Filter

Chrome incompatibilities

In this article

The WebExtension APIs aim to provide compatibility across all the main browsers, so extensions should run on any browser with minimal changes.

However, there are significant differences between Chrome (and Chromium-based browsers), Firefox, and Safari. In particular:

- Support for WebExtension APIs differs across browsers. See Browser support for JavaScript APIs for details.
- Support for `manifest.json` keys differs across browsers. See the "Browser compatibility" section on the `manifest.json` page for more details.
- Extension API namespace:
  - **In Firefox and Safari:** Extension APIs are accessed under the `browser` namespace. The `chrome` namespace is also supported for compatibility with Chrome.
  - **In Chrome:** Extension APIs are accessed under the `chrome` namespace. (cf. Chrome bug 798169)
- Asynchronous APIs:
  - **In Firefox and Safari:** Asynchronous APIs are implemented using promises.
  - **In Chrome:** In Manifest V2, asynchronous APIs are implemented using callbacks. In Manifest V3, support is provided for promises on most appropriate methods. (cf. Chrome bug 328932) Callbacks are supported in Manifest V3 for backward compatibility.

The rest of this page details these and other incompatibilities.

JavaScript APIs

chrome.* and browser.* namespace

- **In Firefox and Safari:** The APIs are accessed using the `browser` namespace.

```
JS
browser.browserAction.setIcon({ path: "path/to/icon.png" });
```

- **In Chrome:** The APIs are accessed using the `chrome` namespace.

```
JS
chrome.browserAction.setIcon({ path: "path/to/icon.png" });
```

Callbacks and promises

- **In Firefox and Safari (all versions), and Chrome (starting from Manifest Version 3):** Asynchronous APIs use promises to return values.

```
JS

function logCookie(c) {
  console.log(c);
}

function logError(e) {
  console.error(e);
}

let setCookie = browser.cookies.set({
  url: "https://developer.mozilla.org/",
```

```
});
setCookie.then(logCookie, logError);
```

- **In Chrome:** In Manifest V2, asynchronous APIs use callbacks to return values and `runtime.lastError` to communicate errors. In Manifest V3, callbacks are supported for backward compatibility, along with support for promises on most appropriate methods.
  JS

```
function logCookie(c) {
  if (chrome.runtime.lastError) {
    console.error(chrome.runtime.lastError);
  } else {
    console.log(c);
  }
}

chrome.cookies.set({ url: "https://developer.mozilla.org/" }, logCookie);
```

Firefox supports both the chrome and browser namespaces

As a porting aid, the Firefox implementation of WebExtensions supports `chrome` using callbacks and `browser` using promises. This means that many Chrome extensions work in Firefox without changes.

> **Note:** The `browser` namespace is supported by Firefox and Safari. Chrome does not offer the `browser` namespace, until Chrome bug 798169 is resolved.

If you choose to write your extension to use `browser` and promises, Firefox provides a polyfill that should enable it to run in Chrome: https://github.com/mozilla/webextension-polyfill .

Partially supported APIs

The Browser support for JavaScript APIs page includes compatibility tables for all APIs that have any support in Firefox. Where there are caveats regarding support for an API method, property, type, or event, this is indicated in these tables with an asterisk "*". Selecting the asterisk expands the table to display a note explaining the caveat.

The tables are generated from compatibility data stored as JSON files in GitHub .

The rest of this section describes the main compatibility issues you may need to consider when building a cross-browser extension. Also, remember to check the browser compatibility tables, as they may contain additional compatibility information.

Notifications API

For `notifications.create()`, with `type "basic"`:

- **In Firefox**: `iconUrl` is optional.
- **In Chrome**: `iconUrl` is required.

When the user clicks on a notification:

- **In Firefox**: The notification is cleared immediately.
- **In Chrome**: This is not the case.

If you call `notifications.create()` more than once in rapid succession:

- **In Firefox**: The notifications may not display. Waiting to make subsequent calls within the `notifications.create()` callback function

is not a sufficient delay to prevent this.

Proxy API

Firefox and Chrome include a Proxy API. However, the design of these two APIs is incompatible.

