Serving

- The fetch event
- Responding to a request
  - Creating responses
  - Responding from the cache
- Caching strategies
  - Cache first
  - Network first
  - Stale while revalidate
  - Network only
  - Cache only
  - Custom strategies
- Updating assets
- Resources

A key aspects of Progressive Web Apps is that they're reliable; they can load assets quickly, keeping users engaged and providing feedback immediately, even under poor network conditions. How is that possible? Thanks to the service worker `fetch` event.

The fetch event

{% BrowserCompat 'api.FetchEvent' %}

The `fetch` event lets us intercept every network request made by the PWA in the service worker's scope, for both same-origin and cross-origin requests. In addition to navigation and asset requests, fetching from an installed service worker allows page visits after a site's first load to be rendered without network calls.

The `fetch` handler receives all requests from an app, including URLs and HTTP headers, and lets the app developer decide how to process them.

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/WExZeP8vV28bPDBlpxuG.png", alt="The service worker sits between the client and the network.", width="800", height="439" %}

Your service worker can forward a request to the network, respond with a previously cached response, or create a new response. The choice is yours. Here's a simple example:

```
self.addEventListener("fetch", event => {
    console.log(`URL requested: ${event.request.url}`);
});
```

{% Aside 'caution' %} While you can have more than one fetch event handler per service worker, only one can respond per request. For example, you can have different handlers that will act on different URL patterns. They are executed in the order they were registered until one of them calls `respondWith()`. {% endAside %}

Responding to a request

When a request comes into your service worker, there are two things you can do; you can ignore it, which lets it go to the network, or you can respond to it. Responding to requests from within your service worker is how you're able to choose what, and how it gets returned to your PWA, even when the user is offline.

To respond to an incoming request, call `event.respondWith()` from within a `fetch` event handler, like this:

```
// fetch event handler in your service worker file
self.addEventListener("fetch", event => {
    const response = .... // a response or a Promise of response
    event.respondWith(response);
});
```

You must call `respondWith()` synchronously and you must return a Response object. But you can't call `respondWith()` after the fetch event handler has finished, like within an async call. If you need to wait for the complete response, you can pass a Promise to `respondWith()` that resolves with a Response.

Creating responses

Thanks to the Fetch API, you can create HTTP responses in your JavaScript code, and those responses can be cached using the Cache Storage API and returned as if they were coming from a

web server.

To create a response, create a new `Response` object, setting its body and options such as status and headers:

```
const simpleResponse = new Response("Body of the HTTP response");

const options = {
   status: 200,
   headers: {
'Content-type': 'text/html'
   }
};
const htmlResponse = new Response("<b>HTML</b> content", options)
```

{% Aside 'warning' %} Be extra careful when creating responses with a service worker buggy code can lead to corner cases and hard to debug issues, such as the browser being unable to load the HTML document that renders your PWA. While debugging, you can always use your browser's reload button with the Shift key pressed, which will bypass the service worker; or use service worker devtools on supported browsers as we'll see in Tools and Debug. {% endAside %}

{% Glitch 'learn-pwa-serving-synthesize' %}

Responding from the cache

Now that you know how to serve HTTP responses from a service worker, it's time to use the Caching Storage interface to store assets on the device.

You can use the cache storage API to check if the request received from the PWA is available in the cache, and if it is, respond to `respondWith()` with it. To do that, you first need to search within the cache. The `match()` function, available at the top-level `caches` interface searches all stores in your origin, or on a single open cache object.

The `match()` function receives an HTTP request or a URL as an argument and it returns a promise that resolves with the Response associated with the corresponding key.

```
// Global search on all caches in the current origin
caches.match(urlOrRequest).then(response => {
   console.log(response ? response : "It's not in the cache");
});

// Cache-specific search
```

```
caches.open("pwa-assets").then(cache => {
  cache.match(urlOrRequest).then(response) {
    console.log(response ? response : "It's not in the cache");
  }
});
```

{% Aside 'gotchas' %} When using `match()` with `respondWith()`, the match promise will resolve either with the found item or with nothing. It will only reject if there was a problem that prevented the search operation from completing. To throw an error if no results are found, you'll need to inspect the results before returning a response to `respondWith()`. {% endAside %}

Caching strategies

Serving files only from the browser cache doesn't fit every use case. For example the user or the browser can evict the cache. That's why you should define your own strategies for delivering assets for your PWA. You're not restricted to one caching strategy. You can define different ones for different URL patterns. For example, you can have one strategy for the minimum UI assets, another for API calls, and a third for image and data URLs. To do this, read `event.request.url` in `ServiceWorkerGlobalScope.onfetch` and parse it through regular expressions or a URL Pattern. (At the time of writing, URL Pattern is not supported on all platforms).

The most common strategies are:

Cache First : Searches for a cached response first and falls back to the network if one isn't found.

Network First : Requests a response from the network first and if none is returned, checks for response in the cache.

Stale While Revalidate : Serves a response from the cache, while in the background requests the latest version and saves it to the cache for the next time the asset is requested.

