Why Oboe.js?

Oboe.js is a Javascript library that runs under Node.js and web browsers for loading JSON using streaming. It combines the convenience of DOM parsing with the speed and fluidity of SAX parsing.

This page looks at why that is a good thing. See also Oboe.js vs SAX vs DOM.

Downloading REST resources

Let's start by examining the standard pattern found on most AJAX-powered sites. We have a client-side web application and a service that it goes to for data. The page isn't updated until the response completes.

{{demo "fast-ajax-discrete"}}

On a good connection there isn't a huge amount of time to save but we can show data sooner and give a more responsive feel by using streaming.

{{demo "fast-ajax-progressive"}}

As the connection gets slower or the response gets larger the improvement is more significant.

Mobile data connections

Mobile networks today are high-bandwidth but can also be high-latency and come with inconsistent packet delivery times. This is why buffered content like streaming HD video plays fluidly but web surfing still feels laggy. The visualisation below approximates a medium-sized download on a mobile network.

{{demo "mobile-discrete"}}

Oboe.js makes it easy for the programmer to use chunks from the response as soon as they arrive. This helps webapps to feel faster when running over mobile networks.

{{demo "mobile-progressive"}}

The visualisation above shows how the data is displayed sooner. In itself, progressive display also improves the *perception* of performance.

Dropped connections

Oboe.js provides improved tolerance if a connection is lost before the response completes. Most AJAX frameworks equate a dropped connection with total failure and discard the partially transferred data, even if 90% transferred correctly.

We can handle this situation better by using the partially transferred data instead of throwing it away. From a streaming approach using this data follows naturally without requiring any extra programming.

In the next visualisation we have a mobile connection which fails when the user enters a building:

{{demo “mobile-fail-discrete”}}

Because Oboe.js views the HTTP response as a series of small, useful parts, when a connection is lost it is simply the case that some parts were successful and were used already, while others did not arrive. Fault tolerance follows naturally from this model and no special cases are required.

In the example below the client is smart enough so that when the network comes back it only requests the data that it missed on the first request:

{{demo “mobile-fail-progressive”}}

Aggregating resources

It is a common architectural pattern for web clients to retrieve their data through an aggreating middle tier. The aggregator connects to several back-end services and combines their data into a single response.

The visualisation below shows an example without streaming. *Origin 1* is slower than *Origin 2* but the system is forced to run at the speed of the slowest service:

{{demo “aggregated-discrete”}}

We can speed this scenario up by using Oboe.js to load data in the aggregator and the client. The aggregator dispatches the data as soon as it has it and the client displays the data as soon as it is arrives. In a Java stack this could also be implemented by using GSON in the middle tier.

{{demo “aggregated-progressive”}}

Despite being a stream, the aggregator’s output is 100% valid JSON so it remains compatible with standard AJAX tools. A streaming parser like Oboe.js reads the resource as a stream but a tool which does not understand streaming will have no problem reading it like a static resource.

Historic and live data on the same transport

It is a common pattern for an application to fetch existing data and then keep the page updated with ‘live’ events as they happen. We traditionally use two transports here but wouldn’t our day be easier if we didn’t have to program for distinct cases?

In the example below the message server intentionally writes a JSON response that never completes. It starts by writing out the existing messages as a chunk and then continues to write out new ones as they happen. The only difference between 'old' and 'new' data is timing.

{{demo "historic-and-live"}}

Cacheable streaming

Above we had a service where the response intentionally never completes. Here we will consider a slightly different case: JSON that streams to reflect live events but which eventually ends.

Most streaming HTTP techniques like Websockets intentionally avoid caches and proxies. Oboe.js is different; by taking a REST-based approach to streaming it remains compatible with HTTP intermediaries and can take advantage of caches to better distribute the content.

The visualisation below is based on a cartogram taken from Wikipedia and simulates each state's results being announced in the 2012 United States presidential election. Time is sped up so that hours are condensed into seconds.

{{demo "caching"}}

This won't work for every use case. Websockets remains the better choice where live data after-the-fact is no longer interesting. REST-based Cacheable streaming works best for cases where the live data is not specific to a single user and remains interesting as it ages.

In Summary

Using streaming to load data is usually faster and more fault tolerant. The biggest advantages come with large responses, mobile networks, or reading from streaming JSON services.

Downsides?

Because it is a pure Javascript parser, Oboe.js requires more CPU time than JSON.parse. Oboe.js is marginally slower for messages that load very quickly but for most real-world cases reacting to i/o sooner beats fussing about CPU usage. If in doubt, benchmark, but don't forget to use the real internet and think about perceptual performance.