Don’t cross the streams

Streams are a very interesting concept and a new set of tools for the web. The idea is that we can read and write, depending on the type of stream we’re using, chunks of content… either write them to a location or read them from a location. This will improve performance because we can start showing things to the user before it has completed loading.

The example below how we can asynchronously download and display content to the user. The problem with this, if you can call it that, is that fetch will wait to download the entire file before settling the promise and only then will populate the content into the page.

```
const url = 'https://jsonplaceholder.typicode.com/photos';
const response = await fetch(url);
document.body.innerHTML = await response.text();
```

Streams seek to provide a better way to fetch content and display it to the user. The content gets to the browser first and we can then render it to the user as it arrives rather than have to wait for all the content to arrive before display.

The example below does the following:

1. Fetches the specified resource
2. Creates a reader from the body of the response object
3. Creates a readable stream
4. In the reader’s start menu we create a push function to do the work and read the first chunk of the stream
5. We create a TextDecoder that will convert the value of the chunk from Uint8 to text
6. If we hit done it’s because there are no more chunks to read so we close the controller and return
7. Enqueue means we add the chunk we read to the stream and then we append the decoded string to the page
8. We call the function again to continue processing the stream until there are no more chunks to read and done returns true
9. We return a new response with the stream as the value and a new `Content-Type` header to make sure it’s served as HTML

```
fetch("https://jsonplaceholder.typicode.com/photos").then((response) => {
const reader = response.body.getReader(); // 2
const stream = new ReadableStream({ //3
  start(controller) {
    function push() {
      reader.read().then(({ done, value }) => { // 4

        let string = new TextDecoder("utf-8").decode(value); "utf-8"// 5

        if (done) { // 6
          controller.close();
          return;
        }
        controller.enqueue(value); // 7
        document.body.innerHTML += string;
        push()
      });
    };
    push(); // 8
  }
});

  return new Response(stream, { // 9aders: {
      "Content-Type": "text/html"
    }
  });
});
"Content-Type"
```

This reader becomes more powerful the larger the document we feed it is.

Creating my own streams

The example above also illustrates some of the functions and methods of `ReadableStream` and `controller`. The syntax looks like this and we're not required to use any of the methods.

```
let stream = new ReadableStream({
  start(controller) {},
  pull(controller) {},
  cancel(reason) {}
}, queuingStrategy);
```

- `start` is called immediately. Use this to set up any underlying data sources (meaning, wherever you get your data from, which could be events, another stream, or just a variable like a string). If you return a promise from this and it rejects, it will signal an error through the stream
- `pull` is called when your stream's buffer isn't full and is called repeatedly until it's full. Again, If you return a promise from this and it rejects, it will signal an error through the stream. Pull will not be called again until the returned promise fulfills
- `cancel` is called if the stream is canceled. Use this to cancel any underlying data sources
- `queuingStrategy` defines how much this stream should ideally buffer, defaulting to one item. Check the spec for more information

And the controller has the following methods:

- `controller.enqueue(whatever)` - queue data in the stream's buffer.
- `controller.close()` - signal the end of the stream.
- `controller.error(e)` - signal a terminal error.
- `controller.desiredSize` - the amount of buffer remaining, which may be negative if the buffer is over-full. This number is calculated using the queuingStrategy.

Streams and service workers

Where I see the biggest benefit of using streams is in fetching resources for a service worker. It'll make it faster to download a resource using streams and it'll give a consistent and fast user experience using cached resources from the service worker.

This is the outline of what I need to do:

1. Create a function to stream the content
2. If the URL is for an HTML file then

        1. If it’s not cache follow these steps:
            1. Fetch the head of the page from the cache
            2. Fetch the body of the page from the network
            3. Catch any errors here
            4. Fetch the footer of the page from the cache
            5. Combine the resources into a single stream
        2. Clone the stream and cache the clone
        3. Return the original stream to the page for display
    3. Otherwise respond with a default network first strategy

Workbox

We’ll first look at streaming content using Workbox.js and the `workbox.streams` plugin.

