# Handle events with service workers

Tutorial that covers extension service worker concepts

## Overview

This tutorial provides an introduction to Chrome Extension service workers. As part of this tutorial, you will build an extension that allows users to quickly navigate to Chrome API reference pages using the omnibox. You will learn how to:

- Register your service worker and import modules.
- Debug your extension service worker.
- Manage state and handle events.
- Trigger periodic events.
- Communicate with content scripts.

## Before you start

This guide assumes that you have basic web development experience. We recommend reviewing Extensions 101 and Hello World for an introduction to extension development.

## Build the extension

Start by creating a new directory called `quick-api-reference` to hold the extension files, or download the source code from our GitHub samples repository.

### Step 1: Register the service worker

Create the manifest file in the root of the project and add the following code:

manifest.json:

```
{
  "manifest_version": 3,
  "name": "Open extension API reference",
  "version": "1.0.0",
  "icons": {
    "16": "images/icon-16.png",
    "128": "images/icon-128.png"
  },
  "background": {
    "service_worker": "service-worker.js",
  },
}
```

Extensions register their service worker in the manifest, which only takes a single JavaScript file. There's no need to call `navigator.serviceWorker.register()`, like you would in a web page.

Create an `images` folder then download the icons into it.

Check out the first steps of the Reading time tutorial to learn more about the extension's metadata and icons in the manifest.

## Step 2: Import multiple service worker modules

Our service worker implements two features. For better maintainability, we will implement each feature in a separate module. First, we need to declare the service worker as an ES Module in our manifest, which allows us to import modules in our service worker:

manifest.json:

```
{
 "background": {
    "service_worker": "service-worker.js",
    "type": "module"
  },
}
```

Create the `service-worker.js` file and import two modules:

```
import './sw-omnibox.js';
import './sw-tips.js';
```

Create these files and add a console log to each one.

sw-omnibox.js:

```
console.log("sw-omnibox.js")
```

sw-tips.js:

```
console.log("sw-tips.js")
```

See Importing scripts to learn about other ways to import multiple files in a service worker.

**Tip:** Remember to set `"type.module"` when using a modern module bundler framework, such as CRXjs Vite plugin.

## Optional: Debugging the service worker

I'll explain how to find the service worker logs and know when it has terminated. First, follow the instructions to Load an unpacked extension.

After 30 seconds you will see "service worker (inactive)" which means the service worker has terminated. Click the "service worker (inactive)" link to inspect it. The following animation shows this.

Did you notice that inspecting the service worker woke it up? Opening the service worker in the devtools will keep it active. To make sure that your extension behaves correctly when your service worker is terminated, remember to close DevTools.

Now, break the extension to learn where to locate errors. One way to do this is to delete ".js" from the `'./sw-omnibox.js'` import in the `service-worker.js` file. Chrome will be unable to register the service worker.

Go back to chrome://extensions and refresh the extension. You will see two errors:

```
Service worker registration failed. Status code: 3.

An unknown error occurred when fetching the script.
```

See Debugging extensions for more ways debug the extension service worker.

**Caution:** Don't forget to fix the filename before moving on.

## Step 4: Initialize the state

Chrome will shut down service workers if they are not needed. We use the `chrome.storage` API to persist state across service worker sessions. For storage access, we need to request permission in the manifest:

manifest.json:

```
{
  ...
  "permissions": ["storage"],
}
```

First, save the default suggestions to storage. We can initialize state when the extension is first installed by listening to the `runtime.onInstalled()` event:

sw-omnibox.js:

```
...
// Save default API suggestions
chrome.runtime.onInstalled.addListener(({ reason }) => {
  if (reason === 'install') {
    chrome.storage.local.set({
      apiSuggestions: ['tabs', 'storage', 'scripting']
    });
  }
});
```

Service workers don't have direct access to the window object and therefore cannot use `window.localStorage` to store values. Also, service workers are short-lived execution environments; they get terminated repeatedly throughout a user's browser session, which makes them incompatible with global variables. Instead, use `chrome.storage.local` which stores data on the local machine.

See Persist data rather than using global variables to learn about other storage options for extension service workers.

## Step 5: Register your events

All event listeners need to be statically registered in the global scope of the service worker. In other words, event listeners shouldn't be nested in async functions. This way Chrome can ensure that all event handlers are restored in case of a service worker reboot.

In this example, we are going to use the `chrome.omnibox` API, but first we must declare the omnibox keyword trigger in the manifest:

manifest.json:

```
{
  ...
  "minimum_chrome_version": "102",
  "omnibox": {
    "keyword": "api"
  },
}
```

**Key point:** The `"minimum_chrome_version"` reference explains how this key behaves when a user tries to install your extension but isn't using a compatible version of Chrome.

