what slows loading?

- the network
- parsing
- thermal throttling
- disk I/O
- thermal throttling
- excess round trips
- too many images
- web fonts + foit
- cache eviction

Optimizing >>>>>>
LOAD ONLY WHAT YOU NEED WHEN YOU NEED IT
LOADING BEST PRACTICES

DON'T BE BIG

✂ Code-split your JavaScript
🗜 Compress resources
⚡ Minify & optimize *.*
🌴 Tree-shake modules
📱 Respect data plans
📰 Don't over-do Web Fonts

ONLY LOAD WHAT YOU NEED

😴 Lazy-load non-critical resources
🚨 Preconnect to important origins
🔗 Preload critical resources
🚦 Minimize redirects & round-trips

ONLY LOAD WHAT CHANGED

📦 Cache resources effectively
.readdir Network resilient with Service Workers
User Expectations
Evolving RESPONSE ANIMATION IDLE LOAD RAIL
User happiness metrics

- First Paint
- First Contentful Paint
- First Meaningful Paint
- Time To Interactive
Time to Interactive

<5s

on an average mobile device over 3G

*2s on repeat-load after Service Worker registered
Latencies are significantly higher than a wired connection.
Rendering a (fast) mobile page on 3G

- **DNS LOOKUP**: 200ms
- **TCP HANDSHAKE**: 200ms
- **HTTP REQUEST & RESPONSE**: 200ms
- **SERVER RESPONSE**: 200ms
- **CLIENT-SIDE RENDERING**: 200ms

3G network overhead

Server response <200ms
- Reduce first-render RTTs
- Reduce redirects
- Optimize JS processing
Can we data-science the web?
SELECT origin,
  ROUND(100* SUM(
    SELECT SUM(bin.density)
    FROM UNNEST(first_contentful_paint.histogram.bin) bin WHERE bin.end <= 1000)
  ) AS fast_percent
FROM `chrome-ux-report.chrome_ux_report.201710`
WHERE origin = 'https://news.google.com'
GROUP BY origin;
THIS CAN'T GO WRONG.
STAGE 1: MEASURE
TOOLS TO SCIENCE THE WEB

Synthetic lab conditions

- WebPageTest
- Chrome DevTools

Synthetic health of the web

- Lighthouse
- Puppeteer

Real-world

- httparchive BETA
- RUM
Queryable RUM for the web?
Chrome User Experience Report

- Real world performance as experienced by Chrome users
  - User experience metrics
  - Origin-level resolution

- Initial release is...
  - Focusing on metrics that capture loading experience
  - Provides a sample of 10K origins

Available as a public dataset on Google BigQuery.

bit.ly/introducing-crux
<table>
<thead>
<tr>
<th>Origin</th>
<th>Form Factor</th>
<th>Effective Connection Type (e.g. 3G, 4G)</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.google.com">https://www.google.com</a></td>
<td>phone</td>
<td>4G</td>
</tr>
</tbody>
</table>

### Chrome UX Report

- **Origin**: https://www.google.com
- **Form Factor**: phone
- **Effective Connection Type**: 4G
- **First Paint**
- **First Contentful Paint**
- **DOMContentLoaded**
- **onLoad**
I want to adapt serving based on estimated network quality

// Network type that browser uses
navigator.connection.type
> 'wifi'

// New: Effective connection type
// using rtt and downlink values
navigator.connection.effectiveType
> '2G'

For more on navigator.connection.*
See ‘Building a modern media experience’
Chrome UX Report

SELECT
effective_connection_type.name AS ect,
ROUND(SUM(bin.density) * 100, 2) AS ratio
FROM
(SELECT
  effective_connection_type,
  first_paint.histogram.bin AS bins
FROM
  "chrome-ux-report.chrome_ux_report.2017110"
WHERE
  origin = 'https://www.buzzfeed.com'
) CROSS JOIN
UNNEST(bins) AS bin
GROUP BY
  ect
go ORDER BY ratio DESC

Added instrumentation for EffectiveConnectionType on buzzfeed.com. Closest wins.

<table>
<thead>
<tr>
<th>ect</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4G</td>
<td>3.51K</td>
</tr>
<tr>
<td>3G</td>
<td>16K</td>
</tr>
<tr>
<td>2g</td>
<td>14K</td>
</tr>
<tr>
<td>2g</td>
<td>5K</td>
</tr>
</tbody>
</table>

9:46 AM - 27 Oct 2017

1 Retweet 14 Likes
“Networks, CPUs and disks all hate you. On the client, you pay for what you send in ways you can't easily see”

- Alex Russell, Chrome
STATE OF JAVASCRIPT ON MOBILE

1MB  600KB+  300KB+
10% sites  25% sites  50% sites

Using Dev Tools mobile emulation, Moto G4 calibrated CPU, Cable (5/1mbps, 28ms)
JavaScript has a cost.

