Manifest V3

devlin@chromium.org

Status: DRAFT. This document may be updated with additions, modifications, or removals.

Updated: November 18th, 2018

This doc on the google intranet: go/manifest-v3

https://crbug.com/896897

THIS DOCUMENT IS PUBLIC

Objective

This document describes a large set of planned changes to the Chrome Extensions platform, ranging from core features to specific APIs, with the motivation of increasing security, privacy, and performance for extensions. These changes will be bundled with a new manifest version. Once announced and implemented, it will eventually be required for all extensions over a year+ rollout process.

See the summary of changes here.

Background

The extension manifest version is a mechanism for restricting certain capabilities to a certain class of extensions. These restrictions can be in the form of either a minimum version or a maximum version. Restricting to a minimum version allows newer APIs or capabilities to only be available to newer extensions, while restricting to a maximum manifest version allows older APIs or capabilities to be gradually deprecated. The implication is that eventually support for old manifest versions is removed, allowing us to fully remove those older capabilities. This is one of the most effective and clear, though heavy-weight, mechanisms for making breaking changes to the extensions platform.

The manifest version is specified through the `"manifest_version"` key in an extension's `manifest.json` file, and is a single integer value.

We have incremented the manifest version once before, from manifest version 1 (which was implicit and wasn't actually specified) to manifest version 2. This introduced a number of breaking changes (many of these changes can be seen here), including requiring resources exposed to web pages to be specified in a `web_accessible_resources` section, adding a default CSP (Content Security Policy), and changing the format of the `page_action` and `background` entries in the manifest.

Motivation

A new manifest version (which will be gradually required by extensions) is one of the most effective ways to make breaking changes and enforce certain restrictions. There are a number of capabilities, practices, and APIs that extensions use that we want to migrate away from due to their negative impact on the user experience. We also plan to restrict new APIs and features to the new manifest version, providing additional incentive for extensions to migrate.

The current extensions platform has a number of issues in the areas of performance, security, privacy, and ergonomics. By implementing a new manifest version, we can enforce certain best practices, ban negative practices, and provide a clear migration path for developers.

Goals

Developers should fall into a pit of success: writing a secure, performant, privacy-respecting extension in Manifest V3 should be easy, while writing an insecure, non-performant, or privacy-leaking extension should be difficult. As a corollary to this, we should have higher confidence in the quality of a Manifest V3 extension, opening the door to ideas like actively recommending extensions to users.

Security

Implementing an extension in Manifest V3 should provide strong security guarantees, both from outside attackers (targeting the extension) and from malicious extensions. The extension should be protected from outside attackers (e.g., malicious websites trying to hijack an extension through XSS). Additionally, users should feel confident in installing an extension and be reasonably assured that the extension cannot easily cause significant, lasting damage. Finally, Manifest V3 should increase our ability to audit extensions using automated systems (like Navitron - internal, public) and manual review processes.

Privacy

Users should have increased control over their extensions. A user should be able to determine what information is available to an extension, and be able to control that privilege.

Performance

Extensions implemented in Manifest V3 should be performant. Long-running background processes should not be allowed. APIs should be guaranteed to be fast and efficient, and non-performant misuse of these APIs should be difficult.

Summary of Changes

The **TL;DR**. More details for these are provided below.

P1

Background Process: Migrate from event/persistent background pages to Service Workers.
Restricting Origin Access: Migrate to an `activeTab`-style model, where access is granted at runtime.
Remotely-Hosted Code: Disallow extensions from using remotely-hosted code.

Cross-Origin Communication: Content scripts share the same cross-origin communication rules as the page. Extension pages can make cross-origin requests to any site they have access to.
Manifest Host Permission Specification: Host permissions will be specified in a new `host_permissions` manifest key; `permissions` will only be used for API permissions.
Promise-Based APIs: Support promise-based APIs.

P2

Web-Accessible Resources: Require all resources that will be committed in a non-extension context to be specified in the web accessible resources and support dynamic resource URLs.
Dynamic Content Scripts: Provide more capabilities to extensions to support dynamic content scripts.