We cache the `page-start` and `page-end` by adding them to our configuration file. This will precache the assets in addition to the other assets we precache for the sample site. The configuration modules looks like this:

```
module.exports = {
  'globDirectory': 'docs',
  'globPatterns': [
    '/',
    'css/index.css',
    'js/zenscroll.min.js',
    'pages/404.html',
    'pages/offline.html',
    'pages/page-start.inc',
    'pages/page-end.inc',
  ],
  'swDest': 'sw.js',
  'swSrc': 'js/sw.js',
};
```

The `globDirectory` constant holds the location of the files we want to work with.

`globPatterns` list the locations of the individual files we want to precache.

`swSrc` indicates the location of the service worker template and `swDest` indicates where the file should be written after Workbox inserts the files to precache.

We first define constants for the elements of the page that we have already precached. We do this to make typing easier later in the script.

```
const HEAD = workbox.precaching.getCacheKeyForURL('partials/page-start.ht
const FOOT = workbox.precaching.getCacheKeyForURL('partials/page-end.html
const ERROR = workbox.precaching.getCacheKeyForURL('partials/404.html');
```

We now create two strategies: one for the precached content and one for the content we'll pull from the network.

We have to register the precache strategy so that Workbox will know where to pull the pieces from. It's not enough to precache them.

The strategy for the content is what we're familiar with. A name for the cache and any plugins that we want to use.

```
const cacheStrategy = new workbox.strategies.CacheFirst({
  cacheName: workbox.core.cacheNames.precache,
});

const networkStrategy = new workbox.strategies.StaleWhileRevalidate({
  cacheName: 'content',
  plugins: [
    new workbox.expiration.Plugin({
      maxEntries: 50,
    }),
  ],
});
```

The route that will stitch the content together is fairly simple.

First we make a request for the HEAD partial that will come from the cache.

Next we make an async request for the body of the page from the network. If we succeed then we return the body of the request as text. If we fail then we make a request for the ERROR partial.

Finally we make a request for the FOOTER template that will also be fetched from the precache.

The fact we're stitching our pages together doesn't mean we don't have to pprovide routes for other assets. The page works but it keeps reminding me that there are no routes defined for other assets we reference in the header and footer.

```
workbox.routing.registerRoute(
    new RegExp('\\.html$'),
    workbox.streams.strategy([
      () => cacheStrategy.makeRequest({
        request: HEAD,
      }),
      async ({event}) => {
        try {
          const contentResponse = networkStrategy.makeRequest({
            request: event.request.url,
          });
          const contentData = await contentResponse.text();
          return contentData;
        } catch (error) {
          return cacheStrategy.makeRequest({
            request: ERROR,
          });
        }
      },
      () => cacheStrategy.makeRequest({
        request: FOOT,
      }),
    ])
);
```

I know what you're thinking… this is too much work for little returns. Consider that we're building smaller pages

The video below from provides a deeper dive on caching strategies and how to use streams with Workbox. It’s cued to start with Phil Walton’s

What’s next?

I’m still working on an equivalent implementation to the Workbox example using vanilla JavaScript. It’s taking longer than expected to wrap my head around the concept so I decided to take it out of this post and make its own later down the road.

Conclusion

The idea is to improve performance by caching the beginning and the end of the page. We then fetch the body of the page and provide an offline fallback if the network is not available.

This will provide a faster using experience by breaking the content into sections and only fetching the minimum necessary from the network, getting the rest from the cache.

Links and Resources

- Streams Living Standard
- Transform Streams
- A Guide to Faster Web App I/O and Data Operations with Streams

- workbox.streams reference
- Stream Demos
- Streams Reference Implementation
- Fetch API + Streams Live Demo
- Microsoft Developer Network document on Streams
- Jake Archibald
    - 2016 - the year of web streams
    - Streaming Template Literals
- MDN
    - Streams API concepts
    - ReadableStream
    - ReadableStream.pipeThrough()
    - WriteableStream
    - Pipe Streams
- Chrome Status
    - Streams API
    - Streams API: Piping
    - Streams API: Writable Stream