Fast = Fast at Download Parse Eval

On mobile devices
2017 JavaScript Parse Costs

~1MB JavaScript (uncompressed)
## JavaScript Parse Cost On Mobile - CNN

<table>
<thead>
<tr>
<th>Device</th>
<th>Script</th>
<th>Layout</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 8 (A11)</td>
<td>3967</td>
<td>2693</td>
<td>1148</td>
</tr>
<tr>
<td>iPhone 7+ (A10)</td>
<td>5669</td>
<td>2002</td>
<td>2582</td>
</tr>
<tr>
<td>iPhone 6s (A9)</td>
<td>6074</td>
<td>4483</td>
<td>1737</td>
</tr>
<tr>
<td>iPhone SE (A9)</td>
<td>5476</td>
<td>4464</td>
<td>1713</td>
</tr>
<tr>
<td>iPhone 6 (A8)</td>
<td>12038</td>
<td>5614</td>
<td>3069</td>
</tr>
<tr>
<td>iPhone 5c (A6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samsung S7 (Exynos 8890)</td>
<td>14354</td>
<td>3250</td>
<td>1048</td>
</tr>
<tr>
<td>Moto G4 (Snapdragon 617)</td>
<td>13355</td>
<td>4107</td>
<td>1002</td>
</tr>
<tr>
<td>Thinkpad T430 (Core i5 3320M)</td>
<td>7179</td>
<td>4955</td>
<td>2851</td>
</tr>
<tr>
<td>Thinkpad Yoga (Core i7 6600U)</td>
<td>2891</td>
<td>1243</td>
<td>422</td>
</tr>
<tr>
<td>Desktop (Core i7-5930K)</td>
<td>2061</td>
<td>8182</td>
<td>265</td>
</tr>
</tbody>
</table>

With thanks to Pat Meenan
**PRPL Pattern**

- **Push the minimal code for the initial route**
  - index.html
  - parsing
  - entry
  - parse/compile
  - vendor
  - parse/compile

- **Render route & get interactive**
  - Navigate next route

- **Lazy-load async (split) routes**
  - route-1
  - route-2
  - parse/compile
  - Navigate to another route

- **Pre-cache using Service Workers**
  - scripts
  - images
  - styles

- **Cache remaining resources**

**Supported by:** CLIs

**Used by sites like:** wego
Where do mobile sites spend their time loading?

With thanks to Camillo and Mathias @ V8
Removing unused code can reduce network transmission times, CPU-intensive code parsing, and memory overhead.
CODE COVERAGE
JS CODE COVERAGE OF TOP 50 SITES

SITES MAY USE ONLY 40% OF THE JAVASCRIPT THEY LOAD UPFRONT.

With thanks to fmeawad@chromium.org
LEARN FROM
GAME DEVELOPERS
BAKE **ONLY** WHAT A SECTION REQUIRES INTO BUNDLES THAT CAN BE LOADED AS NEEDED.
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
- **Minify _everything_*
  - Babelified ES5 w/Uglify
  - ES2015+ with babel-minify
  - 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()
Pinterest’s old Mobile Site - 1st load

First Paint: 4.2s
First Meaningful Paint: 6.2s
Time To Interactive: 23s
Pinterest’s new Mobile Site - 1st load

First Paint: 1.8s
First Meaningful Paint: 5.1s
Time To Interactive: 5.6s

JS Bundles: 620KB ➔ 150KB
CSS Bundles: 150KB ➔ 6KB inline
P90 for Pin pages: 20s ➔ 6.5s
Webpack Bundle Analyzer: Before splitting out common async route code
Webpack Bundle Analyzer: After moving out common code from async chunks into entryChunk

60-90% decrease in size of async route chunks (e.g. 13.9KB ➡ 1KB)

20% increase in size of entry (59KB ➡ 71KB)
INTRODUCE **WORKFLOWS** THAT FORCE EVERYBODY TO THINK ABOUT LOADING TIMES FROM THE BEGINNING.
Performance Budgets for JS

