The Service Worker Lifecycle

By Jake Archibald
Human boy working on web standards at Google

The lifecycle of the service worker is its most complicated part. If you don't know what it's trying to do and what the benefits are, it can feel like it's fighting you. But once you know how it works, you can deliver seemless, unobtrusive updates to users, mixing the best of web and native patterns.

This is a deep dive, but the bullets at the start of each section cover most of what you need to know.

The intent

The intent of the lifecycle is to:

- Make offline-first possible.
- Allow a new service worker to get itself ready without disrupting the current one.
- Ensure an in-scope page is controlled by the same service worker (or no service worker) throughout.
- Ensure there's only one version of your site running at once.

That last one is pretty important. Without service workers, users can load one tab to your site, then later open another. This can result in two versions of your site running at the same time. Sometimes this is ok, but if you're dealing with storage you can easily end up with two tabs having very different opinions on how their shared storage should be managed. This can result in errors, or worse, data loss.

Caution: Users actively dislike data loss. It causes them great sadness.

The first service worker

In brief:

- The **`install`** event is the first event a service worker gets, and it only happens once.
- A promise passed to **`installEvent.waitUntil()`** signals the duration and success or failure of your install.
- A service worker won't receive events like **`fetch`** and **`push`** until it successfully finishes installing and becomes "active".
- By default, a page's fetches won't go through a service worker unless the page request itself went through a service worker. So you'll need to refresh the page to see the effects of the service worker.
- **`clients.claim()`** can override this default, and take control of non-controlled pages.

Take this HTML:

```
<!DOCTYPE html>
An image will appear here in 3 seconds:
<script>
  navigator.serviceWorker.register('/sw.js')
    .then(reg => console.log('SW registered!', reg))
    .catch(err => console.log('Boo!', err));

  setTimeout(() => {
    const img = new Image();
    img.src = '/dog.svg';
    document.body.appendChild(img);
  }, 3000);
</script>
```

It registers a service worker, and adds image of a dog after 3 seconds.

Here's its service worker, `sw.js`:

```
self.addEventListener('install', event => {
  console.log('V1 installing…');

  // cache a cat SVG
  event.waitUntil(
    caches.open('static-v1').then(cache => cache.add('/cat.svg'))
  );
});

self.addEventListener('activate', event => {
  console.log('V1 now ready to handle fetches!');
});

self.addEventListener('fetch', event => {
  const url = new URL(event.request.url);

  // serve the cat SVG from the cache if the request is
  // same-origin and the path is '/dog.svg'
  if (url.origin == location.origin && url.pathname == '/dog.svg') {
    event.respondWith(caches.match('/cat.svg'));
  }
});
```

It caches an image of a cat, and serves it whenever there's a request for `/dog.svg`. However, if you run the above example, you'll see a dog the first time you load the page. Hit refresh, and you'll see the cat.

Note: Cats are better than dogs. They just *are*.

