

The Offline Cookbook



By Jake Archibald

Human boy working on web standards at Google

When AppCache arrived on the scene it gave us a couple of patterns to make content work offline. If those were the patterns you needed, congratulations, you won the AppCache lottery (the jackpot remains unclaimed), but the rest of us were left huddled in a corner rocking back & forth.

With ServiceWorker we gave up trying to solve offline, and gave developers the moving parts to go solve it themselves. It gives you control over caching and how requests are handled. That means you get to create your own patterns. Let's take a look at a few possible patterns in isolation, but in practice you'll likely use many of them in tandem depending on URL & context.

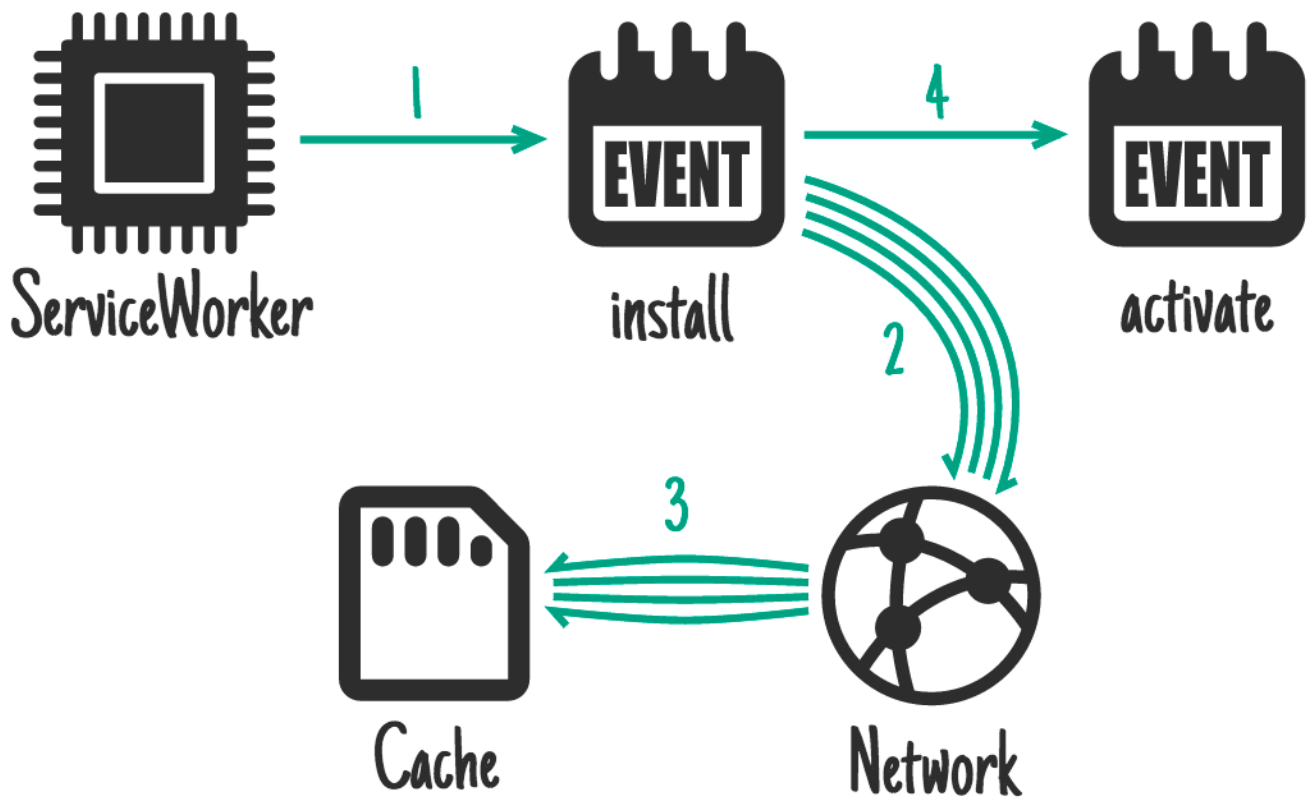
All code examples work today in Chrome & Firefox, unless otherwise noted. For full details on service worker support, see "Is Service Worker Ready?".

For a working demo of some of these patterns, see Trained-to-thrill, and this video showing the performance impact.

The cache machine - when to store resources

ServiceWorker lets you handle requests independently from caching, so we'll look at them separately. First up, caching, when should it be done?

On install - as a dependency



ServiceWorker gives you an `install` event. You can use this to get stuff ready, stuff that must be ready before you handle other events. While this happens any previous version of your ServiceWorker is still running & serving pages, so the things you do here mustn't disrupt that.

Ideal for: CSS, images, fonts, JS, templates... basically anything you'd consider static to that "version" of your site.

These are things that would make your site entirely non-functional if they failed to fetch, things an equivalent native-app would make part of the initial download.

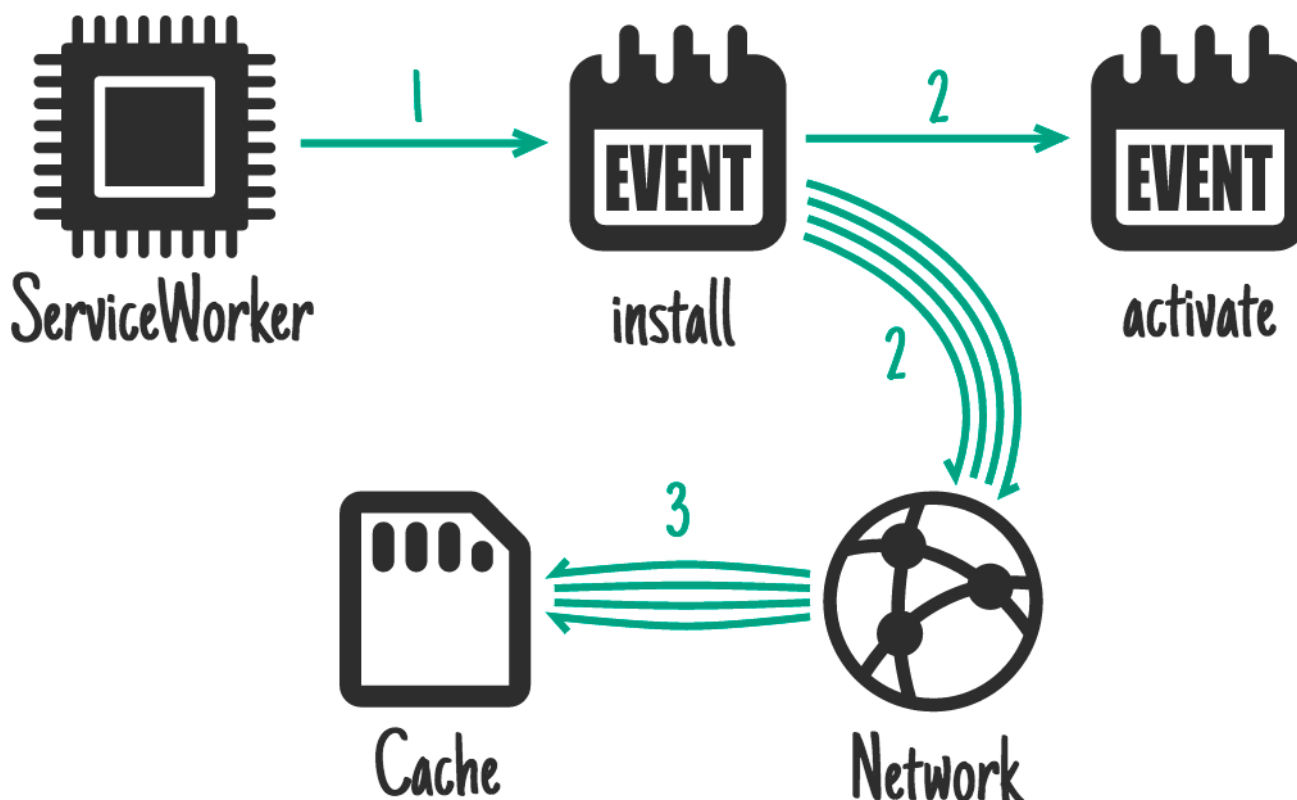
```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('mysite-static-v3').then(function(cache) {
      return cache.addAll([
        '/css/whatever-v3.css',
        '/css/imgs/sprites-v6.png',
        '/css/fonts/whatever-v8.woff',
        '/js/all-min-v4.js'
        // etc
      ]);
    })
  );
});
```



`event.waitUntil` takes a promise to define the length & success of the install. If the promise rejects, the installation is considered a failure and this ServiceWorker will be abandoned (if an older version is running, it'll be left intact). `caches.open` and `cache.addAll` return promises. If any of the resources fail to fetch, the `cache.addAll` call rejects.

