Background scripts

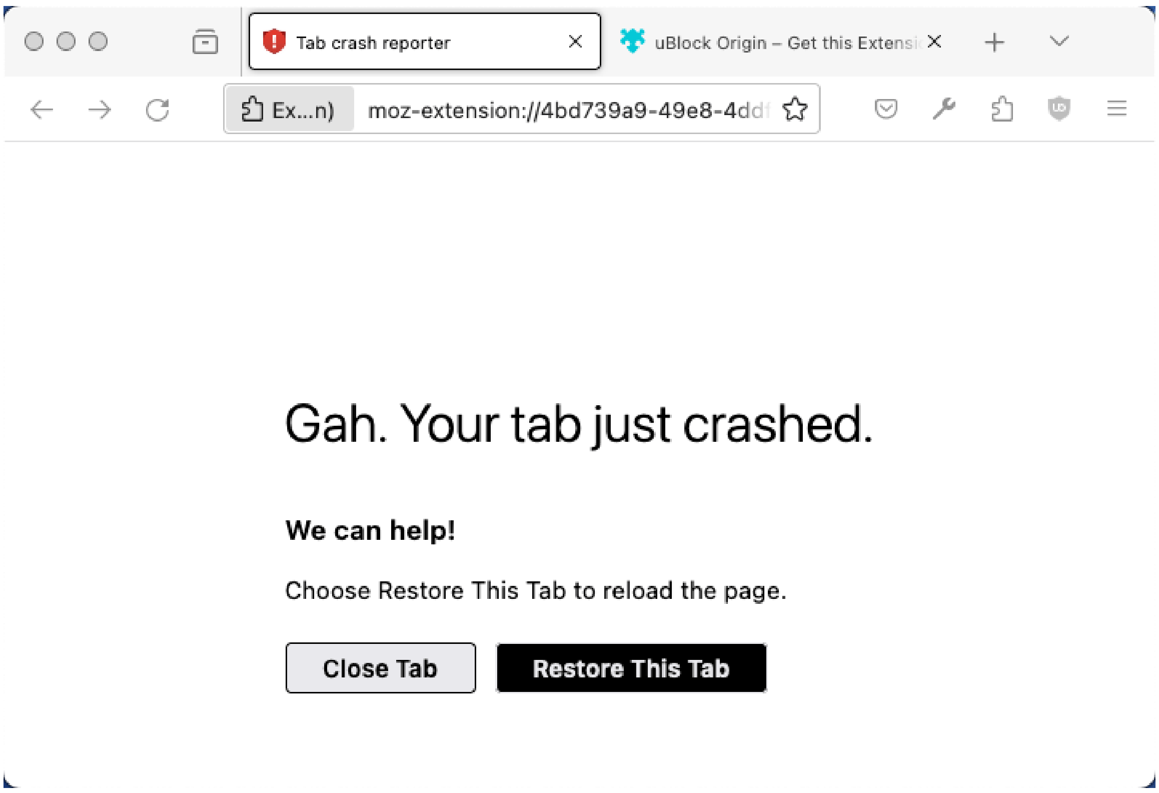
Background scripts or a background page enable you to monitor and react to events in the browser, such as navigating to a new page, removing a bookmark, or closing a tab.

Background scripts or a page are:

- Persistent – loaded when the extension starts and unloaded when the extension is disabled or uninstalled.
- Non-persistent (which are also known as event pages) – loaded only when needed to respond to an event and unloaded when they become idle. However, a background page does not unload until all visible views and message ports are closed. Opening a view does not cause the background page to load but does prevent it from closing.

Note: In Firefox, if the extension process crashes:

- persistent background scripts running at the time of the crash are reloaded automatically.
- non-persistent background scripts (also known as "Event Pages") running at the time of the crash are not reloaded. However, they are restarted automatically when Firefox calls one of their WebExtensions API events listeners.
- extension pages loaded in tabs at the time of the crash are not automatically restored. A warning message in each tab informs the user the page has crashed and enables the user to close or restore the tab.


