Client hints: What they are? How do they work? Why do we need them?

Thank you to Rowan Merewood for his patient explanation and review of this post.

What are client hints?

Client Hints are a content negotiation tool and can help with making responsive images easier to work with and give you other tools to help create a better user experience.

They are a set of HTTP request headers allowing clients to indicate a list of device and browser preferences.

Using client hints isn't automatic: servers must tell clients what hints they want clients to send using the Accept-CH (accept client hints) header:

```
Accept-CH: Width, Viewport-Width, Downlink
```

or an equivalent HTML meta element with the http-equiv attribute.

```
<meta http-equiv="Accept-CH" content="Width, Viewport-Width, Downlink">
```

In subsequent requests, the client will send these headers with the appropriate values based on their characteristics and the server can use the values in these headers to craft customized responses for each client request based on the values the client sends.

One of the potential drawbacks and the reason why neither Mozilla nor Apple has implemented client hints is the fear that they allow for easier fingerprinting of users and their devices

Case Studies

There are many use cases for Client Hints. I'll illustrate what I consider the most interesting ones.

A new way to do user agent sniffing

The user agent string has been around since the World Wide Web first became a thing. This was defined all the way back in 1996 (RFC 1945 for HTTP/1.0), where you can find the original definition for the User-Agent string, which includes an example:

```
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
```

This header was intended to specify, in order of significance, the product (browser or library) and a comment that usually contained the version.

Since the header was first introduced, it has evolved into something that is much more complicated and easier to use in fingerprinting you and your connection.

I ran the following command in multiple browsers to see if they produced any result that was less frightening:

```
console.log(navigator.userAgent);
```

The first result is from Safari Technology Preview 119. It lies about the version of macOS it's running under but it provides all the necessary information to fingerprint a user:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/14.1 Safari/605.1.15
```

Next one is from Firefox Nightly (version 87 when I wrote this). It lies about the exact version of macOS it uses but in a very transparent way:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10.16; rv:87.0)
Gecko/20100101
Firefox/87.0
```

The following one from Chrome 84 in Android (taken from Improving user privacy and developer experience with User-Agent Client Hints)

```
Mozilla/5.0 (Linux; Android 10; Pixel 3)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/84.0.4076.0
Mobile Safari/537.36
```

And finally Chrome stable (version 88) on the laptop where I'm writing this:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 11_1_0)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/88.0.4324.96
Safari/537.36
```

Do you see the privacy problem? You can use any of these strings to identify the operating system name and version, version of the browser without the person reading the string having to do anything about it.

Once they have the basic information about your browser and your computer they can try to exploit vulnerabilities in the OS or the browser. The worst of those are zero-day exploits.

I can hear the comments. You don't need browser detection when you can do feature detection in Javascript.

The user agent string can also be used to lie to all sites or only to specific ones. There are also things like the OS the user is running that can't be feature detected.

For most cases I would agree but if you follow the Intent to Deprecate and Freeze: The User-Agent string discussion in blink-dev you'll see that there are still valid use cases for sniffing browser versions. Client hints provide a more granular way to do it by breaking down the string into components and forcing developers to request the components they need or want to use.

The following user agent client hints are available since Chrome 84 and were run against my current browser (Chrome 88 stable)

Server Response Accept-CH Client Request header	Client Request Example value	Description
Sec-CH-UA	“Chromium”;v=“88” “Google Chrome”;v=“88”	List of browser brands and their significant version
Sec-CH-UA-Mobile	?0	Boolean indicating if the browser is on a mobile device (?1) or not (?0)
Sec-CH-UA-Full-Version	“88.0.4324.150”	The complete browser version
Sec-CH-UA-Platform	“macOS”	The platform for the device, usually the operating system (OS)
Sec-CH-UA-Platform-Version	“11_2_0”	The version for the platform or OS
Sec-CH-UA-Arch	“x86”	The underlying architecture for the device. While this may not be relevant to displaying the page, the site may want to offer a download which defaults to the right format
Sec-CH-UA-Model	“”	The device model. This is mostly applicable to mobile devices where the type of device may be relevant

To start we can use the meta tag to tell the server what we want to send to it.

