One of the biggest pains when working on the web is how to deal with large files (images, videos, large chunks of markup) without blocking the browser from doing other things like rendering whatever content is not part of the stream.

At I/O this year Surma and Paul Lewis did a live coding demo of streams and web components. This got me thinking about streams and how to best leverage them to increase performance of our applications.

What are streams and what are they good for

According to the WHATWG Streams Living Standard:

> This specification provides APIs for creating, composing, and consuming streams of data. These streams are designed to map efficiently to low-level I/O primitives, and allow easy composition with built-in backpressure and queuing. On top of streams, the web platform can build higher-level abstractions, such as filesystem or socket APIs, while at the same time users can use the supplied tools to build their own streams which integrate well with those of the web platform.

The idea behind streams is that they'll make things go faster by downloading smaller chunks of contents and displaying it or doing something with each chunk as it arrives. If we combine this with service workers it makes for an even faster user experience. Streams can also be used outside of Service Workers wherever you use Fetch.

It's important to note that the streams we're discussing here (and defined in a WHATWG Living Standard) are not the same as the node streams. There is a FAQ where these differences are discussed in some detail, for readable streams these are:

- An asynchronous `.read()` method, instead of async `.on("readable", ...)` plus synchronous `.read()`
- Addition of the exclusive reader and locking concepts, to better support off-main-thread piping
- Addition of cancelation semantics
- Addition of more precise flow control via the `desiredSize` signal

- Built-in teeing support
- Removal of the `"data"` event, which competes conceptually with other ways of reading
- Removal of pause/resume for managing backpressure
- Removal of the `unshift` method for putting chunks back into the queue after reading them
- No “binary/string mode” vs. “object mode” switch; instead, queueing strategies allow custom chunk types
- No optional and only sometimes-working size parameter while reading; instead use BYOB readers

First example

In the example below we fetch the JSON data for my blog using streams. Because it’s a non-blocking API it tells us after every chunk how much it has downloaded and when the download is completed we tell the user.

```
fetch('https://publishing-project.rivendellweb.net/wp-json/')
  .then(response => {
  var reader = response.body.getReader();
  var bytesReceived = 0;

  reader.read().then(function processResult(result) {
    if (result.done) {
      console.log("All Done!");
      return;
    }
    bytesReceived += resul"All Done!"gth;
    console.log(`Received ${bytesReceived} bytes of data so far`);

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

The idea is that the page that triggered the fetch will receive data as soon as possible and will continue to do so until the full transfer is complete.

By default the stream that uses is a `UintArray8` of binary data and the full response are all the `UintArray8` chunks joined together. If we want the response

as text we'll have to use `TextDecoder` and modify the fetch handler to look like the one below:

```
fetch('https://publishing-project.rivendellweb.net/wp-json/')
  .then(response => {
  const decoder = new TextDecoder();
  const reader = response.body.getReader();

  reader.read()
  .then(function processResult(result) {
    if (result.done) {
      console.log("All Done");
      return;
    }
    console.log(
        decoder.decode(result.value, {stream: true})
    );
    return reader.read().then(processResult);
  });
});
```

This decode will log each chunk to console as it arrives, not having to wait for all 160Kb of JSON to download... Win!

What's next?

I'm researching how to combine streams inside a service worker's fetch event. Its proving surprisingly harder than I expected to wrap my head around how to build such streams for a single file.

Rather than stop this post from going out I will continue researching how to build a streaming fetch even in service workers and post it as a separate article.

The article will also include building custom streams.