preload)</td>
<td>Image</td>
</tr>
<tr>
<td></td>
<td>Image (in viewport)</td>
<td></td>
<td>Media</td>
</tr>
</tbody>
</table>
ResourceLoadPriorityVeryHigh
Everything is high priority
Everything is high priority

JS + CSS is high priority

CSS + fonts are high prior
FIRST DO IT.
THEN DO IT RIGHT.
THEN DO IT BETTER.
<html>
  <head>
    <link href="style.css" rel="stylesheet">
    <script src="a.js"></script>
    <script src="b.js"></script>
  </head>
  <body>...</body>
</html>
<link rel="preload">

<head>
<link rel="preload" as="script" href="1.js">
<link rel="preload" as="script" href="2.js">
<link rel="preload" as="script" href="3.js">
..

Link: 1.js; rel="preload"; as="script"
Before preload, the network request started here

After preload, it has shifted left - right at parse time

```html
<link rel="preload" href="/chunk.acee36bce77619e8458e.js" as="script"/>
<link rel="preload" href="/chunk.dd281f26035de6418fb2.js" as="script"/>
```
PRPL

**Push** the minimal code for initial route

- `index.html`
- `index.html: parsing`

App loading

Push essential code

**Render** the initial route

- `entry chunk`
- `entry chunk: evaluate`
- `vendor chunk`
- `vendor chunk: evaluate`

Navigate other route

**Lazy-load** code split by routes

- `route-1 chunk: evaluate`
- `route-1-1`
- `route-1-1 chunk: evaluate`

Pre-cache by Service Worker

- `chunks`
- `images.png`

Navigate next route

Cache remaining resources

bit.ly/prpl-pattern
HTTP/2 with 3G
HTTP/2 + preload with 3G
const express = require('express'),
let app = express();

app
  .use('/js', express.static('js'))
  .get('/', function (req, res) {
    res.set('Link', `</style.css>; rel=preload; as='style',
                   </js/vendor.bundle.js>; rel=preload; as='script',
                   </js/app.bundle.js>; rel=preload; as='script'`)
When can we run into problems?

- Client may already have the resource
- H/2 Push might delay response from origin
H/2 Server Push + Service Worker
Alternatively: Track cache content using cookies

```php
if (supports_http2() && !http_cached('/app.js')) {
    header('link:/app.js>;  rel=preload;  as=script');
    setcookie('/app.js', 'is-cached', 0, '/');
}
```
function http_cached($filename) {
    if ('is-cached' === $_COOKIE[$filename]) {
        return true;
    } else {
        return false;
    }
}
Push Vs. Preload

Cuts out an RTT
Useful if you have a Service Worker or CacheDigests
Not cache aware
No prioritization

Move resource download time closer to initial request
Cross-origin Cache & cookies
Load/error events
Content negotiation
Repeat visit with Service Worker

Fully loaded from Disk Cache
Next: Differential Serving based on browser compatibility?

HTTP/2 works better when resources are more granular (unbundled)

Serve an unbundled build for server/browser combinations supporting HTTP/2. Trigger delivery with `<link rel="preload">` or HTTP/2 Push

HTTP/1 works better when resources are concatenated (bundled)

Serve a bundled build to minimize round-trips to get the app running on server/browser combinations that don't support HTTP/2 Push
Debugging: HTTP/2 Server Push in DevTools
Debugging: HTTP/2 Server Push in DevTools

```
link: </image.jpg>; rel=preload; as=image
```
HTTP/2 Server Push Rules Of Thumb

bit.ly/h2push

1. Push just enough resources to fill idle network time, and no more.
2. Push resources in evaluation-dependence order.
3. Consider using strategies to track the client-side cache.
4. Use the right cookies when pushing resources.
5. Use server push to fill the initial cwnd. Consider preload links to reveal remaining critical resources.
"HTTP/2 push will solve that" is something I've heard a lot when it comes to page load performance problems, but I didn't know much about it, so I decided to dig in.

HTTP/2 push is more complicated and low-level than I initially thought, but what really caught me off-guard is how inconsistent it is between browsers – I'd assumed it was a done deal & totally ready for production.