API Changes

WebRequest: Restrict the blocking capabilities of the `webRequest` API.
DeclarativeNetRequest: Launch `declarativeNetRequest`, which provides an alternative to the blocking capabilities of `webRequest`.
Browser Action and Page Action: Merge `browserAction` and `pageAction` into a single `action` API.
chrome://favicon: Migrate the `chrome://favicon` capabilities to a new `chrome.favicon` API.
Capturing APIs: Coalesce capturing capabilities from `tabs`, `pageCapture`, `tabCapture`, and `desktopCapture` APIs into a single `capture` API.
Remove APIs replaced by the Open Web Platform: Remove any APIs whose functionality is now available on the Open Web Platform.
Update APIs for ServiceWorker-based processes: Update APIs for use with ServiceWorker-based processes, especially those that assume running on the main thread.
Remove Deprecated APIs: Remove any publicly-deprecated APIs.
Miscellaneous API Changes: Other API updates for performance, utility, or ergonomics.

P1 Changes

The following are significant changes to the extensions platform or core APIs, which are currently being planned as part of Manifest V3.

Background Process

Summary

In Manifest V3, the only allowable background presence type will be ServiceWorkers.

Description

Many extensions have a form of "background" process. This process allows the extension to perform operations outside of a given tab or visible web contents, as well as react to different events. Most APIs are also restricted to extension processes, which are commonly (though not exclusively) the background process. As an example, an email-checking extension would use its background process to communicate with the email server and check the number of unread emails for an account, and could then update the user-visible UI with the result. This way, the UI is up to date, even if the user does not have a tab open to the extension. This is often desired or even critical to the extension's usefulness; in the case above, if the extension required a tab, it might be no better than always having an email tab open.

There are currently two options for the type of extension background processes: persistent background pages and event pages (also known as "lazy" background pages). Both of these are essentially a web page, complete with DOM, HTML capabilities, and JavaScript, that runs outside the view of the user.

A persistent background page runs from the moment Chrome starts until it shuts down. This has the advantage of making development easy (state is easy to keep, at least in a single run, since it is never destroyed) and reducing "lag time" in extension responses, since the process is always in a ready state, able to respond to relevant events or inputs. It has the disadvantage that the extension is always consuming memory, CPU, and other resources (including the process itself), even when it is not performing any work. In the example extension above, even if there is no email coming in, the extension would be running 100% of the time that Chrome is open.

An event page is created in response to a certain event occurring. The extension registers for the events it wishes to respond to (such as tab creation), and is kept alive while doing work or responding to events. When the extension is not doing work and not responding to events, the event page is suspended, allowing the resources and the process to be reallocated to a different task. This has the obvious advantage of being more resource-friendly, but has the disadvantage that the extension will be slower to respond to an event if the process needs to be started first.

Event pages are almost always preferable to persistent background pages, since they allow valuable resources to be returned to Chrome (or the system) when the extension is not active.

The web has evolved significantly since event pages were first implemented, and now websites can create ServiceWorkers. ServiceWorkers are very similar to (and were motivated by) event pages. Each is temporal, being set up and torn down in response to certain events, each allows the client to register for different relevant events, and each is a background-running presence that is unseen by the user.

We plan to replace background pages with ServiceWorkers, registered at install time and granted access to extension APIs. The concepts will be more familiar to developers (both going from extensions to the web and the web to extensions), since the mechanisms will be largely the same. We will be able to reduce the amount of extension-specific code since, instead of implementing a custom background-running presence, we can extend the existing ServiceWorker context. Most importantly, the platform will evolve with the open web; as improvements to ServiceWorkers and the web platform are made, the extensions platform benefits from those same improvements.

Restricting Origin Access

Summary

In Manifest V3, host permissions will be granted by the user at runtime (similar to `activeTab`, but with options for the user to choose to always run on a specific site or all sites), rather than install-time.

Description

Origin access permissions are used to determine which sites an extension can interact with. This affects many APIs, including script injection, the webRequest API, cookies, and others. These can be used for any number of purposes, from ad blocking to accessibility to website enhancement and others.

Extensions can request different host patterns and scopes in the manifest. They can request specific hosts (`https://google.com`), host patterns (`https://*.google.com`, allowing access on all google.com domains and subdomains), or even request permission to all sites (`<all_urls>` or `*://*/*` for all HTTP/HTTPS sites). The latter allows the extension to inject scripts on, intercept network requests from, and read cookies for any domain, including social networks, financial websites, corporate sites, etc. - all without any further indication or permission from the user.

