Blink principles of web compatibility

First version Feb 27, 2017 - rbyers@chromium.org
Maintained by blink API owners, discussion
Public doc, linked from blink launch process documentation.
Access to comment should be open to all chromium.org accounts, feel free to request access.
—

The Chromium project aims to reduce the pain of breaking changes on web developers. But Chromium's mission is to advance the web, and in some cases it's realistically unavoidable to make a breaking change in order to do that. Since the web is expected to continue to evolve incrementally indefinitely, it's essential to its survival that we have some mechanism for shedding some of the mistakes of the past. It's always been part of the culture of the Chromium project to (carefully and thoughtfully) make changes that are in the best long-term interest of the web ecosystem, despite causing some short-term pain.

In the Chromium web platform change process, non-trivial intentional breaking changes must be approved by 3 Blink API owners. Between the Blink fork (April 2013) and March 2017, we analyzed over 200 such proposals, ultimately approving some plan for ~85% of them. There are a number of tools for analyzing web compat, but it's still difficult to know when a breaking change is acceptable in Blink.

Ideally we'd have clear guidelines that allow anyone to determine which breaking changes are acceptable and which are not. As a step towards that goal, this document aims to document the primary factors the Blink API owners consider, along with references to specific examples. Many of these factors are about predictability, but they are also about other cross-cutting concerns like performance, security and the user experience. Based on a public, inclusive and data-driven discussion of these trade-offs, it's the Blink API owners' responsibility to judge whether the overall balance is good for the Chromium project and its user and developer constituency.

Overview

Minimizing end-user impact

First and foremost we have a responsibility to users of Chromium-based browsers to ensure they can expect the web at large to continue to work correctly.

Page views impacted

The primary signal we use is the fraction of page views impacted in Chrome, usually computed via Blink’s UseCounter UMA metrics. As a general rule of thumb, 0.1% of PageVisits (1 in 1000) is large, while 0.001% is considered small but non-trivial. Anything below about 0.00001% (1 in 10 million) is generally considered trivial. There are around 771 billion web pages viewed in Chrome every month (not counting other Chromium-based browsers). So seriously breaking even 0.0001% still results in someone being frustrated every 3 seconds, and so not to be taken lightly!

Keep in mind that our UseCounters have some blind-spots including environments where UMA is disabled (there is debate on the extent this is true in Enterprise), China (where Google’s metrics servers are blocked), and Chromium derivatives (which don’t have access to Google metrics servers). PageVisit-weighted usage is also biased against single-page apps such as games. If there’s reason to expect such cases to be much more likely to be impacted, UseCounter data may be less relevant. When there is reason to believe usage might be concentrated in some sub-population (eg. ChromeOS users, users in Japan), then subpopulation-specific metrics should also be considered.

When using some other metric like fraction of network requests, care is needed to interpret it in a context that makes sense to a user. Eg. impacting 0.0001% of HTTP requests may seem inconsequential except when you consider a typical page view has around 100 HTTP requests and so if breaking just one makes the site unusable, then that would disable 1 in 10,000 page views which is actually quite a lot.

Relevant example cases:

- Support for legacy protocols (`ftp:`) in subresource requests.
  Usage of 0.0003%, no reported complaints (but a couple low-impact cases found in HTTP Archive)
- Remove SVGPathSeg interfaces
  Usage was around 0.001% but generated quite a lot of complaints (and hasn't been removed in other browsers). See ease of adaptation for details.
- Remove filesystem: URLs in iframes
  Usage of 0.0000008% but still severely broke some use cases resulting in a need for emergency mitigations for those customers

Unique sites impacted

Sometimes the compat impact is dominated by a few particularly popular sites. The most popular sites are also often the easiest to get updated. In such cases, "page views impacted" is a poor indicator of the risk. In general we do not let any single site prevent us from making a breaking change (assuming other conditions below such as "ease of adaptation", "outreach" and "interoperability" are satisfied). This is especially true when the only site(s) impacted are Google properties; we go out of our way to make it clear that Google properties do not receive special treatment in Blink.

In addition, a UseCounter analysis is sometimes not possible (eg. due to popular code which enumerates all properties on Event and Node instances). A site-impact analysis can often form a good proxy for the lack of "page views" analysis.

When using HTTP Archive for this, getting hits on over 0.01% of sites is considered an indication of moderate risk, while fewer than 0.001% typically represents low risk. However, it depends highly on the shape of the using code, removal can be possible even with >1% hits.

