

Automating Resource Selection with Client Hints



By Ilya Grigorik

Ilya is a Developer Advocate and Web Perf Guru

Building for the web gives you unparalleled reach. Your web application is a click away and available on most every connected device—smartphone, tablet, laptop and desktop, TV, and more—regardless of the brand or the platform. To deliver the best experience you've built a responsive site that adapts the presentation and functionality for each form-factor, and now you're running down your performance checklist to ensure that the application loads as quickly as possible: you've optimized your critical rendering.path, you've compressed and cached your text resources, and now you're looking at your image resources, which often account for majority of transferred bytes. Problem is, **image optimization is hard**:

- Determine the appropriate format (vector vs. raster)
- Determine the optimal encoding formats (jpeg, webp, etc.)
- Determine the right compression settings (lossy vs. lossless)
- Determine which metadata should be kept or stripped
- Make multiple variants of each for each display + DPR resolution
- ...
- Account for user's network type, speed, and preferences

Individually, these are well-understood problems. Collectively, they create a large optimization space that we (the developers) often overlook or neglect—*"ain't nobody got time fo that"*. Humans do a poor job of exploring the same search space repetitively, especially when many steps are involved. Computers, on the other hand, excel at these types of tasks.

The answer to a good and sustainable optimization strategy for images, and other resources with similar properties is simple: automation. If you're hand-tuning your resources, you're doing it wrong: you'll forget, you'll get lazy, or someone else will make these mistake for you—guaranteed.

The saga of the performance-conscious developer

The search through image optimization space has two distinct phases: build-time and run-time.

- Some optimizations are intrinsic to the resource itself—e.g. selecting the appropriate format and encoding type, tuning compression settings for each encoder, stripping unnecessary metadata and so on. These steps can be performed at "build-time".
- Other optimizations are determined by the type and properties of the client requesting it and must be performed at "run-time": selecting the appropriate resource for client's DPR and intended display width, accounting for client's network speed, user and application preferences, and so on.

The build-time tooling exists but could be made better. For example, there are a lot of savings to be had by dynamically tuning the "quality" setting for each image and each image format, but I'm yet to see anyone actually use it outside of research. This is an area ripe for innovation, but for the purposes of this post I'll leave it at that. Let's focus on the run-time part of the story.

```

```



The application intent is very simple: fetch and display the image at 50% of the user's viewport. This is where most every designer washes their hands and heads for the bar. Meanwhile, the performance-conscious developer on the team is in for a long night:

1. To get the best compression she wants to use the optimal image format for each client: WebP for Chrome, JPEG XR for Edge, and JPEG to the rest.
2. To get the best visual quality she needs to generate multiple variants of each image at different resolutions: 1x, 1.5x, 2x, 2.5x, 3x, and maybe even a few more in between.
3. To avoid delivering unnecessary pixels she needs to understand what "50% of the user's viewport actually means"—there are a lot of different viewport widths out there!
4. Ideally, she also wants to deliver a resilient experience where users on slower networks will automatically fetch a lower resolution. After all, it's all about time to glass.
5. The application also exposes some user controls that affect which image resource ought to be fetched, so there's that to factor in as well.