Scope and control

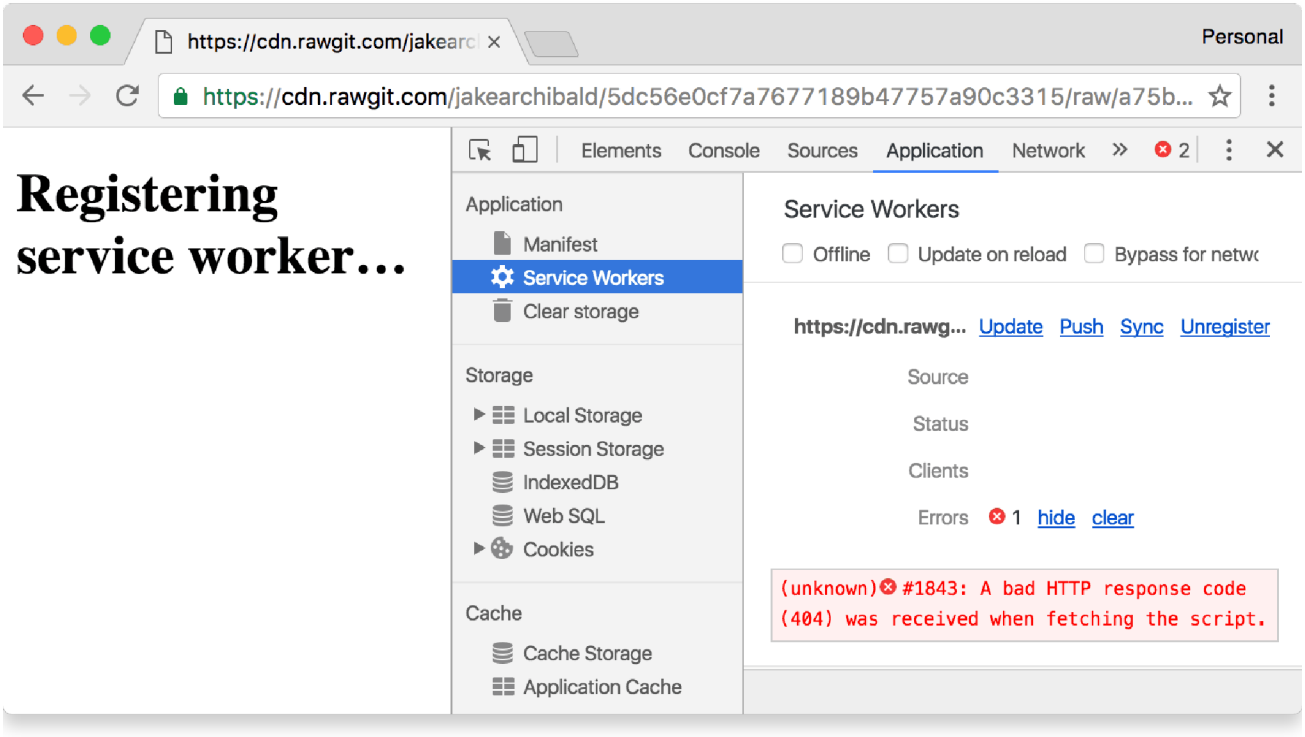
The default scope of a service worker registration is `./` relative to the script URL. This means if you register a service worker at `//example.com/foo/bar.js` it has a default scope of `//example.com/foo/`.

We call pages, workers, and shared workers `clients`. Your service worker can only control clients that are in-scope. Once a client is "controlled", its fetches go through the in-scope service worker. You can detect if a client is controlled via `navigator.serviceWorker.controller` which will be null or a service worker instance.

Download, parse, and execute

Your very first service worker downloads when you call `.register()`. If your script fails to download, parse, or throws an error in its initial execution, the register promise rejects, and the service worker is discarded.

Chrome's DevTools shows the error in the console, and in the service worker section of the application tab:



Install

The first event a service worker gets is **`install`**. It's triggered as soon as the worker executes, and it's only called once per service worker. If you alter your service worker script the browser considers it a different service worker, and it'll get its own **`install`** event. I'll cover updates in detail later.

The **`install`** event is your chance to cache everything you need before being able to control clients. The promise you pass to **`event.waitUntil()`** lets the browser know when your install completes, and if it was successful.

If your promise rejects, this signals the install failed, and the browser throws the service worker away. It'll never control clients. This means we can't rely on "cat.svg" being present in the cache in our **`fetch`** events. It's a dependency.

Activate

Once your service worker is ready to control clients and handle functional events like **push** and **sync**, you'll get an **`activate`** event. But that doesn't mean the page that called **`.register()`** will be controlled.

The first time you load the demo, even though **`dog.svg`** is requested long after the service worker activates, it doesn't handle the request, and you still see the image of the dog. The default is *consistency*, if your page loads without a service worker, neither will its subresources. If you load the demo a second time (in other words, refresh the page), it'll be controlled. Both the page and the image will go through **`fetch`** events, and you'll see a cat instead.

clients.claim

You can take control of uncontrolled clients by calling **`clients.claim()`** within your service worker once it's activated.

Here's a variation of the demo above which calls **`clients.claim()`** in its **`activate`** event. You *should* see a cat the first time. I say "should", because this is timing sensitive. You'll only see a cat if the service worker activates and **`clients.claim()`** takes effect before the image tries to load.

If you use your service worker to load pages differently than they'd load via the network, **`clients.claim()`** can be troublesome, as your service worker ends up controlling some clients that loaded without it.