In some cases, this broad permission can be necessary to the extension's functionality (most content blocking is desired to run on every site, as are accessibility features). In other cases, extensions request this permission even if they don't need it, partially because there is very little penalty for doing so now and partially because requesting permissions after installation results in users being prompted for permissions, which leads to many users uninstalling or disabling the extension.

As of August 2018, more than 80% of the top 1000 (internal-only link) extensions request access to all domains or an all-domains-like pattern (e.g., `*://*.com/*`).

One alternative to requesting access to all URLs is to request access to an arbitrary URL through the user of the `activeTab` permission. This permission allows the extension to run on any arbitrary site *once invoked by the user while the tab remains on the origin*. This is an important permission, since many extensions may need to act on an unknown number of sites, but not want to run without the user's knowledge (such as an extension to dim the background DOM during video playback or pin an image). Unfortunately, this permission has seen very limited usage (roughly 3% of the top 1000 extensions), with most extensions simply opting for access to all domains (indeed, the two examples above, though they could use `activeTab`, do not).

Another alternative to broad permission requests is optional permissions, which allow developers to request a given permission at runtime. This can give the user greater context into the permission request, and doesn't require them to approve the permission prior to installation. Like `activeTab`, optional permissions are very underutilized (roughly 6% of the top 1000 extensions).

In Manifest V3, we want `activeTab`-style host permissions to be the default, with a number of extra options. Instead of being granted access to all URLs on installation, extensions will be unable to request `<all_urls>`, and instead the user can choose to invoke the extension on certain websites, like they would with `activeTab`. Additional settings will be available to the user post-installation, to allow them to tweak behavior if they so desire.

This has a number of advantages. In the default case (click-to-run), it is clear to the user when the extension is running, and has a safe default (not running on any site). When the user chooses to invoke the extension on a given site, there is implicit understanding that the extension will "see" the contents of the page. Finally, this avoids giving the users an ultimatum. Currently, we force users to accept all permissions and install the extension, or accept none and refuse the extension. This provides a middle ground - install the extension, but use it on the user's terms.

This is also being implemented for Manifest V2. See the Runtime Host Permissions design document (internal-only) and PRD (internal-only).

Remotely-Hosted Code

Summary

Beginning in Manifest V3, we will disallow extensions from using remotely-hosted code. This will require that all code executed by the extension be present in the extension's package uploaded to the webstore. Server communication (potentially changing extension behavior) will still be allowed. This will help us better review the extensions uploaded, and keep our users safe. We will leverage a minimum required CSP to help enforce this (though it will not be 100% unpreventable, and we will require policy and manual review enforcement as well).

See the full document here (internal only).

Cross-Origin Communication

Summary

Extension origins will continue to be able to make cross-origin requests to any sites they have permission to access. Content scripts will have the same permission as the page they are injected in.

Description

Extensions currently have the ability to perform cross-origin requests to any domain listed as part of the extension's host permissions, whereas these would normally be blocked by the Same Origin Policy and subject to Cross-Origin Resource Sharing (CORS). These requests can be made from either one of the extension's pages or from its content scripts. Unfortunately, the fact that these are allowed in content scripts is detrimental to the security guarantees of the site isolation project, since it limits our ability to determine, from the browser process, if a request should be allowed for a renderer. This is because we can no longer block a request from a renderer hosting `evil.com` to `google.com`, since the request could have been made on behalf of an extension's content script on `evil.com`.

Beginning in Manifest V3, content scripts will not be given special privileges regarding the requests they can make. If a content script needs access to data fetched from a cross-origin server that requires authentication, it can proxy the request through its background page using extension messaging.

Extension pages can continue to make authenticated requests to any origin for which they have permission. There are a number of use cases that require this. Additionally, if an extension already has permission to that host, preventing these requests would be meaningless, since the extension could simply script the host instead.

Manifest Host Permission Specification

Summary

Move host permission specification in the manifest to a separate key, `host_permissions`. The `permissions` manifest key will only be used for API permissions. Host permissions should omit the path. Specific host permission patterns may be limited to a certain number.

