Oboe.js

Why Oboe.js?

This page was written to show how streaming can speed up applications. The examples illustrated show web interfaces using AJAX to pull in new data but the same techniques would apply equally well anywhere that REST is used.

编写此页面是为了展示流式传输如何加速应用程序。所示示例显示了使用 AJAX 提取新数据的 Web 界面，但相同的技术同样适用于使用 REST 的任何地方。

Stream any JSON REST resource 流式传输任何 JSON REST 资源

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 provides it with data. The page isn’t updated until the response

Send historic and live data using the same transport
使用相同的传输方式发送历史数据和实时数据

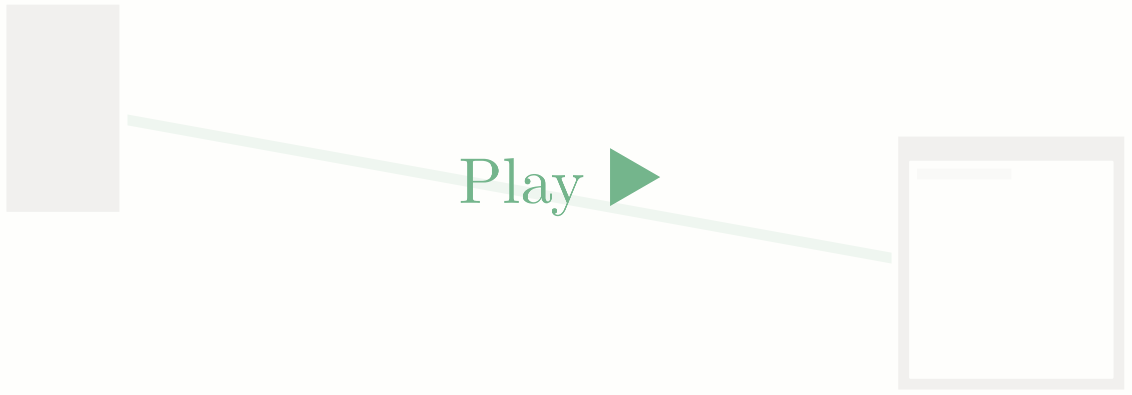
Publish cacheable, streamed content
发布可缓存的流式内容

What downsides?

Improve this page 改进此页面

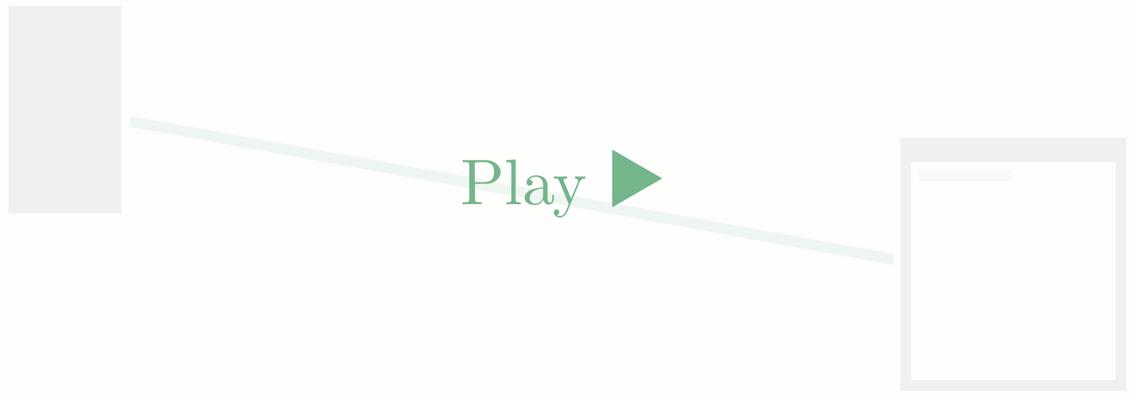
completes:

让我们从检查大多数 AJAX 支持的站点上的标准模式开始。我们有一个客户端 Web 应用程序和一个为其提供数据的服务。在响应完成之前页面不会更新：



Oboe is different from most streaming JSON libraries since the JSON does not have to follow a special format. On a good connection there isn't a lot of time to save but it is likely that progressive display in itself will improve the *perception* of performance:

Oboe 不同于大多数流式 JSON 库，因为 JSON 不必遵循特殊格式。在良好的连接上，没有太多时间可以节省，但渐进式显示本身可能会改善性能感知：



With a slower connection or larger data the improvement is more significant.