On trained-to-thrill I use this to cache static assets.

On install - not as a dependency



Similar to above, but won't delay install completing and won't cause installation to fail if caching fails.

Ideal for: Bigger resources that aren't needed straight away, such as assets for later levels of a game.

```
self.addEventListener('install', function(event) {  
  event.waitUntil(  
    caches.open('mygame-core-v1').then(function(cache) {  
      cache.addAll(  
        // levels 11-20  
      );  
      return cache.addAll(  
        // core assets & levels 1-10  
      );  
    });  
  });  
});
```



```

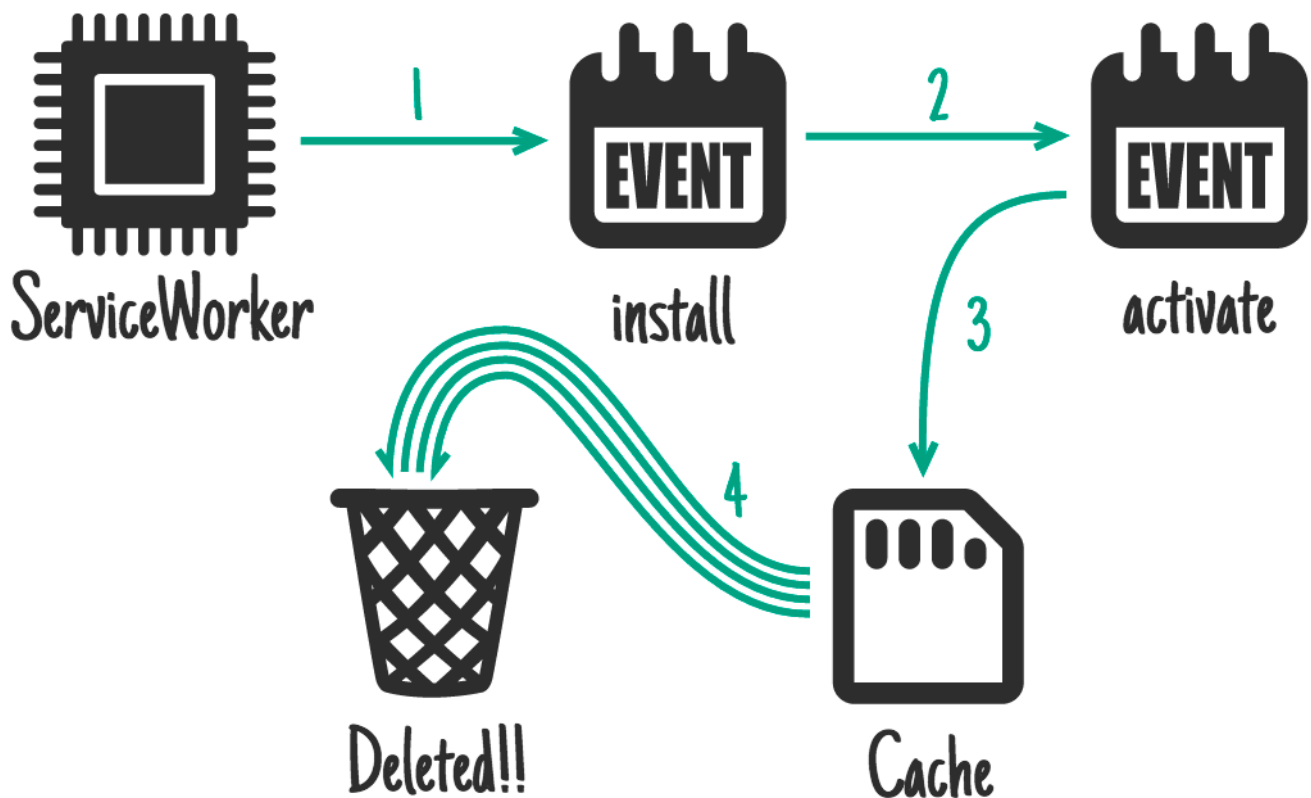
    })
  );
});

```

We're not passing the `cache.addAll` promise for levels 11-20 back to `event.waitUntil`, so even if it fails, the game will still be available offline. Of course, you'll have to cater for the possible absence of those levels & reattempt caching them if they're missing.

The ServiceWorker may be killed while levels 11-20 download since it's finished handling events, meaning they won't be cached. In future we plan to add a background downloading API to handle cases like this, and larger downloads such as movies.

On activate



Ideal for: Clean-up & migration.

Once a new ServiceWorker has installed & a previous version isn't being used, the new one activates, and you get an `activate` event. Because the old version is out of the way, it's a good time to handle schema migrations in IndexedDB and also delete unused caches.

```

self.addEventListener('activate', function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(

```



```

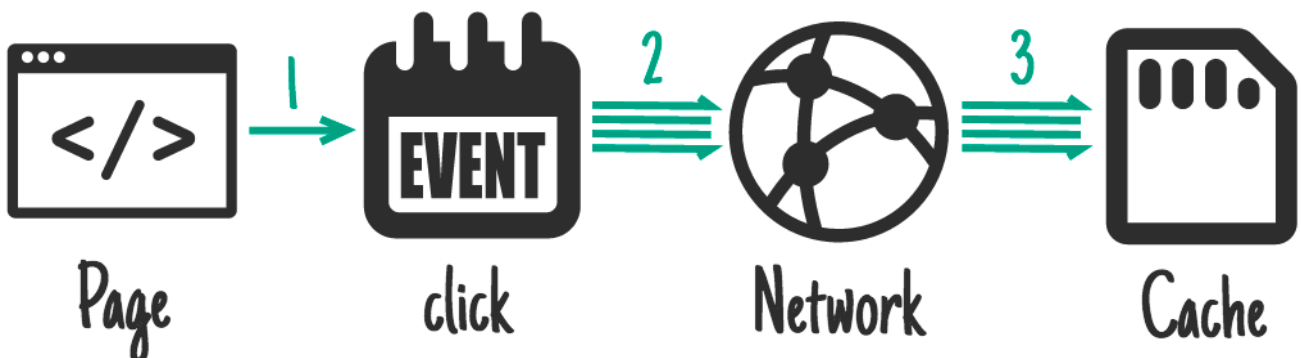
cacheNames.filter(function(cacheName) {
  // Return true if you want to remove this cache,
  // but remember that caches are shared across
  // the whole origin
}).map(function(cacheName) {
  return caches.delete(cacheName);
})
);
});

```

During activation, other events such as `fetch` are put into a queue, so a long activation could potentially block page loads. Keep your activation as lean as possible, only use it for things you *couldn't* do while the old version was active.