Note: I see a lot of people including **`clients.claim()`** as boilerplate, but I rarely do so myself. It only really matters on the very first load, and due to progressive enhancement the page is usually working happily without service worker anyway.

Updating the service worker

In brief:

- An update is triggered:
  - On navigation to an in-scope page.
  - On functional events such as **push** and **sync**, unless there's been an update check within the previous 24 hours.

    - On calling `.register()` *only if* the service worker URL has changed.
- Most browsers, including Chrome 68 and later, default to ignoring caching headers when checking for updates of the registered service worker script. They still respect caching headers when fetching resources loaded inside a service worker via `importScripts()`. You can override this default behavior by setting the `updateViaCache` option when registering your service worker.
- Your service worker is considered updated if it's byte-different to the one the browser already has. (We're extending this to include imported scripts/modules too.)
- The updated service worker is launched alongside the existing one, and gets its own `install` event.
- If your new worker has a non-ok status code (for example, 404), fails to parse, throws an error during execution, or rejects during install, the new worker is thrown away, but the current one remains active.
- Once successfully installed, the updated worker will `wait` until the existing worker is controlling zero clients. (Note that clients overlap during a refresh.)
- `self.skipWaiting()` prevents the waiting, meaning the service worker activates as soon as it's finished installing.

Let's say we changed our service worker script to respond with a picture of a horse rather than a cat:

```
const expectedCaches = ['static-v2'];
```

```
self.addEventListener('install', event => {
  console.log('V2 installing…');

  // cache a horse SVG into a new cache, static-v2
  event.waitUntil(
    caches.open('static-v2').then(cache => cache.add('/horse.svg'))
  );
});

self.addEventListener('activate', event => {
  // delete any caches that aren't in expectedCaches
  // which will get rid of static-v1
  event.waitUntil(
    caches.keys().then(keys => Promise.all(
      keys.map(key => {
        if (!expectedCaches.includes(key)) {
          return caches.delete(key);
        }
      })
    )).then(() => {
      console.log('V2 now ready to handle fetches!');
    })
  );
});

self.addEventListener('fetch', event => {
  const url = new URL(event.request.url);

  // serve the horse SVG from the cache if the request is
  // same-origin and the path is '/dog.svg'
  if (url.origin == location.origin && url.pathname == '/dog.svg') {
    event.respondWith(caches.match('/horse.svg'));
  }
});
```

Note: I have no strong opinions on horses.

Check out a demo of the above. You should still see an image of a cat. Here's why…

Install

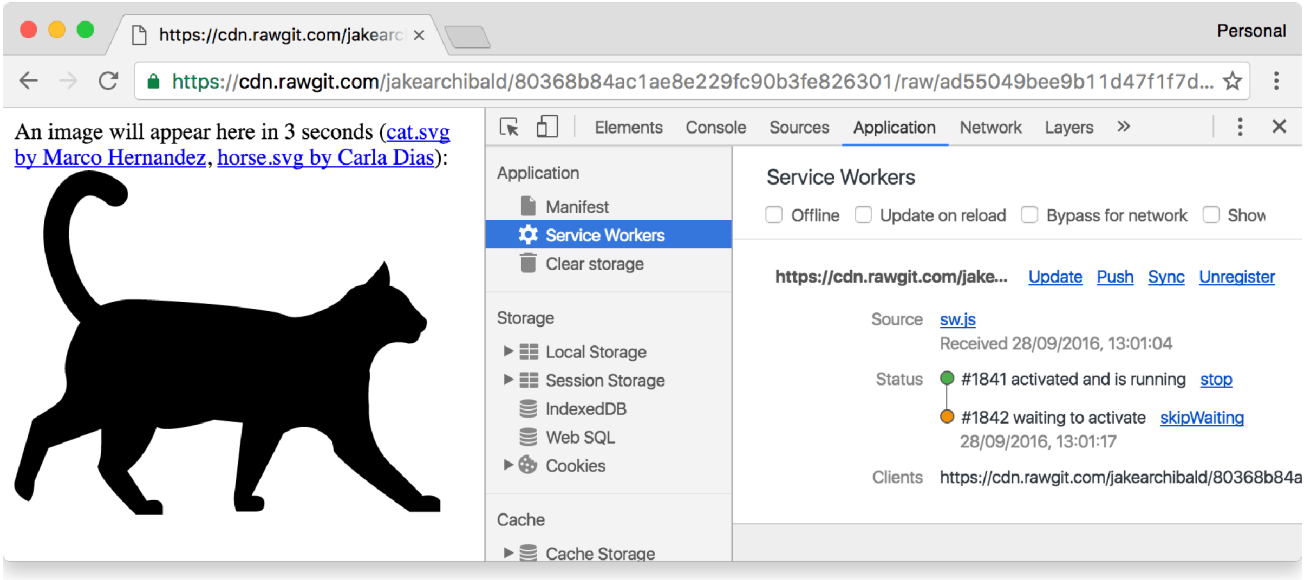
Note that I've changed the cache name from `static-v1` to `static-v2`. This means I can set up the new cache without overwriting things in the current one, which the old service worker is still using.

