Improve your web experience

One thing we as developers tend to forget is that not all web experiences are equal. One of the positive things I've found about AMP is that it reduces the fat of your web content by reducing the number of requests, reducing the ammount of CSS that you can use and, mostly, eliminating Javascript.

That's good but I don't think we need a whole new platform or library to do that. We need to be smarter on how we pack and serve our content. This is not new, some of the tools and tricks we'll discuss in this section are old (in web terms).

Experiences outside the big city

The first thing we need to do is decide who our audience is. I don't mean just figure out where they are coming from (although that's important) but also figure out what their bandwidth and bandwidth cost is as this may have an effect on how they access the web and whether they will keep your application on their devices or not.

I love the two parallel views of the next billion users of the web, where they come from and what they can and cannot do with their data.

Bruce Lawson presents a more user centered view of these next billion users and offers insights of emerging markets and things we can do to make their experiences easier.

Tal Oppenheimer presents a more technical view (and Chrome-centered) view of what we can do to improve the experience of your users who are outside the US and other bandwidth rich markets.

Some of the biggest takeways for me:

- How much does bandwidth cost for your users?
- What devices are they using to access your app?
- What kind of network are your users on?
- How fast does your application load?

Answering these questions will help not only users in emerging markets but all

your users, improve the overall site's performance, and give users the perception that this is a fast page.

(Real) Performance Test

If you've read previous posts in this blog you've probably seen Alex's video on why you should test performance on actual devices and why this is important so I'll spare you the gory details and leave some important points to consider.

Even if you simulate a network connection, adjust network latency and modify other aspects of the browser's network connection or use third party tool like Apple's Network Link Conditioner the connection will not be a real mobile connection.

Alex goes into a lot of details and in much more depth than I could but the main take away is: **Test your site in an actual device plugged into your laptop and using chrome://inspect oor its equivalent for iOS devices.**

Server side tools to improve performance

There are a few ways that we can help the user save data and improve their web experiences from the server without making any changes to the content and letting the server make the changes or fetch resources for you.

Proxy Browsers

Opera Mini is a proxy browser that is very popular in emerging markets where bandwidth is expensive and storage may not be as available as we are used to. It takes your request and forwards it to one of a number of Data Centers where they will make the request, download and process assets before sending the processed request back to your device for rendering.

Chrome in Android and UCBrowser do the same thing using different strategies.

Chrome uses the PageSpeed Module to process the pages it proxies.

UCBrowser uses a similar technology but I'm not fully familiar on the details of how it works.

The fact that these browsers proxy the page you want to access means that highly interactive applications will not work consistently in a proxy browser and some technologies will not work at all.

But if you pay a large percentage of your monthly salary for internet access, you have to swap memory cards to make sure that all your content is available to you or you expect pages to load faster than they do, then it makes more sense.

Configuring the server to preload resources

One of the cool things about resource hints is that we can implement them either on the server or the client. In this section we'll look at implementing them on the server with an example of preloading assets for a single page using Nginx or Apache.

The idea is as follows:

1. When we hit `demo.html` we add link preload headers to load 3 resources: a stylesheet and two images
2. The server will also set cookie
3. In subsequent visits the server will check if the cookie exists and it will only load the link preload headers if it does `not` exist. If it exists it means the

resources were already loaded and don't need to be downloaded again

This is what the configuration looks like on Nginx. Note that this is the full configuration.

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate ssl/certificate.pem;
    ssl_certificate_key ssl/key.pem;

    root /var/www/html;
    http2_push_preload on;

    location = /demo.html {
        add_header Set-Cookie "resloaded=1";
        add_header Link $resources;
    }
}

map $http_cookie $resources {
    "~*resloaded=1" "";
    default "</style.css>; as=style; rel=preload,
    </image1.jpg>; as=image; rel=preload,
    </image2.jpg>; as=style; rel=preload";
}
```

And this is how preloading the resources looks like in an Apache HTTPD configuration. Unlike Nginx this is a partial configuration that does the following:

1. Checks if the HTTP2 module is installed
2. Checks if a `resloaded` cookie already exists
3. Tests if the file requested is demo.html
4. If it does then add the link preload headers
5. Adds the `resloaded` cookie to check for in subsequent visits