On [trained-to-thrill](#) I use this to [remove old caches](#).

On user interaction



Ideal for: If the whole site can't be taken offline, you may allow the user to select the content they want available offline. E.g. a video on something like YouTube, an article on Wikipedia, a particular gallery on Flickr.

Give the user a "Read later" or "Save for offline" button. When it's clicked, fetch what you need from the network & pop it in the cache.

```

document.querySelector('.cache-article').addEventListener('click', function(event) {
  event.preventDefault();

  var id = this.dataset.articleId;
  caches.open('mysite-article-' + id).then(function(cache) {
    fetch('/get-article-urls?id=' + id).then(function(response) {
      // /get-article-urls returns a JSON-encoded array of
      // resource URLs that a given article depends on
    });
  });
});

```

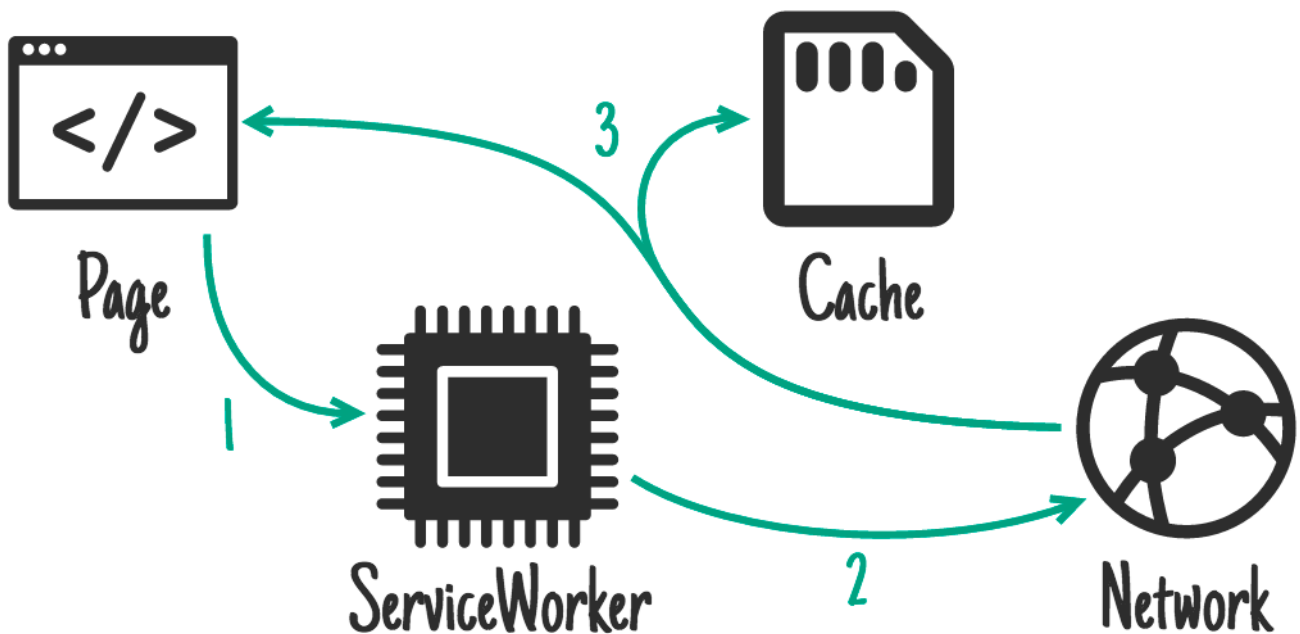
```

    return response.json();
  }).then(function(urls) {
    cache.addAll(urls);
  });
});
});
});

```

The caches API is available from pages as well as service workers, meaning you don't need to involve the service worker to add things to the cache.

On network response



Ideal for: Frequently updating resources such as a user's inbox, or article contents. Also useful for non-essential content such as avatars, but care is needed.

If a request doesn't match anything in the cache, get it from the network, send it to the page & add it to the cache at the same time.

If you do this for a range of URLs, such as avatars, you'll need to be careful you don't bloat the storage of your origin – if the user needs to reclaim disk space you don't want to be the prime candidate. Make sure you get rid of items in the cache you don't need any more.

```

self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    });
  );
});

```



```

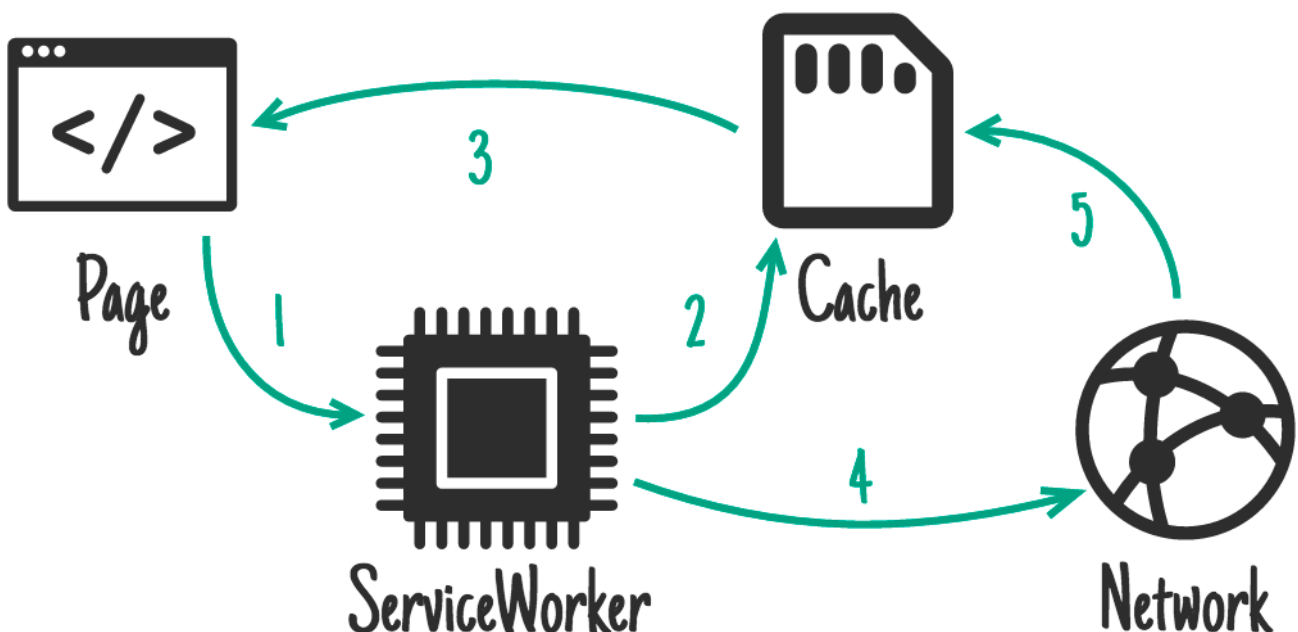
    });
  });
})
);
});

```

To allow for efficient memory usage, you can only read a response/request's body once. In the code above, `.clone()` is used to create additional copies that can be read separately.

On [trained-to-thrill](#) I use this to [cache Flickr images](#).

Stale-while-revalidate



Ideal for: Frequently updating resources where having the very latest version is non-essential. Avatars can fall into this category.

