HTTP/2 Server Push, Link Preload And Resource Hints

We've become performance obsessed, it's important and the obsession shows. Firtunately we're also given the tools to acommodate the need for speed. We'll look at three ways of helping our servers give us the resources we need before we actually need them.

We're not covering service workers in this post. Even thoough they are very much a performance feature they are client-side and we want to discuss server side performance improvements or improvements t hat are used directly from the HTML code, not Javascript.

What is Server Push

Accessing websites has always followed a request and response pattern. The user sends a request to a remote server, and with some delay, the server responds with the requested content.

The initial request to a web server is usually for an HTML document. The server returns the requested HTML resource. The browser parses the HTML and discoversreferences to other assets (style sheets, scripts, fonts and images). The browser requests these new assets which runs the process again (a stylesheet may have links to fonts or to images being used as background).

The problem with this mechanism is that users must wait for the browser to discover and retrieve critical assets until after an HTML document has been downloaded. This delays rendering and increases load times.

What problem does server push solve?

With server push we now have a way to preemptively "push" assets to the browser before the browser explicitly request them. If we do this carefully then we can

increase perceived performance by sending things we know users are going to need.

Let's say that our site uses the same fonts throughout and it uses one common stylesheet named `main.css`. When the user requests our site's main page, `index.html` we can push these files we know we'll need right after we send the reponse for `index.html`.

This push will increase perceived speed because it enables the browser to rende the page faster than waiting for the server to respond with the HTML file and then parsing it to discover additional resources that it must request and parse.

Server push also acts as a suitable alternative for a number of HTTP/1-specific optimizations, such as inlining CSS and JavaScript directly into HTML, as well as using the data URI scheme to embed binary data into CSS and HTML.

If you inline CSS into an HTML document within `<style>` tags, the browser can begin applying styles immediately to the HTML without waiting to fetch them from an external source. This concept holds true with inlining scripts and inlining binary data with the data URI scheme.

However the big pitfall of these techniques is that they can't be cached outside of the page they are embedded in and, if the same code is used in more than one page, we end up with duplicated code in our pages.

Say, for example, that we want to push a CSS style sheet and a Javascript file for all requests that are HTML pages. In an Apache HTTP Server (version 2.4.17 and later) you can configure the server to push with something like this:

```
<If "%{DOCUMENT_URI} =~ /\.html$/">
  H2PushResource add css/site.css
  H2PushResource add js/site.js
</If>
```

We can then configure push resources that are specific to each section of our site, for example:

```
<If "%{DOCUMENT_URI} == '/portfolio/index.html'">
  H2PushResource add /css/dist/critical-portfolio.css?01042017
```

```
</If>

<If "%{DOCUMENT_URI} == '/code/index.html'">
  H2PushResource add /css/dist/critical-code.css?01042017
</If>
```

Yes, this means we have to play with server configurations, either the default configuration, a virtual host or on a directory basis using `.htaccess`. I still consider this an effort worth making.

It's not all positive. Some things to be aware of:

YOU CAN PUSH TOO MUCH STUFF

Just like when building an application shell with a service worker, we need to be extemely careful about the size of the assets we choose to push. Too many files or files that are too large will defeat the purpose of pushing assets to improve performance as they'll delay rendering.

YOU CAN PUSH SOMETHING THAT'S NOT ON THE PAGE

This is not necessarily a bad thing if you have visitor analytics to back up this strategy or you know that the asset will be used elsewhere on your site. When in doubt don't push. Remember that you may be costing people real money whe pushing unnecessary resources to people who are on restricted mobile data plans.

CONFIGURE YOUR HTTP/2 SERVER PROPERLY

Some servers give you a lot of server push-related configuration options. Apache's mod_http2 has some options for configuring how assets are pushed. Check your server's configuration for details about what options can be configured and how to do it.

PUSHES MAY NOT BE CACHED

There has been some questions on whether server push could cause assets to be unnecessarily pushed to users when they return to our site. One way to control this is to only push assets when a cookie indicating the assets were pushed is not present.

