Layered APIs

née permafills
ojan@chromium.org, domenic@chromium.org
Last update: 2018-02
https://bit.ly/lapis-design
PUBLIC

Context and scope

Guided by the extensible web, Chrome has been focused on exposing lower-level primitives as web APIs. *Layered web APIs* is a proposed project for fulfilling the other half of the extensible web promise, by creating a new effort for standardizing and shipping higher-level APIs, in a way that scales.

For more on the motivation and benefits of layered APIs, see the public explainer document.

Goals

- Decrease page load time, data consumption, and script compilation by reducing the need for external JavaScript libraries.
- Improve the experience of developing for the web platform by providing, out of the box, a "standard library" of high-level features that making building a modern web application as easy as building for native platforms.
- The platform only pays the cost for new features sites use instead of for every feature added to the platform (e.g. binary size, initial JS context size and startup time, etc.), so that we can technically scale to a much larger standard library for the platform.
- Advance a well-layered, maintainable platform; as part of building these features, uncover additional low-level primitives that might be needed as a foundation for layered APIs or for web developers' apps.
- Expose these higher-level features in a way that allows web developers to easily use them when they're available, or fall back to polyfills when they are not, to encourage adoption.
- Make it easier to conform to Bonsai web policies by providing substitutes for disallowed features.

Non-Goals

- Bypass the standards process. Layered web APIs, like all web platform APIs, will be produced via web standards and tested with shared web platform tests.

- Bypass the Blink launch process. In Blink, the details of how we implemented layered web APIs are different than for other web APIs; see below. But they require the same level of rigor when designing and shipping them.
- Ship all features as layered web APIs. By definition, layered APIs can only use APIs that are exposed to web developers; they will not have "backdoor" APIs to get extra capabilities. They are a second track for evolving the web platform, in symbiosis—not competition—with efforts to add new lower-level capabilities.
- Mandate a particular implementation strategy across all browsers. Previous drafts of this idea (under the name permafills) emphasized heavily the idea that browsers could share JavaScript implementations of the standardized feature. That characteristic is not an essential goal; how a browser chooses to implement a standardized layered web API is up to that browser.

Layered web APIs overview

Layered web APIs are a new class of standardized web APIs with two important requirements:

- They are loaded via JavaScript imports, via a special syntax that incorporates automatic fallback to a polyfill; they are not provided automatically, e.g. in the global namespace.
- Their specification must not use any primitive operations that are not directly accessible to web developers. Stated another way, a web developer must be able to implement a given layered API's specification, purely in unprivileged JavaScript.

See the public explainer for more details on these essential characteristics of layered APIs, as well as the benefits these restrictions give, both for web developers and for standardization/implementation.

A backlog of potential layered APIs is being collected; we are also polling developers to get their interest. Our initial targets are an infinite list API (currently being researched in the infinite list study group) and an async local storage API. This initial targets are tentative, however, and may shift in response to interest from the layered APIs community.

Layered web APIs in Blink

In Blink, we plan to take advantages of the layering requirement built in to layered APIs by actually implementing them in JavaScript. This automatically gives us several advantages:

- It provides **a hard check that the layered API specification is properly layered**; if the specification accidentally calls for using some primitive not exposed to web developers, then Blink will be unable to implement it.
- We can easily **leverage existing loading infrastructure for JavaScript modules**, over time evolving it to allow such possibilities as lazily-loading from a CDN for uncommon layered APIs, or doing bytecode caching for particular popular ones.

- It **increases potential web developer involvement** in the development and implementation of layered APIs; instead of requiring C++ expertise and a deep understanding of Blink's architecture to grok why features are coded a certain way, web developers can read the code directly, and perhaps even contribute to it.
- It **allows other implementations to reuse the code** if they wish. Like all parts of Blink, layered API implementations will be open source. But, because of the layering boundary, reusing this code will generally be easier than reusing Blink's C++, which has deep ties to core libraries, common types, and other Blink-specific assumptions.

