Ideas and Tips for thinking about performance

In the time I’ve been researching performance I’ve become even more convinced that it’s not just an issue with Javascript, CSS, images, HTML or what framework you’re using, although anyone of those items, if not used carefully, can have negative impact on your users’ experience.

We have to think deeper than just the tools and ask ourselves why are we using these technologies and how to leverage them to improve the user experience of our sites and applications.

What technology stack to use

I will not preach one technology over another one, particularly frameworks. I have an opinion and, if you know me, you know what it is. However there are things to consider when starting a new project or when reachitecting for performance.

Don’t pay much attention to the latest and greatest tool. As long as you get the results you want and are happy maintaining the tool, you’re doing just fine. The only exception might be a bundler that provides code splitting, tree shaking and a good plugin ecosystem.

Besides images JavaScript is one of the largest components. Staying within the boundaries of our budget that sill contains the critical-path HTML/CSS/JavaScript, all the resources, and the app logic necessary for the route the bundle will handle we need to be extra careful when creating the bundle and the costs (network, transfer, parse/compile and runtime) of the components that you choose to use

Not every project needs a framework, and not every part of a SPA needs to load the framework.

Be deliberate in your choices. Be thorough in evaluating third-party JS regarding these areas:

- features
- accessibility
- stability

- performance
- package ecosystem
- community
- learning curve
- documentation
- tooling
- track record
- team
- compatibility
- security

Pick your framework battles wisely. Make sure that your chosen framework has all the feature you need for your project and understand the way your framework works.

If you make multiple calls to an API or use multiple APIs in your app, they might become a performance bottleneck. Explore ways to reduce your dependencies on foreign APIs

Depending on how much dynamic data you may be able to move your content to a static site and serve it through a content delivery network. Make sure that your static site generator has plugins to do what you need to increase performance (e.g. image optimization in terms of formats, compression and resizing at the edge), support for servers workers and other tasks you've identified as necessary

Optimize your build

Run an inventory on all of your assets (JavaScript, images, fonts, third-party scripts, "expensive" modules on the page), and break them down in groups. Break the content into groups:

- The basic core experience (fully accessible core content for legacy browsers)
- An enhanced, full experience for capable browsers
- Whatever extras assets that are nice to have and, thus, that can be lazy-loaded)

Define what "cutting-the-mustard" means for your site

Either use feature detection to send features only to those browsers that support them or use ES2015 modules to break the difference between core and enhanced tools.

If you haven't seen them, feature detection queries the Browser's Javascript engine if it supports a given feature.

```
 if ('querySelector' in document) {
  console.log('Yay!')
} else {
  console.log('Boo')
}
```

We can have multiple items in the if statement to give us finer control over how we cut the mustard for the enchanced experience.

This example uses the and logical operator to only return true if both arguments are true, meaning that the browser must support both features to return true.

```
if (('querySelector' in document) && ('serviceWorker' in navigator)) {
  console.log('Yay!')
} else {
  console.log('Boo')
}
```

Modules in browsers

Another, coarser, way to cut the mustard is to use ES2016 ES Modules and the `type="module` attribute in the `script` tag for loading JavaScript. Modern browsers will interpret the script as a JavaScript module and run it as expected, while legacy browsers won't recognize it and hence ignore it

> Be aware that cheap Android phones will cut the mustard

> despite their limited memory and CPU capabilities, so consider feature detect Device Memory API and and then decide on what features you send the user, using feature detection to make sure they are supported).

```
const memory = navigator.deviceMemory
console.log ("This device has at least " + memory + "GiB of RAM.")
```

Remember that parsing JavaScript is expensive, so keep it small. Look for modules and techniques to speed up the initial rendering time (loading, parsing and rendering). Be mindful that this can take significantly longer in low-end mobile devices like those used in emerging markets.

Reducing the size of your payload

Use tree-shaking, scope hoisting and code-splitting to reduce payloads where appropriate. I'm not against tree shaking and minimizing, as long as we can understand the code when we expand the minimized files.

- Tree-shaking is a way to clean up your build process by only including code that is actually used in production
- Code-splitting splits your code base into "chunks" that are loaded on demand
- Scope hoisting detects where import chaining can be flattened and converted into one inlined function without compromising the code.

Make use of these techniques via your bunndler of choice.

Figure out if you can you offload JavaScript

Offloading expensive computations to a worker (running in a separate thread) or to WebAssembly improve your site's performance and free the main thread for UI work and user interaction.