Oh, and then the designer realizes that she needs to display a different image at 100% width if the viewport size is small to optimize legibility. This means we now have to repeat the same process for one more asset, and then make the fetch conditional on viewport size. Have I mentioned this stuff is hard? Well, ok, let's get to it. The picture element will get us pretty far:

```

<picture>
  <!-- serve WebP to Chrome and Opera -->
  <source
    media="(min-width: 50em)"
    sizes="50vw"
    srcset="/image/thing-200.webp 200w, /image/thing-400.webp 400w,
      /image/thing-800.webp 800w, /image/thing-1200.webp 1200w,
      /image/thing-1600.webp 1600w, /image/thing-2000.webp 2000w"
    type="image/webp">
  <source
    sizes="(min-width: 30em) 100vw"
    srcset="/image/thing-crop-200.webp 200w, /image/thing-crop-400.webp 400w,
      /image/thing-crop-800.webp 800w, /image/thing-crop-1200.webp 1200w,
      /image/thing-crop-1600.webp 1600w, /image/thing-crop-2000.webp 2000w"
    type="image/webp">
  <!-- serve JPEGXR to Edge -->
  <source
    media="(min-width: 50em)"
    sizes="50vw"
    srcset="/image/thing-200.jpgxr 200w, /image/thing-400.jpgxr 400w,
      /image/thing-800.jpgxr 800w, /image/thing-1200.jpgxr 1200w,
      /image/thing-1600.jpgxr 1600w, /image/thing-2000.jpgxr 2000w"
    type="image/vnd.ms-photo">
  <source
    sizes="(min-width: 30em) 100vw"
    srcset="/image/thing-crop-200.jpgxr 200w, /image/thing-crop-400.jpgxr 400w,
      /image/thing-crop-800.jpgxr 800w, /image/thing-crop-1200.jpgxr 1200w,
      /image/thing-crop-1600.jpgxr 1600w, /image/thing-crop-2000.jpgxr 2000w"
    type="image/vnd.ms-photo">
  <!-- serve JPEG to others -->
  <source
    media="(min-width: 50em)"
    sizes="50vw"
    srcset="/image/thing-200.jpg 200w, /image/thing-400.jpg 400w,
      /image/thing-800.jpg 800w, /image/thing-1200.jpg 1200w,
      /image/thing-1600.jpg 1600w, /image/thing-2000.jpg 2000w">
  <source
    sizes="(min-width: 30em) 100vw"
    srcset="/image/thing-crop-200.jpg 200w, /image/thing-crop-400.jpg 400w,
      /image/thing-crop-800.jpg 800w, /image/thing-crop-1200.jpg 1200w,
      /image/thing-crop-1600.jpg 1600w, /image/thing-crop-2000.jpg 2000w">
  <!-- fallback for browsers that don't support picture -->
  
</picture>

```

We've handled the art direction, format selection, and provided six variants of each image to account for variability in DPR and viewport width of the client's device. Impressive!

Note: For the courageous, Jason Grigsby has a great 10-part ['responsive images 101'](#) series that explains in depth all the in's and out's of the `picture` element.

Unfortunately, the `picture` element does not allow us to define any rules for how it should behave based on client's connection type or speed. That said, its processing algorithm does allow the user agent to adjust what resource it fetches in some cases—see step 5. We'll just have to hope that the user agent is smart enough. (Note: none of the current implementations are). Similarly, there are no hooks in the `picture` element to allow for app-specific logic that accounts for app or user preferences. To get these last two bits we'd have to move all of the above logic into JavaScript, but that forfeits the preload scanner optimizations offered by `picture`. Hmm.

Those limitations aside, it works. Well, at least for this particular asset. The real, and the long-term challenge here is that we can't expect the designer or the developer to hand-craft code like this for each and every asset. It's a fun brain puzzle on the first try, but it loses its appeal immediately after that. **We need automation.** Perhaps the IDE's or other content-transform tooling can save us and automatically generate the boilerplate above.

Automating resource selection with client hints

Take a deep breath, suspend your disbelief, and now consider the following example:

```
<meta http-equiv="Accept-CH" content="DPR, Viewport-Width, Width">
...
<picture>
  <source media="(min-width: 50em)" sizes="50vw" srcset="/image/thing">
  
</picture>
```