Now, register the omnibox event listeners at the top level of the script. When the user enters the omnibox keyword (`api`) in the address bar followed by tab or space, Chrome will display a list of suggestions based on the keywords in storage. The `onInputChanged()` event, which takes the current user input and a `suggestResult` object, is responsible for populating these suggestions.

sw-omnibox.js:

```
...
const URL_CHROME_EXTENSIONS_DOC =
  'https://developer.chrome.com/docs/extensions/reference/';
const NUMBER_OF_PREVIOUS_SEARCHES = 4;

// Display the suggestions after user starts typing
chrome.omnibox.onInputChanged.addListener(async (input, suggest) => {
  await chrome.omnibox.setDefaultSuggestion({
    description: 'Enter a Chrome API or choose from past searches'
  });
  const { apiSuggestions } = await chrome.storage.local.get('apiSuggestions');
  const suggestions = apiSuggestions.map((api) => {
    return { content: api, description: `Open chrome.${api} API` };
  });
  suggest(suggestions);
});
```

After the user selects a suggestion, `onInputEntered()` will open the corresponding Chrome API reference page.

sw-omnibox.js:

```
...
// Open the reference page of the chosen API
chrome.omnibox.onInputEntered.addListener((input) => {
  chrome.tabs.create({ url: URL_CHROME_EXTENSIONS_DOC + input });
  // Save the latest keyword
  updateHistory(input);
});
```

The `updateHistory()` function takes the omnibox input and saves it to `storage.local`. This way the most recent search term can be used later as an omnibox suggestion.

sw-omnibox.js:

```
...
async function updateHistory(input) {
  const { apiSuggestions } = await chrome.storage.local.get('apiSuggestions');
  apiSuggestions.unshift(input);
  apiSuggestions.splice(NUMBER_OF_PREVIOUS_SEARCHES);
  return chrome.storage.local.set({ apiSuggestions });
}
```

**Key point:** Extension service workers can use both web APIs and Chrome APIs, with a few exceptions. For more information, see Service Workers events.

## Step 6: Set up a recurring event

The `setTimeout()` or `setInterval()` methods are commonly used to perform delayed or periodic tasks. However, these APIs can fail because the scheduler will cancel the timers when the service worker is terminated. Instead, extensions can use the `chrome.alarms` API.

Start by requesting the `"alarms"` permission in the manifest. Additionally, to fetch the extension tips from a remote hosted location, you need to request host permission:

manifest.json:

```
{
  ...
  "permissions": ["storage", "alarms"],
  "permissions": ["storage"],
  "host_permissions": ["https://extension-tips.glitch.me/*"],
}
```

The extension will fetch all the tips, pick one at random and save it to storage. We will create an alarm that will be triggered once a day to update the tip. Alarms are not saved when you close Chrome. So we need to check if the alarm exists and create it if it doesn't.

sw-tips.js:

```
// Fetch tip & save in storage
const updateTip = async () => {
  const response = await fetch('https://extension-tips.glitch.me/tips.json');
  const tips = await response.json();
  const randomIndex = Math.floor(Math.random() * tips.length);
  return chrome.storage.local.set({ tip: tips[randomIndex] });
```

```
};

const ALARM_NAME = 'tip';

// Check if alarm exists to avoid resetting the timer.
// The alarm might be removed when the browser session restarts.
async function createAlarm() {
  const alarm = await chrome.alarms.get(ALARM_NAME);
  if (typeof alarm === 'undefined') {
    chrome.alarms.create(ALARM_NAME, {
      delayInMinutes: 1,
      periodInMinutes: 1440
    });
    updateTip();
  }
}

createAlarm();

// Update tip once a day
chrome.alarms.onAlarm.addListener(updateTip);
```

**Key point:** All Chrome API event listeners and methods restart the service worker's 30-second termination timer. For more information, see the Extension service worker lifecycle.

## Step 7: Communicate with other contexts

Extensions use content scripts to read and modify the content of the page. When a user visits a Chrome API reference page, the extension's content script will update the page with the tip of the day. It sends a message to request the tip of the day from the service worker.

Start by declaring the content script in the manifest and add the match pattern corresponding to the Chrome API reference documentation.

manifest.json:

```
{
  ...
  "content_scripts": [
    {
      "matches": ["https://developer.chrome.com/docs/extensions/reference/*"],
      "js": ["content.js"]
    }
  ]
}
```

Create a new content file. The following code sends a message to the service worker requesting the tip. Then, adds a button that will open a popover containing the extension tip. This code uses the new web platform Popover API.