Budgets Tinder tries not to exceed

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>main-3ab35329c6cd0f2ad515.js</td>
<td>137 KB</td>
</tr>
<tr>
<td>vendor-b8328f0f4b9974de408e.js</td>
<td>134 KB</td>
</tr>
<tr>
<td>precache-acync-common-72c1fa6c9b806d75ea.js</td>
<td>46.6 KB</td>
</tr>
<tr>
<td>pc-r-Settings-4ebd6f19cbee51e848aa.js</td>
<td>28.1 KB</td>
</tr>
<tr>
<td>pc-r-DesktopChat-49018926360a06f3f021.js</td>
<td>11.1 KB</td>
</tr>
<tr>
<td>pc-r-Chat-3b8fc719055af6eb432.js</td>
<td>10.0 KB</td>
</tr>
<tr>
<td>ModalManager-2e1092ddb5cf889ee593.js</td>
<td>7.9 KB</td>
</tr>
<tr>
<td>pc-ReqsGamepad-a93849b42ab45408bcdf.js</td>
<td>6.4 KB</td>
</tr>
<tr>
<td>Onboarding-a4da9b76adc7a2534648.js</td>
<td>5.3 KB</td>
</tr>
<tr>
<td>manifest-068ac21131d77a56f41e.js</td>
<td>3.8 KB</td>
</tr>
<tr>
<td>RecsModals-6919b482e9bb59dc1449.js</td>
<td>3.2 KB</td>
</tr>
<tr>
<td>fastclick-ae3a1fe598c27b06f56b.js</td>
<td>3.1 KB</td>
</tr>
<tr>
<td>NotificationManager-d6bf38a62991485e5913.js</td>
<td>2.8 KB</td>
</tr>
<tr>
<td>pc-r-MatchlistPage-9301b6ea8215034d992f.js</td>
<td>2.8 KB</td>
</tr>
<tr>
<td>pc-r-RecProfile-aed14020095393af69f1.js</td>
<td>2.0 KB</td>
</tr>
</tbody>
</table>
import A from '../A';
import B from '../B';

const route = [
    {
        route: '/
        regions: {
            side: A,
            main: B
        }
    }
];
import Loadable from 'react-loadable';

const A = Loadable({
  loader: () => import('../A' /* webpackChunkName: "pc-r-A" */),
  loading: () => null
});

const B = Loadable({
  loader: () => import('../B' /* webpackChunkName: "pc-r-B" */),
  loading: () => null
});

const route = [
  {
    route: '/
    regions: {
      side: A,
      main: B
    },
    preload: [ /* next page chunk to preload */ ]
  }
]
JavaScript Route-based code-splitting + budgets

**Before**
Main bundle size: **166kb**
DOMContentLoaded: **5.46s**
load: **11.91s**

**After**
Main bundle size: **101kb**
DOMContentLoaded: **4.69s**
load: **4.69s**
Cache-Control
max-age=315360000
max-age=28599
max-age=0, no-cache
private, no-transform, max-age=259

Press "Start" to race
# Chrome’s Cache Hit Rates

<table>
<thead>
<tr>
<th>Type</th>
<th>End to end</th>
<th>Memory cache</th>
<th>HTTP cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Android</td>
<td>Windows</td>
<td>Android</td>
</tr>
<tr>
<td>CSS</td>
<td>95%</td>
<td>91.8%</td>
<td>84%</td>
</tr>
<tr>
<td>JS</td>
<td>76.3%</td>
<td>79.4%</td>
<td>50.2%</td>
</tr>
<tr>
<td>Fonts</td>
<td>67.2%</td>
<td>75.2%</td>
<td>24.4%</td>
</tr>
<tr>
<td>Images</td>
<td>80.2%</td>
<td>97.5%</td>
<td>70.2%</td>
</tr>
</tbody>
</table>

* Chrome has 4+ caches. The above reflects the main two - the HTTP and memory caches.
Cache-Control Policies

- **/page**
  - HTML
    - Cache-Control: `no-cache`

- **/style.3da37df.css**
  - CSS
    - Cache-Control: `max-age=31536000`

- **/script.8sd34ff.js**
  - JavaScript
    - Cache-Control: `private, max-age=31536000`

- **/photo.jpg**
  - Image
    - Cache-Control: `max-age=86400`

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
Before Service Worker

- App can load on offline/flaky connections
- Ensuring your web app can respond when the network connection is critical to providing your users a good experience. This is accomplished with a Service Worker.
- Registers a Service Worker
- Responds with a 200 when offline

Load Time: 6.10s

After Service Worker

- Load Time: 1.49s

DOMContentLoaded: 1.31s | Load: 1.49s
Caching best practices & max-age gotchas

Getting caching right yields huge performance benefits, saves bandwidth, and reduces server costs, but many sites half-arse their caching, creating race conditions resulting in interdependent resources getting out of sync.