对于较慢的连接或较大的数据，改进更为显著。

Transmit fluently over mobile 通过手机流畅传输

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:

今天的移动网络是高带宽的，但也可能是高延迟的，并且数据包传送时间不一致。这就是为什么像流式高清视频这样的缓冲内容可以流畅地播放，

但网上冲浪仍然感觉迟钝的原因。下面的可视化近似于移动网络上的中型下载：

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:

Oboe.js 使程序员可以轻松地在响应到达后立即使用响应中的块。这有助于 webapps 在移动网络上运行时感觉更快：

Handle dropped connections with grace
优雅地处理断开的连接

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% was received correctly.

如果连接在响应完成之前丢失，Oboe.js 提供改进的容忍度。大多数 AJAX 框架将断开的连接等同于完全失败并丢弃部分传输的数据，即使 90% 已正确接收。

We can handle this situation better by using the partially transferred data instead of throwing it away. Given an incremental approach to parsing, using partial data follows naturally without requiring any extra programming.

我们可以通过使用部分传输的数据而不是将其丢弃来更好地处理这种情况。给定增量解析方法，自然会使用部分数据，而无需任何额外编程。

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

在下一个可视化中，我们有一个移动连接，当用户进入建筑物时连接失败：

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 comes for free.

因为 Oboe.js 将 HTTP 响应视为一系列小而有用的部分，所以当连接丢失时，只是一些部分成功并且已经使用，而其他部分没有到达的情况。容错是免费的。

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

在下面的示例中，客户端足够聪明，因此当网络恢复时，它只请求第一次尝试时丢失的数据：

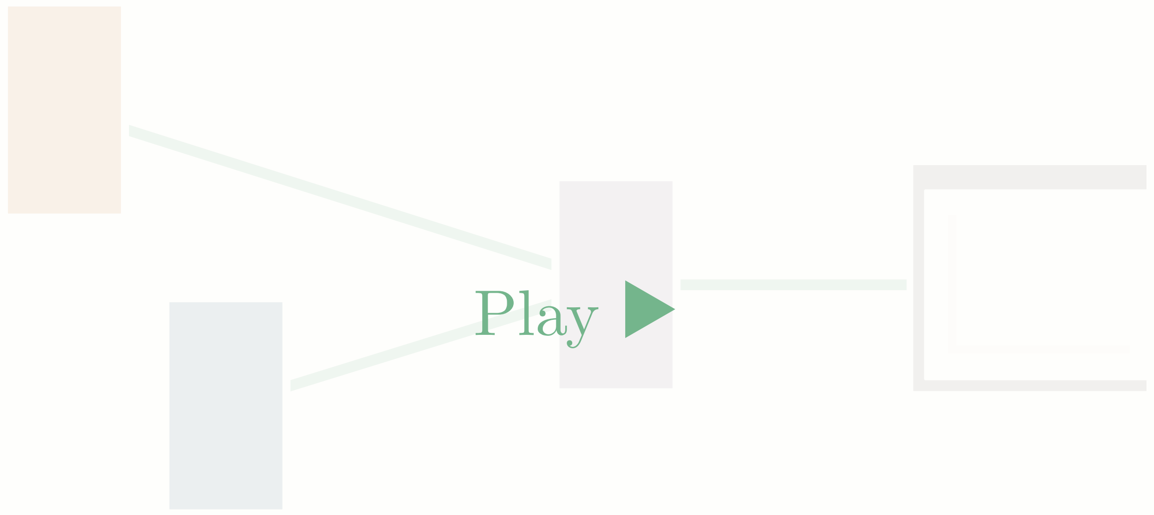
Streamline resource aggregation 简化资源聚合

It is a common pattern for web clients to retrieve data through a middle tier. Nodes in the middle tier connect to multiple back-end services and create a single, aggregated response by combining their data.

