Filter

Content scripts

A content script is a part of your extension that runs in the context of a particular web page (as opposed to background scripts which are part of the extension, or scripts which are part of the website itself, such as those loaded using the `<script>` element).

Background scripts can access all the WebExtension JavaScript APIs, but they can't directly access the content of web pages. So if your extension needs to do that, you need content scripts.

Just like the scripts loaded by normal web pages, content scripts can read and modify the content of their pages using the standard DOM APIs. However, they can only do this when host permissions to the web page's origin have been granted.

Content scripts can only access a small subset of the WebExtension APIs, but they can communicate with background scripts using a messaging system, and thereby indirectly access the WebExtension APIs.

Loading content scripts

You can load a content script into a web page:

1. At install time, into pages that match URL patterns.
   - Using the `content_scripts` key in your `manifest.json`, you can ask the browser to load a content script whenever the browser loads a page whose URL matches a given pattern.
2. At runtime, into pages that match URL patterns.
   - Using `scripting.registerContentScripts()` or (only in Manifest V2 in Firefox) `contentScripts`, you can ask the browser to load a content script whenever the browser loads a page whose URL matches a given pattern. (This is similar to method 1, *except* that you can add and remove content scripts at runtime.)
3. At runtime, into specific tabs.
   - Using `scripting.executeScript()` or (in Manifest V2 only) `tabs.executeScript()`, you can load a content script into a specific tab whenever you want. (For example, in response to the user clicking on a browser action.)

There is only one global scope *per frame, per extension*. This means that variables from one content script can directly be accessed by another content script, regardless of how the content script was loaded.

Using methods (1) and (2), you can only load scripts into pages whose URLs can be represented using a match pattern.

Using method (3), you can also load scripts into pages packaged with your extension, but you can't load scripts into privileged browser pages (like "`about:debugging`" or "`about:addons`").

> **Note:** Dynamic JS module imports are now working in content scripts. For more details, see Firefox bug 1536094 . Only URLs with the *moz-extension* scheme are allowed, which excludes data URLs (Firefox bug 1587336).

Permissions, restrictions, and limitations

Registered content scripts are only executed if the extension is granted host permissions for the domain.

To inject scripts programmatically, the extension needs either the `activeTab` permission or host permissions. The `scripting` permission is required to use methods from the `scripting` API.

Starting with Manifest V3, host permissions are not automatically granted at install time. Users may opt in or out of host permissions after installing the extension.

Restricted domains

Both host permissions and the `activeTab` permission have exceptions for some domains. Content scripts are blocked from executing on these domains, for example, to protect the user from an extension escalating privileges through special pages.

In Firefox, this includes the following domains:

- accounts-static.cdn.mozilla.net
- accounts.firefox.com
- addons.cdn.mozilla.net
- addons.mozilla.org
- api.accounts.firefox.com
- content.cdn.mozilla.net
- discovery.addons.mozilla.org
- install.mozilla.org
- oauth.accounts.firefox.com
- profile.accounts.firefox.com
- support.mozilla.org
- sync.services.mozilla.com

Other browsers have similar restrictions over the websites extensions can be installed from. For example, access to chrome.google.com is restricted in Chrome.

> **Note:** Because these restrictions include addons.mozilla.org, users who try to use your extension immediately after installation may find that it doesn't work. To avoid this, you should add an appropriate warning or an onboarding page to move users away from `addons.mozilla.org`.

The set of domains can be restricted further through enterprise policies: Firefox recognizes the `restricted_domains` policy as documented at ExtensionSettings in mozilla/policy-templates . Chrome's `runtime_blocked_hosts` policy is documented at Configure ExtensionSettings policy .

Limitations

Whole tabs or frames may be loaded using `data:` URI, `Blob` objects, and other similar techniques. Support of content scripts injection into such special documents varies across browsers, see the Firefox bug #1411641 comment 41 for some details.

Content script environment

DOM access

Content scripts can access and modify the page's DOM, just like normal page scripts can. They can also see any changes that were made to the

DOM by page scripts.

