PHILIP WALTON
Engineer @ Google

Untangling Deeply-Nested Promise Chains

August 17, 2016

If you've been writing JavaScript for a while, you've probably heard terms like callback hell or the pyramid of doom. When promises were added to JavaScript a few years ago, I remember reading a lot of blog posts claiming that these problems would be solved; unfortunately, that was a little too optimistic. With more and more web APIs becoming promise-based, we've proven that even promises don't prevent us from writing overly-nested, hard-to-read code.

The place I'm seeing this happen a lot these days is in Service Worker scripts that are heavily promise-based. Specifically, blog posts and tutorials that are supposed to be showing us the proper way to use Service Worker.

The following code is a so-called "basic" example of how use Service Worker to implement a *network-first with cache fallback* strategy for offline support. This exact code (or a variation of it) can be found in numerous tutorials on the web:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('cache:v1').then((cache) => {
      return fetch(event.request).then((response) => {
        cache.put(event.request, response.clone());
        return response;
      }).catch(() => {
        return cache.match(event.request).then((response) => {
          return response || Response.error();
        });
      });
    })
  );
});
```

While I suppose this code could be considered "basic" in the sense that it's only a few lines and the strategy is conceptually simple, I'd argue that the control flow makes it anything but basic. Unless you're *very* familiar with the new `Fetch` and `CacheStorage` APIs, and unless you have a solid understanding of all the nuances of promises, it's probably going to take you a few reads to really grok how this code works.

Speaking from personal experience, about a week ago I started playing around with Service Worker for the first time: I wanted to add basic caching and offline analytics to this site. But after completing my initial implementation, I was pretty unsatisfied with the code I'd written. It wasn't clear, it wasn't self-documenting, and it seemed far too complex for the relatively simple problem I was trying to solve.

After spending a few hours refactoring and really exploring these new APIs, I came up with a few strategies for improving the readability of my code that I wanted to share. The strategies range from general software development tips and best-practices to leveraging new JavaScript language features like async functions.

Note: my intent here is not to critique people who have written Service Worker tutorials. I've learned a great deal from that content, and I think it's invaluable. I also understand that when writing blog posts, concise examples often have a place.

This article is primarily meant to encourage readers of such tutorials to make sure their own implementations are readable and maintainable; to resist the urge to simply copy and paste boilerplate examples without fully understanding them.

What the code is doing

Before I start talking about how to improve this code, I want to make sure everyone reading is very clear on exactly what this code is doing.

When adding a `fetch` event listener to a Service Worker, you typically call `event.respondWith` and pass a promise that resolves to a `Response` object. But when you pass a promise chain like in the above code example (with multiple levels of nested `then` and `catch` calls), it can be pretty hard to see all the points at which the resolution can happen.

Here is a step-by-step explanation, in plain English, of what happens inside the `fetch` event handler in the code example above:

1. The `event` object calls `respondWith()` and passes it a promise chain that will eventually resolve to a `Response` object.
2. The promise chain starts by opening the cache called: `cache:v1`.
3. Once the cache is open, it makes a `fetch()` over the network for a request object specified by `event.request`.
4. If the `fetch()` succeeds:
    a. It puts a copy of the network response in the cache.
    b. It resolves the promise with the network response.
5. If the `fetch()` fails:

    a. It attempts to find a matching request in the cache.
    b. If a match is found:
        i. It resolves the promise with the cached response.
    c. If a match is not found:
        i. It resolves the promise with a generic `Response.error()` object.

I mentioned above that `respondWith()` takes a promise that eventually resolves to a `Response` object. In the above logic outline, that resolution could happen in three different places: `4.b`, `5.b.i`, or `5.c.i`.

How the code could be improved

Each of the follow sections introduces a technique or a principle to help make your code more readable and ultimately easier for you (and others) to work with in the future. The techniques start out simple and get more complex as they go—each one building on the previous one.

Give variables more descriptive names

When writing Service Worker code you'll find yourself dealing with a lot of `Request` and `Response` objects, and it can be tempting to just use those names with every occurrence of these objects in your code. And if each `Request` or `Response` object appears in its own distinct scope, there isn't a technical reason to name them anything else.

However, in the case of our Service Worker `fetch` example, there is a distinct fork in the logic that could result in two very different types of responses: a network response or a cache response.

An easy way to indicate to a reader which condition they're in is to give each `Response` object a name specific to that condition:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('cache:v1').then((cache) => {
      return fetch(event.request).then((networkResponse) => {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(() => {
        return cache.match(event.request).then((cacheResponse) => {
          return cacheResponse || Response.error();
        });
      });
    })
  );
});
```