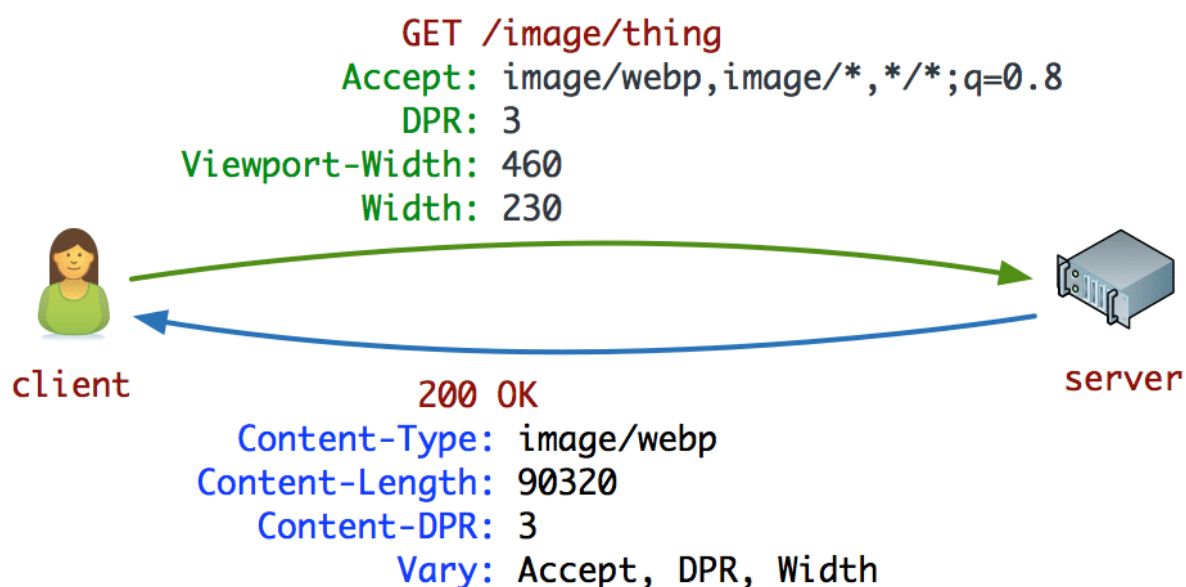
Believe it or not, the above example is sufficient to deliver all the same capabilities as the much longer `picture` markup above plus, as we will see, it enables full developer control over how, which, and when the image resources are fetched. The "magic" is in the first line that enables client hints reporting and tells the browser to advertise the device pixel ratio (DPR), the layout viewport width (`Viewport-Width`), and the intended display width (`Width`) of the resources to the server.

Note: Client hints support is enabled by default in Chrome 46.

With client hints enabled, the resulting client-side markup retains just the presentation requirements. The designer does not have to worry about image types, client resolutions,

optimal breakpoints to reduce delivered bytes, or other resource selection criteria. Let's face it, they never did, and they shouldn't have to. Better, the developer also does not need to rewrite and expand the above markup because the actual resource selection is negotiated by the client and server.

Chrome 46 provides native support for the DPR, Width, and Viewport-Width hints. The hints are disabled by default and the `<meta http-equiv="Accept-CH" content="...">` above serves as an opt-in signal that tells Chrome to append the specified headers to outgoing requests. With that in place, let's examine the request and response headers for a sample image request:



Chrome advertises its support for the WebP format [via the Accept request header](#); the new Edge browser similarly advertises support for JPEG XR via the Accept header.

The next three request headers are the client-hint headers advertising the device pixel ratio of the client's device (3x), the layout viewport width (460px), and the intended display width of the resource (230px). This provides all the necessary information to the server to select the optimal image variant based on its own set of policies: availability of pregenerated resources, cost of re-encoding or resizing a resource, popularity of a resource, current server load, and so on. In this particular case, the server uses the DPR and Width hints and returns a WebP resource, as indicated by the Content-Type, Content-DPR and Vary headers.

Note: The server uses Content-DPR header to [confirm the DPR of the returned asset](#) to allow the user agent to calculate the correct 'intrinsic size' of the image resource.

There is no magic here. We moved the resource selection from HTML markup and into the request-response negotiation between the client and server. As a result, the HTML is only

concerned with presentation requirements and is something we can trust any designer and developer to write, while the search through the image optimization space is deferred to computers and is now easily automated at scale. Remember our performance-conscious developer? Her job now is to write an image service that can leverage the provided hints and returns the appropriate response: she can use any language or server she likes, or, let a third party service or a CDN do this on her behalf.

```

```



Also, remember this guy above? **With client hints the humble image tag is now DPR-, viewport-, and width-aware without any additional markup.** If you need to add art-direction you can use the `picture` tag, as we illustrated above, and otherwise all of your existing image tags just became a lot smarter. **Client hints enhance existing `img` and `picture` elements.**