Description

Today, the `permissions` manifest key contains both API permissions (like `tabs`) and host permission patterns (like `https://example.com/*`, `*://*/*`, or `<all_urls>`). With the origin

access changes as a result of the RuntimeHostPermissions feature, specifying these permissions in the same key as permissions that will be auto-granted can cause developer confusion. Additionally, host permissions and API permissions are significantly different already in terms of format, capability granted, and treatment.

In Manifest V3, extensions will specify host permissions in a new `host_permissions` key. The `permissions` and `optional_permissions` manifest keys will be reserved for API permissions. Since host permissions are going to require runtime approval by the user, they will not need a separate `optional_host_permissions` entry (whether such permissions will be removable via the `permissions.remove` API method is TBD.)

Open question: Should `activeTab` be removed in favor of specifying `<all_urls>` in the `host_permissions` key of the manifest?

Detailed design doc required.

Remove Support for NaCl/PNaCl

Summary

Extensions will no longer be allowed to use NaCl and PNaCl. Instead, they should use Web Assembly.

Description

NaCl and PNaCl have been deprecated for the web since May, 2017. We have allowed extensions to continue using these technologies. However, WebAssembly is now mature enough to serve as an alternative. Extensions should use WebAssembly instead, which will allow Chrome to fully remove support for NaCl and PNaCl.

Promise-Based APIs

Summary

Extension APIs will be promise-based. The older callback version will continue to be supported.

Description

JavaScript Promises are a tool used in asynchronous programming that allows for easier chaining of calls and cleaner code. Promises also provide an ability to indicate failure via a promise rejection. Using promises for extension APIs would be significantly cleaner and more modern than the current callback-based approach, where errors are surfaced through `chrome.runtime.lastError`. There is currently a bug to migrate extension APIs to be promise-based.

One potential downside is that this would not allow us to optimize API return values if the extension provided a callback (since we would not know if the promise was going to be used or not). However, we currently do this exceedingly rarely, and very few APIs have expensive return values.

This would be a good step for the platform as a whole, to bring the APIs more closely to modern web APIs, and would also make development of extensions easier and closer to web development. Additionally, this is a helpful motivation for developers to upgrade to manifest V3, as it is a widely-desired feature.

In order to maintain backwards compatibility (and not force developers to rewrite their extension more than they already have to), providing a callback to an API method will continue to work. If a callback is provided, a promise will not be returned.

P2 Changes

The following are changes that are being discussed for Manifest V3, but are not fully decided yet. These may or may not ship with the initial version of Manifest V3, and could be added subsequently.

Web-Accessible Resource Hardening

Web-accessible resources control which resources can be embedded in a web page. For instance, an extension that embeds an iframe in the web page must specify the iframe's HTML file in its manifest. This prevents web pages from being able to embed extension resources that shouldn't be exposed to the web.

This is most important for security - extensions are more protected from third-party interaction if they are not embedded in the DOM. It is also important for privacy, as embedding extension resources in the DOM is an easy way to identify if the extension is installed, which should be generally opaque to sites (unless the extension takes some action).

Web-accessible resources were launched in Manifest Version 2, but there are two areas for improvement.

Stricter Resource Restrictions

Currently, any resource loaded by an extension frame is allowed to load. This means that if `iframe.html` includes `some_script.js`, only `iframe.html` need be present in the `web_accessible_resources` section of the manifest. This addresses most major concerns, particularly since site isolation is now enabled, but fails to ensure that the developer has total control over what may or may not be embedded in an untrusted frame.

With Manifest V3, we can tighten these restrictions and require any resource that will be loaded in an untrusted frame to be specified in the `web_accessible_resources`.

Unique Identifiers

Currently, resources are embeddable by referencing their URL, which is `chrome-extension://<extension-id>/<resource-path>`. However, this means that any site that knows the extension ID and file structure (which are both trivially determined) can attempt to embed an extension resource. This is unfortunate for extensions that want to conditionally embed extension resources in a web page, but only if initiated by the extension itself. This also leads to frequent fingerprinting of popular extensions that expose web accessible resources.

We could improve this by (optionally) allowing resources to only be exposed through a unique identifier, rather than through their path. The extension (e.g. in a content script) could use an API to retrieve this identifier. This way, untrusted web pages would be unable to embed resources without the extension's cooperation.