Network-Only : Always replies with a response from the network or errors out. The cache is never consulted.

Cache-Only : Always replies with a response from the cache or errors out. The network will never be consulted. The assets that will be served using this strategy must be added to the cache before they are requested.

Cache first

Using this strategy, the service worker looks for the matching request in the cache and returns the corresponding Response if it's cached. Otherwise it retrieves the response from the network

(optionally, updating the cache for future calls). If there is neither a cache response nor a network response, the request will error. Since serving assets without going to the network tends to be faster, this strategy prioritizes performance over freshness.

{% Aside %} Remember the cache storage interface won't return a Response object. The reference will be `undefined`, if the request is not cached. {% endAside %}

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/ki9e4T6ik7mfrN67Zysx.png", alt="The Cache First strategy", width="800", height="439" %}

```
self.addEventListener("fetch", event => {
   event.respondWith(
     caches.match(event.request)
     .then(cachedResponse => {
      // It can update the cache to serve updated content on the next request
          return cachedResponse || fetch(event.request);
     }
   )
  )
});
```

{% Glitch 'learn-pwa-serving-offline' %}

Network first

This strategy is the mirror of the Cache First strategy; it checks if the request can be fulfilled from the network and, if it can't, tries to retrieve it from the cache. Like cache first. If there is neither a network response nor a cache response, the request will error. Getting the response from the network is usually slower than getting it from the cache, this strategy prioritizes updated content instead of performance.

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/bGCR5LiMRlt9urRcpnk4.png", alt="The Network First strategy", width="800", height="439" %}

```
self.addEventListener("fetch", event => {
   event.respondWith(
     fetch(event.request)
     .catch(error => {
       return caches.match(event.request) ;
     })
   );
```

```
});
```

Stale while revalidate

The stale while revalidate strategy returns a cached response immediately, then checks the network for an update, replacing the cached response if one is found. This strategy always makes a network request, because even if a cached resource is found, it will try to update what was in the cache with what was received from the network, to use the updated version in the next request. This strategy, therefore, provides a way for you to benefit from the quick serving of the cache first strategy and update the cache in the background.

{% Aside %} With this strategy, the assets are updated in the background. This means your users won't get the new version of the assets until the next time that path is requested. At that time the strategy will again check if a new version exists and the cycle repeats. {% endAside %}

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/C7OcnPRHNdTLe2043JHG.png", alt="The stale while revalidate strategy", width="800", height="439" %}

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request).then(cachedResponse => {
        const networkFetch = fetch(event.request).then(response => {
          // update the cache with a clone of the network response
          const responseClone = response.clone()
          caches.open(url.searchParams.get('name')).then(cache => {
            cache.put(event.request, responseClone)
          })
          return response
        }).catch(function (reason) {
          console.error('ServiceWorker fetch failed: ', reason)
        })
        // prioritize cached response over network
        return cachedResponse || networkFetch
      }
    )
  )
})
```

{% Aside 'caution' %} The Response's body is a ReadableStream that can only be consumed once. That means that if you call `fetch()` to deliver the response to the PWA because the asset was not cached, you cannot reuse the same response for updating it in the cache. You need to call

`Response.clone()` to use it twice. {% endAside %}

{% Glitch 'learn-pwa-serving-stale-while-revalidate' %}

Network only

The network only strategy is similar to how browsers behave without a service worker or the Cache Storage API. Requests will only return a resource if it can be fetched from the network. This is often useful for resources like online-only API requests.

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/NGxIKl6Lq8TY9GqXT8r8.png", alt="The Network only strategy", width="800", height="439" %}

Cache only

The cache only strategy ensures that requests never go to the network; all incoming requests are responded to with a pre-populated cache item. The following code uses the `fetch` event handler with the `match` method of the cache storage to respond cache only:

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

{% Aside %} With the cache only strategy you cache your assets before you need them, usually in the service worker's `install` event handler. Because this strategy never goes to the network, these assets will get updated only when you update your service worker and a new `install` event is fired. {% endAside %}

{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/rDI3LvI9y0SBUFfRCdgm.png", alt="Cache only strategy.", width="800", height="439" %}

{% Glitch 'learn-pwa-serving-cache-only' %}

Custom strategies

While the above are common caching strategies, you are in charge of your service worker and how requests are handled. If none of these work for your needs, create your own.

You can, for example, use a network first strategy with a timeout to prioritize updated content, but only if the response appears within a threshold you set. You can also merge a cached response with a network response and build a complex response from the service worker.

Updating assets

Keeping your PWA's cached assets up-to-date can be a challenge. While the stale while revalidate strategy is one way to do so, it's not the only one. In the Update chapter you will learn different techniques to keep your app's content and assets updated.

Resources

- Fetch event on MDN
- The Offline Cookbook
- Cache Match on MDN

□□□ #2
□ □□□ □□□ 2026-10-31 22:55:57 UTC
□ □□□ □□□ 2026-10-31 22:55:59 UTC