Taking control over resource selection with service worker

ServiceWorker is, in effect, a client-side proxy running in your browser. It intercepts all outgoing requests and allows you to inspect, rewrite, cache, and even synthesize responses. Images are no different and, with client hints enabled, the active ServiceWorker can identify the image requests, inspect the provided client hints, and define its own processing logic.

```
self.onfetch = function(event) {
  var req = event.request.clone();
  console.log("SW received request for: " + req.url)
  for (var entry of req.headers.entries()) {
    console.log("\t" + entry[0] + ": " + entry[1])
  }
  ...
}
```



2015-08-27 10:16:41.965	SW received request for: https://client-hints.herokuapp.com/image?2	sw.js:19
2015-08-27 10:16:41.965	accept: image/webp,image/*,*/*;q=0.8	sw.js:20
2015-08-27 10:16:41.965	width: 360	sw.js:20
2015-08-27 10:16:41.965	dpr: 2	sw.js:20
2015-08-27 10:16:41.965	viewport-width: 720	sw.js:20
2015-08-27 10:16:41.965	SW received request for: https://sw/style.css	sw.js:19
2015-08-27 10:16:41.965	accept: text/css,*/*;q=0.1	sw.js:20
2015-08-27 10:16:41.966	dpr: 2	sw.js:20
2015-08-27 10:16:41.966	viewport-width: 720	sw.js:20

ServiceWorker gives you full client-side control over the resource selection. This is critical. Let that sink in, because the possibilities are near infinite:

- You can rewrite the client hints header values set by the user agent.
- You can append new client hints headers values to the request.
- You can rewrite the URL and point the image request at an alternative server (e.g. CDN).
 - You can even move the hint values from headers and into the URL itself if that makes things easier to deploy in your infrastructure.
- You can cache responses and define own logic for which resources are served.
- You can adapt your response based on users' connectivity.
 - You can use the NetInfo API to query and advertise your preferences to the server.
 - You can return an alternate response if the fetch is slow.
- You can account for application and user preference overrides.
- You can ... do anything your heart desires, really.

The `picture` element provides the necessary art-direction control in the HTML markup. Client hints provide annotations on resulting image requests that enable resource selection automation. ServiceWorker provides request and response management capabilities on the client. This is the extensible web in action.

The client hints FAQ

1. Where are client hints available?

Shipped in [Chrome 46](#). Under consideration in [Firefox](#) and [Edge](#) [\[?\]](#).

2. Why are the client hints opt-in?

We want to minimize overhead for sites that won't use client hints. To enable client hints the site should provide the [Accept-CH header](#) or equivalent `<meta http-equiv>` directive in the page markup. With either of those present, the user agent will append the appropriate hints to all subresource requests. In the future, we may provide an additional mechanism to persist this preference for a particular origin, which will allow the same hints to be delivered on navigation requests.

3. Why do we need client hints if we have ServiceWorker?

ServiceWorker does not have access to layout, resource and viewport width information. At least, not without introducing costly roundtrips and significantly delaying the image request - e.g. when an image request is initiated by the preload parser. Client hints integrates with the browser to make this data available as part of the request.


4. Are client hints for image resources only?

The core use case behind DPR, Viewport-Width, and Width hints is to enable resource selection for image assets. However, the same hints are delivered for all subresources regardless of type – e.g. CSS and JavaScript requests also get the same information and can be used to optimize those resources as well.

5. Some image requests do not report Width, why?

The browser may not know the intended display width because the site is relying on the intrinsic size of the image. As a result, the Width hint is omitted for such requests, and for requests that don't have "display width" - e.g. a JavaScript resource. To receive Width hints make sure to specify a sizes value on your images.

6. What about *<insert my favorite hint>*?

ServiceWorker enables developers to intercept and modify (e.g. add new headers) all outgoing requests. As an example, it is easy to add [NetInfo](#) -based information to indicate current connection type – see "[Capability reporting with ServiceWorker](#)". The "native" hints shipped in Chrome (DPR, Width, Resource-Width) are implemented in the browser because a pure SW-based implementation would delay all image requests.

7. Where can I learn more, see more demos, and what about...?

Check out the [explainer document](#) and feel free to [open an issue on GitHub](#) if you have feedback or other questions.

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.