Dynamic Content Scripts

With the changes to origin access to restrict extensions' ability to automatically inject scripts, the specification of scripts to always run on a site begins to make less sense. In particular, extensions may wish to surface the request to run to the user in different contexts, and may wish to provide more information. This is good, as it will (hopefully) allow the user to make more informed decisions.

To aid in this, we can allow extensions to dynamically add and remove, or enable and disable, content scripts. This would allow extensions to only add these scripts once they have permission to do so.

There are additional situations in which this is beneficial, as well. Currently, the advice for extensions wishing to dynamically inject scripts based on some knowledge at runtime is to use the tabs.executeScript API; however, this is insufficient for certain use cases. In particular, this cannot (reliably) insert a script before the page finishes loading, which is a feature that content scripts provide. Allowing dynamic content scripts would solve this use case.

API Changes

The following are changes to specific APIs.

WebRequest

Summary

In Manifest V3, we will strive to limit the blocking version of `webRequest`, potentially removing blocking options from most events (making them observational only). Content blockers should instead use `declarativeNetRequest` (see below). It is unlikely this will account for 100% of use cases (e.g., onAuthRequired), so we will likely need to retain webRequest functionality in some form.

Description

The current `webRequest` API allows extensions to intercept network requests in order to modify, redirect, or block them. It is frequently used by content blockers. Currently, with the `webRequest` permission, an extension can delay a request for an arbitrary amount of time, since Chrome needs to wait for the result from the extension in order to continue processing the request. The basic flow is that when a network request begins, Chrome sends information about it to interested extensions, and the extensions respond with which action to take. This begins in the browser process, involves a process hop to the extension's renderer process, where the extension then performs arbitrary (and potentially very slow) JavaScript, and returns the result back to the browser process. This can have a significant effect on every single network request, even those that are not modified, redirected, or blocked by the extension (since Chrome needs to dispatch the event to the extension to determine the result).

In Manifest V3, this API will be discouraged (and likely limited) in its blocking form. The non-blocking implementation of the `webRequest` API, which allows extensions to observe network requests, but not modify, redirect, or block them (and thus doesn't prevent Chrome from continuing to process the request) will not be discouraged. As an alternative, we plan to provide a `declarativeNetRequest` API (see below). The details of what limitations we may put in the `webRequest` API are to be determined.

DeclarativeNetRequest

Summary

The new `declarativeNetRequest` API will be used as the primary content-blocking API in extensions, as it is more performant and offers better privacy guarantees to users.

Description

The declarativeNetRequest API is an alternative to the `webRequest` API. At its core, this API allows extensions to tell Chrome what to do with a given request, rather than have Chrome forward the request to the extension. Thus, instead of the above flow where Chrome receives the request, asks the extension, and then eventually gets the result, the flow is that the

extension tells Chrome how to handle a request and Chrome can handle it synchronously. This allows us to ensure efficiency since a) we have control over the algorithm determining the result and b) we can prevent or disable inefficient rules. This is also better for user privacy, as the details of the network request are never exposed to the extension.

This API is currently being implemented, and will be available to both the current manifest version and Manifest V3, but will be the primary way to modify network requests in Manifest V3.

Browser Action and Page Action

Summary

We will merge `browserAction` and `pageAction` APIs in Manifest V3 into a single `action` API.

Description

The browserAction and pageAction APIs allow extensions to declare an "action" in their manifest, which becomes the toolbar icon. When these APIs were originally created, `pageActions` were designed to apply to a specific page, and would appear ephemerally in the omnibox, while `browserActions` were designed to apply to the browser as a whole, and would appear persistently in the toolbar.

This changed as part of the Extension Toolbar Redesign, which gave every extension a permanent UI surface in the toolbar or, if overflowed, the Chrome menu. This was intended to increase extension visibility to users and help them understand which extensions were installed. Overall, this change was a success (though of course not a panacea).

However, the difference between `pageActions` and `browserActions` is now heavily blurred, and can cause more confusion than anything else. In Manifest V3, we can combine these into a single `action` key in the extension's manifest, and expose a single `action` API. Optionally, developers could specify a `"default_state"` to control whether the extension was default-on or default-off.

This simplified UI would reduce code complexity, as well as present a more unified and simple UI for the platform.]