- **In Firefox**: Proxies are set using the proxy.settings property or proxy.onRequest to provide ProxyInfo dynamically. See proxy for more information on the API.
- **In Chrome**: Proxy settings are defined in a `proxy.ProxyConfig` object. Depending on Chrome's proxy settings, the settings may contain `proxy.ProxyRules` or a `proxy.PacScript` . Proxies are set using the proxy.settings property. See chrome.proxy for more information on the API.

Tabs API

When using `tabs.executeScript()` or `tabs.insertCSS()` :

- **In Firefox**: Relative URLs passed are resolved relative to the current page URL.
- **In Chrome**: Relative URLs are resolved relative to the extension's base URL.

To work cross-browser, you can specify the path as an absolute URL, starting at the extension's root, like this:

```
/path/to/script.js
```

When calling `tabs.remove()` :

- **In Firefox**: The `tabs.remove()` promise is fulfilled after the `beforeunload` event.
- **In Chrome**: The callback does not wait for `beforeunload` .

WebRequest API

- **In Firefox:**
  - Requests can be redirected only if their original URL uses the `http:` or `https:` scheme.
  - The `activeTab` permission does not allow for intercepting network requests in the current tab. (See bug 1617479)
  - Events are not fired for system requests (for example, extension upgrades or search bar suggestions).
    - **From Firefox 57 onwards:** Firefox makes an exception for extensions that need to intercept `webRequest.onAuthRequired` for proxy authorization. See the documentation for `webRequest.onAuthRequired` .
  - If an extension wants to redirect a public (e.g., HTTPS) URL to an extension page, the extension's `manifest.json` file must contain a `web_accessible_resources` key with the URL of the extension page.

    > **Note:** *Any* website may link or redirect to that URL, and extensions should treat any input (POST data, for example) as if it came from an untrusted source, as a normal web page should.

  - Some of the `browser.webRequest.*` APIs allow for returning Promises that resolves `webRequest.BlockingResponse` asynchronously.
- **In Chrome:** Only `webRequest.onAuthRequired` supports asynchronous `webRequest.BlockingResponse` by supplying `'asyncBlocking'` , through a callback instead of a Promise.

Windows API

- **In Firefox:** `onFocusChanged` of the `windows` API triggers multiple times for a focus change.

Unsupported APIs

DeclarativeContent API

- **In Firefox:** Chrome's declarativeContent API is not implemented. In addition, Firefox will not support the `declarativeContent.RequestContentScript` API (which is rarely used and is unavailable in stable releases of Chrome).

Miscellaneous incompatibilities

URLs in CSS

- **In Firefox:** URLs in injected CSS files are resolved relative to *the CSS file*.
- **In Chrome:** URLs in injected CSS files are resolved relative to *the page they are injected into*.

Support for dialogs in background pages

- **In Firefox:** `alert()`, `confirm()`, and `prompt()` are not supported in background pages.

web_accessible_resources

- **In Firefox:** Resources are assigned a random UUID that changes for every instance of Firefox: `moz-extension://«random-UUID»/«path»`. This randomness can prevent you from doing things, such as adding your extension's URL to another domain's CSP policy.
- **In Chrome:** When a resource is listed in `web_accessible_resources`, it is accessible as `chrome-extension://«your-extension-id»/«path»`. The extension ID is fixed for an extension.

Manifest "key" property

- **In Firefox:** As Firefox uses random UUIDs for `web_accessible_resources`, this property is unsupported. Firefox extensions can fix their extension ID through the `browser_specific_settings.gecko.id` manifest key (see browser_specific_settings.gecko).
- **In Chrome:** When working with an unpacked extension, the manifest may include a `"key"` property to pin the extension ID across different machines. This is mainly useful when working with `web_accessible_resources`.

Content script HTTP(S) requests

- **In Firefox:** When a content script makes an HTTP(S) request, you *must* provide absolute URLs.
- **In Chrome:** When a content script makes a request (for example, using `fetch()`) to a relative URL (like `/api`), it is sent to `https://example.com/api`.

