Workers overview

- How workers can improve your website {: #how-workers-can-improve-your-website }
- Web workers and service workers {: #web-workers-and-service-workers }
  - Similarities {: #similarities }
  - Differences {: #differences }
- Use cases {: #use-cases }
- Tools and libraries {: #tools-and-libraries }
  - Comlink {: #comlink }
  - Workbox {: #workbox }
- Next steps {: #next-steps }

This overview explains how web workers and service workers can improve the performance of your website, and when to use a web worker versus a service worker. Check out the rest of this series for specific patterns of window and service worker communication.

How workers can improve your website {: #how-workers-can-improve-your-website }

The browser uses a single thread (the main thread) to run all the JavaScript in a web page, as well as to perform tasks like rendering the page and performing garbage collection. Running excessive JavaScript code can block the main thread, delaying the browser from performing these tasks and leading to a poor user experience.

In iOS/Android application development, a common pattern to ensure that the app's main thread remains free to respond to user events is to offload operations to additional threads. In fact, in the latest versions of Android, blocking the main thread for too long leads to an app crash.

On the web, JavaScript was designed around the concept of a single thread, and lacks capabilities needed to implement a multithreading model like the one apps have, like shared memory.

Despite these limitations, a similar pattern can be achieved in the web by using workers to run scripts in background threads, allowing them to perform tasks without interfering with the main thread. Workers are an entire JavaScript scope running on a separate thread, without any shared memory.

In this post you'll learn about two different types of workers (web workers and service workers), their similarities and differences, and the most common patterns for using them in production websites.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/eN5kePr9U0aZMgCyekhJ.png", alt="Diagram showing two links between the Window object and a web worker and service worker.", width="728", height="418" %}

Web workers and service workers {: #web-workers-and-service-workers }

Similarities {: #similarities }

Web workers and service workers are two types of workers available to websites. They have some things in common:

- Both run in a secondary thread, allowing JavaScript code to execute without blocking the main thread and the user interface.
- They don't have access to the `Window` and `Document` objects, so they can't interact with the DOM directly, and they have limited access to browser APIs.

Differences {: #differences }

One might think that most things that can be delegated to a web worker can be done in a service worker and vice versa, but there are important differences between them:

- Unlike web workers, service workers allow you to intercept network requests (via the `fetch` event) and to listen for Push API events in the background (via the `push` event).

- A page can spawn multiple web workers, but a single service worker controls all the active tabs under the scope it was registered with.
- The lifespan of the web worker is tightly coupled to the tab it belongs to, while the service worker's lifecycle is independent of it. For that reason, closing the tab where a web worker is running will terminate it, while a service worker can continue running in the background, even when the site doesn't have any active tabs open.

{% Aside %} For relatively short bits of work like sending a message, the browser won't likely terminate a service worker when there are no active tabs, but if the task takes too long the browser will terminate the service worker, otherwise it's a risk to the user's privacy and battery. APIs like Background Fetch, that can let you avoid the service worker's termination. {% endAside %}

Use cases {: #use-cases }

The differences between both types of workers suggest in which situations one might want to use one or the other:

Use cases for web workers are more commonly related to offloading work (like heavy computations) to a secondary thread, to avoid blocking the UI.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/ZCC24V8uqi6HfFjRzuPq.png", alt="Diagram showing a link from the Window object to a web worker.", width="500", height="175" %}

- **Example:** the team that built the videogame PROXX wanted to leave the main thread as free as possible to take care of user input and animations. To achieve that, they used web workers to run the game logic and state maintenance on a separate thread.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/xXNCbahCtPrS8rw6uf1z.png", alt="A screenshot of the videogame PROXX.", width="225", height="403", style="width: inherit; margin: auto;" %}

Service workers tasks are generally more related to acting as a network proxy, handling background tasks, and things like caching and offline.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/GmGcVnb2y1yNc4ZIFFQ8.png", alt="A screenshot of the videogame PROXX.", width="624", height="267" %}

Example: In a podcast PWA, one might want to allow users to download complete episodes to listen to them while offline. A service worker, and, in particular, the Background Fetch API can be

used to that end. That way, if the user closes the tab while the episode is downloading, the task doesn't have to be interrupted.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/oUH6K2JvcmfdAynTMjxQ.png", alt="A screenshot of a Podcast PWA.", width="500", height="310" %}
> The UI is updated to indicate the progress of a download (left). Thanks to service workers, the operation can continue running when all tabs have been closed (right).

Tools and libraries {: #tools-and-libraries }

Window and worker communication can be implemented by using different lower level APIs. Fortunately, there are libraries that abstract this process, taking care of the most common use cases. In this section, we'll cover two of them that take care of window to web workers and service workers respectively: Comlink and Workbox.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/MNMwX5KuJt5iOLZJbdO0.png", alt="A screenshot of the videogame PROXX.", width="500", height="269" %}

Comlink {: #comlink }

Comlink is a small (1.6k) RPC library that takes care of many underlying details when building websites that use Web Workers. It has been used in websites like PROXX and Squoosh. A summary of its motivations and code samples can be found here.

Workbox {: #workbox }

Workbox is a popular library to build websites that use service workers. It packages a set of best practices around things like caching, offline, background synchronization, etc. The `workbox-window` module provides a convenient way to exchange messages between the service worker and the page.

Next steps {: #next-steps }

The rest of this series focuses on patterns for window and service worker communication:

- Imperative caching guide: Calling a service worker from the page to cache resources in advance (e.g. in prefetching scenarios).
- Broadcast updates: Calling the page from the service worker to inform about important updates (e.g. a new version of the website is available).
- Two-way communication: Delegating a task to a service worker (e.g. a heavy download), and keeping the page informed on the progress.

For patterns of window and web worker communication check out: Use web workers to run JavaScript off the browser's main thread.

□□□ #2
□ □□□ □□□ 2026-10-31 23:13:07 UTC
□ □□□ □□□ 2026-10-31 23:13:08 UTC