This isn't an "HTTP/2 push is a douchebag" hatchet job – I think HTTP/2 push is really powerful and will improve over time, but I no longer think it's a silver bullet from a golden gun.
Hacker News readers as Progressive Web Apps
Ask HN: For those programming 10+ years, what do you wish you knew 4 years in?

223 points by u/hacker

I've been programming for ~3 years now, first just on the side and now full-time. My learning has been in stops and starts. Now I can feel the cumulative learning paying off. I'm also realizing there's SO much more to learn. What advice do you have for programmers just hitting their stride?

377 comments
Preact HN
kristoberbaxter/preact-hn

Lighthouse: 93/100
Interactive (Emerging Markets): 2.3s
Interactive (Faster 3G): 1.7s

Framework/UI libraries: Preact, Preact Router
Module bundling: Webpack
Service Worker: Application Shell with OfflinePlugin
Performance patterns: HTTP/2 with Server Push, Brotli and Zopfli static assets
Server-side rendering: Yes
API: In-memory cached Hacker News Firebase API
Hosting: Webfaction + Cloudflare
Author:

 VIEW APP  SOURCE CODE
With Service Workers
babel-preset-env + per-browser bundles

Request URL: https://hn.kristoferbaxter.com/dist/chrome/bundle.application.e61940299d085eeada9e.js
Request Method: GET
Status Code: 200
Remote Address: [2400:cb00:2048:1::681c:797]:443
Referrer Policy: no-referrer-when-downgrade

Response Headers
access-control-allow-credentials: true
access-control-allow-origin: *
cache-control: public, max-age=31536000
cf-cache-status: HIT
cf-ray: 36e0e1e63ece716c4c-SJC
content-encoding: br
content-type: text/javascript
date: Wed, 14 Jun 2017 00:17:08 GMT
expires: Thu, 14 Jun 2018 00:17:08 GMT
server: cloudflare-nginx
set-cookie: __cfduid=ded1a50372cf3d6c807c164a01f4e08cc1497399428; expires=T
babel-preset-env + per-browser bundles
Interactive in <5s on 3G

Page load performance is fast

Users notice if sites and apps don't perform well. These top-level metrics capture:

- First meaningful paint: **1543.0ms** (target: 1600ms)
- Perceptual Speed Index: **2399** (target: 1250)
- First Visual Change: **233ms**
- Last Visual Change: **3739ms**
- Estimated Input Latency: **16.2ms** (target: 50ms)
- Time To Interactive (alpha): **3647ms** (target: 5000ms)
Can we get fast 3G numbers across the board or regular 3G?
Push / Preload
18% improvement
36% improvement

<link rel=preload>
Render
HTML Streaming reduced TTFB by 30% (200ms), increasing time user’s spent in the app.

Nicolas Gallagher, Technical lead for Twitter Lite
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>C9qFrS8VAAAvQq.Q.jpg</td>
<td>jpeg</td>
<td>147 KB</td>
<td>46 ms</td>
<td>Low</td>
</tr>
<tr>
<td>Cv-mxS0WEA9kAU.jpg</td>
<td>jpeg</td>
<td>124 KB</td>
<td>32 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9qGryV0AEH-Qw.jpg</td>
<td>jpeg</td>
<td>113 KB</td>
<td>34 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9qFfpFAAAa50h.jpg</td>
<td>jpeg</td>
<td>96.2 KB</td>
<td>19 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9nwvAYWsaA0c0.jpg</td>
<td>jpeg</td>
<td>88.1 KB</td>
<td>20 ms</td>
<td>Low</td>
</tr>
<tr>
<td>nyantocat_1__normal.gif</td>
<td>gif</td>
<td>35.2 KB</td>
<td>32 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9pvESTUAEA1Mc0.jpg</td>
<td>jpeg</td>
<td>32.7 KB</td>
<td>28 ms</td>
<td>Low</td>
</tr>
<tr>
<td>600x200</td>
<td></td>
<td>29.6 KB</td>
<td>14 ms</td>
<td>Low</td>
</tr>
<tr>
<td>vauUFZMl?format=jpg&amp;name=386x202.jpg</td>
<td>jpeg</td>
<td>21.4 KB</td>
<td>15 ms</td>
<td>Low</td>
</tr>
<tr>
<td>HkwNSLGk4?format=jpg&amp;name=386x202.jpg</td>
<td>jpeg</td>
<td>12.0 KB</td>
<td>18 ms</td>
<td>Low</td>
</tr>
<tr>
<td>mikeyyyyy_normal.png</td>
<td>png</td>
<td>7.6 KB</td>
<td>19 ms</td>
<td>Low</td>
</tr>
<tr>
<td>Picture_24_normal.png</td>
<td>png</td>
<td>7.6 KB</td>
<td>13 ms</td>
<td>Low</td>
</tr>
<tr>
<td>me05_normal.jpg</td>
<td>jpeg</td>
<td>7.6 KB</td>
<td>389 ms</td>
<td>Low</td>
</tr>
<tr>
<td>horse-js_normal.png</td>
<td>png</td>
<td>7.6 KB</td>
<td>18 ms</td>
<td>Low</td>
</tr>
<tr>
<td>squirrelfish_bigger_normal.png</td>
<td>png</td>
<td>7.6 KB</td>
<td>39 ms</td>
<td>Low</td>
</tr>
<tr>
<td>player-placeholder.png</td>
<td>png</td>
<td>7.2 KB</td>
<td>22 ms</td>
<td>Low</td>
</tr>
<tr>
<td>IT16de6_A_normal.png</td>
<td>png</td>
<td>6.5 KB</td>
<td>17 ms</td>
<td>Low</td>
</tr>
<tr>
<td>HbkkX_H_normal.png</td>
<td>png</td>
<td>6.0 KB</td>
<td>14 ms</td>
<td>Low</td>
</tr>
</tbody>
</table>
4x improvement to render perf by using requestIdleCallback() to defer JS loading of images.