Tab crash reporter
uBlock Origin – Get this Extensi
Ex...n)
moz-extension://4bd739a9-49e8-4dd
Close Tab
Restore This Tab

Gah. Your tab just crashed.

We can help!

Choose Restore This Tab to reload the page.

You can test this condition by opening a new tab and navigating to `about:crashextensions`, which silently triggers a crash of the extension process.

In Manifest V2, background scripts or a page can be persistent or non-persistent. Non-persistent background scripts are recommended as they reduce the resource cost of your extension. In Manifest V3, only non-persistent background scripts or a page are supported.

If you have persistent background scripts or a page in Manifest V2 and want to prepare your extension for migration to Manifest V3, [Convert to non-persistent](#) provides advice on transitioning the scripts or page to the non-persistent model.

In this article

Background script environment

DOM APIs

Background scripts run in the context of a special page called a background page. This gives them a `window` global, along with all the standard DOM APIs provided by that object.

Warning: In Firefox, background pages do not support the use of `alert()`, `confirm()`, or `prompt()`.

WebExtension APIs

Background scripts can use any WebExtension APIs, as long as their extension has the necessary permissions.

Cross-origin access

Background scripts can make XHR requests to hosts they have host permissions for.

Web content

Background scripts do not get direct access to web pages. However, they can load content scripts into web pages and communicate with these content scripts using a message-passing API.

Content security policy

Background scripts are restricted from certain potentially dangerous operations, such as the use of `eval()`, through a Content Security Policy.

See Content Security Policy for more details.

Implementing background scripts

This section describes how to implement a non-persistent background script.

Specify the background scripts

In your extension, you include a background script or scripts, if you need them, using the `"background"` key in `manifest.json`. For Manifest V2 extensions, the `persistent` property must be `false` to create a non-persistent script. It can be omitted for Manifest V3 extensions or must be set to `false`, as script are always non-persistent in Manifest V3. Including `"type": "module"` loads the background scripts as ES modules.

JSON

```
"background": {
  "scripts": ["background-script.js"],
  "persistent": false,
  "type": "module"
}
```

These scripts execute in the extension's background page, so they run in the same context, like scripts loaded into a web page.

Filter

Browser extensions

▶ Getting started

▼ Concepts

JavaScript APIs

Content scripts

Background scripts

Match patterns

Work with files

Internationalization

Content Security Policy

Native messaging

Differences between API implementations

Chrome incompatibilities

▶ User interface

▶ How to

▶ JavaScript APIs

▶ Manifest keys

▶ Extension Workshop

Contact us

▶ Channels

However, if you need certain content in the background page, you can specify one. You then specify your script from the page rather than using the `"scripts"` property. Before the introduction of the `"type"` property to the `"background"` key, this was the only option to include ES modules. You specify a background page like this:

- manifest.json

  JSON

  ```
  "background": {
    "page": "background-page.html",
    "persistent": false
  }
  ```

- background-page.html

  HTML

  ```
  <!doctype html>
  <html lang="en">
    <head>
      <meta charset="utf-8" />
      <script type="module" src="background-script.js"></script>
    </head>
  </html>
  ```

You cannot specify background scripts and a background page.

Initialize the extension

Listen to `runtime.onInstalled` to initialize an extension on installation. Use this event to set a state or for one-time initialization.

For extensions with event pages, this is where stateful APIs, such as a context menu created using `menus.create` , should be used. This is because stateful APIs don't need to be run each time the event page reloads; they only need to run when the extension is installed.

JS

```
browser.runtime.onInstalled.addListener(() => {
  browser.contextMenus.create({
    id: "sampleContextMenu",
    title: "Sample Context Menu",
    contexts: ["selection"],
  });
});
```

Add listeners

Structure background scripts around events the extension depends on. Defining relevant events enables background scripts to lie dormant until those events are fired and prevents the extension from missing essential triggers.