The vast majority of best-practice caching falls into one of two patterns:

Pattern 1: Immutable content + long max-age

Cache-Control: max-age=31536000

- The content at this URL never changes, therefore...
- The browser/CDN can cache this resource for a year without a problem

https://jakearchibald.com/2016/caching-best-practices/
CIRCUIT

Priority
Low
Low
High
High
Low

Press "Start" to race
TIME TO INTERACTIVE ON MOBILE

35s
10% sites

22s
25% sites

14s
50% sites
<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, Idle</td>
</tr>
<tr>
<td><strong>Blink Priority</strong></td>
<td>VeryHigh</td>
<td>High</td>
<td>Medium, 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, Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lowest</td>
</tr>
<tr>
<td><strong>Main Resource</strong></td>
<td></td>
<td></td>
<td>CSS (match)</td>
</tr>
<tr>
<td><strong>CSS (match)</strong></td>
<td></td>
<td></td>
<td>CSS (mismatch)</td>
</tr>
<tr>
<td><strong>Font</strong></td>
<td>Font (preload)</td>
<td></td>
<td>Image</td>
</tr>
<tr>
<td><strong>Import</strong></td>
<td></td>
<td></td>
<td>Media</td>
</tr>
<tr>
<td><strong>Image (in viewport)</strong></td>
<td></td>
<td></td>
<td>Image</td>
</tr>
</tbody>
</table>
<link rel="preload" as="script" href="bundle.js">
I have critical resources I want to load earlier than discovery.

```html
<link rel="preload" href="movies.json" as="fetch" crossorigin="use-credentials">
<script>
(async () => {
  try {
    const response = await fetch(new Request("movies.json", {credentials: "include"}));
    const data = await response.json();
    console.log(data);
  } catch (exception) {
    console.log("Booo");
  }
})();
</script>
```
How are sites are using link rel=preload?

BBC News - Stylesheets
BBC News stylesheets are loaded by blocking JS for Reasons™ so I cheated a little with `<link rel="preload"> to reduce start render time
36% improvement
Reduce first paint by 500ms, load time by 1 second
RESOURCE HINTS

- `link rel="dns-prefetch"`
- `link rel="prefetch"`
- `link rel="preconnect"`

**DNS Lookup**

**HTTP Request**

- **Socket Connection**
- **Content Download**

Total time: 683 ms
**Link element**

```html
<link rel='dns-prefetch' href='//foo.com'>
<link rel='preconnect' href='//cdn.foo.com'>
<link rel='prefetch' href='//foo.com/next-page.html'>
<link rel='preload' href='//foo.com/main.js' as='script'>
```

**Link header**

```
Link: <https://foo.com>; rel=dns-prefetch
Link: <https://foo.com>; rel=preconnect
Link: <https://foo.com/next-page.html>; rel=prefetch;
Link: <https://foo.com/main.js>; rel=preload; as=script;
```
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')`)
HTTP/2 with 3G ~8s
HTTP/2 SERVER PUSH, BASICALLY
When can we run into problems?

1. Client may already have the resource
2. H/2 Push might delay response from origin
H/2 Server Push + Service Worker
Push Vs. Preload

**Push**
- Cuts out an RTT
- Useful if you have a Service Worker or Cache Digests
- Not cache aware
- No prioritization

**Preload**
- Move resource download time closer to initial request
- Cross-origin Cache & cookies
- Load/error events
- Content negotiation
HTTP/2 push is tougher than I thought

"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.

Map of fetching

Between your page and the destination server there’s a series of caches & things that can

https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/
INLINE CRITICAL CSS

AUTOMATE

Critical - extract & inline critical CSS using Chrome Headless

https://github.com/addyosmani/critical

.inline
<head>
  <style>
    /* critical */
  </style>
  <script>
    /* loadCSS */
  </script>
</head>
WEB PAGE WEIGHT ON MOBILE

- 5.4MB: 10% sites
- 2.9MB: 25% sites
- 1.4MB: 50% sites

Using Dev Tools mobile emulation, Moto G4 calibrated CPU, Cable (5/1mbps, 28ms)
How many sites do not serve content compressed?

~30%
Compress!

