Service worker mindset

- The same, but different
  - A new layer
- Powerful, but limited
- Long-lived, but short-lived
  - Stopped
  - waitUntil
  - Watch out for global state
- Together, but separate
  - Messing with another service worker's caches
  - Skip skipWaiting
  - Start clean
  - End clean
- Service worker mindset

Service workers are powerful and absolutely worth learning. They let you deliver an entirely new level of experience to your users. Your site can load *instantly*. It can work *offline*. It can be installed as a platform-specific app and feel every bit as polished—but with the reach and freedom of the web.

But service workers are unlike anything most of us web devs are used to. They come with a steep learning curve and a handful of snags you've got to watch out for.

Google Developers and I recently collaborated on a project—Service Workies—a free game for understanding service workers. While building it and working with the complex ins and outs of service workers, I ran into a few snags. What helped me the most was coming up with a handful of depictive metaphors. In this post we'll explore these mental models and wrap our brains around the paradoxical traits that make service workers both tricky and awesome.

The same, but different

While coding your service worker, many things will feel familiar. You get to use your favorite new JavaScript language features. You listen to lifecycle events just like with UI events. You manage control flow with promises like you're used to.

But other service worker behavior causes you to scratch your head in confusion. Especially when you refresh the page and don't see your code changes applied.

A new layer

Normally when building a site you have just two layers to think about: the client and the server. The service worker is a brand new layer that sits in the middle.

A service worker acts as a middle layer between the client and the server

Think of your service worker as a sort of *browser extension*—one that your site can install in your user's browser. Once installed, the service worker *extends* the browser for your site with a powerful middle layer. This service worker layer can intercept and handle all of the requests your site makes.

The service worker layer has its **own lifecycle** independent of the browser tab. A simple page refresh isn't enough to update a service worker—just like you wouldn't expect a page refresh to update code deployed on a server. Each layer has its own unique rules for updating.

In the Service Workies game we cover the many details of the service worker lifecycle and give you a ton of practice working with it.

Think of your service worker as a new middle layer with its own lifecycle and methods for updating.

Powerful, but limited

Having a service worker on your site gives you incredible benefits. Your site can:

- work flawlessly even when the user is offline
- gain massive performance improvements through caching
- use push notifications
- be installed as a PWA

With as much as service workers can do, they are limited by design. They can't do anything synchronous or in the same thread as your site. So that means no access to:

- localStorage
- the DOM

- the window

The good news is there are a handful of ways your page can communicate with its service worker, including direct `postMessage`, one-to-one Message Channels and one-to-many Broadcast Channels.

Think of your service worker as something that lives outside of your page. You can talk to it, but it can't access your page directly.

Long-lived, but short-lived

An active service worker goes on living even after a user leaves your site or closes the tab. The browser keeps this service worker around so that it will be ready the next time the user comes back to your site. Before the very first request is made, the service worker gets a chance to intercept it and take control of the page. This is what allows a site to work offline—the service worker can serve a cached version of the page itself, even if the user has no connection to the internet.

In Service Workies we visualize this concept with Kolohe (a friendly service worker) intercepting and handling requests.

Stopped

Despite service workers appearing to be immortal, they can be **stopped** at almost any time. The browser doesn't want to waste resources on a service worker that isn't currently doing anything. Getting stopped isn't the same thing as getting *terminated*—the service worker remains installed and activated. It's just put to sleep. The next time it's needed (e.g., to handle a request), the browser wakes it back up.

waitUntil

Because of the constant possibility of getting put to sleep, your service worker needs a way to let the browser know when it's doing something important and doesn't feel like taking a nap. This is where `event.waitUntil()` comes into play. This method extends the lifecycle it's used in, keeping it both from being stopped and from moving on to the next phase of its lifecycle until we're ready. This gives us time to setup caches, fetch resources from the network, etc.

This example tells the browser that our service worker isn't done installing until the `assets` cache has been created and populated with the picture of a sword:

```
self.addEventListener("install", event => {
  event.waitUntil(
    caches.open("assets").then(cache => {
      return cache.addAll(["/weapons/sword/blade.png"]);
    })
  );
});
```

Watch out for global state

When this start/stop happens the service worker's global scope is reset. So be careful not to use any global state in your service worker or you'll be sad the next time it wakes back up and has different state from what it was expecting.

