Engineer

Service worker

a script that answers requests when the network cannot

What it is

A service worker is a script that a site registers with the browser, which runs it apart from any page and routes the network requests of the pages it controls through it. Service workers are a kind of web worker: each runs on a thread of its own, with no document and no window. The browser starts one when there is something for it to handle and may stop it as soon as nothing is left, so anything it keeps in a variable can be gone by the next event.

A page installs one with a single call, and the specification puts three conditions on it. The page must be a secure context, which in practice means HTTPS, with `localhost` and the loopback addresses allowed for development. The script must come from the page's own origin, so it cannot live on a CDN, although it may import code from one. And the registration covers a scope: by default the folder the script sits in, and nothing wider unless the server sends a `Service-Worker-Allowed` header with the script.

```
// on the page
navigator.serviceWorker.register("/sw.js");

// in /sw.js
async function answer(request) {
    const copy = await caches.match(request);
    return copy ?? fetch(request);
}

self.addEventListener("fetch", (event) => {
    event.respondWith(answer(event.request));
});
```

That worker answers each request from its own copy when it holds one and from the network otherwise. Putting the copies there is the lifecycle's job. Once the script has been fetched, the browser fires `install`, the moment to download and store what the worker will need; if a promise handed to `event.waitUntil()` there rejects, the installation fails. Then `activate` fires, the moment to clear out what an older version left behind, and only after that does the worker receive `fetch` events. A page is matched to its worker when it loads, so the page that registered the first worker carries on without one until it is reloaded, unless the worker calls `clients.claim()`.

The copies live in Cache Storage, which the same specification defines: named caches of request and response pairs that the worker fills, reads and deletes itself. They are separate from the browser's HTTP cache. An entry never expires, never refreshes on its own and survives an update of the worker script, so it stays until the worker deletes it or the browser clears the site's storage.

Use the concept in a decision

Put "What is a service worker?" to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

Directed full-stack GIS map development Get in touch →

Why it matters

Service workers give a site explicit control over how it keeps working with no network at all. A service worker decides request by request, in code the site wrote, whether to answer from the network, from its own copies or from both, which is what makes an installed web app usable on a ship or in a basement. Push messages reach it too, when no page of the site is open.

Most workers come down to choosing a strategy per kind of request, and Workbox, a library for writing them, names five. Cache first answers from the copy and goes to the network only when there is none, which suits files that never change under the same name. Network first asks the network and falls back to the copy, which suits pages. Stale-while-revalidate answers from the copy and refreshes it in the background. Network only and cache only are the two extremes.

It is also code that stands between every page in its scope and the network, on the reader's device, and it outlives the visit: a mistake in it stays until a corrected script replaces it. The specification is built around that risk. Every navigation makes the browser check for a new script, and other events do so once a day has passed since the last check. By default the main script is revalidated with the server rather than taken from the HTTP cache, so a long `max-age` on the worker file cannot keep a broken version in place; imported scripts get the same treatment once a day has passed, or always when `updateViaCache` is `none`. The browser compares the fetched script with the installed one byte for byte, and only a difference starts an update. A shift-reload skips the worker for that one navigation, which is why a bug can look fixed on a developer's machine.

The waiting rule is the other half of that design. A new version installs alongside the old one and then waits, unused, until no open page still depends on the old one, so two versions never serve one page between them. Calling `skipWaiting()` lets it take over at once, at the price of running new code under pages the old one loaded.

Storage is the last cost. Cache Storage counts against the space the site holds on the reader's device, and under the Storage Standard that space is best-effort by default: the browser may clear it when the device runs short, unless the site has asked for persistent storage and been granted it.

What it looks like in practice

The case for one is clearest where the network is weakest. The progressive web app built for seafarers and field staff serves people who are often offline or on a poor connection. Its service worker is set up through the next-pwa plugin, and Workbox runtime caching applies cache first to what does not change from one request to the next, fonts, images, audio, video and CDN assets, alongside HTTP cache-control headers. Because service workers only really behave over HTTPS, the offline and install behaviour was tested over HTTPS on real devices, and repeat loads come back from the cache instead of the network.

This site makes the opposite choice, and says so. The cookies page states that it has no service worker, no IndexedDB and no Cache Storage: nothing from the site runs in the background before a reader opens it or after they close it. Its pages ship no executable JavaScript. The search code is retained but disabled and does not register a worker.

The install page draws the consequence for anyone who adds the site to a home screen: installation gives it its own window without creating a managed offline copy. This site ships no service worker or script-managed offline storage, so the installed app needs a connection for normal use.

A worker is registered from executable script, which this site's Content-Security-Policy now forbids with `script-src 'none'` on every page. JSON-LD remains inert data. An earlier head-template comment recorded a build where 550 of 793 documents stated `'none'` and 243 stated `'self'`, matching the pages that carried the list filter then. Those figures describe the earlier script-enabled build, not the current policy.

Where it goes wrong

The old version keeps serving after a deploy. The new worker installs and then waits while any open tab still uses the old one, so a reader who never closes that tab keeps running the old code. Call `skipWaiting()` when the new version is safe under pages the old one loaded, or show an update notice and have the page message the waiting worker to call it.

Pages stay at an old version for weeks. A cache-first strategy applied to HTML answers every navigation from Cache Storage, whose entries never expire and pay no attention to the server's cache headers. Keep cache first for files whose names change when their contents do, and use network first for pages.

The install fails and the old worker stays. One address in the install step's list answers with an error, so `cache.addAll()` rejects, the promise passed to `waitUntil()` rejects with it, and the new version is discarded. Build that list from files that exist, and fetch optional files later rather than at install.

The worker ignores most of the site. A script served from `/js/sw.js` has `/js/` as both its default and its widest scope, so pages outside that folder load without it. Serve the script from the root of the site, or send `Service-Worker-Allowed: /` with it.

Storage on the device grows with every release. Caches survive an update of the worker script, so each version that opens a cache under a new name leaves the previous one behind. Put a version in every cache name, and in the `activate` handler delete each cache the current version does not use.

Further reading

- Service Workers — registration, scope, the update check, the lifecycle and Cache Storage.
- Secure Contexts — which origins count as potentially trustworthy, `localhost` and the loopback addresses included.
- Fetch Standard — the `Request` and `Response` objects a `fetch` handler receives and returns, and how a request is handed to a service worker.
- Storage Standard — the storage buckets Cache Storage lives in, quota estimates and how a site asks for persistent storage.
- Workbox: workbox-strategies — the five caching strategies and the kind of request each one suits.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a service worker?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/service-worker/