Relevant example cases:

- border-image spec compliance
  UseCounter was high but hypothesized (but not proven) to be dominated by mobile GMail. Landed removal prior to getting commitment from GMail to update.
- KeyboardEvent.keyLocation
  UseCounter is useless because of popular Event-property-enumeration code which

copies all properties on every input event. Relied on detailed HTTP Archive analysis instead.

- Smoothscroll.js breakage impacting scrollTopLeftInterop fix
  Fixing the scrollTop bug was blocked for a long time due to this library used on 0.3% of the top 500k. We ultimately added a library-specific workaround which we saw triggered on ~0.03% of PageVisits.

Severity of breakage

To evaluate the impact of a breaking change, it's important to understand the severity of the impact. If the break typically results in a page being unusable (such as by preventing scrolling), this is extremely serious. If, on the other hand, the break typically results in a small visual difference (such as a missing border or incorrect font being used) then the risk is lower.

To estimate the severity we will sometimes manually analyze a random sample of hits from HTTP Archive, grouping them into buckets by the impact observed on that site.

Relevant example cases:

- data: URL in SVGUseElement
  UseCounter was moderate (>0.01%). Outreach led to some important fixes, but lots of minor breakage wasn't fixed. These were usually just some minor icon on the page not showing up, not a functional blocker. No issues reported for at least 3 milestones after shipping.
- Remove: legacy constants on Event
  Couldn't use UseCounter. Got lots of hits in static search, but analysis of those hits suggested most were non-breaking. No reports of breakage.
- Temporarily Remove: sendBeacon() with a Blob whose type is not CORS-safelisted
  We chose to throw an exception to report the failure in order to increase developer visibility, but this broke at least one site which would have worked correctly if we had followed the pre-existing failure-mode of returning false. Guidance for increasing developer visibility is covered under Outreach.

Keep in mind that UseCounter and HTTP Archive often don't find Enterprise sites, since these are usually not indexed by the search engine, and whose users don't opt into stats reporting. Enterprise Policy settings may also skew use counter results which are reported.

Chrome's release process

We depend on Chrome's release process (finch and canary, dev, beta, stable releases) and Blink's reliable bug triaging to reduce the cost of mistakes, and allow us to still make breaking change decisions in the face of uncertainty. In particular, if a user or developer (or two) files a bug about a breaking change before that change has hit stable, that probably means the change has a high risk of being disruptive (expect ~100x the pain when the change hits stable).

Unfortunately the inverse is not true, just because a change causes no reports of issues in beta, it still may cause a major problem when it reaches stable - especially for environments like enterprises which rarely have any beta user deployments. As of late 2022 we've decided our best defense is to ensure that changes with any real risk are guarded by flags where possible so that they can be disabled in the field without requiring a respin.

We trust our QA process to rapidly connect reports of regressions back to the CL that caused them (via our per-CL bisect tools), and we trust Blink engineers and team leads to act swiftly and responsibly in response to any such bugs being filed, re-evaluating the decision based on any new information.

Relevant example cases:

- Deprecate and Remove: HTTP/0.9 Support
  Landed in M54, reported to break access to one type of router a canary user about a month later. Re-landed in M55 and breakage reported during the final week of beta, reverted for M56 and fixed in M57.
- Expose reporting API interfaces to Javascript
  There was some concern about the compat risk of exposing a generic name like 'Report' on the window object, but an HTTP Archive search suggested the risk was very low so we decided to try but be prepared to revert if we heard of any issue in beta. Unfortunately a popular enterprise application depended on `window.Report` not existing, and broke when Chrome hit stable, necessitating an emergency respin. Having the API change be RuntimeEnabled would have allowed for a finch killswitch to be deployed rather than doing a respin.

User opt-out

In some cases (such as when we expect a small number of users to have high impact breakage), the compat risk can be mitigated by providing a user opt-out. This can be as simple as a temporary chrome://flags entry we can tell impacted users to set in an emergency, or a new enterprise policy setting administrators can set when a critical in-house application depends on it. Or, in extreme cases, it may be some form of user-visible UI similar Chrome's pop-up blocking UI and it's "always allow pop-ups for this site" option.