This patterns creates version-specific caches, akin to assets a native app would bundle with its executable. You may also have caches that aren't version specific, such as `avatars`.

Waiting

After it's successfully installed, the updated service worker delays activating until the existing service worker is no longer controlling clients. This state is called "waiting", and it's how the browser ensures that only one version of your service worker is running at a time.

If you ran the updated demo, you should still see a picture of a cat, because the V2 worker hasn't yet activated. You can see the new service worker waiting in the "Application" tab of DevTools:



Even if you only have one tab open to the demo, refreshing the page isn't enough to let the new version take over. This is due to how browser navigations work. When you navigate, the current page doesn't go away until the response headers have been received, and even then the current page may stay if the response has a `Content-Disposition` header. Because of this overlap, the current service worker is always controlling a client during a refresh.

To get the update, close or navigate away from all tabs using the current service worker. Then, when you navigate to the demo again, you should see the horse.

This pattern is similar to how Chrome updates. Updates to Chrome download in the background, but don't apply until Chrome restarts. In the mean time, you can continue to use the current version without disruption. However, this is a pain during development, but DevTools has ways to make it easier, which I'll cover later in this article.

Activate

This fires once the old service worker is gone, and your new service worker is able to control clients. This is the ideal time to do stuff that you couldn't do while the old worker was still in use, such as migrating databases and clearing caches.

In the demo above, I maintain a list of caches that I expect to be there, and in the **`activate`** event I get rid of any others, which removes the old **`static-v1`** cache.

Caution: You may not be updating from the previous version. It may be a service worker many versions old.

If you pass a promise to **`event.waitUntil()`** it'll buffer functional events (**`fetch`**, **`push`**, **`sync`** etc.) until the promise resolves. So when your **`fetch`** event fires, the activation is fully complete.

Caution: The cache storage API is "origin storage" (like localStorage, and IndexedDB). If you run many sites on the same origin (for example, **`yourname.github.io/myapp`**), be careful that you don't delete caches for your other sites. To avoid this, give your cache names a prefix unique to the current site, eg **`myapp-static-v1`**, and don't touch caches unless they begin with **`myapp-`**.

Skip the waiting phase

The waiting phase means you're only running one version of your site at once, but if you don't need that feature, you can make your new service worker activate sooner by calling **`self.skipWaiting()`**.

This causes your service worker to kick out the current active worker and activate itself as soon as it enters the waiting phase (or immediately if it's already in the waiting phase). It *doesn't* cause your worker to skip installing, just waiting.

It doesn't really matter when you call **`skipWaiting()`**, as long as it's during or before waiting. It's pretty common to call it in the **`install`** event:

```
self.addEventListener('install', event => {
  self.skipWaiting();

  event.waitUntil(
    // caching etc
  );
});
```

But you may want to call it as a results of a `postMessage()` to the service worker. As in, you want to `skipWaiting()` following a user interaction.

Here's a demo that uses `skipWaiting()`. You should see a picture of a cow without having to navigate away. Like `clients.claim()` it's a race, so you'll only see the cow if the new service worker fetches, installs and activates before the page tries to load the image.

Caution: `skipWaiting()` means that your new service worker is likely controlling pages that were loaded with an older version. This means some of your page's fetches will have been handled by your old service worker, but your new service worker will be handling subsequent fetches. If this might break things, don't use `skipWaiting()`.