However, content scripts get a "clean" view of the DOM. This means:

- Content scripts cannot see JavaScript variables defined by page scripts.
- If a page script redefines a built-in DOM property, the content script sees the original version of the property, not the redefined version.

As noted at "Content script environment" at Chrome incompatibilities, the behavior differs across browsers:

- In Firefox, this behavior is called Xray vision. Content scripts may encounter JavaScript objects from its own global scope or Xray-wrapped versions from the web page.
- In Chrome this behavior is enforced through an isolated world , which uses a fundamentally different approach.

Consider a web page like this:

HTML

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  </head>

  <body>
    <script src="page-scripts/page-script.js"></script>
  </body>
</html>
```

The script `page-script.js` does this:

JS

```
// page-script.js

// add a new element to the DOM
let p = document.createElement("p");
p.textContent = "This paragraph was added by a page script.";
p.setAttribute("id", "page-script-para");
document.body.appendChild(p);

// define a new property on the window
window.foo = "This global variable was added by a page script";

// redefine the built-in window.confirm() function
window.confirm = () => {
  alert("The page script has also redefined 'confirm'");
};
```

Now an extension injects a content script into the page:

JS

```
// content-script.js

// can access and modify the DOM
let pageScriptPara = document.getElementById("page-script-para");
pageScriptPara.style.backgroundColor = "blue";

// can't see properties added by page-script.js
console.log(window.foo); // undefined

// sees the original form of redefined properties
window.confirm("Are you sure?"); // calls the original window.confirm()
```

The same is true in reverse; page scripts cannot see JavaScript properties added by content scripts.

This means that content scripts can rely on DOM properties behaving predictably, without worrying about its variables clashing with variables from the page script.

One practical consequence of this behavior is that a content script doesn't have access to any JavaScript libraries loaded by the page. So, for example, if the page includes jQuery, the content script can't see it.

If a content script needs to use a JavaScript library, then the library itself should be injected as a content script *alongside* the content script that wants to use it:

JSON

```
"content_scripts": [
  {
    "matches": ["*://*.mozilla.org/*"],
    "js": ["jquery.js", "content-script.js"]
  }
]
```

Note: Firefox *does* provide some APIs that enable content scripts to access JavaScript objects created by page scripts, and to expose their own JavaScript objects to page scripts.

See Sharing objects with page scripts for more details.

WebExtension APIs

In addition to the standard DOM APIs, content scripts can use the following WebExtension APIs:

From `extension` :

- `getURL()`
- `inIncognitoContext`

From `runtime` :

- `connect()`
- `getManifest()`
- `getURL()`
- `onConnect`
- `onMessage`
- `sendMessage()`

From `i18n` :

- `getMessage()`
- `getAcceptLanguages()`
- `getUILanguage()`
- `detectLanguage()`

From `menus` :

- `getTargetElement`

Everything from:

- `storage`

XHR and Fetch

Content scripts can make requests using the normal `window.XMLHttpRequest` and `window.fetch()` APIs.

> **Note:** In Firefox in Manifest V2, content script requests (for example, using `fetch()`) happen in the context of an extension, so you must provide an absolute URL to reference page content.
>
> In Chrome and Firefox in Manifest V3, these requests happen in context of the page, so they are made to a relative URL. For example, `/api` is sent to `https://«current page URL»/api`.

Content scripts get the same cross-domain privileges as the rest of the extension: so if the extension has requested cross-domain access for a domain using the `permissions` key in `manifest.json`, then its content scripts get access that domain as well.

> **Note:** When using Manifest V3, content scripts can perform cross-origin requests when the destination server opts in using CORS; however, host permissions don't work in content scripts, but they still do in regular extension pages.

This is accomplished by exposing more privileged XHR and fetch instances in the content script, which has the side effect of not setting the `Origin` and `Referer` headers like a request from the page itself would; this is often preferable to prevent the request from revealing its cross-origin nature.

> **Note:** In Firefox in Manifest V2, extensions that need to perform requests that behave as if they were sent by the content itself can use `content.XMLHttpRequest` and `content.fetch()` instead.
>
> For cross-browser extensions, the presence of these methods must be feature-detected.
>
> This is not possible in Manifest V3, as `content.XMLHttpRequest` and `content.fetch()` are not available.

> **Note:** In Chrome, starting with version 73, and Firefox, starting with version 101 when using Manifest V3, content scripts are subject to the same CORS policy as the page they are running within. Only backend scripts have elevated cross-domain privileges. See Changes to Cross-Origin Requests in Chrome Extension Content Scripts .

Communicating with background scripts

Although content scripts can't directly use most of the WebExtension APIs, they can communicate with the extension's background scripts using the messaging APIs, and can therefore indirectly access all the same APIs that the background scripts can.

There are two basic patterns for communicating between the background scripts and content scripts:

- You can send **one-off messages** (with an optional response).
- You can set up a **longer-lived connection between the two sides**, and use that connection to exchange messages.

One-off messages

To send one-off messages, with an optional response, you can use the following APIs:

	In content script	In background script
Send a message	`browser.runtime.sendMessage()`	`browser.tabs.sendMessage()`
Receive a message	`browser.runtime.onMessage`	`browser.runtime.onMessage`

For example, here's a content script that listens for click events in the web page.

If the click was on a link, it sends a message to the background page with the target URL:

JS

