Why Stream-loading?

This page makes the case for load JSON using streaming by examining several use cases.

Tl;dr

Streaming is usually faster. Or in the worst case it is about the same. For messages that load *very* quickly and never fail streaming might be slower due to extra processing but this is very likely to be negligible.

Using Oboe in the browser, the biggest advantages are found on mobile networks where requests can stall or fail, or if the server is writing out the JSON as a stream. For some use cases writing out REST streams can be used as an alternative to Websockets and similar.

When aggregating, streaming will usually be faster because everything else doesn't have to wait for the slowest resource.

Downloading from standard REST

In this use case we have a web application running on a browser and a server which provides it with JSON. The page won't be updated until the response is complete. This is the standard pattern used on almost all AJAX-powered sites.

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

Streaming downloading from standard REST

With a non-streamed response there is only a little time that can be saved.

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

Streaming one, with a server writing out using GSON or Node.js. Although this is a stream, the contents when it eventually completes are 100% valid JSON and is compatible with standard tools. Compatible with legacy or non-interactive tools.

{{demo "streaming-ajax-progressive"}}

Mobile data connections

Mobile networks today can be high-bandwidth but they are also high-latency and give poor guarantees of packet delivery time. This is why mobile networks, with buffering on the device, streaming HD video fluidly while web surfing feels

laggy. If we wait until we have everything we're wasting the chance to show data earlier.

{{demo "mobile-discrete"}}

With some stream-loading we show everything at the earliest opportunity:

{{demo "mobile-progressive"}}

Dropped connections

Most AJAX frameworks consider requests to be either wholly successful or wholly unsuccessful.

{{demo "mobile-fail-discrete"}}

Oboe takes a less dimorphic approach by viewing the HTTP response as a collection of many small parts. If the connection is lost it is simply the case that some parts where successful and can be used immediately, while others failed.

If we use the data that we have, as soon as we get it, we can show it *now*. When the network returns we need only request the parts that we missed.

{{demo "mobile-fail-progressive"}}

Aggregating resources

- waiting for the last one

{{demo "aggregated-discrete"}}

- don't let one slow resource slow everything down.
- don't slow everything's display down to the speed of the slowest component

{{demo "aggregated-progressive"}}

Historic and live data on the same transport

REST talks in the language of resources, not services. URLs should identify things, not endpoints. It shouldn't matter if the server has the thing now or if it will send it later when it does have it, or some combination of both.

You got to a page, you get the 'old' data and then are kept up to date with 'live' events. We normally use two transports for this, but wouldn't it be nicer if we didn't have to handle distinct cases?

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

If we treat the historic part as a stream and the streaming becomes trivial. Handle both with the same code, with no divergent code to write.

write: Why this is more like REST. Why REST is good.

Cacheable streaming

Above we had a JSON server which intentionally never completes its response. Here we have a different example: a datastream which will complete. Although streaming is used this case can be treated according to the standard REST paradigm and plays nice with well designed intermediaries such as caches.

The visualisation below includes a cartogram inspired by this file on Wikipedia and simulates each state's results being announced in the 2012 United States presidential election by condensing several hours into a minute to so.

{{demo "caching"}}

Incorporate and break up:

The REST service gives results per-state for the for the United States presidential election, 2012. While the results are being announced, requesting them returns an incomplete JSON with the states known so far be immediately sent, followed by the remainder dispatched individually as the results are called. When all results are known the JSON would finally close leaving a complete resource.

After the event, somebody wishing to fetch the results would use the *same URL for the historic data as was used on the night for the live data.* This is possible because the URL refers only to the data that is required, not to whether it is current or historic. Because it eventually forms a complete HTTP response, the data that was streamed is not incompatible with HTTP caching and a cache which saw the data while it was live could later serve it from cache as historic. More sophisticated caches located between client and service would recognise when a new request has the same URL as an already ongoing request, serve the response received so far, and then continue by giving both inbound requests the content as it arrives from the already established outbound request. Hence, the resource would be cacheable even while the election results are streaming and a service would only have to provide one stream to serve the same live data to multiple users fronted by the same cache. An application developer programming with Oboe would not have to handle live and historic data as separate cases because the node and path events they receive are the same. Without branching, the code which displays results as they are announced would automatically be able to show historic data.