Manual updates

As I mentioned earlier, the browser checks for updates automatically after navigations and functional events, but you can also trigger them manually:

```
navigator.serviceWorker.register('/sw.js').then(reg => {
  // sometime later…
  reg.update();
});
```

If you expect the user to be using your site for a long time without reloading, you may want to call `update()` on an interval (such as hourly).

Avoid changing the URL of your service worker script

If you've read my post on caching best practices, you may consider giving each version of your service worker a unique URL. **Don't do this!** This is usually bad practice for service workers, just update the script at its current location.

It can land you with a problem like this:

1. `index.html` registers `sw-v1.js` as a service worker.
2. `sw-v1.js` caches and serves `index.html` so it works offline-first.
3. You update `index.html` so it registers your new and shiny `sw-v2.js`.

If you do the above, the user never gets `sw-v2.js`, because `sw-v1.js` is serving the old version of `index.html` from its cache. You've put yourself in a position where you need to update your service worker in order to update your service worker. Ew.

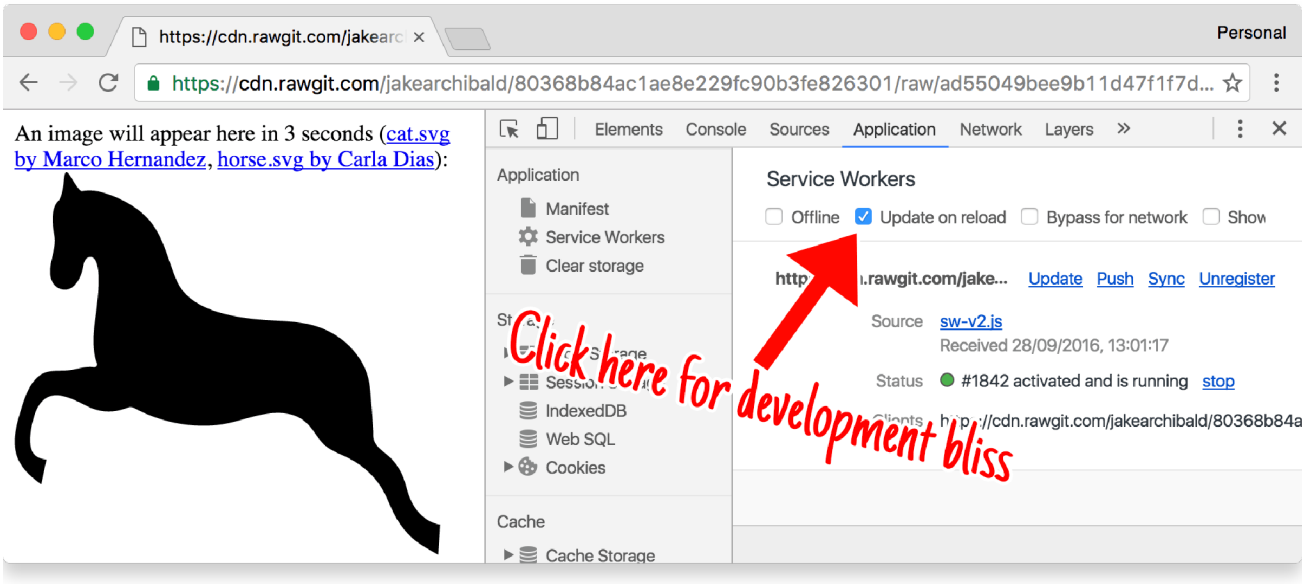
However, for the demo above, I *have* changed the URL of the service worker. This is so, for the sake of the demo, you can switch between the versions. It isn't something I'd do in production.

Making development easy

The service worker lifecycle is built with the user in mind, but during development it's a bit of a pain. Thankfully there are a few tools to help out:

Update on reload

This one's my favourite.