WebAssembly provides several benefits when working with Javascript for performance

- Access to libraries that are not available in Javascript or that provide better performance than built in libraries in current browsers. (See Emscripting a

C library to Wasm)
- A mechanism to improve your application's hot path code, those pieces of script that slow down your application. In Replacing a hot path in your app's JavaScript with WebAssembly we get a second way to work around
- Integrate modules written in other languages into your web projects. Emscripten and NPM give an example of how you can do this

Trim unused CSS/JavaScript: Differential loading and using smaller packages

You can also serve different code to browsers based on the features that they support (called differential loading). This is different from from using `type="module"` on script tags.

Use `babel-preset-env` to only transpile ES2015+ features unsupported by the modern browsers you are targeting. Then set up two builds, one in ES6 and one in ES5.

```
<script type="module" src="path/to/module.mjs">
<script async nomodule src="path/to/es5/script.js">
```

Browsers that support modules natively will load the first script element and ignore the `nomodule` script.

Browsers that don't support modules will ignore the first script and load the second.

Tools like `Autoprefixer` allow to write clean CSS and only use prefixes needed for your target browsers, in a similar way to what `babel-preset-env` does for Javascript.

Chrome Dev Tool's `coverage` menu helps in identifying parts of your CSS and Javascript that are not used and can be moved to separate scripts and lazy loaded.

If you work with libraries like Moment or Lodash you can load only the methods you actually use rather than load the entire library.

Using Lodash as an example, instead of loading the complete library using methods like these

```
// Load the full build.
const _ = require('lodash');
'lodash'// Load the core build.
const _ = require('lodash/core');
```

You can load method categories:

```
const array = require('lodash/array');
const object = require('lodash/fp/object');
```

Or even individual methods:

```
// Cherry-pick methods for smaller bundles.
const at = require('lodash/at');
const curryN = require('lodash/fp/curryN');
```

Loading individual methods guarantees the smallest possible build but depending on what you use, you may still end up loading a lot of code anyways but, likely less than you would loading the full library.

Moment is very heavy and it doesn't seem to have a way to load individual methods or categories. You may want to look at date-fns as an alternative.

```
const formatDistance = require('date-fns/formatDistance')
// Require english and spanish locales
const en = require('date-fns/locale/en-US')
const es = require('date-fns/locale/es')

const resultES = formatDistan'date-fns/locale/en-US' 1),
  new Date(2015, 0, 1),
  {locale: es} // Pass the locale as an option
)

const resultEN = formatDistance(
  new Date(2016, 7, 1),
  new Date(2015, 0, 1),
  {locale: en} // Pass the locale as an option
)

console.log(resultES)
//más de 1 año

console.log(resultEN)
// over 1 year
```

A lot of the functionality of libraries like Moment or date-fns can be done natively using the Intl object and its methods in modern browsers.

```
const date = new Date(Date.UTC(2019, 5, 20, 3, 0, 0));

let dateUS = new Intl.DateTimeFormat('en-US').format(date);
console.log(dateUS)
// -> 6/19/2019

let dateGB = new Intl.DateTimeFormat('en-GB').format(date);
console.log(dateGB)
'en-GB'// -> 19/06/2019
```

See the following articles for more information:

- The Intl.ListFormat API
- The Intl.RelativeTimeFormat API
- ECMA-402/Intl Status update - May 2018

Restrict Third-Party code loading additional assets

Too often one single third-party script ends up calling many additional scripts that have little or no usefulness to your page or its content. Establish a Content Security Policy (CSP) to restrict the impact of third-party scripts, e.g. disallowing the download of audio or video. Embed scripts via iframe and sandbox them, so scripts don’t have access to the DOM and run with only the permissions you assign them.

Make sure cache headers are set correctly

Assuming that you have access to your server's configuration file, double-check that expires, cache-control, max-age and other HTTP cache headers are set properly.

In general, resources should be cacheable either for a very short time (if they are likely to change) or indefinitely (if they are static). Use cache-control: immutable, designed for fingerprinted static resources, to avoid revalidation.

Check that you aren't sending unnecessary headers or headers that may expose your server to potential hacks.

Evaluate if service workers are a good solution

Consider using a Service Worker to optimize future visits. Service workers, inside or outside a PWA give you finer control over the duration of items in the cache without any new server configurations. Libraries like Workbox make the job easier.

Treat service workers as a progressive enhancement. If the browser doesn't support them or has Javascript disables then the browser will not cache the resources and will not work offline so plan accordingly.