Consider this example that uses a global state:

```
const favoriteNumber = Math.random();
let hasHandledARequest = false;

self.addEventListener("fetch", event => {
  console.log(favoriteNumber);
  console.log(hasHandledARequest);
  hasHandledARequest = true;
});
```

On each request this service worker will log a number—let's say `0.13981866382421893`. The `hasHandledARequest` variable also changes to `true`. Now the service worker sits idle for a bit, so the browser stops it. The next time there's a request, the service worker is needed again, so the browser wakes it up. Its script is **evaluated again**. Now `hasHandledARequest` is reset to `false`, and `favoriteNumber` is something completely different—`0.5907281835659033`.

You can't rely on stored state in a service worker. Also, creating instances of things like Message Channels can cause bugs: you'll get a brand new instance every time the service worker stops/starts.

This snag is especially important to keep in mind while working on your service worker code because when Chrome DevTools is open, the start/stop behavior is disabled. You may not even see bugs caused by relying on global state until they've shipped to your users.

In Service Workies chapter 3 we visualize our stopped service worker as losing all color while it waits to be woken up.

visualization of a stopped service worker

Think of your service worker as a whippet dog. It's fast, loyal and awesome. It'll stick around by your side no matter what. But mostly it just wants to sleep. All the time. You've got to let it know when you want it to stay awake. Good dog!

Together, but separate

Your page can only be *controlled* by one service worker at a time. But it can have two service workers *installed* at once. When you make a change to your service worker code and refresh the page, you aren't actually editing your service worker at all. Service workers are *immutable*. You're instead making a brand new one. This new service worker (let's call it SW2) will *install* but it won't *activate* yet. It has to *wait* for the current service worker (SW1) to terminate (when your user leaves your site).

Messing with another service worker's caches

While installing, SW2 can get things setup—usually creating and populating caches. But heads up: this new service worker has access to everything that the current service worker has access to. If you're not careful, your new waiting service worker can really mess things up for your current service worker. Some examples that could cause you trouble:

- SW2 could delete a cache that SW1 is actively using.
- SW2 could edit the contents of a cache that SW1 is using, causing SW1 to respond with assets that the page isn't expecting.

Skip skipWaiting

A service worker can also use the risky `skipWaiting()` method to take control of the page as soon as it's done installing. This is generally a bad idea unless you're intentionally trying to replace a buggy service worker. The new service worker might be using updated resources that the current page isn't expecting, leading to errors and bugs.

Start clean

The way to prevent your service workers from clobbering each other is to make sure they use different caches. The easiest way to accomplish that is to version the cache names they use.

```
const version = 1;
const assetCacheName = `assets-${version}`;

self.addEventListener("install", event => {
  caches.open(assetCacheName).then(cache => {
    // confidently do stuff with your very own cache
  });
});
```

When you deploy a new service worker, you'll bump the `version` so that it does what it needs with an entirely separate cache from the previous service worker.

visualization of a cache

End clean

Once your service worker reaches the `activated` state, you know it has taken over, and the previous service worker is redundant. At this point it's important to clean up after the old service worker. Not only does it respect your users' cache storage limits, but it can also prevent unintentional bugs.

The `caches.match()` method is an often-used shortcut for retrieving an item from *any* cache where there's a match. But it iterates through the caches in the order they were created. So let's say you've got two versions of a script file `app.js` in two different caches—`assets-1` and `assets-2`. Your page is expecting the newer script that's stored in `assets-2`. But if you haven't deleted the old cache, `caches.match('app.js')` is going to return the old one from `assets-1` and most likely break your site.

All it takes to clean up after previous service workers is to delete any cache that the new service worker doesn't need:

```
const version = 2;
const assetCacheName = `assets-${version}`;

self.addEventListener("activate", event => {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cacheName => {
          if (cacheName !== assetCacheName){
            return caches.delete(cacheName);
          }
```

```
        });
      );
    });
  );
});
```

Preventing your service workers from clobbering each other takes a bit of work and discipline but is worth the trouble.

Think of the combination of your service worker and your site as an installable app. Each version should work. Each version should be separate from the others. Imagine how buggy a game would be if the developer accidentally released a patch that used new game logic but outdated assets. You'd rage on the forums so fast! Keep your app versions tidy & clean.

Service worker mindset

Getting into the right mindset while thinking about service workers will help you build yours with confidence. Once you get the hang of them you'll be able to create incredible experiences for your users.

If you want to understand all this by playing a game, then you're in luck! Go play Service Workies where you'll learn the ways of the service worker in order to slay the offline beasts.

□□□ #3
□ □□□ □□□ 2026-10-31 22:15:33 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:00 UTC