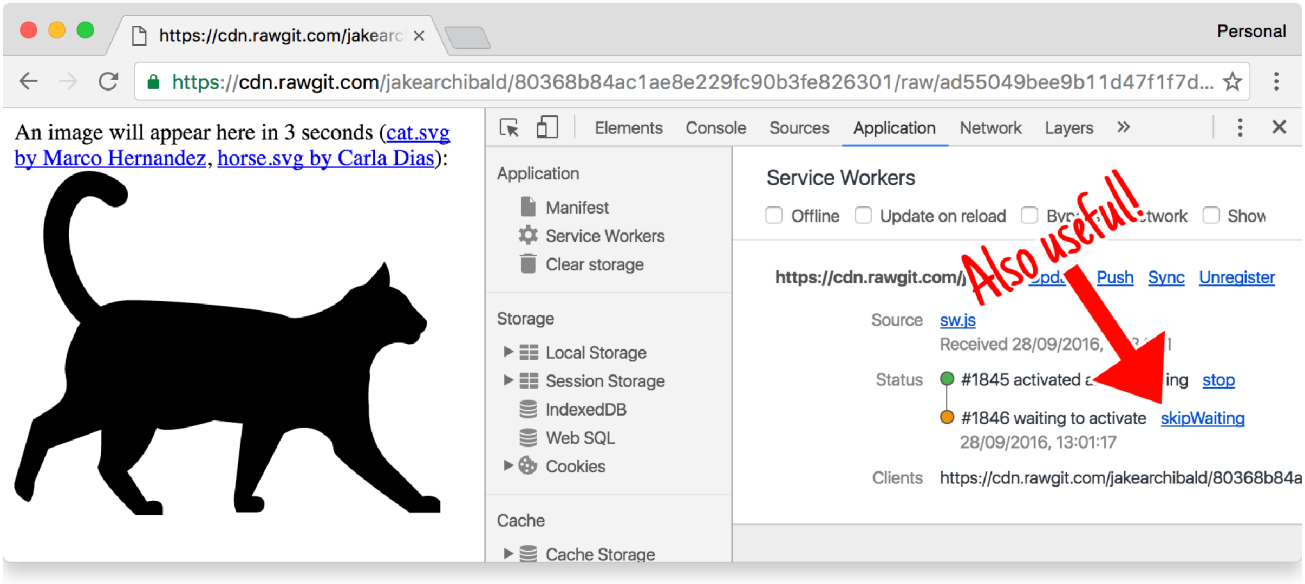


This changes the lifecycle to be developer-friendly. Each navigation will:

1. Refetch the service worker.
2. Install it as a new version even if it's byte-identical, meaning your `install` event runs and your caches update.
3. Skip the waiting phase so the new service worker activates.
4. Navigate the page.

This means you'll get your updates on each navigation (including refresh) without having to reload twice or close the tab.

Skip waiting



If you have a worker waiting, you can hit "skip waiting" in DevTools to immediately promote it to "active".

Shift-reload

If you force-reload the page (shift-reload) it bypasses the service worker entirely. It'll be uncontrolled. This feature is in the spec, so it works in other service-worker-supporting browsers.

Handling updates

The service worker was designed as part of the extensible web. The idea is that we, as browser developers, acknowledge that we are not better at web development than web developers. And as such, we shouldn't provide narrow high-level APIs that solve a particular problem using patterns *we* like, and instead give you access to the guts of the browser and let you do it how you want, in a way that works best for *your* users.

So, to enable as many patterns as we can, the whole update cycle is observable:

```
navigator.serviceWorker.register('/sw.js').then(reg => {
  reg.installing; // the installing worker, or undefined
  reg.waiting; // the waiting worker, or undefined
  reg.active; // the active worker, or undefined

  reg.addEventListener('updatefound', () => {
    // A wild service worker has appeared in reg.installing!
    const newWorker = reg.installing;
```

```
    newWorker.state;
    // "installing" - the install event has fired, but not yet complete
    // "installed"  - install complete
    // "activating" - the activate event has fired, but not yet complete
    // "activated"  - fully active
    // "redundant"  - discarded. Either failed install, or it's been
    //                replaced by a newer version

    newWorker.addEventListener('statechange', () => {
      // newWorker.state has changed
    });
  });
});

navigator.serviceWorker.addEventListener('controllerchange', () => {
  // This fires when the service worker controlling this page
  // changes, eg a new worker has skipped waiting and become
  // the new active worker.
});
```

You survived!

Phew! That was a lot of technical theory. Stay tuned in the coming weeks where we'll dive into some practical applications of the above.

Last updated July 2, 2018.