Optimize assets

Compressing plain text assets (HTML, CSS, and Javascript) provides good results for little effort. Use Brotli or Zopfli in addition to GZip for compressing text files.

Evaluate what compressioon strategy works best for your content. Usually you can compress static assets with Brotli+Gzip at the highest level, compress HTML on the fly with Brotli lower levels.

Use responsive images with srcset, sizes and the `<picture>` element. Make use of the WebP format, by serving WebP images in `<picture>` and a JPEG fallback.

Note: WebP will make your payload smaller but with JPEG you may get better perceived performance. Users might see an actual image faster with JPEG files although WebP images might travel faster through the network.

Tools like gulp-responsive will generate files for your reponsive images as part of your build process. This assumes that you have large, high density, images to use as your source.

Make sure you optimize images, as much and as often as possible. Even if you don’t use responsive images you owe it to your users to optimize your images.

Tools like Imagemin either standalone, as a Gulp plugin or a plugin for your favorite build system automate the compression process mdssszxcxxc.

Video formats and compression

Likewise, we need to ensure that videos are properly encoded. Use WebM or MP4 with HEVC encoding videos instead of animated GIFs.

Evaluate what codecs will work best for your video. WebM, MP4 and HEVC have wide support but AV1 has finally stabilized and it’s gaining adoption both in browsers (see Caniuse.com AV1 support matrix for more information) and hardware.

Test your video in all the formats your target audience can play and choose wich one is the most efficient for you to encode and for your users to play. Prioritize user experience over encoding speed.

See the following entries in my blog for more information about video encoding, codecs and containers:

- Tools for video on the web
- AV1 video in browsers
- Quick Note: Video Containers
- Revisiting Video Encoding: MP4 and WebM

Optimizing fonts

Be very careful when working with webfonts. They carry a loot of extra baggage that you may not need.

Subsetting fonts will shrink them to only the glyphs you’re actually using. Depending on the function of the font it may reduce the number of glyphs (characters) considerably. A good example is if you’re using a different font for your pages’ headings, that’s a prime candidate for subsetting.

More information on subsetting:

- Subsetting fonts
- More on font subsetting

Where possible choose WOFF2 as your primary font format and fallback to WOFF. These two formats will produce smaller files for the same content and some tools use zoplfi compression, giving you better overall compression.

Variable fonts offer another way to save on font size and download. Consolidating multiple axes of variation (from normall to italic and normal to bold) means only one font at a smaller size that will do double duty for both italics and bold.

We can subset variable fonts but, this is important, we can’t subset the custom axes. This becomes important when we start working with fonts with too many axes.

I love Roboto and I particularly love the variable font implementation

Roboto VF provides twenty named axes, each produces a different visual effect, whether you use them or not. The weight of this version of RobotoVF is 1MB as a WOFF2 file. Not small but not overtyly large either.

In Improving Font Performance: Subset fonts using Glyphhanger I discuss how to use Glyphhanger to create smaller subsets of fonts.

As good as they are I have a problem with variable fonts. You can’t subset the variations to only keep those you need for a specific project.

Jason Pamental reports on a promising development on this front. The W3C’s Web Fonts Working Group charter has been extended with the mandate to explore improvint the performance of web fonts, particularly in light of new Variable Fonts and their potential impact on overall page performance.

According to Jason:

> A simplistic description would be something like ‘font streaming’ but in truth that wouldn’t actually solve the problem: users would still be constantly downloading entire font files even if they only needed a small portion to render the

> one or two pages they might view on a given site. The problem with existing subsetting solutions is that either the subset is thrown away with each page view or the solution requires a proprietary server resource, thereby greatly reducing the usefulness of the subset while increasing the complexity and resource requirements on the server.
>
> The ideal solution would combine the benefits of both of these approaches: subset a font request to what's necessary for a given page, but add to the original font asset on subsequent content requests, thereby enabling the gradual enrichment of the font file. Adobe has been doing something like this for a while with their own custom implementation, which shows it's possible to preserve the enriched font's cacheability and greatly enhances the viability of using web fonts with very large character sets like Arabic and CJK.
>
> Responsive Web Typography — Progressive Font Enrichment: reinventing web font performance

So, hopefully in the not-so-distant future, we should be able to have smaller and faster web fonts where the size would be less of an issue and where variable fonts

Still I think that Variable Fonts are the best solution to handle font size and bloat.

Optimize delivery

Use progressive enhancement as a default.