GZIP, BROTLI, ZOPFLI
AddOutputFilterByType DEFLATE text/plain
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE application/xml
AddOutputFilterByType DEFLATE application/xhtml+xml
AddOutputFilterByType DEFLATE application/rss+xml
AddOutputFilterByType DEFLATE application/javascript
AddOutputFilterByType DEFLATE application/x-javascript

gzip on;
gzip_vary on;
gzip_comp_level 6;
gzip_http_version 1.1;
gzip_proxied any;
gzip_min_length 256;
gzip_buffers 16 8k;
gzip_types text/plain text/html text/css application/x-javascript text/xml application/xml application/xml+rss text/javascript;
Brotli Accept-Encoding/Content-Encoding - OTHER

More effective lossless compression algorithm than gzip and deflate.

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td></td>
<td>10.1</td>
<td>47</td>
<td>4.4</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>55</td>
<td>60</td>
<td>61</td>
<td>10.1</td>
<td>47</td>
<td>10.2</td>
<td>48</td>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>16</td>
<td>56</td>
<td>62</td>
<td>11</td>
<td>48</td>
<td>all</td>
<td>11</td>
<td>all</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td>63</td>
<td></td>
<td></td>
<td>56</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td>64</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

1. Supported in Chrome and Opera behind the 'Brotli Content-Encoding' flag
2. Enabled since 27 May 2016
3. Support starting with macOS 10.13 High Sierra
Brotli

- Improved load time by 7% in India & 4% U.S

- Decreased the size of static assets by 20%

- 17% improvement for largest JS bundles

- 1.5 petabytes (million gigs) saved a day
Brotli Use: Mostly JavaScript, CSS and HTML

From “Tracking the performance of the web with HTTP Archive”
Remove unnecessary downloads

1. The fastest and best-optimized resource is a resource not sent.
2. Inventory your own assets and third-party assets on your pages.
3. Measure the perf of each asset: its value and its technical performance.
4. Determine if the resources are providing sufficient value.
5. Lazy-load/defer resources that are non-critical as much as possible.
ABCDEF

NOPQRSTUVWXYZ

abcdefghijklm

WEB FONTS

PRESS “START” TO RACE
WEB FONT WEIGHT ON MOBILE

- 200KB: 10% sites
- 100KB: 25% sites
- 80KB: 50% sites

Using Dev Tools mobile emulation, Moto G4 calibrated CPU, Cable (5/1mbps, 28ms)
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.

<table>
<thead>
<tr>
<th>Fallback font</th>
<th>Web font</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Georgia</strong></td>
<td><strong>Merriweather</strong></td>
</tr>
</tbody>
</table>

- Font size: 16px
- Line height: 1
- Font weight: 300
- Download from Google Fonts
- Font size: 16px
- Line height: 1
- Font weight: 300

https://meowni.ca/font-style-matcher/
If my Web Fonts can’t load quickly, don’t load them at all.

```css
@font-face {
  font-family: 'Roboto';
  font-display: optional;
  src: url(Roboto.woff) format('woff'),
       url(Roboto.eot) format('eot');
  font-weight: 400;
  font-style: normal;
}
```
Heaviest use of rel=preload is for Web Fonts
Preloading Web Fonts = 50% (1.2s) improvement in time-to-text-paint
Use System Fonts when you can
The fastest font is one that doesn’t need to load.

Try `font-display: optional;`
If a Web Font can’t load fast, load a fallback instead. If the Web Font is cached, it’ll get used the next time the user loads the page.

Try `<link rel=preload as=font> to request Web Fonts with a higher priority`
If Web Fonts are a critical to your UX, preload them to minimize FOIT.

Try subsetting to limit the range of Web Font characters needed
Subsetting removes characters & Open-Type features from fonts, reducing file size. Google Fonts, TypeKit & Font Squirrel support it. Be careful with use.

Try the CSS Font Loading API if you need more control
Track font download progress & apply once fetched, manipulate font faces and override default lazy load behavior.
IMAGE WEIGHT ON MOBILE

3.9MB - 10% sites
1.9MB - 25% sites
0.8MB - 50% sites

Using Dev Tools mobile emulation, Moto G4 calibrated CPU, Cable (5/1mbps, 28ms)
Image Quality Matters

q=80 is a good baseline for web
JPEG Encoders

**MozJPEG**

- **Source**
  - 982KB

- **q=90**
  - 841KB
  - Butteraugli: 1.576957
  - SSIM: 0.999936

- **q=85**
  - 562KB
  - Butteraugli: 2.483837
  - SSIM: 0.999968

- **q=75**
  - 324KB
  - Butteraugli: 3.661270
  - SSIM: 0.999478

**Guetzli**

- **Source**
  - 982KB

- **q=100**
  - 945KB
  - Butteraugli: 0.408840
  - SSIM: 0.999998

- **q=90**
  - 687KB
  - Butteraugli: 1.580555
  - SSIM: 0.999710

- **q=85**
  - 542KB
  - Butteraugli: 2.099600
  - SSIM: 0.999508

Image is https://unsplash.com/photos/nI4OwYnz0dw by Ray Hennessy.
Image Optimisation