Relevant example cases:

- Flash deprecation
  - An important step in deprecation flash on all browsers was to increasingly rely on "click to play" behavior and site-specific settings.
- Document Level Passive Event Listeners
  - While evaluating the cost/benefit tradeoff of this intervention a tri-state flag was added (chrome://flags/#document-passive-event-listeners). To launch, "default" was changed from "disabled" to "enabled" but the flag preserved so users could still switch back to "disabled" if necessary.

Maximizing user experience

The Chromium project has also made a promise to end-users to work to constantly improve their experience using the web.

Security / privacy

We will generally tolerate more risk for a breaking change which substantially improves user security or privacy. The Chrome Security team is actively deprecating powerful features on insecure origins. In extreme cases, a breaking change may be necessary to stop serious active exploitation of a browser flaw, and in such cases the security improvement may outweigh most other factors.

Relevant example cases:

- Remove: Insecure origin usage of geolocation
  This caused some site/user pain (eg. store locator features being harder to use) but was removed primarily with the justification that we don't think users want to share their precise location with everyone who has access to the network.
- Deprecate and Remove: Top-frame navigations to data URLs
  Actively used in phishing attacks. Substantial usage (0.05% of all navigations).

Performance

We have greater tolerance for breaking changes which improve performance, especially if the performance improvement in the wild can be quantified precisely (eg. via a finch trial) and is substantial. In extreme cases, a performance improvement may result in a substantial decrease in page abandonment.

Relevant example cases:

- Blocking the load of cross-site, parser-blocking scripts inserted via document.write in the main frame, for users on 2G.
  Caused quite a bit of pain, but resulted in 25% more page loads finishing so was ultimately trading one form of breakage (abandonment due to extremely slow page loads) for another (3rd party scripts not loading).
- Document Level Passive Event Listeners
  Caused mostly subtle breakage on a large fraction of sites (and a bit of less subtle breakage) but cut our #1 scrolling perf metric in half. Highly contentious but has stuck in Chrome and may be on a path to interop.

User annoyance

We have greater tolerance for breaking changes which reduce behavior that users are actively complaining is annoying. The browser is called a “user agent” because it will explicitly make decisions on behalf of the user in some cases (such as blocking pop-ups).

Relevant example cases:

- Allow autoplay video if muted
  Early in Chrome for Android we blocked all autoplaying video - requiring a user gesture. This led the ecosystem to animated images and JS-based video decoders which were worse in almost every way (higher data / battery usage). Ultimately we (and other browsers) shifted to allow video to autoplay but (to prevent user annoyance) require a user gesture in order to play audio.
- Remove: touch drag as a user gesture in cross-origin iframes (intervention)
  Prevented pop-ups from opening when users happened to touch-scroll on top of certain buggy advertisements.
- Block navigator.vibrate in cross-origin iframes
  Users were reporting issues with ads causing their phone to vibrate. Ultimately took a path similar to audio - permitted after a user activation in the frame.

Minimizing web developer impact

We also have a responsibility to web developers to minimize the pain and cost incurred by breaking changes.

Ease of adaptation

A change which can be trivially accounted for (such as by replacing a webkit-prefixed API name with the non-prefixed version) is generally considered to be lower risk than one which requires more effort. At the extreme end, a breaking change which takes away a capability (no matter how minor) which cannot be achieved by any other mechanism is generally considered high risk. Generally we avoid breaking any use cases which cannot be shown to have a reasonable alternate implementation (with exceptions such as under “user annoyance“ and “security / privacy”).

In some cases it may reduce risk to provide a JavaScript polyfill which acts as a high-fidelity replacement for the removed functionality. When one customer in particular claims they cannot easily adapt to a removal, we may ask the team involved to work with them to demonstrate the feasibility of an alternate implementation.

Sometimes the risk and complaints around ease of adaptability are more FUD than real issues (developers who know nothing is ever as easy as claimed would rather fight against a breaking change than go down the uncertain path of trying to cope with it). Here a successful polyfill deployment can be extremely powerful in dispelling the FUD.

Note however that there is a substantial portion of the web which is unmaintained and will effectively never be updated (including historical archives like archive.org). It may be useful to look at how long chromium has had the behavior in question to get some idea of the risk that a lot of unmaintained code will depend on it. A breaking change being trivial to accommodate does not necessarily mean it is low risk. In general we believe in the principle that the vast majority of websites should continue to function forever.

