Migrate to User-Agent Client Hints

- Audit collection and use of user-agent data
  - Are you only using basic user-agent data?
- Strategy: On-demand client-side JavaScript API
- Strategy: Static server-side header
- Strategy: Delegating hints to cross-origin requests
- Strategy: Delegating hints to iframes
- Strategy: Dynamic server-side hints
- Strategy: Server-side hints required on first request
- Strategy: Legacy support
- Further support

The User-Agent string is a significant passive fingerprinting surface in browsers, as well as being difficult to process. However, there are all kinds of valid reasons for collecting and processing user-agent data, so what's needed is a path to a better solution. User-Agent Client Hints provide both an explicit way to declare your need for user-agent data and methods to return the data in an easy-to-use format.

{% Aside %} For more information on Client Hints and expanding them with user-agent data, read the introductory article on User-Agent Client Hints. {% endAside %}

This article will take you through auditing your access to user-agent data and migrating user-agent string usage to User-Agent Client Hints.

Audit collection and use of user-agent data

As with any form of data collection, you should always understand **why** you are collecting it. The first step, regardless of whether or not you will be taking any action, is to understand where and why you are using user-agent data.

If you don't know if or where user-agent data is being used, consider searching your front-end code for use of `navigator.userAgent` and your back-end code for use of the `User-Agent` HTTP header. You should also check your front-end code for use of already deprecated features, such as `navigator.platform` and `navigator.appVersion`.

From a functional point of view, think about anywhere in your code where you are recording or processing:

- Browser name or version
- Operating system name or version
- Device make or model
- CPU type, architecture, or bitness (for example, 64-bit)

It's also likely that you may be using a third-party library or service to process the user-agent. In this case, check to see if they are updating to support User-Agent Client Hints.

Are you only using basic user-agent data?

The default set of User-Agent Client Hints includes:

- `Sec-CH-UA`: browser name and major/significant version
- `Sec-CH-UA-Mobile`: boolean value indicating a mobile device
- `Sec-CH-UA-Platform`: operating system name
  - *Note that this has been updated in the spec and will be reflected in Chrome and other Chromium-based browsers shortly.*

The reduced version of the user-agent string that is proposed will also retain this basic information in a backwards-compatible way. For example, instead of `Chrome/90.0.4430.85` the string would include `Chrome/90.0.0.0`.

If you are only checking the user-agent string for browser name, major version, or operating system, then your code will continue to work though you are likely to see deprecation warnings.

While you can and should migrate to User-Agent Client Hints, you may have legacy code or resource constraints that prevent this. The reduction of information in the user-agent string in this backwards-compatible way is intended to ensure that while existing code will receive less detailed information, it should still retain basic functionality.

Strategy: On-demand client-side JavaScript API

If you are currently using `navigator.userAgent` you should transition to preferring `navigator.userAgentData` before falling back to parsing the user-agent string.

```
if (navigator.userAgentData) {
  // use new hints
} else {
  // fall back to user-agent string parsing
}
```

If you are checking for mobile or desktop, use the boolean `mobile` value:

```
const isMobile = navigator.userAgentData.mobile;
```

`userAgentData.brands` is an array of objects with `brand` and `version` properties where the browser is able to list its compatibility with those brands. You can access it directly as an array or you may want to use a `some()` call to check if a specific entry is present:

```
function isCompatible(item) {
  // In real life you most likely have more complex rules here
  return ['Chromium', 'Google Chrome', 'NewBrowser'].includes(item.brand);
}
if (navigator.userAgentData.brands.some(isCompatible)) {
  // browser reports as compatible
}
```

{% Aside 'gotchas' %} `userAgentData.brands` will contain varying values in a varying order, so don't rely on something appearing at a certain index. {% endAside %}

If you need one of the more detailed, high-entropy user-agent values, you will need to specify it and check for the result in the returned `Promise`:

```
navigator.userAgentData.getHighEntropyValues(['model'])
  .then(ua => {
    // requested hints available as attributes
    const model = ua.model
  });
```

You may also want to use this strategy if you would like to move from server-side processing to client-side processing. The JavaScript API does not require access to HTTP request headers, so user-agent values can be requested at any point.