```
<IfModule http2_module>
  SetEnvIf Cookie "resloaded=1" resloaded
```

```
  <Files "demo.html">
    Header add Link "</style.css>; as=style; rel=preload,
  </image1.jpg>; as=image; rel=preload,
  </image2.jpg>; as=style; rel=preload" env=!resloaded
    Header add Set-Cookie "resloaded=1; Path=/; Secure; HttpOnly" env=!res
  </Files>
</IfModule>
```

We can load diffeerent resources based on the page we are working with and we can load resources that are needed by all pages and then additional resources that are specific to a page or set of pages.

The one problem when working with preload (and any other resource hint) on the server is that we don't have an easy way to check if the client has already downloadded the file. That's why we use a session cookie to handle the check, if it exists we'll skip preloading the resources.

The Save-Data header

In addition to the tricks we've discussed above, we can take advantage of Chrome's data saver feature and have Chrome send the request to a data compression proxy server that will reduce the size of the requested files by 60% according to Google.

As of Chrome 49 when the user has enabled the Data Saver feature in Chrome, Chrome will add a save-data HTTP Header with the value 'on' to each HTTP Request. The HTTP Header will not be present when the feature is turned off. Use this as a signal of intent from the user that they are conscious of the amount of data that they are using and not that their connection is going through the Data Compression Proxy. We can use this header to write conditional code as we'll see in the following sections.

The first step is to add a header on the server side if the user agent (browser) sent the `Save-Data` header. This is what it looks like for Apache.

We also tell downstream proxies that data may change based on whether the `Save-Data` header is present or not.

```
# If the browser sent the Save-Data header
SetEnvIfNoCase ^Save-Data$ "on" data_saver="on"
# Unset link
Header unset Link env=data_saver
# Tell downstream servers that the response may change
# Based on the Save-Data header
Vary: Save-Data
```

Now that we've done the work on the server side we can look at client side code. These examples use PHP because that's what I'm most familiar with because of my work on Wordpress.

The first code block does the following:

1. Creaate a `saveData` variable and set it to false by default
2. Check the existance of a `save_data` header and that its value is on
3. If both of the conditions in the prior step are true then set the `saveData` variable to true

```
// false by default.
$saveData = false;

// Check if the Save-Data header exists
// Check if Save-Data is set to a value of "on".
if  (isset($_SERVER["HTTP_SAVE_DATA"]) &&
    strtolower($_SERVER["HTTP_SAVE_DATA"]) === "on") {
      "HTTP_SAVE_DATA"// Set saveData to true.
      $saveData = true;
}
```

Now that we know whether the `Save_Data` was enabled in the client we can use it to conditionally do things for people who are in data saver mode. For example, we may want to skip preloading resources for people in data saver mode.

We create a string with the elements we need to preload a resource, in this case a stylesheet.

We check if the `saveData` variable is set to true and, if it is, we append the

nopush option to the preload string. The nopush directive will skip preloading the resource.

We then add a `Link` header with the value of our `preload` variable as configured (with or without the nopush directive)

```
// `preload` like usual...
$preload = "</css/styles.css>; rel=pr"</css/styles.css>; rel=preload; as=s
  // ...but don't push anything if `Save-Data` is detected!
  $preload .= "; nopush";
}

header("Link: " . $preload);
"; nopush"
```

Another thing we can do is conditionally use responsive images in our pages. In the example below we will load a single image if the `saveData` variable is true and load a set of responsive images othwerwise.

```
if ($saveData === true) {
  // Send a low-resolution version of the image for clients specifying `Sa
  ?><img src="butterfly-1x.jpg" alt="A butterfly perched on a flower."><?p
}
else {
  // Send the usual assets for everyone else.
  ?><img src="butterfly-1x.jpg" srcset="butterfly-2x.jpg 2x, butterfly-1x
  alt="A butterfly perched on a flower."><?php
}
```

We can also choose whether to load images at all or not. Just like we did with responsive image sets, we can choose to load images when not saving data (the `saveData` variable is set to false).

```
<p>This paragraph is essential content.</p>
<p>The image below may be humorous, but it
is not critical to the content.</p>
<?php
```

```
if ($saveData === false) {
  // Only send this image if `Save-Data` has NOT been detected.
  ?><img src="meme.jpg" alt="One does not simply consume data."><?php
}?>
```

The last thing I wanted to highlight is a way to conditionally work with fonts and CSS in general.

The first part is to add a class to the HTML element based on whether `saveData` is true or false; this example will only add the class if we are saving data.