Run an inventory on all of your assets (JavaScript, images, fonts, third-party scripts, "expensive" modules on the page), and break them down in groups. Break the content into groups:

- The basic core experience (fully accessible core content for legacy browsers)
- An enhanced, full experience for capable browsers
- Whatever extras assets that are nice to have and, thus, that can be lazy-loaded).

Design and build the core experience first, and then enhance the experience with advanced features for capable browsers, creating resilient experiences.

If your website runs fast on a slow machine with a poor screen in a poor browser on a suboptimal network, then it will only run faster on a fast machine with a good browser on a decent network.

As developers, we have to explicitly tell the browser not to wait and to start rendering the page. The `defer` and `async` attributes of the script element handle this.

Which attribute you use will depend on what you need for the specific script in the page, and if you need the scripts to load in a specific order.

Use IntersectionObserver

lazy-load all expensive components, such as heavy JavaScript, videos, iframes, widgets, and potentially images using Intersection Observers, native support in Chromium browsers (behind a flag) and through libraries like yall.js.

Having some browsers support lazy loading natively and some not introduces some interesting conundrums. We need to feature detect native support and use it if available and load a library if it’s not supported natively.

Following Addy Osmani’s example we can do something like this. Then we do two things

- We set up a class for the assets we want to lazy load
- We add the `loading` attribute to flag the lazy loading behavior we want in native support

```
<!-- Let's load this in-viewport image normally -->
<img src="hero.jpg" alt=".."/>

<img src="hero.jpg" alt=".."/><!-- Let's lazy-load the rest of these image
<img data-src="unicorn.jpg"
     loading="lazy"
     alt=".."
     class="lazyload"/>
<img data-src="cats.jpg"
```

```
     loading="lazy"
     alt=".."
     class="lazyload"/>
<img data-src="dogs.jpg"
     loading="lazy"
     alt=".."
     class="lazyload"/>
```

The script does the following

1. Feature detects if the browser supporst lazy loading natively
2. Collect all the images we want to lazy load
3. For all the images we want to lazy load, copy the `data-src` into `src`
4. Load a lazy-load library and initialize it

```
(async () => {
  // 1
  if ('loading' in HTMLImag'loading'prototype) {
    // 2
    const images = document.querySelectorAll("img.lazyload");
    "img.lazyload"// 3
    images.forEach(img => {
        img.src = img.dataset.src;
    });
  } else {
    // 4lLib = await import('/scripts/yall.js');
    // Initiate ya'/scripts/yall.js'// Initiate yall
    document.addEventListener("DOMContentLoaded", yall);
  }
  })();
```

Push critical CSS quickly

Collect all of the above `the fold` CSS required to start rendering the first visible portion of the page and inline it in the <head> of the page.

Experiment with regrouping your CSS rules into purpose specific modules or queries for individual breakpoints and import them as needed.

Make sure you don’t place `<link rel="stylesheet" />` before async scripts. Cache inlined CSS with a service worker and experiment with in-body CSS.

Consider using client hints and Network Information API to customize your users’ experiences

The Save-Data client hint request header allows us to customize the application and the payload to cost- and performance-constrained users.

- Serve low resolution images devices that request it
- Omit non-essential imagery
- Omit non-essential web fonts
- Opting out of server pushes

See Delivering Fast and Light Applications with Save-Data for more specifics. Of course, your projects may need.

Another tool supported in some browsers is the Network Information API. The API enables web applications to access information about the network connection in use by the device; you can then use this information to decide what assets to serve based on the reported networ conditions from the users.

```
navigator.connection.addEventListener('change', logNetworkInfo);

function logNetworkInfo() {
  // Network type that browser uses
  console.log('type: ' + navigator.connection.type);

  'type: '// Effective bandwidth estimatele.log('downlink: ' + navigator.

  // Effec'downlink: '// Effective round-trip time estimate ' + navigator

  // Upper bound on th' + navigator.connection.rtt + '// Upper bound on t

  // Effective connection type determined using a
  //' + navigator.connection.downlinkMax + '// Effective connection type
  // combination of recently observed rtt and
  // downlink valueser has requested a reduced
  // data usage mode from the user agent.
  cons// data usage mode from the user agent.// True if the user has reque
  // data usage mode from the user agent.
  console.log('saveData: ' + navigator.connection.saveData);
}

logNetworkInfo();
```

Be aware that the information the API provides can change drastically and without warning, even on desktop machines

Service Workers

Service Workers provide a programmatic way to cache content on the client, intercept requests and provide fallbacks and custom offline fallbacks and pages. You can select the types of assets you cache and how long do we cache these individual assets for.