If there's a cached version available, use it, but fetch an update for next time.

```

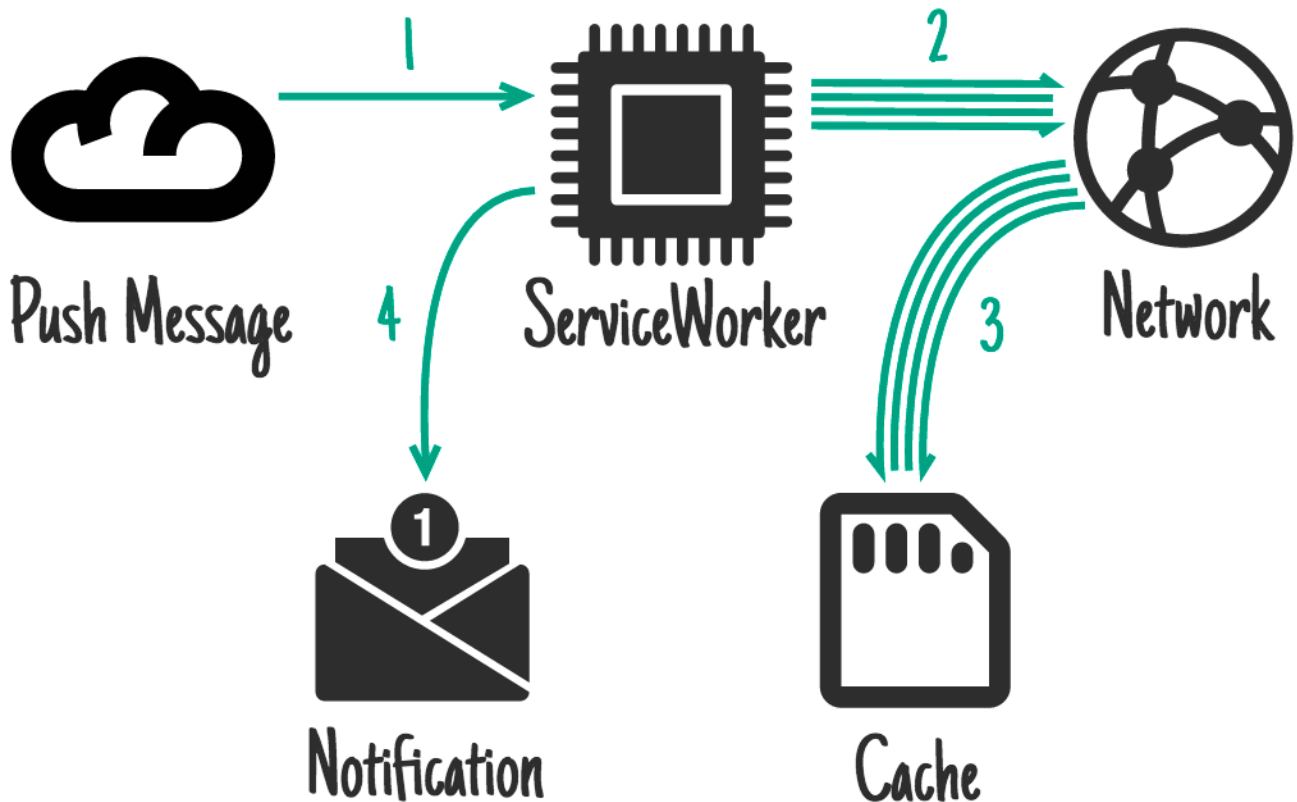
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function(response) {
        var fetchPromise = fetch(event.request).then(function(networkResponse) {
          cache.put(event.request, networkResponse.clone());
          return networkResponse;
        });
        return response || fetchPromise;
      })
    })
  )
});

```

```
);  
});
```

This is very similar to HTTP's stale-while-revalidate.

On push message



The Push API is another feature built on top of ServiceWorker. This allows the ServiceWorker to be awoken in response to a message from the OS's messaging service. This happens even when the user doesn't have a tab open to your site, only the ServiceWorker is woken up. You request permission to do this from a page & the user will be prompted.

Ideal for: Content relating to a notification, such as a chat message, a breaking news story, or an email. Also infrequently changing content that benefits from immediate sync, such as a todo list update or a calendar alteration.

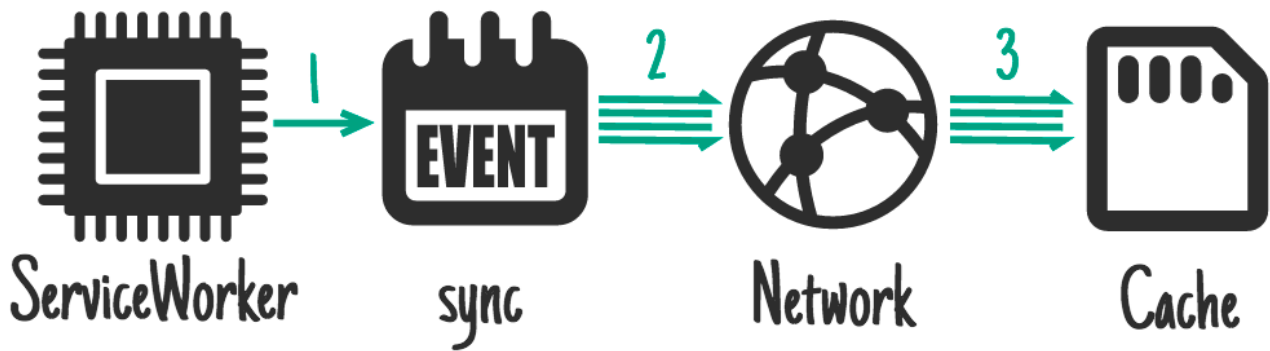
The common final outcome is a notification which, when tapped, opens/focuses a relevant page, but updating caches before this happens is *extremely* important. The user is obviously online at the time of receiving the push message, but they may not be when they finally interact with the notification, so making this content available offline is important. The Twitter native app, which is for the most part an excellent example of offline-first, gets this a bit wrong.

Without a connection, Twitter fails to provide the content relating to the push message. Tapping it does remove the notification however, leaving the user with less information than before they tapped. Don't do this!

This code updates caches before showing a notification:

```
self.addEventListener('push', function(event) {  
  if (event.data.text() == 'new-email') {  
    event.waitUntil(  
      caches.open('mysite-dynamic').then(function(cache) {  
        return fetch('/inbox.json').then(function(response) {  
          cache.put('/inbox.json', response.clone());  
          return response.json();  
        });  
      }).then(function(emails) {  
        registration.showNotification("New email", {  
          body: "From " + emails[0].from.name  
          tag: "new-email"  
        });  
      })  
    );  
  }  
});  
  
self.addEventListener('notificationclick', function(event) {  
  if (event.notification.tag == 'new-email') {  
    // Assume that all of the resources needed to render  
    // /inbox/ have previously been cached, e.g. as part  
    // of the install handler.  
    new WindowClient('/inbox/');  
  }  
});
```

On background-sync



Background sync is another feature built on top of ServiceWorker. It allows you to request background data synchronization as a one-off, or on an (extremely heuristic) interval. This happens even when the user doesn't have a tab open to your site, only the ServiceWorker is woken up. You request permission to do this from a page & the user will be prompted.

Ideal for: Non-urgent updates, especially those that happen so regularly that a push message per update would be too frequent, such as social timelines or news articles.

```
self.addEventListener('sync', function(event) {  
  if (event.id == 'update-leaderboard') {  
    event.waitUntil(  
      caches.open('mygame-dynamic').then(function(cache) {  
        return cache.add('/leaderboard.json');  
      })  
    );  
  }  
});
```



Cache persistence

Your origin is given a certain amount of free space to do what it wants with. That free space is shared between all origin storage: LocalStorage, IndexedDB, Filesystem, and of course Caches.