Old design doc (internal only). Updated (public) design doc needed.

chrome://favicon API

Summary

In Manifest V3, the `chrome://favicon` permission and utility will move to a new Chrome API with the `favicon` permission and `chrome.favicon` namespace.

Description

Currently, extensions can request `chrome://favicon as host permission, and this allows them to fetch a website's favicon by fetching` chrome://favicon/https://example.com `(for` example.com`'s favicon).

This has caused us endless grief. This is one of the few areas that extensions are allowed access to the `chrome:`-scheme (which is otherwise off-limits), and this permission is silently added as an additional host permission if the extension requests `<all_urls>`. Rather than have this complexity, there will be a full extensions API (under `chrome.favicon`) to support retrieving a website's favicon. This API would be available with either a new `favicon` permission, or with granted host permission for the requested favicon.

(More detailed doc required.)

Capturing APIs

There are currently five different methods (tabs.captureVisibleTab(), tabCapture.capture(), tabCapture.getMediaStreamId(), pageCapture.saveAsMHTML(), and desktopCapture.chooseDesktopMedia()) across four different APIs to allow extensions to capture the content of a user's screen. Each of these has different capabilities and permission models. In Manifest V3, we should coalesce all of these into a single API namespace, `chrome.capture`, and ensure that the API has a strong and clear permissions model.

This also helps "clean up" the `chrome.tabs` API, which currently has too many capabilities beyond tab management.

See the main document here.

APIs Replaced by the Web

The web has continued to evolve in the time since extensions were created. While many of the extension capabilities are still fundamentally limited to extensions, web alternatives exist to some APIs. When migrating to Manifest V3, we should take the opportunity to remove support for features that are now part of the open web platform. Extensions are always designed to be "the web plus some", but not an alternative to the web. There should never be an extension-way and a web-way - if it's possible on the web platform, that should be the only way.

This helps us standardize both UIs shown to users and APIs used by developers, and also allows us to advance with the open web platform.

The following APIs are now available on the web and should be removed.

- Extension Notifications (Replaced by Web Notifications)

- clipboardRead/clipboardWrite? (The extension APIs may currently allow for copying more data types than the web APIs; how widely used are these capabilities?)

API Updates For Service Worker

With ServiceWorkers as the background processes in Manifest V3, certain extension APIs will need to be updated or removed because they may no longer make sense or be possible. One major reason for this is that ServiceWorkers run on a separate thread in the renderer (similar to other web Workers); thus, they have no access to the DOM. Any APIs that return a `DOMWindow` or `HTMLElement` will need to be updated or removed.

These APIs include:

- chrome.extension.getViews(): Returns a collection of extension views and frames. We should be able to remove this in favor of ServiceWorker's ability to get and claim related clients.
- chrome.runtime.getBackgroundPage(): Returns the extension's background page as a DOMWindow object. Replace with ServiceWorker alternatives.
- chrome.app.window (only used for Chrome Apps)
- ...

Deprecated API Methods

There are a number of APIs or API methods that have been marked as deprecated for an extended period of time, but have not been effectively discontinued. These should be removed.

Deprecated APIs:

- chrome.extension.sendMessage() (undocumented, but used)
- chrome.extension.connect() (undocumented, but used)
- chrome.extension.onConnect (undocumented, but used)
- chrome.extension.onMessage (undocumented, but used)
- chrome.extension.sendRequest()
- chrome.extension.onRequest
- chrome.extension.onRequestExternal
- chrome.extension.lastError
- chrome.extension.getURL()
- chrome.extension.getExtensionTabs()
- chrome.tabs.Tab.selected
- chrome.tabs.sendRequest()
- chrome.tabs.getSelected()
- chrome.tabs.getAllInWindow()
- chrome.tabs.onSelectionChanged
- chrome.tabs.onActiveChanged
- chrome.tabs.onHighlightChanged

- …

Unused, Unpopular, and Limbo APIs

API usage varies widely, with some being very prolific and others only being used by a small handful of extensions. Given the resources available to the extensions team and the maintenance cost of APIs, any APIs that are sufficiently unpopular should likely be removed. This allows the team to execute more efficiently, as well as keeping the platform lean. Additionally, a number of APIs have been kept in limbo for long periods of time, such as the `declarativeWebRequest` API. These should be evaluated and either plan to launch, or removed.

