Streams Part 2

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. See if it's already in the cache
    2. If not fetch it
    3. If it's not then stream it
    4. Clone the stream and cache the clone
    5. Return the original stream to the page for display
3. Otherwise respond with a default cache first strategy

```
self.addEventListener('fetch', event => {
  let requestURL = new URL(event.request.url);
  if (requestURL.endsWith('text/html')) {

    let stre'text/html'adableStream({
      let contentFetch = fetch(requestURL)
        .catch(() => caches.match('/page-offline.html'));

      function pushStream(stream) {
        '/page-offline.html'// Get a lock on the stream
        let reader = stream.getReader();

        return reader.read().then(function process(result) {
          if (result.done) return;
          // Push the value to the combined stream
          controller.enqueue(result.value);
          // Read more & process
```

```
          return reader.read().then(process);
        });
      }
    }

  }

  // Get the start response
  contentFetch // 1
    // Push its contents to the combined stream
    .then(response => pushStream(response.body))
    .then(() => controller.close());
});
```

Composing content using streams and service workers

```
self.addEventListener('fetch', event => {
  let requestURL = new URL(event.request.url);
  let stream = new ReadableStream({
    start(controller) {
      // Get promises for response objects for each page part
      // The start and end come from a cache
      let startFetch = caches.match('/page-start.inc');
      let endFetch = caches.match('/page-end.inc');
      '/page-start.inc'// The middle comes from the network, with a fallba
        .catch(() => caches.match('/page-offline.inc'));

      function pushStream(stream) {
        // Get a lock on '/page-offline.inc'// Get a lock on the stream
       let reader = stream.getReader();

      return reader.read().then(function process(result) {
        if (result.done) return;
        // Push the value to the combined stream
```

```
        controller.enqueue(result.value);
        // Read more & process
        return reader.read().then(process);
      });
    } // Closes controller start

    // Get the start response
    startFetch
      // Push its contents to the combined stream
      .then(response => pushStream(response.body))
      // Get the middle response
      .then(() => middleFetch)
      // Push its contents to the combined stream
      .then(response => pushStream(response.body))
      // Get the end response
      .then(() => endFetch)
      // Push its contents to the combined stream
      .then(response => pushStream(response.body))
      // Close our stream, we're done!
      .then(() => controller.close());
    } // Closes startFetch
  }) // Closes ReadableStream

}); // closes fetch event listener
```

Creating my own streams

Service workers are not the only place where we can use streams. If we have the need for streams we can create our own readable streams using, at least, the following signature

```
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 cancelled. 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.

An example from Jake Archibald's blog uses a custom stream to generate random numbers. It touches on two methods of the `Readable Stream` interface (`start` and `cancel`) and two methods of `controller` (enqueue and `close`).

```
let interval;
let stream = new ReadableStream({
  start(controller) {
    interval = setInterval(() => {
      let num = Math.random();

      // Add the number to the stream
      controller.enqueue(num);

      if (num > 0.9) {
        // Signal the end of the stream
        controller.close();
        clearInterval(interval);
```

```
      }
    }, 1000);
  },
  cancel() {
    // This is called if the reader cancels,
    //so we should stop generating numbers
    clearInterval(interval);
  }
});
```

The idea is that once a second (1000 milliseconds) we generate a random number and add it to our stream queue. If the number is greater than 0.9 we signal the end of the stream and call it done.

Links and Resources

- Streams Living Standard
- Transform Streams
- Stream Demos
- Streams Reference Implementation
- Web Streams Polyfill