The amount you get isn't spec'd, it will differ depending on device and storage conditions. You can find out how much you've got via:

```
navigator.storageQuota.queryInfo("temporary").then(function(info) {  
  console.log(info.quota);  
  // Result: <quota in bytes>  
  console.log(info.usage);  
  // Result: <used data in bytes>  
});
```



However, like all browser storage, the browser is free to throw it away if the device becomes under storage pressure. Unfortunately the browser can't tell the difference between those movies you want to keep at all costs, and the game you don't really care about.

To work around this, there's a proposed API, [requestPersistent](#) [↗](#):

```
// From a page:
navigator.storage.requestPersistent().then(function(granted) {
  if (granted) {
    // Hurrah, your data is here to stay!
  }
});
```



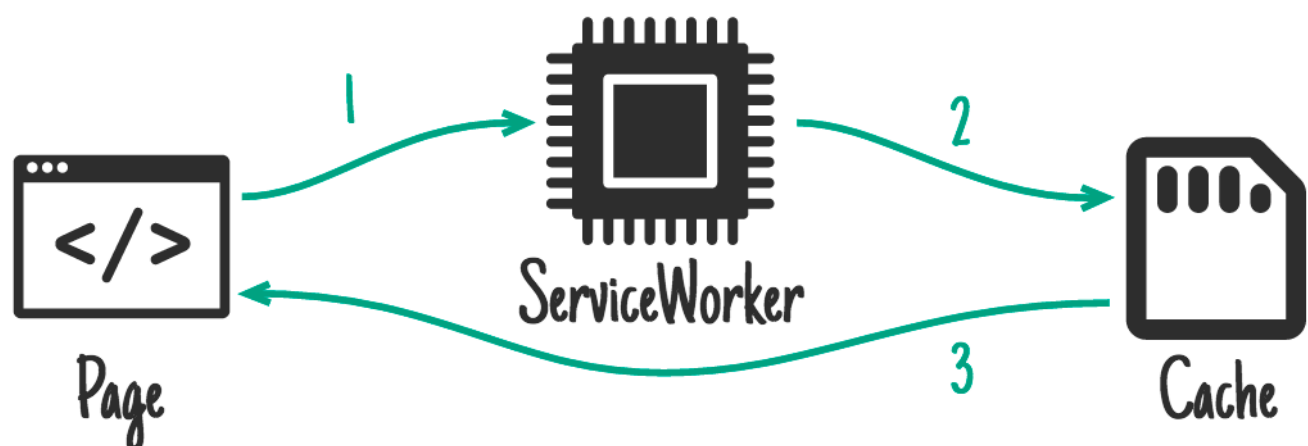
Of course, the user has to grant permission. Making the user part of this flow is important, as we can now expect them to be in control of deletion. If their device comes under storage pressure, and clearing non-essential data doesn't solve it, the user gets to make a judgment call on which items to keep and remove.

For this to work, it requires operating systems to treat "durable" origins as equivalent to native apps in their breakdowns of storage usage, rather than reporting the browser as a single item.

Serving Suggestions - responding to requests

It doesn't matter how much caching you do, the ServiceWorker won't use the cache unless you tell it when & how. Here are a few patterns for handling requests:

Cache only



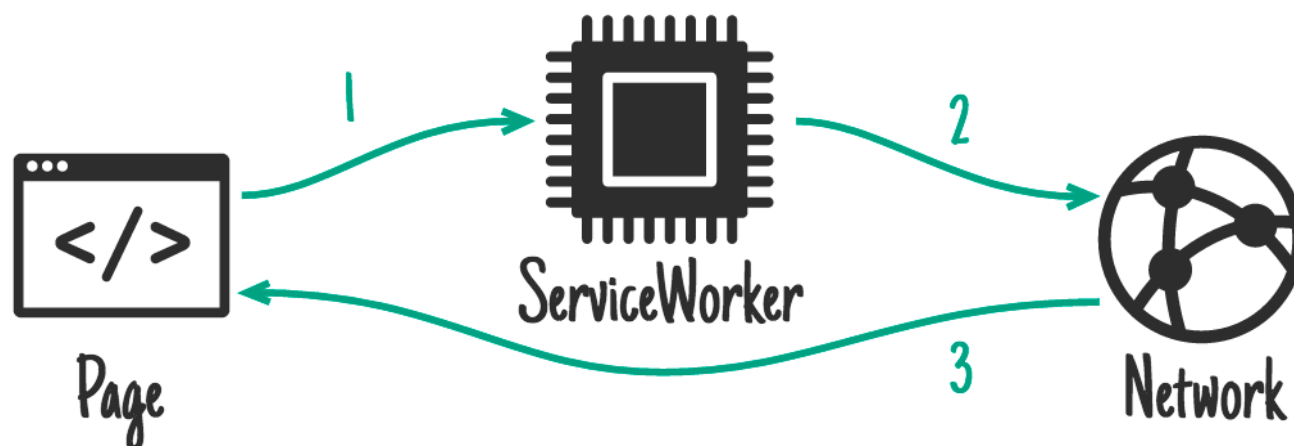
Ideal for: Anything you'd consider static to that "version" of your site. You should have cached these in the install event, so you can depend on them being there.

```
self.addEventListener('fetch', function(event) {  
  // If a match isn't found in the cache, the response  
  // will look like a connection error  
  event.respondWith(caches.match(event.request));  
});
```



...although you don't often need to handle this case specifically, Cache, falling back to network covers it.

Network only



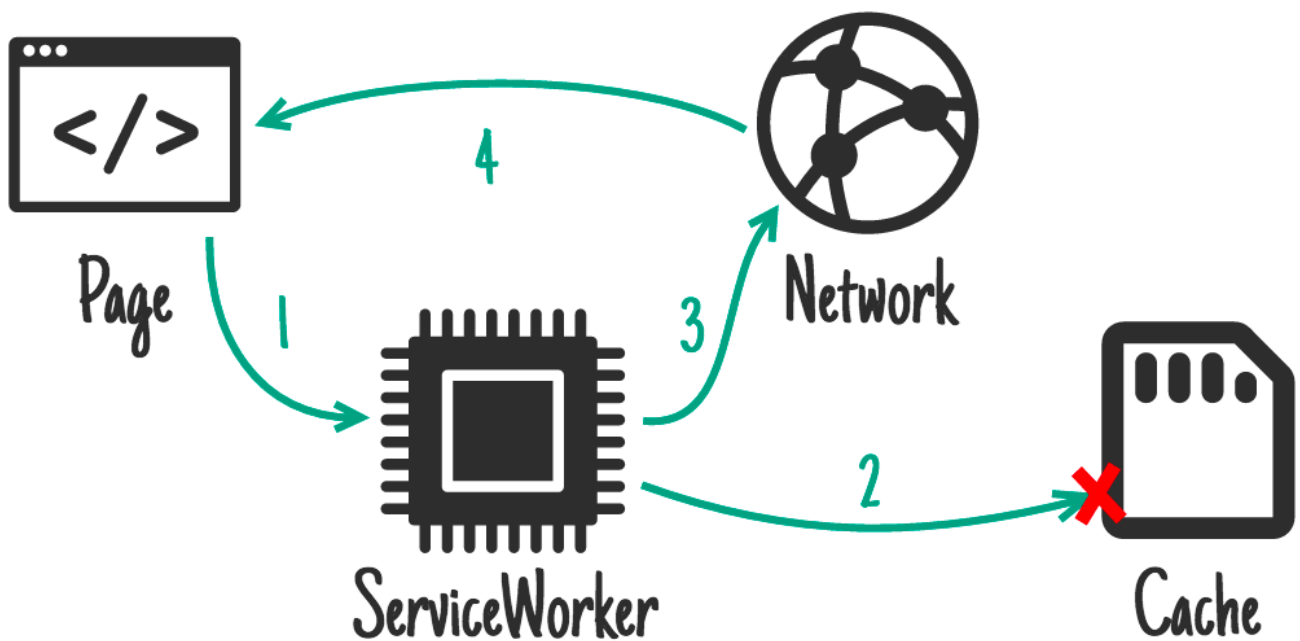
Ideal for: Things that have no offline equivalent, such as analytics pings, non-GET requests.

```
self.addEventListener('fetch', function(event) {  
  event.respondWith(fetch(event.request));  
  // or simply don't call event.respondWith, which  
  // will result in default browser behaviour  
});
```



...although you don't often need to handle this case specifically, Cache, falling back to network covers it.