Listeners must be registered synchronously from the start of the page.

JS

```
browser.runtime.onInstalled.addListener(() => {
  browser.contextMenus.create({
    id: "sampleContextMenu",
    title: "Sample Context Menu",
    contexts: ["selection"],
  });
});

// This will run when a bookmark is created.
browser.bookmarks.onCreated.addListener(() => {
  // do something
});
```

Do not register listeners asynchronously, as they will not be properly triggered. So, rather than:

JS

```
window.onload = () => {
  // WARNING! This event is not persisted, and will not restart the event page.
  browser.bookmarks.onCreated.addListener(() => {
    // do something
  });
};
```

Do this:

JS

```
browser.tabs.onUpdated.addListener(() => {
  // This event is run in the top level scope of the event page, and will persist, allowing
  // it to restart the event page if necessary.
});
```

Extensions can remove listeners from their background scripts by calling `removeListener`, such as with [`runtime.onMessage`](runtime.onMessage) `removeListener`. If all listeners for an event are removed, the browser no longer loads the extension's background script for that event.

JS

```
browser.runtime.onMessage.addListener(
  function messageListener(message, sender, sendResponse) {
    browser.runtime.onMessage.removeListener(messageListener);
  },
);
```

Filter events

Use APIs that support event filters to restrict listeners to the cases the extension cares about. If an extension is listening for [`tabs.onUpdated`](tabs.onUpdated), use the [`webNavigation.onCompleted`](webNavigation.onCompleted) event with filters instead, as the tabs API does not support filters.

JS

```
browser.webNavigation.onCompleted.addListener(
  () => {
    console.log("This is my favorite website!");
  },
  { url: [{ urlMatches: "https://www.mozilla.org/" }] },
);
```

React to listeners

Listeners exist to trigger functionality once an event has fired. To react to an event, structure the desired reaction inside the listener event.

When responding to events in the context of a specific tab or frame, use the `tabId` and `frameId` from the event details instead of relying on the "current tab". Specifying the target ensures your extension does not invoke an extension API on the wrong target when the "current tab" changes while waking the event page.

For example, [`runtime.onMessage`](runtime.onMessage) can respond to [`runtime.sendMessage`](runtime.sendMessage) calls as follows:

JS

```
browser.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.data === "setAlarm") {
    browser.alarms.create({ delayInMinutes: 5 });
  } else if (message.data === "runLogic") {
    browser.scripting.executeScript({
      target: {
        tabId: sender.tab.id,
        frameIds: [sender.frameId],
      },
      files: ["logic.js"],
    });
  } else if (message.data === "changeColor") {
    browser.scripting.executeScript({
      target: {
        tabId: sender.tab.id,
        frameIds: [sender.frameId],
      },
      func: () => {
        document.body.style.backgroundColor = "orange";
      },
    });
  }
});
```

Unload background scripts

Data should be persisted periodically to not lose important information if an extension crashes without receiving `runtime.onSuspend`. Use the storage API to assist with this.

JS

```
// Or storage.session if the variable does not need to persist pass browser shutdown.
browser.storage.local.set({ variable: variableInformation });
```

Message ports cannot prevent an event page from shutting down. If an extension uses message passing, the ports are closed when the event page idles. Listening to the `runtime.Port` `onDisconnect` lets you discover when open ports are closing, however the listener is under the same time constraints as `runtime.onSuspend`.

JS

```
browser.runtime.onConnect.addListener((port) => {
  port.onMessage.addListener((message) => {
    if (message === "hello") {
      let response = { greeting: "welcome!" };
      port.postMessage(response);
    } else if (message === "goodbye") {
      console.log("Disconnecting port from this end");
      port.disconnect();
    }
  });
  port.onDisconnect.addListener(() => {
    console.log("Port was disconnected from the other end");
  });
});
```

Background scripts unload after a few seconds of inactivity. However, if during the suspension of a background script another event wakes the background script, `runtime.onSuspendCanceled` is called and the background script continues running. If any cleanup is required, listen to `runtime.onSuspend`.