Avoid code that looks like it might be a mistake

If you look at the last five lines of our original code example, here's what you see. Notice that there's no semicolon at the end of the middle line:

```
          // ...
        });
      });
    })
  );
});
```

When I first saw this, I assumed it was simply an omission, but I was wrong.

In reality, the expression starting with `caches.open()` was moved to its own line and indented—presumably to avoid having too much logic on a single line.

While I support the desire to break the code into more manageable chunks, doing it this way just lead to something that looked like a mistake (which is confusing).

If you have an expression that's so long/complex that you feel the need to visually separate it, why not *actually* separate it: evaluate the expression elsewhere and assign the result to a variable.

```
self.addEventListener('fetch', (event) => {
  const networkOrCacheResponse = caches.open('cache:v1').then((cache) => {
    return fetch(event.request).then((networkResponse) => {
      cache.put(event.request, networkResponse.clone());
      return networkResponse;
    }).catch(() => {
      return cache.match(event.request).then((cacheResponse) => {
        return cacheResponse || Response.error();
      });
    });
  });

  event.respondWith(networkOrCacheResponse);
});
```

Abstract logical units into single-purpose functions

Assigning the result of a complex expression to a variable prior to passing it to another method is good for readability, but we can still do better.

As it stands in the above code example, the `networkOrCacheResponse` object can only be used inside this particular `fetch` handler. If you wanted to use the *network-first with cache fallback* strategy elsewhere, you'd have to rewrite the logic.

To solve this problem we can write a utility function that accepts a `Request` object and returns a promise that will resolve to either the network or cache response to that request. Such a function might look like this:

```
const getNetworkOrCacheResponse = (request) => {
  return caches.open('cache:v1').then((cache) => {
    return fetch(request).then((networkResponse) => {
      cache.put(request, networkResponse.clone());
      return networkResponse;
    }).catch(() => {
      return cache.match(request).then((cacheResponse) => {
        return cacheResponse || Response.error();
      });
    });
  });
};

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

While this function is now more reusable, it's still a bit complex.

You'll notice there are multiple levels of nested promises, and whenever you see a lot of nesting in a function, it's usually a sign that the function is doing too much. In other words, it has too many responsibilities.

The `getNetworkOrCacheResponse` function contains logic that deals with two very separate concerns:

- Making the network request.
- Interacting with the cache storage.

To improve readability, we can abstract the cache-related logic into separate, self-contained functions:

```
const addToCache = (request, networkResponseClone) => {
  return caches.open('cache:v1')
    .then((cache) => cache.put(request, networkResponseClone));
}

const getCacheResponse = (request) => {
  return caches.open('cache:v1').then((cache) => {
    return cache.match(request);
  });
}

const getNetworkOrCacheResponse = (request) => {
  return fetch(request).then((networkResponse) => {
    addToCache(request, networkResponse.clone());
    return networkResponse;
  }).catch(() => {
    return getCacheResponse(request)
      .then((cacheResponse) => cacheResponse || Response.error());
  });
};

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

If you compare this new logic to the logic in the original function, you'll notice one significant difference. In the original code, the first thing to happen was a call to `caches.open()`, and that call only happened once. In the refactored code, there is a call to `caches.open()` in each of the utility functions that directly deals with the cache.

While at first it might seem like this abstraction will end up doing more work, it's actually an optimization over the original code—one I only discovered *after* I started separating the concerns.

Consider the case where the Service Worker makes a network request that is successful. This is the most common case, so we should optimize for it (i.e. get the response to the user as quickly as possible). In the original code, the first step of the logic was to open the cache, and then only respond with a network request once the cache was open.

This is absolutely not necessary. While it's true that we need to write to the cache even in the network case, this write doesn't need to block the response to the user. It can easily happen in parallel.