Cache, falling back to network



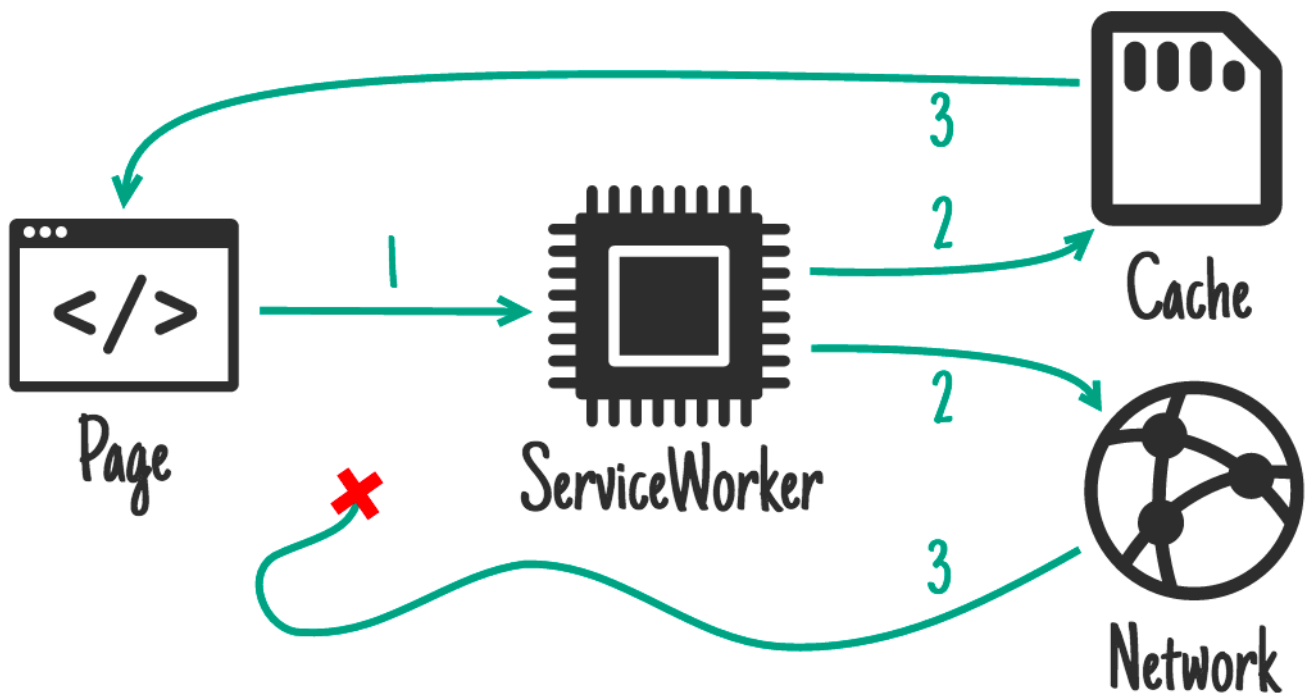
Ideal for: If you're building offline-first, this is how you'll handle the majority of requests. Other patterns will be exceptions based on the incoming request.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);
    })
  );
});
```



This gives you the "Cache only" behaviour for things in the cache and the "Network only" behaviour for anything not-cached (which includes all non-GET requests, as they cannot be cached).

Cache & network race



Ideal for: Small assets where you're chasing performance on devices with slow disk access.

With some combinations of older hard drives, virus scanners, and faster internet connections, getting resources from the network can be quicker than going to disk. However, going to the network when the user has the content on their device can be a waste of data, so bear that in mind.

```

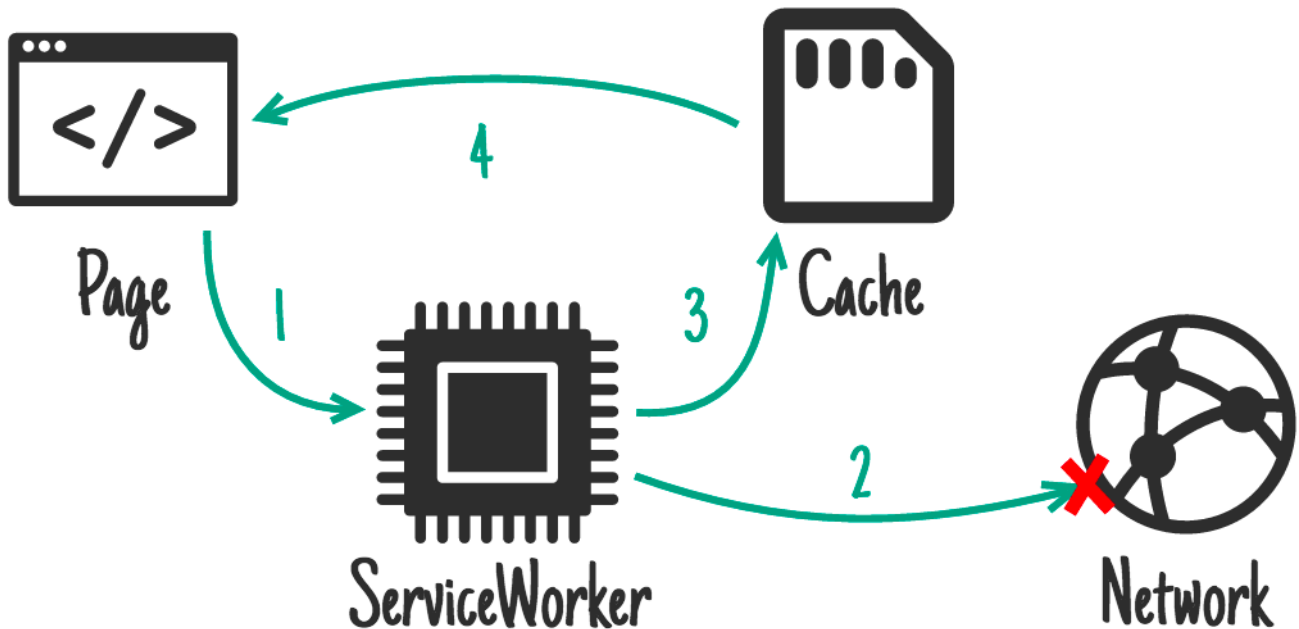
// Promise.race is no good to us because it rejects if
// a promise rejects before fulfilling. Let's make a proper
// race function:
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    // make sure promises are all promises
    promises = promises.map(p => Promise.resolve(p));
    // resolve this promise as soon as one resolves
    promises.forEach(p => p.then(resolve));
    // reject if all promises reject
    promises.reduce((a, b) => a.catch(() => b))
      .catch(() => reject(Error("All failed")));
  });
};

self.addEventListener('fetch', function(event) {
  event.respondWith(
    promiseAny([
      caches.match(event.request),
      fetch(event.request)
    ])
  )
});
  
```