```js
// content-script.js

window.addEventListener("click", notifyExtension);

function notifyExtension(e) {
  if (e.target.tagName !== "A") {
    return;
  }
  browser.runtime.sendMessage({ url: e.target.href });
}
```

The background script listens for these messages and displays a notification using the `notifications` API:

JS

```js
// background-script.js

browser.runtime.onMessage.addListener(notify);

function notify(message) {
  browser.notifications.create({
    type: "basic",
    iconUrl: browser.extension.getURL("link.png"),
    title: "You clicked a link!",
    message: message.url,
  });
}
```

(This example code is lightly adapted from the notify-link-clicks-i18n example on GitHub.)

Connection-based messaging

Sending one-off messages can get cumbersome if you are exchanging a lot of messages between a background script and a content script. So an alternative pattern is to establish a longer-lived connection between the two contexts, and use this connection to exchange messages.

Both sides have a `runtime.Port` object, which they can use to exchange messages.

To create the connection:

- One side listens for connections using `runtime.onConnect`
- The other side calls:
  - `tabs.connect()` (if connecting to a content script)
  - `runtime.connect()` (if connecting to a background script)

This returns a `runtime.Port` object.

- The `runtime.onConnect` listener gets passed its own `runtime.Port` object.

Once each side has a port, the two sides can:

- Send messages using `runtime.Port.postMessage()`
- Receive messages using `runtime.Port.onMessage()`

For example, as soon as it loads, the following content script:

- Connects to the background script
- Stores the `Port` in a variable `myPort`
- Listens for messages on `myPort` (and logs them)
- Uses `myPort` to sends messages to the background script when the user clicks the document

JS

```js
// content-script.js

let myPort = browser.runtime.connect({ name: "port-from-cs" });
myPort.postMessage({ greeting: "hello from content script" });

myPort.onMessage.addListener((m) => {
  console.log("In content script, received message from background script: ");
  console.log(m.greeting);
});

document.body.addEventListener("click", () => {
  myPort.postMessage({ greeting: "they clicked the page!" });
});
```

The corresponding background script:

- Listens for connection attempts from the content script
- When receiving a connection attempt:
  - Stores the port in a variable named `portFromCS`
  - Sends the content script a message using the port
  - Starts listening to messages received on the port, and logs them
- Sends messages to the content script, using `portFromCS`, when the user clicks the extension's browser action

JS

```js
// background-script.js

let portFromCS;

function connected(p) {
  portFromCS = p;
  portFromCS.postMessage({ greeting: "hi there content script!" });
  portFromCS.onMessage.addListener((m) => {
    portFromCS.postMessage({
      greeting: `In background script, received message from content script: ${m.greeting}`,
    });
  });
}

browser.runtime.onConnect.addListener(connected);

browser.browserAction.onClicked.addListener(() => {
  portFromCS.postMessage({ greeting: "they clicked the button!" });
});
```

Multiple content scripts

If you have multiple content scripts communicating at the same time, you might want to store connections to them in an array.

JS