- Choose the right format
- Size appropriately
- Adapt intelligently
- Compress carefully
- Prioritize critical images
- Lazy-load the rest
- Take care with tools
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>C9qFrS8UAAVqQ.jpg</td>
<td>jpeg</td>
<td>147 KB</td>
<td>46 ms</td>
<td>Low</td>
</tr>
<tr>
<td>Cv-mxS0WEAA9kAU.jpg</td>
<td>jpeg</td>
<td>124 KB</td>
<td>32 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9qGr7YwAHEH-qw.jpg</td>
<td>jpeg</td>
<td>113 KB</td>
<td>34 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9qFfpFlA0Aq50h.jpg</td>
<td>jpeg</td>
<td>96.2 KB</td>
<td>19 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9npwAYW0aAocY.jpg</td>
<td>jpeg</td>
<td>88.1 KB</td>
<td>20 ms</td>
<td>Low</td>
</tr>
<tr>
<td>nyanocat_1_normal.gif</td>
<td>gif</td>
<td>35.2 KB</td>
<td>32 ms</td>
<td>Low</td>
</tr>
<tr>
<td>C9pVESTUAEE1Mc0.jpg</td>
<td>jpeg</td>
<td>32.7 KB</td>
<td>28 ms</td>
<td>Low</td>
</tr>
<tr>
<td>600x200</td>
<td>jpeg</td>
<td>29.6 KB</td>
<td>14 ms</td>
<td>Low</td>
</tr>
<tr>
<td>vauUfZMu?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>HkNSLgkJ4?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>IT16ds_A_normal.png</td>
<td>png</td>
<td>6.5 KB</td>
<td>17 ms</td>
<td>Low</td>
</tr>
<tr>
<td>HbdcmXJ1X_normal.png</td>
<td>png</td>
<td>6.0 KB</td>
<td>14 ms</td>
<td>Low</td>
</tr>
</tbody>
</table>
Macbook Pro 2016 Desktop - 138.76ms

Pixel XL - 188.67ms - 1.3x slower than desktop

Moto G - 300.90ms - 2x slower than desktop

Samsung Galaxy S6 - 737.62ms - 5x slower than desktop
385.98ms to decode a single image!

Heavy image decode

Largest image
19.09ms to decode!

Lower image decode
Data Saver Mode introduced up to 70% savings for Twitter Lite

Do this with the browser using the Save-Data client hint
The tl;dr

We should all be automating our image compression.

In 2017, image optimization should be automated. It’s easy to forget, best practices change, and content that doesn’t go through a build pipeline can easily slip. To automate: Use imagemini or libvips for your build process. Many alternatives exist.

https://images.guide
EVERYONE IS RESPONSIBLE FOR performance.
Time to Interactive Budget

5s

DNS LOOKUP
TCP HANDSHAKE
HTTPS HANDSHAKE

At 400Kbps we can send 3.4 x 50KB = 170KB

Baseline is ~$200 Android phone
On a slow 3G network, emulated at:
400ms RTT, 400Kbps transfer
File size Budget

170KB gzipped JS
= ~0.8-1MB decompressed
= ~1s to parse/compile

Critical-path JS/CSS/HTML

Framework
Router,
State management,
Utilities

App

170KB
All bytes are not equal

170KB JS

!=

170KB JPG
Performance Budget Tools

Budgets

Total image transferred must be less than 204.88 KB

Moto G4

JS Size for Polaris has gone under your set budget of 200KB. 1 other test also crossed this budget threshold.

CALIBRE  SPEEDCURVE  BUNDLESIZE
REAL-WORLD WEB PERF BUDGETS

Graph showing time (s) vs data (KB) with key points at 131 KB and 170 KB and a budget of 5 seconds.
PERFORMANCE IS A JOURNEY. LOTS OF SMALL CHANGES CAN LEAD TO BIG GAINS.
KEEP RACING TOWARDS BETTER PERF
Thank you

PERF MATTERS

@ADDYOSMANI