```
);  
});
```

Network falling back to cache



Ideal for: A quick-fix for resources that update frequently, outside of the "version" of the site. E.g. articles, avatars, social media timelines, game leader boards.

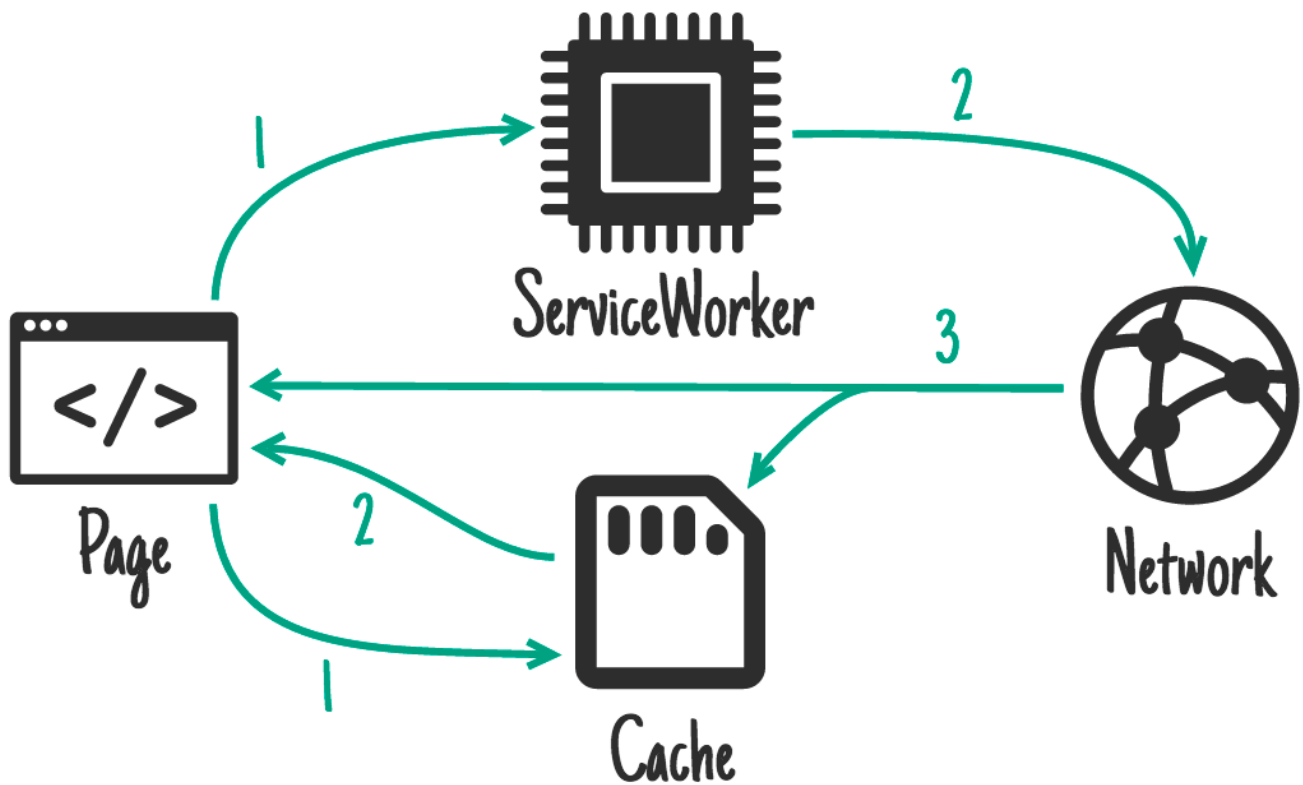
This means you give online users the most up-to-date content, but offline users get an older cached version. If the network request succeeds you'll most-likely want to update the cache entry.

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get the perfectly acceptable content already on their device. This can take an extremely long time and is a frustrating user experience. See the next pattern, Cache then network, for a better solution.

```
self.addEventListener('fetch', function(event) {  
  event.respondWith(  
    fetch(event.request).catch(function() {  
      return caches.match(event.request);  
    })  
  );  
});
```



Cache then network



Ideal for: Content that updates frequently. E.g. articles, social media timelines, game leaderboards.

This requires the page to make two requests, one to the cache, one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

Sometimes you can just replace the current data when new data arrives (e.g. game leaderboard), but that can be disruptive with larger pieces of content. Basically, don't "disappear" something the user may be reading or interacting with.

Twitter adds the new content above the old content & adjusts the scroll position so the user is uninterrupted. This is possible because Twitter mostly retains a mostly-linear order to content. I copied this pattern for trained-to-thrill to get content on screen as fast as possible, but still display up-to-date content once it arrives.

Code in the page:

```
var networkDataReceived = false;

startSpinner();

// fetch fresh data
var networkUpdate = fetch('/data.json').then(function(response) {
  return response.json();
}).then(function(data) {
  networkDataReceived = true;
```




```

    updatePage(data);
  });

  // fetch cached data
  caches.match('/data.json').then(function(response) {
    if (!response) throw Error("No data");
    return response.json();
  }).then(function(data) {
    // don't overwrite newer network data
    if (!networkDataReceived) {
      updatePage(data);
    }
  }).catch(function() {
    // we didn't get cached data, the network is our last hope:
    return networkUpdate;
  }).catch(showErrorMessage).then(stopSpinner);

```

Code in the ServiceWorker:

We always go to the network & update a cache as we go.

```

self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return fetch(event.request).then(function(response) {
        cache.put(event.request, response.clone());
        return response;
      });
    })
  );
});

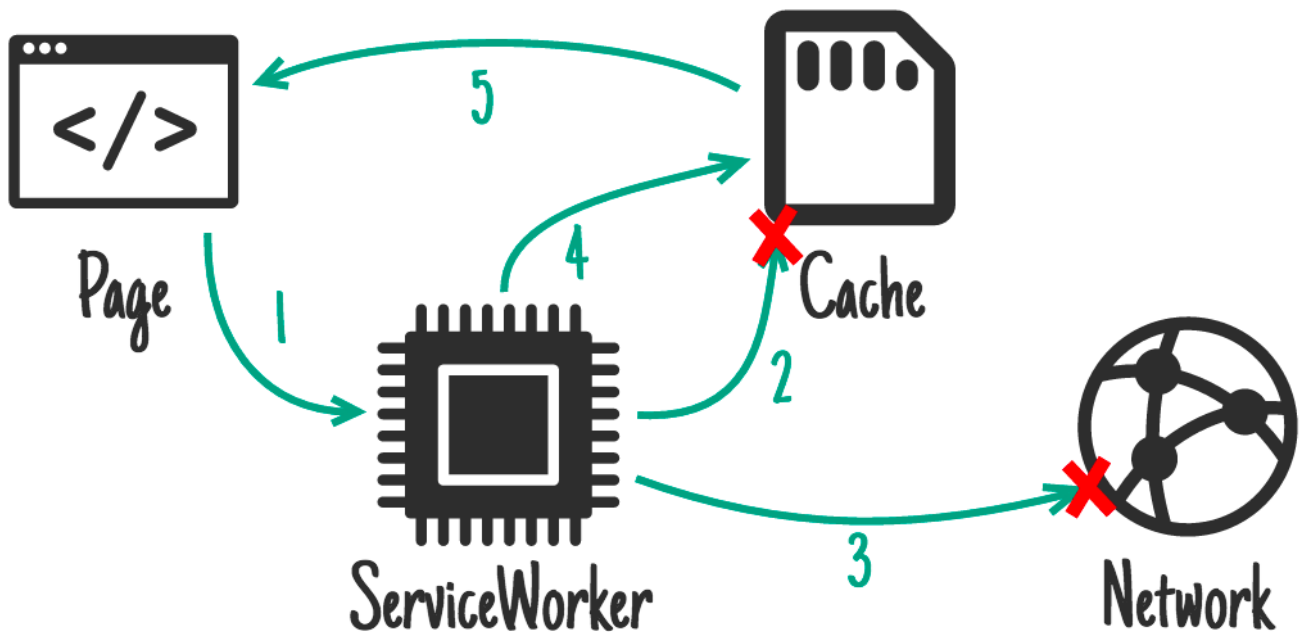
```



Note: The above doesn't work in Chrome yet, as we've yet to expose **fetch** and **caches** to pages ([ticket #1](#), [ticket #2](#)).

In [trained-to-thrill](#) I worked around this by using XHR instead of fetch, and abusing the Accept header to tell the ServiceWorker where to get the result from (page code, ServiceWorker code).

Generic fallback



If you fail to serve something from the cache and/or network you may want to provide a generic fallback.

Ideal for: Secondary imagery such as avatars, failed POST requests, "Unavailable while offline" page.