Nicolas Gallagher, Technical lead for Twitter Lite
385.98ms to decode a single image!

Heavy image decode

Largest image 19.09ms to decode!

Lower image decode
High-perf Images

Choose the right format
Size appropriately
Adapt intelligently

Compress carefully
Prioritize critical images
Lazy-load the rest
Take care with tools
Data Saver Mode introduced up to 70% savings

Next up: Save-Data client hint
Precache
Application Shell

A skeleton representing the user interface that can be offline cached & instantly rendered on repeat visits.
Before Service Worker

Ensuring your web app can respond when the network connection is flaky is critical to providing your users a good experience. This is achieved by:

1. Registering a Service Worker
2. Responding with a 200 status code when offline

6.10s Load Time

After Service Worker

Load Time! 1.49s
Lazy-load
Page load performance is fast

Users notice if sites and apps don't perform well. These top-level metrics capture the most important perceived performance concerns.

- First meaningful paint: **15647.7ms** (target: 1,600ms)
- Perceptual Speed Index: **11328** (target: 1,250)
- First Visual Change: **5854ms**
- Last Visual Change: **15864ms**
- Estimated Input Latency: **54.5ms** (target: 50ms)
- Time To Interactive (alpha): **15836.2ms** (target: 5,000ms)
Before code-splitting
const plugins = [
    // extract vendor and webpack's module manifest
    new webpack.optimize.CommonsChunkPlugin({
        names: ['vendor', 'manifest'],
        minChunks: Infinity
    }),
    // extract common modules from all the chunks (requires no 'name' property)
    new webpack.optimize.CommonsChunkPlugin({
        async: true,
        children: true,
        minChunks: 4
    })
];
After code-splitting

bit.ly/twitterlite-perf

bit.ly/twitter-case-study

FAST @ 3.00 seconds
Have a Web Font Loading Strategy
Font style matcher

If you’re using a web font, you’re bound to see a flash of unstyled text (or FOUC), between the initial render of your websafe font and the webfont that you’ve chosen. This usually results in a jarring shift in layout, due to sizing discrepancies between the two fonts. To minimize this discrepancy, you can try to match the fallback font and the intended webfont’s x-heights and widths [1]. This tool helps you do exactly that.