Note that our layered APIs implementations, by virtue of shipping with Blink, have significant advantages over other JavaScript libraries:

- They can be aggressively cached and optimized, even including first-load roundtrips.
- They do not need to support multiple browsers, or support older versions of Blink; they can use the latest platform primitives to ensure a good user experience, and avoid the bloating of traditional polyfills with their large multiple-browser vs. multiple-versions support matrix.
- They evolve in tandem with their standard, which is necessarily in a backward compatible way ("don't break the web").
- Like all web platform features, only the latest version is available (i.e. they "version with the browser"). Thus, all web pages using a layered API will be sharing the same codebase, instead of the fragmented ecosystem we see for popular libraries today.

Detailed design

Fallback syntax

The fallback syntax shown in the explainer, of a `std:layered-api-identifier|fallback-url` syntax, is one of several possibilities that were exploring in a a dedicated document. Notably, this choice of syntax means that browsers which do not implement the special fetching logic for std: URLs are not able to receive the fallback. We believe this will be reasonable, since layered APIs require module script support anyway, which limits them to somewhat-recent browsers.

The main alternative considered after the chosen one, B, was G. We remain open to switching based on feedback from web developers and other browsers.

Loading

Specification

Moved to explainer

Implementation

See the separate doc.

JavaScript infrastructure

Function.prototype.toString()

`toString()` leaks a method or class's implementation details. We believe it should be relatively straightforward to make toString on layered APIs behave the same as it does for built-ins to mitigate this, i.e. `function foo() { [native code] }`. However, doing so adds new capabilities to layered APIs that web platform features do not have. Domenic is exploring a general opt-out for web platform libraries.

Optimizing layered APIs

In addition to the network benefits of layered APIs, we can theoretically pre-compile them and get a JS parse and initialization time benefit as well. This won't be in the initial implementation though as it won't matter until we have sufficient layered APIs that a significant percentage of a site's JS might plausibly be layered APIs.

V8 already contains several mechanisms for this sort of thing, including snapshotting and storage of Ignition bytecode. In the future we may want to explore storage of an optimized representation as well.

Mitigating leaking implementation details

The specification for layered web APIs, like for any web APIs, constrains the API's observable features. For example, it would not be allowed to have extra properties like `myLayeredAPI._privateState`. We'd need to use the upcoming private fields and methods for our implementation.

The biggest implementation detail leak is the reliance on globals and prototypes, which are modifiable by user script. For example, if the async local storage layered API uses `self.indexedDB.open()`, this is susceptible to tampering in multiple ways:

- The global `self` could be overridden
- The `indexedDB` property of self could be overridden
- `IndexedDBFactory.prototype` could be overridden
- The open property on `IndexedDBFactory.prototype` could be overridden

To mitigate this, we need some way of getting a "clean" copy of various browser built-ins, for use in our layered web API implementation. Per the maxim that layered web APIs must not use any

capabilities that are not already exposed, we need to find a way to expose this ability to get clean copies to all JavaScript modules.

At this time we are still exploring the possible options for solving this. Some notable incomplete solutions are:

- Creating an iframe and using its clean globals does not work in workers; additionally, someone might have tampered with document or document.createElement, preventing us from creating an iframe.
- The JavaScript realms proposal only includes the JavaScript built-ins, not the web platform ones we would need.

(See this document for previous discussion and brainstorming on this subject; most of which has made its way here.)

Interaction with policies

Layered API code must run under the (TBD, as of the time of this writing) Bonsai policies. It is specifically meant to be synergistic with them, and should never stray outside the bounds of "best practice" that they establish. This may require updating layered APIs over time, as the set of Bonsai policies grows.

Additionally, layered APIs should endeavor to work under more-restrictive policies, beyond just the Bonsai policies. For example, as of the time of this writing, the Bonsai policies do not include a CSP policy for disallowing eval. However, this is a relatively common policy for websites to apply. As such, layered APIs code should not use eval, or other features that we expect to be disallowed by some modern web apps.been decided has been incorporated into the above.)

Considerations beyond Blink

Standardization and API design

Layered APIs are web platform features like any other: they require a specification, web platform tests, cross-browser consensus, and are meant to integrate well with the rest of the web platform. As such, they'll require similar standardization and design work as existing web platform features do.