```
<html class="<?php if ($saveData === true): ?>save-data<?php endif; ?>">
```

The second part is the reverse of what I do when working with Fontface observer. I use web fonts by default and have a second set of selectors where the user is saving data and we don't want to force them to load web fonts.

```
p,
li {
  font-family: 'Fira'Fira Sans'ial', sans-serif;
}

.save-data p,
.save-data li {
  font-family: 'Arial', sans-serif;
}
'Arial'
```

If we've decided we want to do the work ourselves to make our content more efficient to download we can opt out of the Data compression proxy. The proxy respects the standard `Cache-Control: no-transform` directive and will not process resources that use that header.

Client side tools to improve performance

We've looked at a lot of server side tricks to reduce the page load and improve users' experience when accessing our content. Now we'll look at client-side client hints and tools to optimize our content.

Client side resource hints

The basic way to preload a resource is to use the `<link>` element with three attributes:

- `rel` tells the browser the relationship between the current page and the resourced linked to
- `href` gives the location of the resource to preload
- `as` specifies the type of content being loaded. This is necessary for content prioritization, request matching, application of correct content security policy, and setting of correct Accept request header.

```
<link rel="preload" href="late_discovered_thing.js" as="script">
```

Early loading fonts and the crossorigin attribute

Loading fonts is just the same as preloading other types of resources with some additional constraints

```
<link rel="preload"
      href="font.woff2"
      as="font"
      type="font/woff2"
      crossorigin>
```

You must add a crossorigin attribute when fetching fonts, since they are fetched using anonymous mode CORS. Yes, even if your fonts are on the same origin as the page.

The type attribute is there to make sure that this resource will only get preloaded on browsers that support that file type. Only Chrome supports preload and it also supports WOFF2, but not all browsers that will support preload in the future may support the specific font type. The same is true for any resource type you're preloading and which browser support isn't ubiquitous.

Responsive Loading Links

Preload links have a media attribute that we can use to conditionally load resources based on a media query condition.

What's the use case? Let's say your site's large viewport uses an interactive map, but you only show a static map for the smaller viewports.

You want to load only one of those resources. The only way to do that would be to load them dynamically using Javascript. If you use a script to do this you hide those resources from the preloader, and they may be loaded later than necessary, which can impact your users' visual experience, and negatively impact your SpeedIndex score.

Fortunately you can use preload to load them ahead of time, and use its media attribute so that only the required script will be preloaded:

```
<link rel="preload"
      as="image"
      href="map.png"
      media="(max-width: 600px)">

<link rel="preload"
      as="script"
      href="map.js"
      media="(min-width: 601px)">
```

Resource Hints

In addition to preload and server push we can also ask the browser to help by providing hints and instructions on how to interact with resources.

For this section we'll discuss

- DNS Prefetching
- Preconnect
- Prefetch
- Prerender

DNS prefetch

This hint tells the browser that we'll need assets from a domain so it should resolve the DNS for that domain as quickly as possible. If we know we'll need assets from example.com we can write the following in the head of the document:

```
<link rel="dns-prefetch" href="//example.com">
```

Then, when we request a file from it, we'll no longer have to wait for the DNS lookup. This is particularly useful if we're using code from third parties or resources from social networks where we might be loading a widget from a `</script><script>`.

Preconnect

Preconnect is a more complete version of DNS prefetch. In addition to resolving the DNS it will also do the TCP handshake and, if necessary, the TLS negotiation. It looks like this:

```
<link rel="preconnect" href="//example.net">
```

Prefetching

This is an older version of preload and it works the same way. If you know you'll be using a given resource you can request it ahead of time using the prefetch hint. For example an image or a script, or anything that's cacheable by the browser:

```
<link rel="prefetch" href="image.png">
```

Unlike DNS prefetching, we're actually requesting and downloading that asset and storing it in the cache. However, prefetching can be ignored by the browser. For example, a client might abandon the request of a large font file on a slow network. Firefox will only prefetch resources when "the browser is idle".

Since we know have the preload API I would recommend using that API (discussed earlier) instead.

Prerender

Prerender is the nuclear option, since it will load all of the assets for a given document like so:

```
<link rel="prerender" href="http://css-tricks.com">
```

Steve Souders wrote a great explanation about this technique:

> This is like opening the URL in a hidden tab – all the resources are downloaded, the DOM is created, the page is laid out, the CSS is applied, the JavaScript is executed, etc. If the user navigates to the specified href, then the hidden page is swapped into view making it appear to load instantly. Google Search has had this feature for years under the name Instant Pages. Microsoft recently announced they're going to similarly use prerender in Bing on IE11.

But beware! You should probably be certain that the user will click that link, otherwise the client will download all of the assets necessary to render the page for no reason at all. It is hard to guess what will be loaded but we can make some fairly educated guesses as to what comes next:

- If the user has done a search with an obvious result, that result page is likely to be loaded next.
- If the user navigated to a login page, the logged-in page is probably coming next.
- If the user is reading a multi-page article or paginated set of results, the page after the current page is likely to be next.

Tools to make your content smaller

These are some of the tools that I use to make my content slimmer, send less bytes trough the wire and make the bytes I send thought the wire load faster.

These are not all the tools that you can use to improve your site's

performance. They are the ones I use most frequently. As with many tools, your mileage may vary.

> All the tool examples use Gulp. For any other systems, you're on your own.

UNCSS

UnCSS takes a CSS stylesheet and one or more HTML files and removes all the unused CSS from the stylesheet and writes it back as CSS.

This is specially useful when working with third party stylesheets and CSS frameworks where you, as the author may not have control over or may have downloaded the full farmework for development and now need to slim it down for production

```
gulp.task('uncss', () => {
  return gulp
    .src('src/css/**/*.css'src/css/**/*.css'cat('main.css'))
    .pipe(
      $.uncss({
        html: ['index.html']
      })
    )
    .pipe(gulp.dest('css/main.css'))
    .pipe(
      $.size({
        pretty: true,
        title: 'Uncss'
      })
    );
});
```

Imagemin

Imagemin compresses images in the most popular formats (gif, jpg, png and svg) to produce images better suited for use in the web.

What I like about Imagemin is that you can configure settings for each image type you're working with.

Note that this uses the older implicit syntax for gulp-imagemin. Newer versions require you to be expicit on what format you're configuring. For the newer syntax check the gulp-imagemin README.

```
gulp.task('imagemin', () => {
  return gulp
    .src('src/images/original'src/images/originals/**'magemin({
        progressive: true,
        svgoPlugins: [{ removeViewBox: false }, { cleanupIDs: false }],
        use: [mozjpeg()]
      })
    )
    .pipe(gulp.dest('src/images'))
    .pipe(
      $.size({
        pretty: true,
        title: 'imagemin'
      })
    );
});
```

Critical

Critical above the fold CSS and inline it on your page. This will reduce the load speed of your above the fold content. The example task below will take the above the fold content for all the specified screen sizes and will remove duplicate CSS before inlining it on the head of the page.

```
gulp.task('critical', () => {
  return gulp
    .src('src/*.html')
    .p'src/*.html'ritical({
        base: 'src/',
        inline: true,
```

```
        css: ['src/css/main.css'],
        minify: true,
        extract: false,
        ignore: ['font-face'],
        dimensions: [
          {
            width: 320,
            height: 480
          },
          {
            width: 768,
            height: 1024
          },
          {
            width: 1280,
            height: 960
          }
        ]
      })
    )
    .pipe(
      $.size({
        pretty: true,
        title: 'Critical'
      })
    )
    .pipe(gulp.dest('dist'));
});
```

Generate Responsive Images

One of the biggest pain points of generating responsive images is that we have to create several versions of each image and then add them to the srcset attribute for each image or figure tag.

gulp-responsive will help with generating the images, not with writing the HTML. This plugin will let you create as many versions of an image as you need. You can target specific images or, like what I did with this example, target all

images in a directory.

```
gulp.task('processImages', () => {
  return gulp
    .src(['src/images/**/*.{jpg,png'src/images/**/*.{jpg,png}'/*.png'])
    .pipe(
      $.responsive({
        '*': [
          {
            // image-small.jpg is 200 pixels wide
            width: 200,
            rename: {
              suffix: '-small',
              extname: '.jpg'
            }
          },
          {
            // image-small@2x.jpg is 400 pixels wide
            width: 200 * 2,
            rename: {
              suffix: '-small@2x',
              extname: '.jpg'
            }
          },
          {
            // image-large.jpg is 480 pixels wide
            width: 480,
            rename: {
              suffix: '-large',
              extname: '.jpg'
            }
          },
          {
            // image-large@2x.jpg is 960 pixels wide
            width: 480 * 2,
            rename: {
              suffix: '-large@2x',
              extname: '.jpg'
```