An example of this technique is in Jeremy Wagner's article Creating a Cache-aware HTTP/2 Server Push Mechanism at CSS Tricks. It provides an example of a way to create cookies to check against when pushing files from the server.

Link Preload

The Preload Specification aims at improving performance and providing more granular loading control to web developers. We can customize loading behavior in a way that doesn't incur the penalites of loader scripts.

With preload the browser can do things that are just no possible with H2 Push:

- The browser can set a resource's priority, so that it will not delay more important resources, or lag behind less important resources
- The browser can enforce the right Content-Security-Policy directives, and not go out to the server if it shouldn't
- The browser can send the appropriate Accept headers based on the resource type
- The browser knows the resource type so it can determine if the resource could be reused

Preload has a functional onload event that we can leverage for additional functionality. It will not block the `window.onload` event unless the resource is blocked by a resource that blocks the event elsewhere in your content.

Loading late-loading resources

The basic way you could use preload is to load late-discovered resources early. Not all resources that make a web page are visible in the initial markup. For example, an image or font can be hidden inside a style sheet or a script. The browser can't know about these resources until it parses the containing style sheet or script and that may end up delaying rendering or loading entire sections of your page.

Preload is basically telling the browser "hey, you're going to need this later so please start loading it now".

Preload works as a new `rel` attribute of the `link` element. It has three attributes:

- **rel** indicates the type of link, for preload links we use the `preload` value

- **href** is the relative URL for the asset we're preloading.
- **as** indicates the kind of resource we're preloading. It can be one of the following:
    - "script"
    - "style"
    - "image"
    - "media"
    - "document"

Knowing what the attributes we can look at how to use it responsibly.

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

Early loading fonts and the crossorigin attribute

Loadinng 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.

Markup-based async loader

Another thing you can do is to use the onload handler in order to create some sort

of a markup-based async loader. Scott Jehl was the first to experiment with that, as part of his loadCSS library. In short, you can do something like:

```
<link rel="preload"
      as="style"
      href="async_style.css"
      onload="this.rel='stylesheet'">
```

The same can also work for async scripts.

We already have `<script async>` you say? Well, `<script async>` is great, but it blocks the window's onload event. In some cases, that's exactly what you want it to do, but in other cases it might not be.

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>`.

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">
```

For more information, Ilya Grigorik wrote a great post about this handy resource hint:

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, this is dependent on a number of conditions, as 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.

Combining h2 push and client side technologies

> Please make sure you test the code in the sections below in your own setup. This may improve your site's performance or it may degrade beyond acceptable levels.
>
> You've been warned

We can combine server and client side teechnologies to further increase performance. Some of the things we can do include:

Gzip the content you serve

One way we can further reduce the size of our payloads is compressing them while in transit. This way we make our files smaller in transite and they are expanded by the browser when they receive them.

How we compress data depends on the server we're using. The first example works with Apache mod_deflate and the configuration goes in the global Apache server configuration, inside a virtual host directive or an `.htaccess file`.

We're not compressing images as I'm not 100% certain that they'll survive the trip as other resources will and I already compress them before uploading them to the server.

We also skip files that already have a `Content-Encoding` header. We don't need to compress them if they are already compressed :)

```
<ifModule mod_gzip.c>
  mod_gzip_on Yes
  mod_gzip_dechunk Yes
  mod_gzip_item_include file .(html?|txt|css|js|php|pl)$
  mod_gzip_item_include handler ^cgi-script$
  mod_gzip_item_include mime ^text/.*
  mod_gzip_item_include mime ^application/x-javascript.*
  mod_gzip_item_exclude mime ^image/.*
  mod_gzip_item_exclude rspheader ^Content-Encoding:.*gzip.*
</ifModule>
```

Using Nginx HTTP GZip Module the code looks like this.

Nginx compression will not work with versions of IE before 6. But, honestly, if you're still serving browsers that old you have more serious issues)

We also add a `vary` header to stop proxy servers from sending gzippedd files to IE6 and older.