Relevant example cases:

- WebSQL
  WebSQL was never broadly supported across browsers and various efforts were made to limit and deprecate it in chromium over many years. In one case, one partner convinced us that there was no way for them to maintain the performance of their application given the removal we wanted to do. Ultimately what allowed WebSQL to be deprecated was the introduction of the high-performance Origin-Private FileSystem API and proof that SQLite could perform well on top of it in Wasm.
- On-screen keyboard resizing behavior
  We struggled for years to unify OSK resizing behavior across browsers and platforms. In one failed attempt our advice to developers who really needed the other behavior was to use a Javascript API to achieve a similar effect. When this change ultimately succeeded it included a very simple meta-tag opt-out which we saw get adopted for ~0.02% of Android page loads without any bugs or complaints of breakage being filed.
- SVGPathSeg interface
  Low usage, but a small number of vocal developers. Developing a polyfill library appeared to reduce the developer frustration and cost dramatically.
- MediaStreamTrack.getSources()
  Low usage. A polyfill was provided with the removal, and no complaints were heard (though we don't know whether any sites actually used the polyfill code).
- Remove showModalDialog
  Usage reported at <0.006% of PageViews but there was no great work-around (especially for sites depending on the blocking nature) and ultimately caused a LOT of user and developer pain, particularly within enterprises (exacerbated by the lack of visibility into enterprise use cases).
- Remove: -webkit-canvas and Document.getCSSCanvasContext
  Showed how SVG was often a better replacement, and ultimately Houdini custom paint was the right answer (but didn't block removal on custom paint). Little to no complaints.
- Deprecate: SMIL
  Many use cases have good alternatives, but removal was blocked in part due to a handful of use cases which were supported fairly broadly across browsers in SMIL

without widely supported alternatives (missing / broken WebAnimations support in other browsers).

Developer opt-in / opt-out

It's quite common that we overestimate the ease of adaptation. For higher risk changes we're not confident developers can easily accommodate, it's often a good idea to provide a staged migration path. First it may be helpful to make the new behavior available in an opt-in fashion and confirm that non-trivial real-world applications are successfully able to opt-in to the new behavior. More importantly, an opt-out can be provided to make the ease of adaptation trivial - explicitly re-enabling the behavior that is now off by default.

For cases where we can get most of the benefit of a deprecation even when a small fraction of sites choose to opt-out indefinitely (eg. performance interventions), a permanent opt-out as part of the API design may be appropriate. Feature Policy is a new tool to enable easy and consistent page-wide opt-outs, especially when removing a capability from iframes by default.

When we expect an opt-out to be temporary, deprecation trials can be a useful tool for giving developers some time to migrate while opening a communication channel with them.

It's important for browser engineers to resist the temptation to treat breaking changes in a paternalistic fashion. It's common to think we know better than web developers, only to find out that we were wrong and didn't know as much about the real world as we thought we did. Providing at least a temporary developer opt-out is an act of humility and respect for developers which acknowledges that we'll only succeed in really improving the web for users long-term via healthy collaborations between browser engineers and web developers.

Relevant examples cases:

- Document Level Passive Event Listeners
  We long wanted to make touch listeners not block scrolling. To provide a migration path we first standardized and shipped an API to allow developers to explicitly opt-in to the non-blocking behavior ("passive"). Only after seeing some (limited) success with developers opting into "passive" listeners did we make the default "passive" in some limited cases, while retaining the ability for developers to explicitly request non-passive. This opt-out turned out to be necessary for Facebook due both to a bug we didn't realize we had, and (to a lesser degree) a design limitation we didn't expect would be very common.
- Block navigator.vibrate in cross-origin iframes
  After blocking vibration in all cross-origin iframes we determined that there were a few legitimate niche scenarios where it was really the behavior that the user and embedding page wants. This ended up being a motivating use case to ship feature policy opt-in to enable a document to easily re-enable this capability in its iframes (without requiring complex postMessage-based delegation of capability).

- Tab discarding with an origin trial opt-out

Enterprise policy opt-out

Enterprises are an important market for Chrome and other chromium browsers. Enterprise environments may be special in that they have little usage of release channels (beta etc.), are often a chromium-only environment, and may depend heavily on web applications not used anywhere else. UMA data from enterprises needs to be interpreted carefully, as enterprise opt-in is low, and possibly biased towards certain types of enterprises—for example, UMA opt-in rate for enterprises that use manual updates is unknown. (google-internal data).