Be aware that Service Workers will not help performance on first load. Service Worker precache and cached content doesn’t exists before the page is fully loaded the first time so another system for improving first load performance (preconnect and preload) is necessary.

Libraries like Workbox make working with Service Workers easier.

Service Workers are also the basis for a series of progressive enhancements: Background Sync (both one-off and periodic), ppush nootifications and others.

Stay consistent in the user experiencee

Isolate expensive components with CSS containment so that the rendering engine will not traverse its children when doing layout, paint or style work.

Where possible, make sure that there is no lag when scrolling the page or when an element is animated, and that you’re consistently hitting 60 frames per second. If that’s not possible, then making the frames per second consistent is at least preferable to a mixed range of 60 to 15.

Use CSS will-change to inform the browser ahead of time of what kinds of changes you are likely to make to an element, so that it can set up the appropriate optimizations before they're needed, therefore avoiding a non-trivial start-up cost which can have a negative effect on the responsiveness of a page. Only use `will-change` with those aspects of an element that you know will change, otherwise you will lose the benefits of optimization if you optimize everythibng.

Perceived performance is important

Don't underestimate the role of perceived performance. While loading assets, try to always be one step ahead of the customer, so the experience feels swift while there is quite a lot happening in the background. To keep the customer engaged, use skeleton screens instead of loading indicators and add transitions and animations.

Server configuration

Yes, I know we're talking about front-end development but the server you host your content on is also important and knowing the basics about your server and how it's configured will help in making the sites hosted in it more performant.

Most, if not all, modern browsers support HTTP/2 and take advantage of its feature to boost performance and, in most cases, you're better off using it. Test your site's performance in HTTP/2 with mobile clients. HTTP/2 is often slower on networks which have a noticeable packet loss rate so mobile may be adversely affected.

Some of the differences between HTTP/2 and HTTP/1.x:

- HTTP/2 is binary, instead of textual. This allows for better compression
- HTTP/2 can send multiple requests for data in parallel over a single TCP connection
- It compresses headers for more efficient communicaation
- It allows servers to "push" responses proactively into client caches instead of waiting for a new request for each resource. **Take this with a grain of salt**.
- It reduces additional round trip times (RTT), making your website load faster without any optimization

HTTP/1.1 (the previous version of HTTP) suggested you package all your page/app resources into as few bundles as possible so as to optimize the server performance, you could also shard your content to different domains or subdomains so they'd come from different origins and not be subject to download restrictions for a single origin. With HTTP/2, domain sharding and asset concatenation are no longer needed

You need to find a fine balance between packaging modules and loading many small modules in parallel to take advantage of HTTP/2.

Break down your entire interface into many small modules; then group, compress and bundle them. Test how different combinations of individual and bundled files work for your application. Sending around 6–10 packages seems like a decent compromise (and isn't too bad for legacy browsers).

There is no one-size-fits-all solution.

Keeping ourselves honest

Once we have the budget we need to enforce it. Webpack has a built in tool that will warn (or error out) if you go over a pre-defined bundle size without an additional plugin. The following Webpack configuration snipet shows how to

```
module.exports = {
  //...
  performance: {
    maxAssetSize: 100000,
    maxEntrypointSize: 100000,
    hints: "warning"
  }
};
"warning"
```

See Setting performance budgets with webpack for more information.

While we're in the Webpack area; Contentful published a series of articles on how to put your Webpack Bundle on a diet and make your gziped bundle less than 100KB.

- Part 1

- Part 2
- Part 3

The techniques discussed in the series may or may not be applicable to your individual needs but they point the way to how to improve the performance of your bundled content.

There are ESLint rules that disallows importing from certain packages or modules based on your team’s criteria (you may have standardized on a specific package).

The Lighthousebot tool provides ways to run Lighthoouse for every pull request on your project and you can choose to reject the PR if the Lighthouse run doesn’t match our criteria.

How we address these performance requirements and how seriously we enforce them is up to us. But I see no other way to really to get out of this bloated mess we’ve turned our web content into.

Closing

Because **Performance Matters** we should all work towards improving performance and the overall user experience.

Echoing the words from Why we focus on front-end performance

> Performance is therefore an integral part of the service we provide and every member of a service team should be involved in the optimisation process. Even minor changes can make a huge difference for our users.