Fallback font
Georgia

Font size: 16px
Line height: 1
Font weight: 300

Web font
Merriweather

✔ Download from Google Fonts

Font size: 16px
Line height: 1
Font weight: 300

https://meowni.ca/font-style-matcher/
“Comprehensive Web Fonts”

- Unceremonious `@font-face`
  - Don't use Web Fonts
  - Data URIs
    - Asynchronous
    - Blocking (Inline or External)
      - FOFT, Two Stage Render
        - Critical FOFT
          - Critical FOFT with Data URI
      - font-display descriptor
        - preload
  - FOUT with a Class

https://www.zachleat.com/web/comprehensive-webfonts/
Without Preload

- HTML
- CSS
- Fonts start loading
- Fonts rendered
- Web fonts rendered
With Preload

```html
<link rel="preload" as="font" href="font.woff" type="font/woff">

Link: <font.woff>; rel=preload; as=font; type='font/woff'
```
Heaviest use of rel=preload is for Web Fonts
Preloading Web Fonts = 50% (1.2s) improvement in time-to-text-paint
Control font performance with font-display

**auto**: uses whatever font display strategy the user-agent uses

**block**: draws "invisible" text at first if the font is not loaded, but swaps the font face in as soon as it loads

**swap**: draws text immediately with a fallback if the font face isn’t loaded, but swaps the font face in as soon as it loads

**fallback**: font face is rendered with a fallback at first if it’s not loaded, but the font is swapped as soon as it loads

**optional**: if the font face can’t be loaded quickly, just use the fallback

Chrome 60
font-display: optional

Fonts not in the cache

font-display: optional

Fonts in the cache
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| À | Á | Â | Ã | Ä | Å | Æ | É | Ê | Ë | Ê | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë | Ë |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |
| Ñ | Ó | Ò | Ô | Õ | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô | Ô |Ô |Ô |Ô |Ô |Ô |Ô |

bit.ly/font-subsetting
Web Font Subsetting
Supported by Google Fonts

https://fonts.googleapis.com/css?family=Inconsolata
~3KB

https://fonts.googleapis.com/css?family=Inconsolata&amp;text=Hello
~880 bytes
The browser can also handle subsetting!

/* Small subset, normal weight */
@font-face {
  font-family: whatever;
  src: url('reg-subset.woff') format('woff');
  font-weight: normal;
}

/* Large subset, normal weight */
@font-face {
  font-family: whatever;
  src: url('reg-extended.woff') format('woff');
  font-weight: normal;
  unicode-range: U+0000-U+FFFF;
}

ABCDEF GHijklm NOPQRSTUVWXYZ
abcdef ghijklm nopqrstuvwxyz
1234567890

https://jakearchibald.com/2014/minimising-font-downloads/
CSS Font Loading API

const font = new FontFace("Awesome Font", "url(/fonts/awesome.woff2)", {
  style: 'normal', unicodeRange: 'U+000-5FF', weight: '400'
});

// don't wait for the render tree, initiate an immediate fetch!
font.load().then(function() {
  // apply the font (which may re-render text and cause a page reflow)
  // after the font has finished downloading
  document.fonts.add(font);
  document.body.style.fontFamily = "Awesome Font, serif";
  // OR... apply your own render strategy here...
});

Web Font Loading Tips

https://meowni.ca/posts/web-fonts/

1. Understand the anatomy of a web font and how browsers load

2. font-display: optional (i.e if you can’t do it fast, load a fallback)

3. Minimize font downloads by limiting range of characters you’re loading

4. Minimize FOIT by using <link rel=“preload”>

5. If you need more control try out the Font Loading API
THE FUTURE? PROGRESSIVE LOADING
Progressive Loading: HTML

Streams API

bit.ly/streams-ftw

Server render: 0.73s / Service worker client render: 0.10s / Service worker client render + hacks: 0.10s / Service worker streamed response: 0.10s / 1.00s
Progressive Loading: CSS

bit.ly/progressive-css

With progressive CSS

Without
<body>
  <!-- HTTP/2 push this resource, or inline it, whichever's faster -->
  <link rel="stylesheet" href="/site-header.css">
  <header>…</header>

  <link rel="stylesheet" href="/article.css">
  <main>…</main>

  <link rel="stylesheet" href="/comment.css">
  <section class="comments">…</section>

  <link rel="stylesheet" href="/about-me.css">
  <section class="about-me">…</section>
</body>
DATA-DRIVEN LOADING

USER RESEARCH

ANALYTICS

MACHINE LEARNING
I got a 200-300ms improvement on render time using rel="preload" for fonts on philna.sh after reading @addyosmani's medium.com/reloading/prel ...
PERF MATTERS
@ADDYOSMANI