When making a breaking change which you have reason to believe may impact enterprise environments, reach out to the chromium-enterprise list and consider adding a temporary enterprise policy opt-out and/or a finch knob. Although it can be turned off, most enterprises leave finch enabled. In some cases it may make sense to leave behavior enabled for Chrome Apps.

Relevant example cases:

- Remove <keygen>
  Since enterprises often have atypical PKI setups/requirements, many PKI-related breaking changes are done with an enterprise policy opt-out knob. UMA metrics for the policy are used in deciding when the code can truly be removed.
- Don't allow popups during page unload
  A major enterprise application was broken by this change. It was reverted and Chrome stable had to be re-spun in a hurry, while leaving the customer broken for 8 days. An enterprise policy opt-out was added, and the change re-enabled.
- Remove filesystesm: URLs in iframes
  Usage was almost zero so we thought it would be safe to remove without even a deprecation period. But Kiosk customers were severely broken, so this was re-enabled specifically for chrome apps.

Debuggability

A breaking change which results in a subtle hard-to-identify bug is generally of greater compatibility risk than one which throws a clear exception with a link for more details on how to address the issue. Most breaking changes will get a specific deprecation warning (with link) for at least one milestone to help with this. When an API is simply removed, this is straightforward (developers searching for "XXX missing in Chrome" are likely to find the details). But when the change is more subtle, it can be important to plan specifically about how developers will figure out what needs to be changed in their code.

Relevant example cases:

- Blocking the load of cross-site, parser-blocking scripts inserted via document.write in the main frame, for users on 2G.
  Console warnings generated even in non-2G scenarios to indicate that behavior will be different on 2G. Also a dedicated 'Intervention' header was added to indicate programmatically to servers who are interested in tracking how often the change was hit.
- Document Level Passive Event Listeners
  After the change in behavior, continues to generate console warnings when a page tries to cancel an event which was uncancelable due to the intervention and the page didn't follow the best practice of declarative annotation with touch-action.

Outreach

We have more tolerance for riskier breaking changes when some outreach has been done for impacted sites / libraries. For example, if a blog post has been published which shows up as one of the first few search results when searching for the console error message generated as a result of the break. Ultimately we want to minimize the cost to web developers of understanding and dealing with breaking changes.

In some cases we have evidence (eg. from HTTP Archive) that most of the breakage users will experience will be as a result of just a couple popular libraries. In those cases, contacting the maintainers and even contributing fixes can be very valuable. Searching GitHub for impacted code or issues mentioning the deprecation warning can also really help to reduce the risk of miscommunication/misunderstanding between blink engineers and web developers.

All non-trivial breaking changes should have a clear and accurate chromestatus.com entry where developers can find information on the justification for the change and the advice for accommodating to it. These will get mentioned in the chromium blog for the relevant release. We expect most breaking changes will also generate a console warning and deprecation report referencing the chromestatus entry for at least one Chrome milestone. For changes of particular relevance to enterprises, notice should also be provided in the Chrome Enterprise release notes.

Relevant example cases:

- New CSS touch-action property breaking hammer.js (talk)
  A popular library had a bug that resulted in many mobile websites being unable to scroll when we attempted to ship a new API already shipped in IE. Extensive outreach to individual impacted sites had near zero impact. What seemed to help most was working with the affected library to publish simple guidance and ensuring that guidance was easily findable by relevant Google searches. When we shipped the new API to stable, almost all impacted sites were fixed within a week with very few regression issues being reported as bugs (or discussed anywhere as far as we could find).
- data: URL in SVGUseElement
  UKM analysis was used to identify top sites, and direct outreach led to some important

fixes, including at least one that would have likely been an emergency (perhaps escalation) due to significant breakage. A blog post and sample page were published which developers found easy to follow to fix their issues. Often developers were appreciative for being given a way to achieve the same effect but that worked across all browsers and performed better.

- Document Level Passive Event Listeners
  Searching GitHub for reports mentioning the chromestatus ID helped us quickly get several developers using the right fix and built a bunch of goodwill, as well as helped us to appreciate just how confusing developers found our documentation and guidance.

Maximizing web ecosystem benefit

We also have a responsibility to the larger web platform community (other browser vendors, standards organization members, etc.) to be a responsible custodian of the web platform in addition to all the ways already mentioned above.