Content script environment

- **In Firefox:** The global scope of the content script environment is not strictly equal to `window` (Firefox bug 1208775). More specifically, the global scope (`globalThis`) is composed of standard JavaScript features as usual, plus `window` as the prototype of the global scope. Most DOM APIs are inherited from the page through `window`, through Xray vision to shield the content script from modifications by the web page. A content script may encounter JavaScript objects from its global scope or Xray-wrapped versions from the web page.
- **In Chrome:** The global scope is `window`, and the available DOM APIs are generally independent of the web page (other than sharing the underlying DOM). Content scripts cannot directly access JavaScript objects from the web page.

Executing code in a web page from content script

- **In Firefox:** `eval` runs code in the context of the content script and `window.eval` runs code in the context of the page. See Using `eval` in content scripts.
- **In Chrome:** `eval` and `window.eval` always runs code in the context of the content script, not in the context of the page.

Sharing variables between content scripts

- **In Firefox:** You cannot share variables between content scripts by assigning them to `this.{variableName}` in one script and then attempting to access them using `window.{variableName}` in another. This is a limitation created by the sandbox environment in

Firefox. This limitation may be removed; see Firefox bug 1208775 .

Content script lifecycle during navigation

- **In Firefox:** Content scripts remain injected in a web page after the user has navigated away. However, window object properties are destroyed. For example, if a content script sets `window.prop1 = "prop"` and the user then navigates away and returns to the page `window.prop1` is undefined. This issue is tracked in Firefox bug 1525400 . To mimic the behavior of Chrome, listen for the pageshow and pagehide events. Then simulate the injection or destruction of the content script.
- **In Chrome:** Content scripts are destroyed when the user navigates away from a web page. If the user clicks the back button to return to the page through history, the content script is injected into the web page.

"per-tab" zoom behavior

- **In Firefox:** The zoom level persists across page loads and navigation within the tab.
- **In Chrome:** Zoom changes are reset on navigation; navigating a tab always loads pages with their per-origin zoom factors.

See `tabs.ZoomSettingsScope` .

manifest.json keys

The main `manifest.json` page includes a table describing browser support for `manifest.json` keys. Where there are caveats around support for a given key, this is indicated in the table with an asterisk "*". Selecting the asterisk expands the table to display a note explaining the caveat.

The tables are generated from compatibility data stored as JSON files in GitHub .

Native messaging

Connection-based messaging arguments

On Linux and Mac: Chrome passes one argument to the native app, which is the origin of the extension that started it, in the form of `chrome-extension://«extensionID/»` (trailing slash required). This enables the app to identify the extension.

On Windows: Chrome passes two arguments:

1. The origin of the extension
2. A handle to the Chrome native window that started the app

allowed_extensions

- **In Firefox:** The manifest key is called `allowed_extensions` .
- **In Chrome:** The manifest key is called `allowed_origins` .

App manifest location

- **In Chrome:** The app manifest is expected in a different place. See Native messaging host location in the Chrome docs.

App persistence

- **In Firefox:** When a native messaging connection is closed, Firefox kills the subprocesses if they do not break away. On Windows, the browser puts the native application's process into a Job object and kills the job. Suppose the native application launches other processes and wants them to remain open after the native application is killed. In that case, the native application must use `CreateProcess` , instead of `ShellExecute` , to launch the additional process with the `CREATE_BREAKAWAY_FROM_JOB` flag.

Data cloning algorithm

Some extension APIs allow an extension to send data from one part of the extension to another, such as `runtime.sendMessage()`, `tabs.sendMessage()`, `runtime.onMessage`, the `postMessage()` method of `runtime.port`, and `tabs.executeScript()`.

- **In Firefox:** The Structured clone algorithm is used.
- **In Chrome:** The JSON serialization algorithm is used. It may switch to structured cloning in the future (issue 248548).

The Structured clone algorithm supports more types than the JSON serialization algorithm. A notable exception are (DOM) objects with a `toJSON` method. DOM objects are not cloneable nor JSON-serializable by default, but with a `toJSON()` method, these can be JSON-serialized (but still not cloned with the structured cloning algorithm). Examples of JSON-serializable objects that are not structured cloneable include instances of `URL` and `PerformanceEntry`.

Extensions that rely on the `toJSON()` method of the JSON serialization algorithm can use `JSON.stringify()` followed by `JSON.parse()` to ensure that a message can be exchanged because a parsed JSON value is always structurally cloneable.