```
<meta http-equiv="Accept-CH" content="Sec-CH-UA, Sec-CH-UA-Mobile, Sec-CH-
```

The server will return the following headers when sending information back to the client:

```
HTTP/1.1 200 OK
Accept-CH: Sec-CH-UA,
Sec-CH-UA-Mobile,
Sec-CH-UA-Full-Version
```

For every subsequent request, the client will send the headers again, this time with the appropriate values, like this:

```
Sec-CH-UA: "Chromium";v="84", "Google Chrome";v="84"
Sec-CH-UA-Mobile: ?0
Sec-CH-UA-Full-Version: "84.0.4143.2"
```

We can use the headers on the server to do some data processing or take into account that some features may not be available in all browsers or may not be fully implemented everywhere.

```php
<?php
// Specific for Chrome
$is_chrome = stristr($_SERVER["Sec-CH-UA"], "Chrome") !== "Sec-CH-UA"e :

if ( $is_chrome ) {
 // Do whatever you want with the information
} else {
 // Do something else
}
```

Because these Client Hints are only available in Chrome, we still need a fallback method for detecting user agents. If at all possible, use feature detection but if you can't then you'll have to parse the `navigator.userAgentData`, or the `navigator.userAgent` object yourself to get the information you need.

The `navigator.userAgentData` object contains basic information about the user agent, the brand (name and version), and whether the device is mobile.

```
if (!'userAgentData' in navigator) {
 console.log('userAgentData is not defined');
} else {
 console.log('userAgentData is defined');
 console.log('do something with it');
}
```

The navigator.userAgentData object provides two children: an array of brands and version. The brands array contains children with browser name and version. In the case of Chrome, the array contains the Chrome and Chromium as valid browser names.

```
const brands = [... Object.entries(navigator.userAgentData.brands)];
const isMobile = navigator.userAgentData.mobile;

brands.forEach((brand) => {
 console.log(`Browser: ${brand[1].brand} ${brand[1].version}`);
});
console.log(`Is mobile: ${isMobile}`);
```

This is not the complete version of the user agent; only the major version. There may be times when more information than what we can get from `navigator.userAgentData` so we parse the user agent string or get more detailed information about the browser by calling `navigator.userAgentData.getHighEntropyValues()` in Chrome 90 and later. The "high entropy" term is a reference to information entropy or the amount of information that these values reveal about the user's browser. It's up to the browser what values, if any, are returned.

```
// Log the full user-agent data
navigator.userAgentData.getHighEntropyValues([
   "architecture",
   "model",
   "platform",
   "platformVersion",
   "uaFullVersion",
 ]).then((ua) => {
   console.log(ua)
 });
```

Performance client hints

Another way we can use client hints is to do performance work. It would be nice if we could keep this within client hints to use a single interface for all the things we want to send to the server, wouldn't it?

The following table presents performance client hints and their Javascript equivalent method in the navigator.connection object.

Type of Resource	Description	Client Hints	JS equivalent
Effective Connection Type	A basic outline of the user's connection type: 4g, 3g, 2g, slow-2g	ECT	`navigator.connection.effectiveType`
Round Trip Time	An estimate of the round trip time of the request (on the application layer). The value of RTT is rounded to the nearest 25 milliseconds to prevent fingerprinting	RTT	`navigator.connection.rtt`
Save data	Whether the user has enabled data saver. Of all the client hints, Save-Data is the only one you can't opt into with Accept-	Save-Data	`navigator.connection.saveData`

Type of Resource	Description	Client Hints	JS equivalent
	CH. Only the user can control whether this hint is sent by enabling Chrome’s Data Saver feature		
Downlink	approximate speed of the user’s connection in megabits per second. To revent fingerprinting the value of Downlink is rounded to the nearest multiple of 25 kilobits per second	Downlink	`navigator.connection.downlink`
Device Memory	Rough estimation of the device’s memory. To prevent fingerprinting the value of Device-Memory is intentionally coarse. Valid values are 0.25, 0.5, 1, 2, 4, and 8	Device-Memory	`navigator.deviceMemory`

The first use that came to mind when looking at these client hints was to lighten the load of slower browsers or browsers in slower connections.

We first tell the server the client hints that we want to work with

```
<meta http-equiv="Accept-CH" content="ECT, RTT, Downlink, Save-Data, Devi
```

We can then play with the data in our PHP templates or Express routes to modify what we send to the user based on the device and network characteristics it sends.

For example, we could provide a lighter library to browsers where device memory is less than 8GB.