Interoperability

Breaking changes which align Chromium's behavior with other engines are much less risky than those which cause it to deviate. See finding a path to interop for some detailed advice and discussion. In general if a change will break only sites coding specifically for Chromium (eg. via UA sniffing), then it's likely to be net-positive towards Chromium's mission of advancing the whole web.

However, be sure to consider Chromium-specific scenarios like Android WebView, enterprise applications (invisible to UMA) and Chrome extensions. If there's a specific reason to believe, for example, that Android apps are likely to be impacted, then we may want to add a temporary "target SDK quirk" to ensure Android applications are impacted only when they're recompiled to a new version of the Android SDK. Ultimately Chromium's mission is to improve the web ecosystem at large, so we are willing to accept some amount of increased risk for such non-open-web scenarios.

Relevant example cases:

- requestAutocomplete
  Only ever shipped in Chrome. Despite quite a lot of evangelism never got any real developer usage.
- Legacy constants on Event
  No legitimate use cases, and supported in Blink and WebKit only. Ultimately removed without a single report of any issue. (Usage wasn't measured and would have been tainted by enumeration.)
- border-image spec compliance
  Mobile GMail was broken on Firefox and Edge because it relied on a WebKit-quirk

- webkitCancelRequestAnimationFrame
  Supported in Blink and WebKit only, and notably Edge supports webkitCancelAnimationFrame but *not* this variant. That indicated low compatibility risk, while removal would increase interoperability by any measure.

Standards conformance

Standards are an important tool in achieving interoperable implementations across browsers. Discussing a change in the context of the relevant standard is also a good way to involve other browser vendors. We generally try to have standards match what is implemented, rather than blindly implement what the standard currently requires. If a standard currently says something is implementation-defined, there may still exist Web content that relies on a particular behavior and so interoperability still matters.

Sometimes we will make a change that hurts interoperability and compatibility temporarily in order to align with a specification because we believe it'll result in better interoperability long-term. For example, when Edge has grudgingly copied some WebKit quirk but Firefox is holding out on principle but experiencing web compat pain as a result (because developers tend to test more on Chrome and Safari than Firefox), we sometimes decide that the spec'd behavior is actually best for the web long-term and resolve the issue by changing Chrome to match the spec and Firefox (which can result in Edge and Safari eventually being able to match the spec also).

We generally expect consensus to be reached and specifications to be updated (or at least have an open issue with proposed pending pull request) to reflect any breaking change.

Relevant example cases:

- border-image spec compliance
  Mobile GMail was broken on Firefox because GMail was depending on a WebKit-specific quirk. Edge copied that quirk but was willing to undo it if GMail was fixed. Mobile GMail wasn't being actively maintained so couldn't justify fixing their issue until Chrome changed to match Firefox and the spec. Edge then changed to also match, but the WebKit change took another 5 years before it was landed.
- The Attr example above illustrates that being removed from the standard is no guarantee that removal will work out.
- <applet>
  This was still in the standard and multiple implementations, but we changed the standard to deprecate, to eventually remove.

IP rights

The chromium project is committed to a free and open web, enabling innovation and competition by anyone in any size organization or of any financial means or legal risk tolerance.

In general the chromium project will accept an increased level of compatibility risk in order to reduce dependence in the web ecosystem on technologies which cannot be implemented on a royalty-free basis.

Relevant example cases:

- Google maintains the WebM project in part to provide royalty-free alternatives to popular video, audio and image codecs. The availability of such free alternatives means chromium is more willing to drop or otherwise limit support for patent-encumbered technologies such as HEVC (H.265).

Accepted interop risk

When an engine ships a new feature there's a judgement made about the interop risk involved (what's the risk we'll regret shipping the API in this form). Sometimes Blink chooses explicitly to ship despite higher interop risk, and we accept responsibility for that risk by being more willing to take on compat risk in the future if the consensus of the web platform community is that the API should change. Effectively, shipping under interop risk incurs some debt which can be paid back by incurring more compat risk in making a breaking change later.

Relevant example cases:

- Shadow DOM v0
  Chrome was ambitious in shipping web components before there was broad consensus. We eventually shipped the consensus API (v1) and in parallel worked on a plan to aggressively remove the (already widely used) v0 API.
- PaymentRequest
  We made a strategic decision to ship web payments support before there was broad consensus on the API, explicitly planning for some additional future compat risk (eg. via messaging / guidance to the small number of partners we knew would be the primary consumers of the API).