If you're wondering why it's important to write functions with only a single responsibility, there are two basic reasons:

- The function is more reusable. (The more responsibilities a function has, the more specific a function is to a particular use-case.)
- The function is easier to test. (N functions that each do one thing can be tested with N tests. By contrast, 1 function that does N things usually needs to account for N! possible outcomes.)

Use async functions to remove nesting entirely

Perhaps the most significant way to improve the readability of complex promise code is to use async functions, a new JavaScript feature that helps make asynchronous logic read more like synchronous code.

Async functions are declared with the new `async` keyword, and instead of returning a value immediately, they return a promise that will eventually resolve to the function's return value.

Within the body of an async function you'll usually find one or more `await` keywords. The `await` keyword is prepended to a promise (or an expression that evaluates to a promise) and when the interpreter encounters the `await` keyword, it halts the execution of the function until the promise is resolved. Once the promise is resolved, the awaited expression "returns" that value.

To make this more clear, consider the `getNetworkOrCacheResponse()` function defined in the section above. Here's how that function would look as an async function:

```
const getNetworkOrCacheResponse = async (request) => {
  try {
    const networkResponse = await fetch(request);
    addToCache(request, networkResponse.clone());
    return networkResponse;
  } catch (err) {
    const cacheResponse = await getCacheResponse(request);
    return cacheResponse || Response.error();
  }
}
```

There are several important things to notice about the `async` version of this function:

- You can use a `try` / `catch` block instead of `.then()` / `.catch()` chains, and the error is handled the way it normally is in a `try` / `catch` block.
- Since `await` expressions halt execution and evaluate to a `Promise`, what was originally multiple levels of nesting can now be represented as successive, top-level assignment expressions (with no nesting).

- Since an `async` function is sugar for a promise that resolves to its `return` value, it makes it much more clear where the promise resolution happens. With nested promise chains, that resolution can be much more obscured.

These improvements are substantial! They make your code much easier to both read and write.

Using async functions today

Async functions have been around for a while now, and you can compile them to ES5 using Babel. However, in order to use that compiled code you *also* have to include the `babel-polyfill` library (which bundles Facebook's regenerator runtime).

While this might make sense for some projects, the added code weight of the polyfill is an unacceptably high cost for a 50-line Service Worker script (in my testing it added ~60K).

Fortunately, when it comes to Service Worker scripts, there's another way!

Since all browsers that support Service Worker *also* support most ES2015 features (specifically, they support generators), you can avoid the regenerator runtime and compile your code with just a single Babel transform: `async-to-generator`.

In my Service Worker code on this site, using the `async-to-generator` transform only added an additional 258 bytes, and since I was already using browserify to load the dependencies, using async functions was a no-brainer!

Wrapping up

With all those improvements in place, compare the readability of the original code to the refactored code.

Original code:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('cache:v1').then((cache) => {
      return fetch(event.request).then((response) => {
        cache.put(event.request, response.clone());
        return response;
      }).catch(() => {
        return cache.match(event.request).then((response) => {
          return response || Response.error();
        });
```

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

Refactored code:

```
const addToCache = async (request, networkResponseClone) => {
  const cache = await caches.open('cache:v1');
  return cache.put(request, networkResponseClone);
};

const getCacheResponse = async (request) => {
  const cache = await caches.open('cache:v1');
  const cachedResponse = await cache.match(request);
  return cachedResponse;
};

const getNetworkOrCacheResponse = async (request) => {
  try {
    const networkResponse = await fetch(request);
    addToCache(request, networkResponse.clone());
    return networkResponse;
  } catch (err) {
    const cacheResponse = await getCacheResponse(request);
    return cacheResponse || Response.error();
  }
};

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

While the refactored version is longer and contains more code, it's definitely easier to read and understand, which means it will be easier to update in the future, both for you and others. It's also more modular, which will make it easier to test and reuse in other contexts.

This article introduced several concepts and strategies to help you refactor complex, promise-based code into individual, reusable parts. Hopefully some of the techniques introduced were helpful; at a minimum, I hope I've encouraged you to strive to make your code as readable and maintainable as possible.

If you liked this article and think others should read it, please share it on Twitter.