content.js:

```
(async () => {
  // Sends a message to the service worker and receives a tip in response
  const { tip } = await chrome.runtime.sendMessage({ greeting: 'tip' });
```

```
  const nav = document.querySelector('.upper-tabs > nav');

  const tipWidget = createDomElement(`
    <button type="button" popovertarget="tip-popover" popovertargetaction="show"
style="padding: 0 12px; height: 36px;">
      <span style="display: block; font: var(--devsite-link-font,500 14px/20px
var(--devsite-primary-font-family));">Tip</span>
    </button>
  `);

  const popover = createDomElement(
    `<div id='tip-popover' popover style="margin: auto;">${tip}</div>`
  );

  document.body.append(popover);
  nav.append(tipWidget);
})();

function createDomElement(html) {
  const dom = new DOMParser().parseFromString(html, 'text/html');
  return dom.body.firstElementChild;
}
```

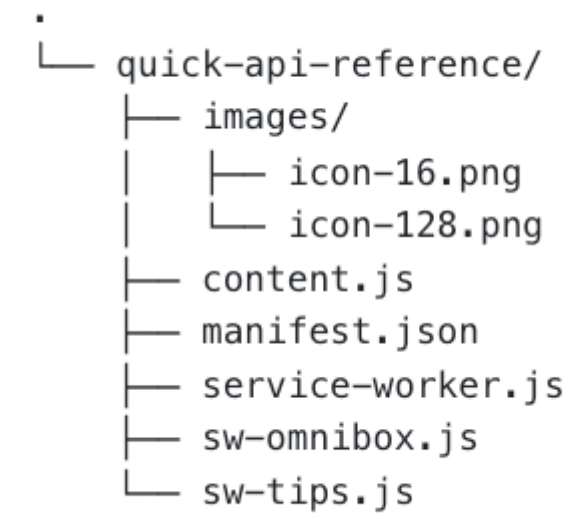
The final step is to add a message handler to our service worker that sends a reply to the content script with the daily tip.

sw-tips.js:

```
...
// Send tip to content script via messaging
chrome.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.greeting === 'tip') {
    chrome.storage.local.get('tip').then(sendResponse);
    return true;
  }
});
```

## Test that it works

Verify that the file structure of your project looks like the following:



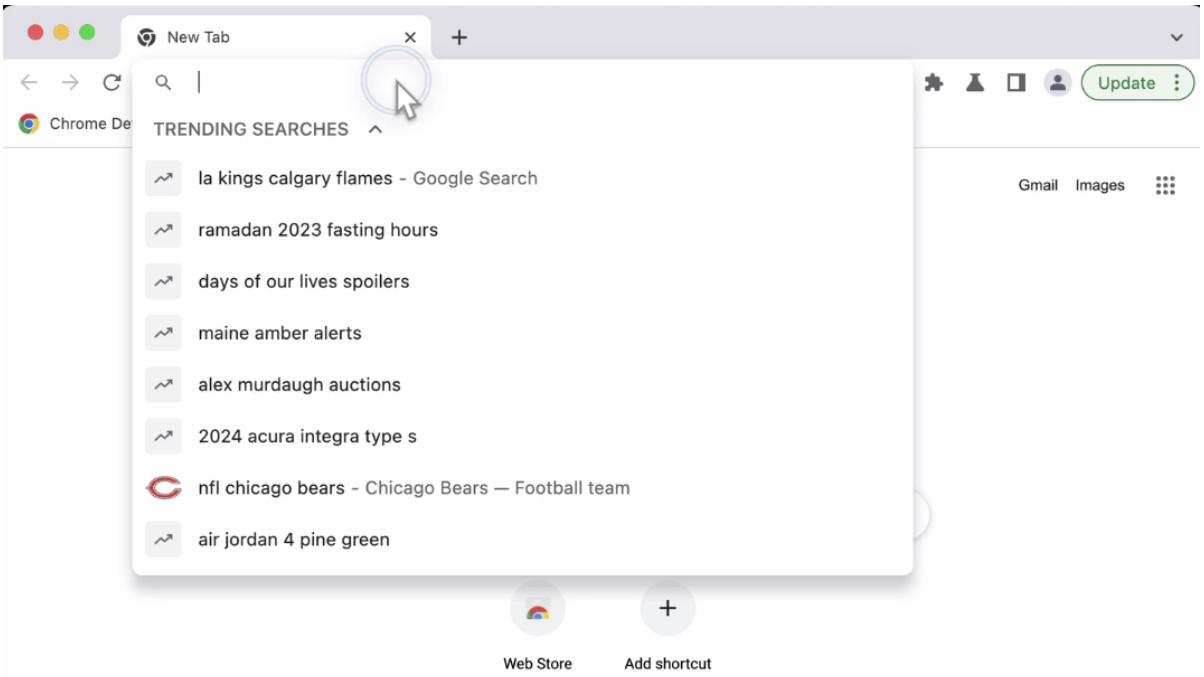

## Load your extension locally

To load an unpacked extension in developer mode, follow the steps in Hello world.