{% Aside %} Try the User-Agent Client Hints JavaScript API demo. {% endAside %}

Strategy: Static server-side header

If you are using the `User-Agent` request header on the server and your needs for that data are relatively consistent across your entire site, then you can specify the desired client hints as a static set in your responses. This is a relatively simple approach since you generally only need to configure it in one location. For example, it may be in your web server configuration if you already add headers there, your hosting configuration, or top-level configuration of the framework or platform you use for your site.

Consider this strategy if you are transforming or customizing the responses served based on the user-agent data.

{% Aside %} You can also consider migrating to the On-demand client-side JavaScript API strategy instead of sending additional headers. {% endAside %}

Browsers or other clients may choose to supply different default hints, so it's good practice to specify everything you need, even if it's generally provided by default.

For example, the current defaults for Chrome would be represented as:

⬇ Response headers

```
Accept-CH: Sec-CH-UA-Mobile, Sec-CH-UA-Platform, Sec-CH-UA
```

If you also wanted to receive the device model in responses, then you would send:

⬇ Response headers

```
Accept-CH: Sec-CH-UA-Mobile, Sec-CH-UA-Model, Sec-CH-UA-Platform, Sec-CH-UA
```

{% Aside 'gotchas' %} Ordering is not important, the example is listed alphabetically. {% endAside %}

When processing this on the server-side you should first check if the desired `Sec-CH-UA` header has been sent and then fallback to the `User-Agent` header parsing if it is not available.

{% Aside %} Try the User-Agent Client Hints HTTP header demo. {% endAside %}

Strategy: Delegating hints to cross-origin requests

If you are requesting cross-origin or cross-site subresources that require User-Agent Client Hints to be sent on their requests then you will need to explicitly specify the desired hints using a Permissions Policy.

{% Aside %} Permissions Policy is the new form of Feature Policy {% endAside %}

For example, let's say that `https://blog.site` hosts resources on `https://cdn.site` which can return resources optimized for a specific device. `https://blog.site` can ask for the `Sec-CH-UA-Model` hint, but needs to explicitly delegate it to `https://cdn.site` using the `Permissions-Policy` header.

The list of policy-controlled hints is available in the Clients Hints Infrastructure draft

⬇ Response from `blog.site` delegating the hint

```
Accept-CH: Sec-CH-UA-Model
Permissions-Policy: ch-ua-model=(self "https://cdn.site")
```

⬆ Request to subresources on `cdn.site` include the delegated hint

```
Sec-CH-UA-Model: "Pixel 5"
```

You can specify multiple hints for multiple origins, and not just from the `ch-ua` range:

⬇ Response from `blog.site` delegating multiple hints to multiple origins

```
Accept-CH: Sec-CH-UA-Model, DPR
Permissions-Policy: ch-ua-model=(self "https://cdn.site"),
                    ch-dpr=(self "https://cdn.site" "https://img.site")
```

{% Aside 'gotchas' %} You do **not** need to include each delegated hint in `Accept-CH`, but you **do** need to include `self` for each hint, even if you are not using it directly at the top-level. {% endAside %}

Strategy: Delegating hints to iframes

Cross-origin iframes work in a similar way to cross-origin resources, but you specify the hints you would like to delegate in the `allow` attribute.

⬇ Response from `blog.site`

```
Accept-CH: Sec-CH-UA-Model
```

↪ HTML for `blog.site`

```
<iframe src="https://widget.site" allow="ch-ua-model"></iframe>
```

⬆ Request to `widget.site`

```
Sec-CH-UA-Model: "Pixel 5"
```

The `allow` attribute in the iframe will override any `Accept-CH` header that `widget.site` may send itself, so make sure you've specified everything the iframe'd site will need.

Strategy: Dynamic server-side hints

If you have specific parts of the user journey where you need a larger selection of hints than across the rest of the site, you may choose to request those hints on demand rather than statically across the entire site. This is more complex to manage, but if you already set different headers on a per route basis it may be feasible.