In particular, layered APIs will need review from those familiar with web platform APIs, such as through the W3C TAG's design reviews, or Chrome's standards mentors (internal) and API mentors (public). Familiar resources, such as the TAG design principles or promises guide, will prove helpful. See also the web standards playbook for guidance on the process of early design

and building cross-browser consensus. Design will start with explainers, then proceed to specifications. In short, for Blink developers, we will be following the full Blink launch process.

Given the requirement that layered APIs be properly layered, and given that at least some browsers will choose to implement them in JavaScript, their specifications will likely be a bit different from traditional ones:

- Instead of using Web IDL, they will use something closer in style to the ECMAScript or Streams specification, since those are much closer to the semantics of normal, non-generated JavaScript code.
- The specifications will often contain instructions of the form "Perform the steps listed in the X method of Y on Z, given arguments A and B." That is, they will directly reuse the steps underlying a given public method, since that more clearly layers on top of existing functionality.

We're prototyping this slightly different style in the async local storage specification. This approach is still tentative and open to feedback, especially from other members of the web standards community.

Open technical considerations

Several technical considerations that are not Blink-specific are tracked externally:

- Custom elements require dashes in the name
- Requiring imports can be suboptimal for custom element layered APIs

Testing

Layered APIs will be tested via web-platform-tests, in the same way as other web platform features.

In Blink, we should also employ standard JavaScript code coverage tools such as Istanbul to keep the tests comprehensive.

Next steps

End of 2017/January 2018

Work:

- Get agreement on the fallback syntax, e.g. <script stdsrc=""> or something related to module specifiers. (Dedicated document.)

    - *Done: settled on std:x|y, at least pending developer/standards community feedback.*
- Figure out what other high-priority layered APIs we hope to work on in 2018 (could just be one).
    - *Done: settled on async local storage.*

Results:

- Initial specification for loading solution
- Technical design doc for its implementation
- Buy-in from loading team for implementing by Q3 2018

End of Q1 2018

Work:

- Research existing infinite list designs, including web and other platforms
- Iterate toward an infinite list design that makes sense for a web API
- Evolve a POC infinite list implementation in tandem with design efforts, dogfooding with partners to validate the design
- Sketch the design for async local storage
    - *Done*, but with open issues: domenic/async-local-storage
- Bring developer partners and other browser vendors into the conversation

Results:

- For both 2018 layered APIs:
    - Explainers
    - An initial public API sketch (e.g. Web IDL or similar)
    - Buy in from at least one other browser vendor on the concept
- For infinite lists:
    - A research document exploring the infinite list space, what the environments do similarly and differently, and what the tradeoffs are
    - A list of features that we're not planning to do in v1, with explanations of how they could be added later, or why we don't ever plan on doing them.
    - A POC implementation validating many of the concepts of layered APIs and our infinite list design. We should *not* expect this code to survive into the next phase; it's for proving out concepts, not prototyping the final high-quality implementation.

This work will be potentially announceable at Google I/O.

End of Q2 2018

Work:

- Write the layered API specs (see standardization and API design)
- TAG review for the layered API fallback syntax

- TAG review for the layered API APIs
- Implement the layered API infrastructure code
- Get other browser vendors to continually engage in the design (and potentially implementation) as it proceeds

Results:

- Beta-quality layered APIs ready for testing in Canary, or as a polyfill, or similar
- Specs that are nearing finality
- At least surface-level web platform tests for all layered APIs

End of Q3 2018

Work:

- Finalize the layered API specs and implementation
- Go through the Blink launch process, and ship them
- Drive web platform tests code coverage of the code as high as possible (100%?), measured using Blink's JavaScript implementation

Results:

- Layered APIs successfully launched in Chrome!
- Hopefully, at least one other browser will announce their intention to ship some of these layered APIs as well, and perhaps will have shipped them already.

This work will be potentially announceable at Chrome Developer Summit.

Security and privacy considerations

In Blink, because layered APIs run in the exact same way as author code, security considerations for layered API code itself are minimal: essentially, they are reduced to the existing problems of running JavaScript code inside the browser's existing sandboxes. (Contrast this with other approaches in the Alternatives section below.)