```

self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
      // However, in reality you'd have many different
      // fallbacks, depending on URL & headers.
      // Eg, a fallback silhouette image for avatars.
    })
  );
});

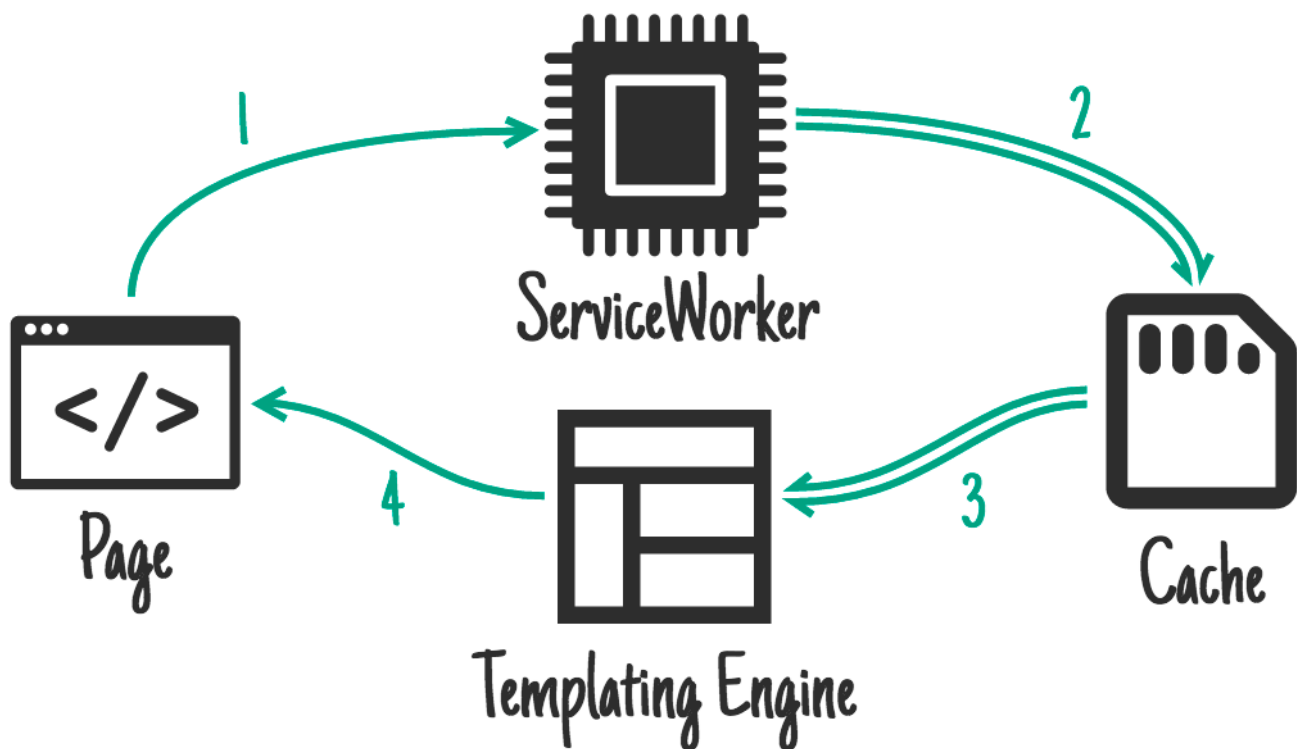
```



The item you fallback to is likely to be an install dependency.

If your page is posting an email, your ServiceWorker may fall back to storing the email in an IDB 'outbox' & respond letting the page know that the send failed but the data was successfully retained.

ServiceWorker-side templating



Ideal for: Pages that cannot have their server response cached.

Rendering pages on the server makes things fast, but that can mean including state data that may not make sense in a cache, e.g. "Logged in as...". If your page is controlled by a ServiceWorker, you may instead choose to request JSON data along with a template, and render that instead.

```
importScripts('templating-engine.js');

self.addEventListener('fetch', function(event) {
  var requestURL = new URL(event.request.url);

  event.respondWith(
    Promise.all([
      caches.match('/article-template.html').then(function(response) {
        return response.text();
      }),
      caches.match(requestURL.path + '.json').then(function(response) {
        return response.json();
      })
    ]).then(function(responses) {
      var template = responses[0];
      var data = responses[1];

      return new Response(renderTemplate(template, data), {
        headers: {
          'Content-Type': 'text/html'
        }
      });
    })
  );
});
```



```

    }
  });
})
);
});

```

Putting it together

You don't have to pick one of these methods, you'll likely use many of them depending on request URL. For example, [trained-to-thrill](#) uses:

- [Cache on install](#), for the static UI and behaviour
- [Cache on network response](#), for the Flickr images and data
- [Fetch from cache, falling back to network](#), for most requests
- [Fetch from cache, then network](#), for the Flickr search results

Just look at the request and decide what to do:

```

self.addEventListener('fetch', function(event) {
  // Parse the URL:
  var requestURL = new URL(event.request.url);

  // Handle requests to a particular host specifically
  if (requestURL.hostname == 'api.example.com') {
    event.respondWith(/* some combination of patterns */);
    return;
  }
  // Routing for local URLs
  if (requestURL.origin == location.origin) {
    // Handle article URLs
    if (/^\/article\/.test(requestURL.pathname)) {
      event.respondWith(/* some other combination of patterns */);
      return;
    }
    if (/\.webp$/ .test(requestURL.pathname)) {
      event.respondWith(/* some other combination of patterns */);
      return;
    }
    if (request.method == 'POST') {
      event.respondWith(/* some other combination of patterns */);
      return;
    }
    if (/cheese/.test(requestURL.pathname)) {
      event.respondWith(

```



```
        new Response("Flagrant cheese error", {
            status: 512
        })
    );
    return;
}

// A sensible default pattern
event.respondWith(
    caches.match(event.request).then(function(response) {
        return response || fetch(event.request);
    })
);
});
```

...you get the picture.

Feedback

Was this page helpful?

YES






NO




Great! Thank you for the feedback.

Sorry to hear that. Please [open an issue](#) and tell us how we can improve.

Credits

...for the lovely icons:

- [Code](#)  by buzzyrobot
- [Calendar](#)  by Scott Lewis
- [Network](#) by  Ben Rizzo
- [SD](#) by Thomas Le Bas
- [CPU](#)  by iconsmind.com
- [Trash](#)  by trasnik

- [Notification](#)  by @daosme
- [Layout](#)  by Mister Pixel
- [Cloud](#)  by P.J. Onori

And thanks to [Jeff Posnick](#) for catching many howling errors before I hit "publish".

Further reading

- [ServiceWorkers - an Introduction](#)
- [Is ServiceWorker ready?](#) - track the implementation status across the main browsers
- [JavaScript Promises - an Introduction](#) - guide to promises

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 24, 2018.