The important thing to remember here is that each instance of the `Accept-CH` header will effectively overwrite the existing set. So, if you are dynamically setting the header then each page must request the full set of hints required.

For example, you may have one section on your site where you want to provide icons and controls that match the user's operating system. For this, you may want to additionally pull in `Sec-CH-UA-Platform-Version` to serve appropriate subresources.

⬇ Response headers for `/blog`

```
Accept-CH: Sec-CH-UA-Mobile, Sec-CH-UA-Platform, Sec-CH-UA
```

⬇ Response headers for `/app`

```
Accept-CH: Sec-CH-UA-Mobile, Sec-CH-UA-Platform, Sec-CH-UA-Platform-Version, Sec-CH-UA
```

Strategy: Server-side hints required on first request

There may be cases where you require more than the default set of hints on the very first request, however this is likely to be rare so make sure you've reviewed the reasoning.

The first request really means the very first top-level request for that origin sent in that browsing session. The default set of hints includes the browser name with major version, the platform, and the mobile indicator. So the question to ask here is, do you require extended data on the initial page load?

{% Aside %} Also consider making use of the On-demand client-side JavaScript API strategy to alter content within the page as opposed to server-side. {% endAside %}

For additional hints on the first request there are two options. First, you can make use of the `Critical-CH` header. This takes the same format as `Accept-CH` but tells the browser that it should immediately retry the request if the first one was sent without the critical hint.

⬆ Initial request

```
[With default headers]
```

⬇ Response headers

```
Accept-CH: Sec-CH-UA-Model
Critical-CH: Sec-CH-UA-Model
```

◫ Browser retries initial request with the extra header

```
[With default headers + …]
Sec-CH-UA-Model: Pixel 5
```

This will incur the overhead of the retry on the very first request, but the implementation cost is relatively low. Send the extra header and the browser will do the rest.

{% Aside 'gotchas' %} Any `Critical-CH` values must be a subset of the values in `Accept-CH`. `Accept-CH` is all the values you would like for the page, `Critical-CH` is the subset of those values you **must** have or you cannot load the page properly. {% endAside %}

For situations where you require really do require additional hints on the very first page load, the Client Hints Reliability proposal is laying out a route to specify hints in the connection-level settings. This makes use of the Application-Layer Protocol Settings(ALPS) extension to TLS 1.3 to enable this early passing of hints on HTTP/2 and HTTP/3 connections. This is still at a very early stage, but if you actively manage your own TLS and connection settings then this is an ideal time to contribute.

Strategy: Legacy support

You may have legacy or third-party code on your site that depends on `navigator.userAgent`, including portions of the user-agent string that will be reduced. Long-term you should plan to move to the equivalent `navigator.userAgentData` calls, but there is an interim solution.

{% Aside 'warning' %} This is not recommended and not supported in any way. This solution is included for completeness but if you spend any time attempting to fix bugs in it, that time would be better spent doing the actual migration. {% endAside %}

UA-CH retrofill is a small library that allows you to overwrite `navigator.userAgent` with a new string built from the requested `navigator.userAgentData` values.

For example, this code will generate a user-agent string that additionally includes the "model" hint:

```
import { overrideUserAgentUsingClientHints } from './uach-retrofill.js';
overrideUserAgentUsingClientHints(['model'])
  .then(() => { console.log(navigator.userAgent); });
```

The resulting string would show the `Pixel 5` model, but still shows the reduced `92.0.0.0` as the `uaFullVersion` hint was not requested:

```
Mozilla/5.0 (Linux; Android 10.0; Pixel 5) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/92.0.0.0 Mobile Safari/537.36
```

Further support

If these strategies do not cover your use case, please start a Discussion in privacy-sandbox-dev-support repo and we can explore your issue together.

Photo by Ricardo Rocha on Unsplash

□□□ #2
□ □□□ □□□ 2026-10-31 22:03:12 UTC
□ □□□ □□□ 2026-10-31 22:03:13 UTC