```
    }
  },
  {
    // image-extralarge.jpg is 1280 pixels wide
    width: 1280,
    rename: {
      suffix: '-extralarge',
      extname: '.jpg'
    }
  },
  {
    // image-extralarge@2x.jpg is 2560 pixels wide
    width: 1280 * 2,
    rename: {
      suffix: '-extralarge@2x',
      extname: '.jpg'
    }
  },
  {
    // image-small.webp is 200 pixels wide
    width: 200,
    rename: {
      suffix: '-small',
      extname: '.webp'
    }
  },
  {
    // image-small@2x.webp is 400 pixels wide
    width: 200 * 2,
    rename: {
      suffix: '-small@2x',
      extname: '.webp'
    }
  },
  {
    // image-large.webp is 480 pixels wide
    width: 480,
    rename: {
```

```
      suffix: '-large',
      extname: '.webp'
    }
  },
  {
    // image-large@2x.webp is 960 pixels wide
    width: 480 * 2,
    rename: {
      suffix: '-large@2x',
      extname: '.webp'
    }
  },
  {
    // image-extralarge.webp is 1280 pixels wide
    width: 1280,
    rename: {
      suffix: '-extralarge',
      extname: '.webp'
    }
  },
  {
    // image-extralarge@2x.webp is 2560 pixels wide
    width: 1280 * 2,
    rename: {
      suffix: '-extralarge@2x',
      extname: '.webp'
    }
  },
  {
    // Global configuration for all images
    // The output quality for JPEG, WebP and TIFF output formats
    quality: 80,
    // Use progressive (interlace) scan for JPEG and PNG output
    progressive: true,
    // Skip enalrgement warnings
    skipOnEnlargement: false,
    // Strip all metadata
    withMetadata: true
```

```
          }
        ]
      }).pipe(gulp.dest('dist/images'))
    );
});
```

Webpack

Webpack (and Rollup and Parcel) are resource bundlers. They will pack together resources to make downloads faster and payloads smaller.

As I documented in Revisiting Webpack I don't take advantage of the full Webpack toolkit since most of the processing happens in Gulp before the resources get to Webpack for bundling and I don't feel the need to reinvent the wheel every time I want to work in a project. Before you destroy me on comments or Twitter... this doesn't mean that I won't use Webpack to its fullest if the teaam or the project warrants it, none of my projects have so far.

Instead I use Webpack inside Gulp to bundle the resources for my application. I use the same configuration file that I would use for a standalone Webpack-based application but most of the heavy lifting has already been done by the time we get here so we can slim the configuration file to only do Javascript bundling.

```
gulp.task('webpack-bundle', function() {
  return gulp
    .src('src/entry.js')
    .pipe(webpack(require('./webpack.config.js')))
    .pipe(gulp.dest('dist/'));
});
```

Summary and final considerations

I realize that and the amount of work we need to put in improving performance is big and that we won't always see the results or that the results may be small and people may not notice.

But people do notice and they will leave your site if it doesn't load fast enough.

According to SOASTA's post Google: 53% of mobile users abandon sites that take longer than 3 seconds to load

> 53% of visits to mobile sites are abandoned after 3 seconds. (This corresponds to research we did at SOASTA last year, where we found that the sweet spot for mobile load times was 2 seconds

Whether you want to or not, performance does affect a site's number of visitors and, potentially, the company's bottom line. So please evaluate your site's performance and improve it where you can.

> Your users and your boss will be thankful. :)

Links and Resources

- Response Times: The 3 Important Limits
- Chrome for a Multi-Device World
- Data Saver
- New Save-Data HTTP header tells websites to reduce their data usage
- Save-Data aware HTTP/2 server push
- Speed up with the PageSpeed Modules
- Introducing HTTP/2 Server Push with NGINX 1.13.9
- Help Your Users `Save-Data`
- Opera Mini
- The need for mobile speed: How mobile latency impacts publisher revenue
- The need for mobile speed (PDF)
- Best of 2016: New Google research, measuring SPA and AMP performance, plus 22 mobile stats you should know
- Preload Specification
- HTTP/2 Spec
- Speed Up Sites with htaccess Caching
- Cache-Control Header