```
// background-script.js

let ports = [];

function connected(p) {
  ports[p.sender.tab.id] = p;
  // …
}

browser.runtime.onConnect.addListener(connected);

browser.browserAction.onClicked.addListener(() => {
  ports.forEach((p) => {
    p.postMessage({ greeting: "they clicked the button!" });
  });
});
```

Choosing between one-off messages and connection-based messaging

The choice between one-off and connection-based messaging depends on how your extension expects to make use of messaging.

The recommended best practices are:

- **Use one-off messages when…**
  - Only one response is expected to a message.
  - A small number of scripts listen to receive messages (`runtime.onMessage` calls).
- **Use connection-based messaging when…**
  - Scripts engage in sessions where multiple messages are exchanged.
  - The extension needs to know about task progress or if a task is interrupted, or wants to interrupt a task initiated using messaging.

Communicating with the web page

By default, content scripts don't get access to the objects created by page scripts. However, they can communicate with page scripts using the DOM `window.postMessage` and `window.addEventListener` APIs.

For example:

JS

```
// page-script.js

let messenger = document.getElementById("from-page-script");

messenger.addEventListener("click", messageContentScript);

function messageContentScript() {
  window.postMessage(
    {
      direction: "from-page-script",
      message: "Message from the page",
    },
    "*",
  );
}
```

JS

```
// content-script.js

window.addEventListener("message", (event) => {
  if (
    event.source === window &&
    event?.data?.direction === "from-page-script"
  ) {
    alert(`Content script received message: "${event.data.message}"`);
  }
});
```

For a complete working example of this, visit the demo page on GitHub and follow the instructions.

> **Warning:** Be very careful when interacting with untrusted web content in this manner! Extensions are privileged code which can have powerful capabilities and hostile web pages can easily trick them into accessing those capabilities.
>
> To give a trivial example, suppose the content script code that receives the message does something like this:
>
> JS
>
> ```
> // content-script.js
>
> window.addEventListener("message", (event) => {
>   if (
>     event.source === window &&
>     event?.data?.direction === "from-page-script"
>   ) {
>     eval(event.data.message);
>   }
> });
> ```
>
> Now the page script can run any code with all the privileges of the content script.

Using `eval()` in content scripts

> **Note:** `eval()` not available in Manifest V3.

In Chrome

`eval` always runs code in the context of the **content script**, not in the context of the page.

In Firefox

If you call `eval()`, it runs code in the context of the **content script**.

If you call `window.eval()`, it runs code in the context of the **page**.

For example, consider a content script like this:

JS

```
// content-script.js

window.eval("window.x = 1;");
eval("window.y = 2");

console.log(`In content script, window.x: ${window.x}`);
console.log(`In content script, window.y: ${window.y}`);
```

```
window.postMessage(
  {
    message: "check",
  },
  "*",
);
```

This code just creates some variables `x` and `y` using `window.eval()` and `eval()`, logs their values, and then messages the page.

On receiving the message, the page script logs the same variables:

JS

```
window.addEventListener("message", (event) => {
  if (event.source === window && event.data && event.data.message === "check") {
    console.log(`In page script, window.x: ${window.x}`);
    console.log(`In page script, window.y: ${window.y}`);
  }
});
```

In Chrome, this produces output like this:

```
In content script, window.x: 1
In content script, window.y: 2
In page script, window.x: undefined
In page script, window.y: undefined
```

In Firefox, this produces output like this:

```
In content script, window.x: undefined
In content script, window.y: 2
In page script, window.x: 1
In page script, window.y: undefined
```

The same applies to `setTimeout()`, `setInterval()`, and `Function()`.

> **Warning:** Be very careful when running code in the context of the page!
>
> The page's environment is controlled by potentially malicious web pages, which can redefine objects you interact with to behave in unexpected ways:
>
> JS
>
> ```
> // page.js redefines console.log
>
> let original = console.log;
>
> console.log = () => {
>   original(true);
> };
> ```
>
> JS
>
> ```
> // content-script.js calls the redefined version
>
> window.eval("console.log(false)");
> ```