JS

```
browser.runtime.onSuspend.addListener(() => {
  console.log("Unloading.");
  browser.browserAction.setBadgeText({ text: "" });
});
```

However, persisting data should be preferred rather than relying on `runtime.onSuspend`. It doesn't allow for as much cleanup as may be needed and does not help in case of a crash.

Convert to non-persistent

If you've a persistent background script, this section provides instructions on converting it to the non-persistent model.

Update your manifest.json file

In your extension's `manifest.json` file, change the persistent property of [`"background"`] key to `false` for your script or page.

JSON

```
"background": {
  …,
  "persistent": false
}
```

Move event listeners

Listeners must be at the top-level to activate the background script if an event is triggered. Registered listeners may need to be restructured to the synchronous pattern and moved to the top-level.

JS

```
browser.runtime.onStartup.addListener(() => {
  // run startup function
});
```

Record state changes

Scripts now open and close as needed. So, do not rely on global variables.

JS

```
var count = 101;
browser.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message === "count") {
    ++count;
    sendResponse(count);
  }
});
```

Instead, use the storage API to set and return states and values:

- Use `storage.session` for in-memory storage that is cleared when the extension or browser shuts down. By default, `storage.session` is only available to extension contexts and not to content scripts.
- Use `storage.local` for a larger storage area that persists across browser and extension restarts.

JS

```
browser.runtime.onMessage.addListener(async (message, sender) => {
  if (message === "count") {
    let items = await browser.storage.session.get({ myStoredCount: 101 });
    let count = items.myStoredCount;
    ++count;
    await browser.storage.session.set({ myStoredCount: count });
    return count;
  }
});
```

The preceding example sends an asynchronous response using a promise, which is not supported in Chrome until Chrome bug 1185241 is resolved. A cross-browser alternative is to return true and use `sendResponse`.

JS

```
browser.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message === "count") {
    browser.storage.session.get({ myStoredCount: 101 }).then(async (items) => {
      let count = items.myStoredCount;
      ++count;
      await browser.storage.session.set({ myStoredCount: count });
      sendResponse(count);
    });
    return true;
  }
});
```

Change timers into alarms

DOM-based timers, such as `setTimeout()`, do not remain active after an event page has idled. Instead, use the `alarms` API if you need a timer to wake an event page.

JS

```
browser.alarms.create({ delayInMinutes: 3.0 });
```

Then add a listener.

JS

```
browser.alarms.onAlarm.addListener(() => {
  alert("Hello, world!");
});
```

Update calls for background script functions

Extensions commonly host their primary functionality in the background script. Some extensions access functions and variables defined in the background page through the `window` returned by `extension.getBackgroundPage`. The method returns `null` when:

- extension pages are isolated, such as extension pages in Private Browsing mode or container tabs.
- the background page is not running. This is uncommon with persistent background pages but very likely when using an Event Page, as an Event Page can be suspended.

Note: The recommended way to invoke functionality in the background script is to communicate with it through `runtime.sendMessage()` or `runtime.connect()`. The `getBackgroundPage()` methods discussed in this section cannot be used in a cross-browser extension, because Manifest Version 3 extensions in Chrome cannot use background or event pages.

If your extension requires a reference to the `window` of the background page, use `runtime.getBackgroundPage` to ensure the event page is running. If the call is optional (that is, only needed if the event page is alive) then use `extension.getBackgroundPage`.

JS

```
document.getElementById("target").addEventListener("click", async () => {
  let backgroundPage = browser.extension.getBackgroundPage();
  // Warning: backgroundPage is likely null.
  backgroundPage.backgroundFunction();
});
```

JS

```
document.getElementById("target").addEventListener("click", async () => {
  // runtime.getBackgroundPage() wakes up the event page if it was not running.
  let backgroundPage = await browser.runtime.getBackgroundPage();
  backgroundPage.backgroundFunction();
});
```