The way in which layered APIs will be loaded requires some care from the security side. The above proto-specification mitigates much of this by intervening at the module resolution layer, instead of the fetching layered. Note that once loaded, the layered API scripts—like all scripts—will execute in the origin of the page, so there shouldn't be any origin-based security concerns.

From the privacy side, the main concern comes if we choose to lazy-load some layered APIs from a (Google-run, for Chrome) CDN. We would need to ensure that a Referer header is not sent in such cases, to limit our ability to track usage. The user's IP address would still be visible, however.

Finally, we note that individual layered APIs may have their own security and privacy considerations. Fortunately, the web standards process has a fairly robust way of auditing these, through wide multi-stakeholder reviews, and mechanisms such as the W3C TAG security and privacy questionnaire.

Will we ship existing libraries as layered APIs?

As layered APIs are just standardized web platform features, this question is essentially the same as "will we ship existing libraries as web platform features?" The answer is the same as the one we have given forever: generally, no.

However, especially given Blink's implementation plan to implement layered APIs as bundled JavaScript, people might ask this same question with renewed fervor. Here we reemphasize the usual reasons why we don't just ship existing libraries with the browser.

First, versioning allows traditional libraries to move forward at an aggressive pace, responding to changes in the platform and differentiating themselves within the evolving competitive landscape. However, version skew means that very few sites are using the same set of code for a library. In contrast, with layered APIs—like all other web platform features—you don't get to choose what version you use; you instead get the one implemented in the current browser version. This mismatch makes traditional JavaScript libraries a poor fit for bundling.

Relatedly, given that they have discrete versions, libraries don't maintain backwards compatibility to the level of web platform features. Much of what allows libraries to present elegant developer experiences is their ability to make a clean break from past versions and build a focused product.

Additionally, in order for Blink to be comfortable with bundling a piece of code, we must have some final say over changes made that land in the browser (e.g. via code review). External parties will still be heavily involved in the layered APIs process, as they are in all standards efforts, but we can't just turn over the final say over our codebase to library developers, and library developers are unlikely to let Blink take over their codebase either.

Finally, we do not want to pick winners in the ecosystem. While many libraries have a large usage base, they always have competitors. We want to layered APIs features that represent repeated—not competing—work, that goes through the usual cross-browser consensus process, instead of being picked by a kingmaker.

These requirements do not **preclude** an external library from becoming a layered API. They only make it **unlikely**. If a library is:

- High usage
- Agreed amongst browsers and developers to be a canonical implementation
- Versionless
- Committed to backwards compatibility
- Willing to be reformulated as a web standard
- Willing to cede governance to the standards process

… then they may be viable candidates for a layered API.

Ideas for future work

Versioning independently of the browser binary

We won't have it for v1, but eventually you could imagine versioning layered APIs more or less frequently than the browser binary.

One notable feature of this is that it would allow for the possibility of Chrome changing out from under web developers without the UA string changing. That could be difficult to reason about. It's possible we could mitigate this if we allow the UA string to also update dynamically, with some kind of layered API version.

Alternatives and adjacent concepts

Aggressively caching libraries on CDNs

A common question is how layered APIs differ from CDNs and aggressive caching. While these two ideas are very similar, a few advantages come from shipping directly with a browser.

Most obviously, shipping with the browser avoids *any* network cost on page load, whereas a CDN only reduces the cost. In emerging markets, where even a single RTT can be on the order of seconds, this benefit is significant. While aggressive caching could solve this problem for future requests, the cost must still be paid at least once.

Shipping libraries directly with the browser also means they can be more streamlined than a generic library. We can store parsed code, reducing bootup cost by hundred of milliseconds for large JS features. We can also strip out unnecessary feature detects or polyfills that aren't needed for that particular browser version.

Another major difference is that because layered APIs are browser features, they go through the same collaborative cross-browser design process that other features do. If we were to instead

promote a set of canonical "Chrome libraries" on a "Chrome CDN", they would simply be seen as yet-another-Google-framework, and not achieve the same kind of staying power as other web platform features. If we were to pile more special-casing on top of this, by pre-caching the contents of this Chrome CDN, we'd still miss many of the benefits: e.g., developers wanting to create pages that are performant in *all* browsers would be unable to use too many layered APIs. Instead, layered APIs are a shared investment by the whole web standards and browser implementer community, which improve the experience for developers and users in all browsers.