## Open a reference page

1. Enter the keyword "api" in the browser address bar.
2. Press "tab" or "space".
3. Enter the complete name of the API.
    - OR choose from a list of past searches
4. A new page will open to the Chrome API reference page.

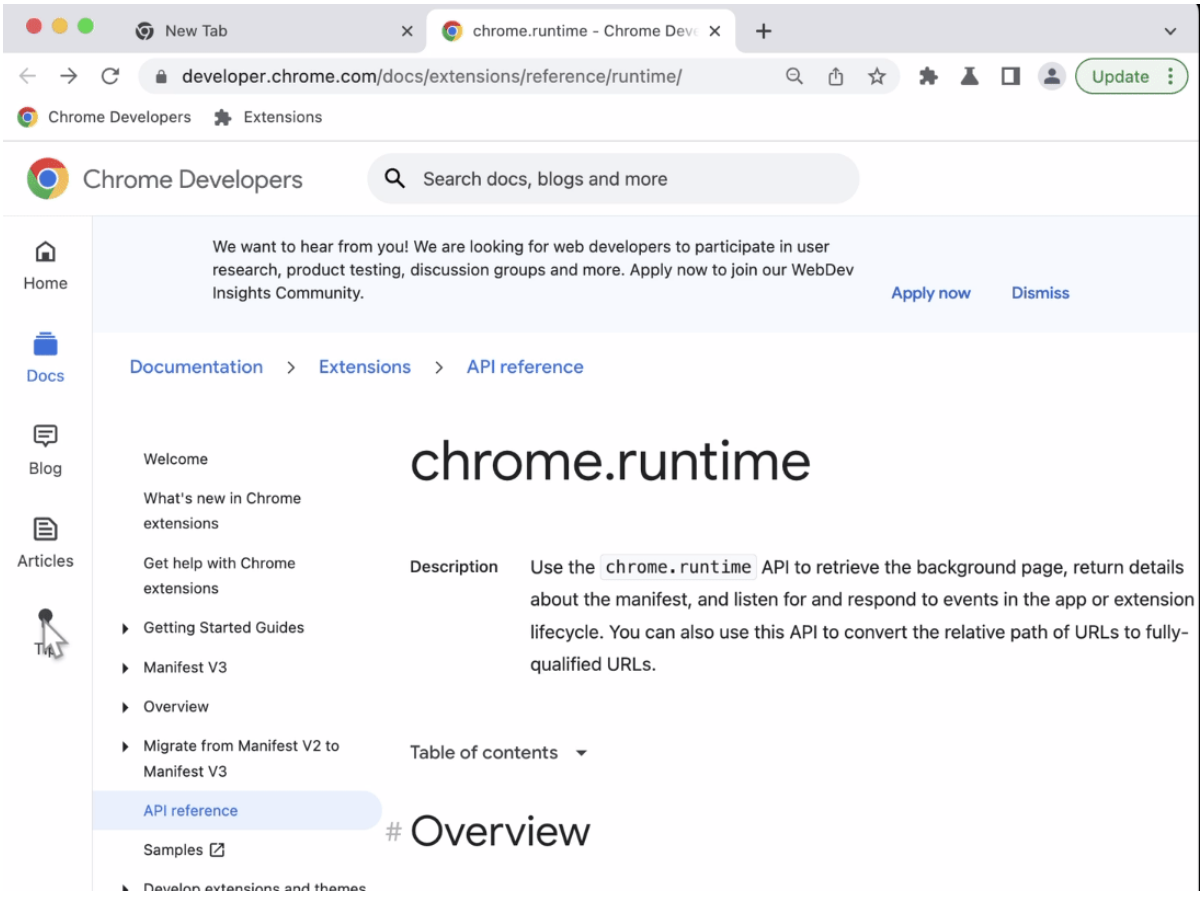
It should look like this:




Quick API extension opening the Runtime API.

## Open the tip of the day

Click the Tip button located on the navigation bar to open the extension tip.




Quick API extension opening the tip of the day.

**Note:** The Popover API was launched in Chrome 114.

## 🎯 Potential enhancements

Based on what you've learned today, try to accomplish any of the following:

- Explore another way to implement the omnibox suggestions.
- Create your own custom modal for displaying the extension tip.
- Open an additional page to the MDN's Web Extensions reference API pages.

# Keep building!

Congratulations on finishing this tutorial 🎉. Continue leveling up your skills by completing other beginner tutorials:

| Extension | What you will learn |
| --- | --- |
| Reading time | To insert an element on a specific set of pages automatically. |
| Tabs Manager | To create a popup that manages browser tabs. |
| Focus Mode | To run code on the current page after clicking on the extension action. |

# Continue exploring

To continue your extension service worker learning path, we recommend exploring the following articles:

- About extension service workers.
- The extension service worker lifecycle.
- Events in service workers