Extension API usage data can be found here (internal only).

APIs with exceedingly low usage that should be removed:

- …

APIs stuck in limbo to be removed:

- `declarativeWebRequest` (obsolete with `declarativeNetRequest`)

Miscellaneous API Changes

Some APIs need to be changed for security or performance reasons, though they may not be fully replaced or removed.

i18n.getMessage

i18n.getMessage is used to retrieve a localized message string. This is currently a synchronous API (from the extension's perspective). However, the first time messages are loaded, this involves a sync (!) IPC to the browser process to load the extension localization bundle on a background sequence and return the result to the renderer. This process can take a significant amount of time, since it involves IPC and disk IO. Since it is a synchronous API call, the renderer cannot continue doing work in this time.

This is already bad, but is made worse by the fact that this API is exposed to content scripts, which can prevent a page from loading. As such, extensions using this API during early-injected content scripts may be significantly slowing down page load.

The alternative to this API is still TBD.

Migration

We will need to gradually migrate extensions over to Manifest V3. In some cases, this may be as simple as incrementing the manifest version (if the extension was not using any APIs affected by this change). However, in many cases, this will involve developer work. As such, we will need to provide a migration period, incentives for migrating, and disincentives for remaining on manifest version 2. Migration details can be found in this doc (internal only for now).

Declined Changes

The following changes were discussed, but are not on the road map for Manifest V3.

Storage API

Storage on the web has come far. We currently provide a storage API to extensions, and have different storage areas including local, sync, and managed. It may be worth pursuing removing all of these but sync, and having extensions instead rely on web standard storage (such as Indexed DB).

This will not be pursued because there is no good alternative on the web for sync storage, which is an important feature to extensions. Given that, the benefit from deprecation is reduced and it is not worth the investment (or cost to developers to migrate) at this time.

Extension Messaging

Extension messaging allows interaction between different portions of an extension. Extensions can use this messaging to communicate between content scripts or extension pages. All of this messaging is a hand-rolled implementation.

ServiceWorkers provide messaging between the ServiceWorker and claimed clients. This would be sufficient for any communication between the service worker and claimed extension frames; however, it would not be sufficient for communication between frames or with the content script. The former may be used rarely enough (and with a workaround of bouncing a message through the ServiceWorker) that we could only support messaging between a worker and a content script.

ServiceWorker messaging has advantages over extension messaging, including being standardized, likely better supported (and thus more efficient and less buggy), and supporting Transferables.

This will not be pursued because we would continue requiring the messaging code from extension processes to content scripts. Given that, the benefit from deprecation is reduced and it is not worth the investment (or cost to developers to migrate) at this time.

Script Injection Main World Capabilities

Currently, extensions can inject scripts into web pages through content scripts or the tabs.executeScript() method. In both of these cases, the renderer injects the script in an isolated world - a separate v8::Context that is unique to the extension. This is important for two main reasons:

1. It prevents the web page from being able to access extension API methods that might be exposed to the content script (such as extension messaging, storage, etc).
2. It prevents collisions in JS. The extensions `window.foo` variable is not the same as the page's `window.foo` variable.

However, this restriction is designed to be one-way: the web page cannot enter the extension's isolated world. Going the other direction, and having the extension execute code in the main world, is trivial (and most easily accomplished by simply appending a `<script>` tag to the page). This can lead to pain points for web developers, since the advantage of the isolated world is gone, and the extension can mutate the web page's variables. In extreme cases, this could include doing things like mutating `Array.prototype` or other similar built-in functionality.

This type of mutation is bad for web developers (who have to deal with it) and bad for users (because developers have to find workarounds, which often come with performance costs, or don't find workarounds, and websites are broken).

Hypothetically, we could restrict an extension's capability to inject scripts into the main world of a web page, and require that all interaction happen in the isolated world, or else require a separate API to do so (thus increasing our ability to audit uses, as well as preventing accidental use). Removal would potentially break some valid use cases, but it is not clear how many.

However, the feasibility of this and the amount of work entailed are both unknown. There are a variety of edge cases that would need to be taken into account, and this would largely need to be a fool-proof change (or else extensions could work around it).

Likely, the extensions team will have neither the domain knowledge nor the resources to staff this project, and it would have to be pursued by the blink team if there is interest in doing it.