Finally, "layered APIs vs. CDNs" is a somewhat misleading framing. Under the hood, layered APIscould even be implemented as cross-browser, collaboratively-developed libraries that are lazily loaded by the browser from CDNs. Indeed, for rarely-used layered APIs, this may be more appropriate than bundling them with the initial download, or requiring each browser to do their own implementation. In that case, the delta between layered APIs and a CDN-based solution is their commitment to a transparent, stable development process and the commitment to first class support within browsers.

Implementing high-level features in C++

Traditional web platform features are implemented in C++ and Web IDL. This makes sense for low-level features that need to deeply integrate with the rest of the web platform code. But for implementing high-level features, our participation in the layered APIs project, and our decision to write them in JavaScript, is a better path.

High-level features implemented in C++ would sit on top of low-level features implemented in C++. However, those C++ APIs are very different across browsers. This means that any code we write in Blink would not be something that other browsers would be able to adopt if they wished. This potential for code sharing is a great accelerant for the layered API ecosystem, and we should not lose it.

Additionally, it would take extreme rigor to maintain the same type of layering between high- and low-level features that is possible by using JavaScript and the web-exposed platform APIs. At the C++ level, there is no clear, programmatic distinction between public, web-exposed APIs and "private" ones meant for use by the rest of the browser code. (This distinction is instead encoded in IDL and the bindings system.) It's too easy to accidentally use one of these private APIs, breaking the layering and thus making your high-level API dependent on the implementation details of the rest of the browser. This hurts the feedback loop in which layered APIs help uncover new low-level features to expose to the web, and increases maintenance and refactoring costs for the private APIs accidentally depended upon.

Finally, at least with our current browser architecture, there is no way to implement a C++ API without adding to the binary size and cost of a new JS context. Every new web API implemented in C++ today causes a small additional memory costs and startup time for every content process and every V8 context. Layered APIs avoid this; they are pay-as-you-go.

Component libraries authored by the Polymer team

The Polymer team already builds several component libraries, both external and internal to Google. These projects share similar motivations and should coordinate/share work wherever possible. Ultimately, though, they are somewhat separate.

Since the Polymer UI libraries are not a cross-browser collaboration, that work can generally move faster and express stronger opinions in terms of design architecture and UI. For example, Material Design does not generally make sense for cross-browser layered APIs (see explanation in the public explainer), and the Polymer components have specific data-binding practices which are not standard across the web.

All that said, some Polymer-built components will be generically useful, and could be good candidates to introduce into the standards process and mold into layered APIs eventually.

Blink-in-JS

Blink-in-JS allows implementing web platform features or other parts of the browser in JavaScript. It is based on Web IDL bindings and running the JavaScript code in an isolated world, causing performance overhead. It is tightly coupled with the C++ implementation, and as such this JavaScript code had special privileges, causing security concerns.

Layered APIs are simpler and less powerful than Blink-in-JS, as they run the code in the context of the main page, and have no access to privileged APIs. As such, they will not be used for implementing security-sensitive features or browser features such as a password manager or find-in-page. This means the security issues with Blink-in-JS do not exist for layered APIs. It also reduces the infrastructure needed, and the performance overhead.

Layered APIs as a project are also broader in scope than Blink-in-JS, as they are meant to open a new way of building cross-browser high-level web platform features. They are not simply an implementation strategy to allow writing JavaScript in place of C++.

V8 extras

V8 extras are another way to implement web platform features in JavaScript. Unlike Blink-in-JS, they do not have Web IDL integration, and like layered APIs, they run directly in the main context.

However, V8 extras are designed for implementing traditional web platform features. As such, programming them requires excessive care not to expose any internals (see Dealing with modifications to the built-ins above, versus V8 extras' security considerations). Additionally, they explicitly include an escape hatch for exposing their functionality to C++, and for exposing

functionality from C++ to them. Layered APIs are written as idiomatic JavaScript, and maintain a stronger layering, due to their goal as building only high-level features.

Similarly to the contrast with Blink-in-JS, V8 extras are largely an alternate implementation strategy, and not a new way of building cross-browser high-level web platform features.