```
gzip on;
gzip_comp_level 2;
gzip_http_version 1.1;
gzip_proxied any;
gzip_min_length 1100;
gzip_buffers 16 8k;
gzip_types text/plain text/html text/css
  application/x-javascript text/xml application/xml
  application/xml+rss text/javascript;

# Disable for IE < 6 because there are some known problems
gzip_disable "MSIE [1-6].(?!.*SV1)";

"MSIE [1-6].(?!.*SV1)"# Add a vary header for downstream proxies
# to avoid sending cached gzipped files to IE6
gzip_vary on;
```

Preload resources and cache them with a service worker cache

> The code below is written in PHP. I'm working on coverting it to Javascript/Node. If you have such an example, please share it :-)

There has been some questions on whether server push could cause assets to be unnecessarily pushed to users when they return to our site. One way to control this is to only push assets when a cookie indicating the assets were pushed is not present; we then store those assets in the service worker.

An example of this technique is in Jeremy Wagner's article Creating a Cache-aware HTTP/2 Server Push Mechanism at CSS Tricks. It provides an example of a way to create cookies to check against when pushing files from the server.

```
function pushAssets() {
  $pushes = array(
    "/css/styles.css" => substr(md5_file("/var/www/css/styles.css"), 0, 8)
    "/js/scripts.js" => subst"/var/www/css/styles.css"cripts.js"), 0, 8)
  );

  if (!isset($_COOKIE["h2pushes"])) {
    $pushString = buildPushString($pushes);
    header($pushString);
    setcookie("h2pushes", json_encode($pushes), 0,"), 0, 8)
  );

  if (!isset($_COOKIE[");
  } else {
    $serializedPushes = json_encode($pushes);

    if ($serializedPushes !== $_COOKIE["h2pushes"]) {
        $oldPushes = json_decode($_COOKIE["h2pushes"], true);
        $diff = array_diff_assoc($pushes, $oldPushes);
        $pushString = buildPushString($diff);
        header($pushString);
```

```
        setcookie("h2pushes", json_encode($pushes), 0, 2592000, "", ".myr
    }
  }
}

function buildPushString($pushes) {
  $pushString = "Link: ";

  foreach($push"h2pushes"et => $version) {
    $pushString .= "<" . $asset . ">; rel=preload";

    if ($asset !== end($pushes)) {
      $pushString .= ",";
    }
  }
  return $pushString;
}
"<"// Push those assets!
pushAssets();
```

This function (taken from Jeremy's article) checks to see if there is a `h2pushes` cookie stored in the user's browser. If there isn't one then it uses the `buildPushString` helper function to generates preload links for the resources specified in the `$pushes` array and send them over as headers for the page and adds the `h2pushes` cookie to the browser with a representation of the paths that were preloaded.

If the user has been here before we need to decide if there's anything to preload. Because we want to re-push assets if they change, we need to fingerprint them for comparison later on. For example, if you're serving a styles.css file, but you change it, you'll use a cache busting strategy like adding a random string to the file name at build time at bor appending a value to the query string to ensure that the browser won't serve a stale version of the resource.

The function will decode the values stored on the cookie and compare the values with what you want to preload. If they are the same then it does nothing and moves forward, if the values are different then the function will take the new values, create preload links and update the cookie with the new values.

If you preload too many files this function may have detrimental effects on performance. As we discussed earlier you need to be mindful of what you preload and how large are the files you preload. But with this method at least we can be confident we're not pushing duplicate assets to the page

Links and resources

- Server Push
  - https://www.smashingmagazine.com/2017/04/guide-http2-server-push/
  - https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/
  - https://blogs.akamai.com/2017/03/http2-server-push-the-what-how-and-why.html
  - https://webapplog.com/http2-server-push-node-express/
  - https://24ways.org/2016/http2-server-push-and-service-workers/
- Link Preload
  - https://developers.google.com/web/updates/2016/03/link-rel-preload
  - https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for
- Resource Hints
  - https://w3c.github.io/resource-hints/
  - https://www.keycdn.com/blog/resource-hints/
  - https://css-tricks.com/prefetching-preloading-prebrowsing/