```
<?php
$memory_capacity = $_SERVER["Devi"Device-Memory" ( $memory_capacity < 8 )
 echo( '<script src="regular-script.js"></script>' );

} else {
 echo( '<script src="light-script.js"></script>' );
}
```

Or we could provide certain files if the client reports a 4G connection (meaning 4g or better, including desktop) and the estimated round trip is less than 200ms.

```
<?php
$estimated_rtt = $_SERVER["HTTP"HTTP_RTT"nnection_type = $_SERVER["HTTP_E

if ( $connection_type == "4g" ) &&"HTTP_ECT"mated_rtt < 200 ) { ?>

<?php } else {
 // don't load the resources
 // or load an alternative
}
?>
```

Improving responsive images

Responsive images are good, but there is too much of a good thing. I've written picture elements like this (adapted from: Automating Resource Selection with Client Hints):

```
<picture>
 <!-- serve AVIF -->
 <source
   media="(min-width: <source
   media="(min-width: 50em)"
   sizes="50vw"
   srcset="/image/thing-200.avif 200w,
   /image/thing-400.avif 400w,
   /image/thing-800.avif 800w,
   /image/thing-1200.avif 1200w,
   /image/thing-1600.avif 1600w,
   /image/thing-2000.avif 2000w"
   type="image/avif">: 30em) 100vw"
   srcset="/image/thing-crop-200.avif 200w,
   /image/thing-crop-400.avif 400w,
   /image/thing-crop-800.avif 800w,
   /image/thing-crop-1200.avif 1200w,
   /image/thing-crop-1600.avif 1600w,
   /image/thing-crop-2000.avif 2000w"
   type="image/avif">
 <!-- serve WebP -->
 <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">
 <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"><!-- serve PNG to others -->width: 30em) 100vw"
   srcset="/image/thing-crop-200.png 200w,
   /image/thing-crop-400.png 400w,
   /image/thing-crop-800.png 800w,
   /image/thing-crop-1200.png 1200w,
   /image/thing-crop-1600.png 1600w,
   /image/thing-crop-2000.png 2000w">
 <!-- fallback for browsers that don't support picture -->
 <img src="/image/thing.jpg" width="50%">
</picture>
<!-- fallback for browsers that don't support picture --><!-- fallback fo
 <img src="/image/thing.jpg" width="50%">
</picture>
```

As wordy as the code is, it shows what an optimal art direction, DPR and Size picture element looks like.

We have three different pairs of picture elements for each image format that we want to support.

It’s awesome that we can do this but it’s a lot of work and the code will continue to grow as new formats work with different browsers.

We could optimize how we write picture elements using client hints and the Accept HTTP Header. The idea is the following:

- The accept header determines if a browser supports a given format or not
- The DPR client header will determine what version of the image to use

We use the information to build source elements only if a given format is supported and use the DPR from the client to generate the approrpriate sources.

The following example only converts one of the image formats we might want to use if supported. It is not complete.

```
<?php
// Check Accept for an "image/avif" substring.
$avif = stristr($_SERVER["HTTP_ACCEPT"], "image/avif") !== false ? true :

// This fu"DPR"// This function will generate the
// sources for the picture elementateSources(name) {
 if ( $avif ) {
   echo( "<source media=\"(max-width: 799px)\"" . "srcset=\"images/" . nar
 } else {}
?>"<source media=\"(max-width: 799px)\""?>
```

We then call the function like this in the HTML. We write the picture element and use the PHP function inside to conditionally generate AVIF and WebP source elements and the PNG version.

```
<picture>
 <?php generateSources('sample-image');>
 <img  src="images/sample-image.jpg"
       alt="sample image">
</picture>
```

This is one example of many. Other examples include

Using the Downlink and RTT client hints to serve lighter content for low bandwidth or poor connections

- Provide OS-specific downloads and resources
- One final thing to remember: Client Hints as currently implemented and deployed need to be a progressive enhancement or you will have unexpected but unpleasant results.