Web 客户端通过中间层检索数据是一种常见的模式。中间层的节点连接到多个后端服务，并通过组合它们的数据创建一个单一的、聚合的响应。

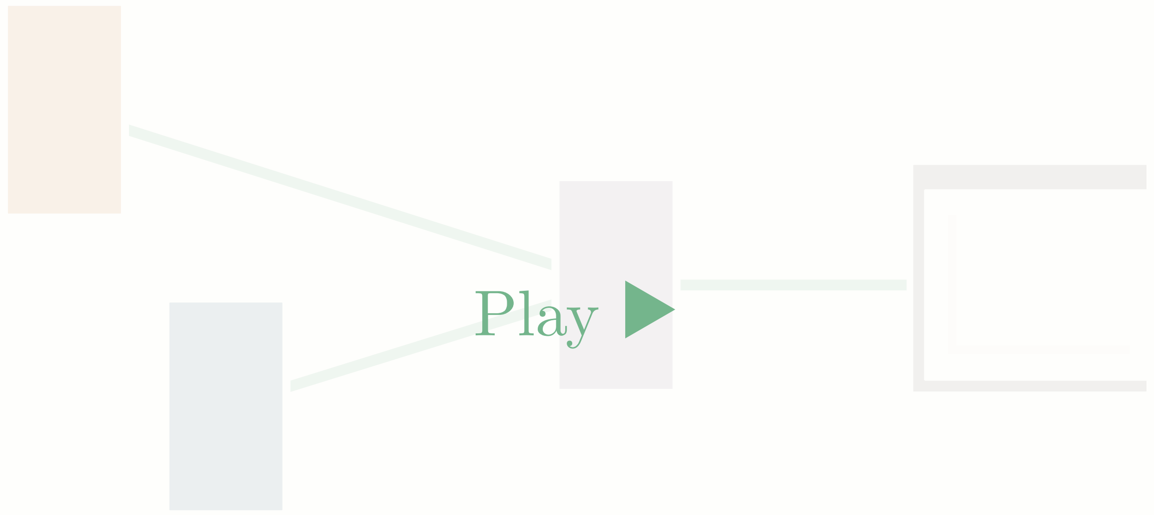
The visualisation below shows an example without streaming. Origin 1 is slower than Origin 2 but the aggregator is forced to respond at the speed of the slowest service:

下面的可视化显示了一个没有流式传输的示例。 Origin 1 比 Origin 2 慢，但聚合器被迫以最慢服务的速度响应：



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.

我们可以通过使用 Oboe.js 在聚合器和客户端中加载数据来加速此场景。聚合器在收到数据后立即发送数据，客户端在数据到达后立即显示数据。



Despite being a stream, the aggregator's output is 100% valid JSON so it remains compatible with standard AJAX tools. A client using a streaming parser like Oboe.js consumes the resource as a stream but a more traditional client has no problem reading it as a static resource.

尽管是一个流，但聚合器的输出是 100% 有效的 JSON，因此它仍然与标准 AJAX 工具兼容。使用像 Oboe.js 这样的流式解析器的客户端将资源作为流使用，但更传统的客户端可以将其作为静态资源读取没有问题。

In a Java stack this could also be implemented by using GSON in the middle tier.

在 Java 堆栈中，这也可以通过在中间层使用 GSON 来实现。

Step outside the trade-off between big and small JSON
走出大小 JSON 之间的权衡

There is often a tradeoff using traditional REST clients:

使用传统 REST 客户端通常需要权衡：

- Request too much data and the application feels unresponsive because each request takes some time to download.

  请求太多数据，应用程序感觉没有响应，因为每个请求都需要一些时间来下载。

- Request less and, while the first data is handled earlier, more requests are needed, meaning a greater http overhead and more

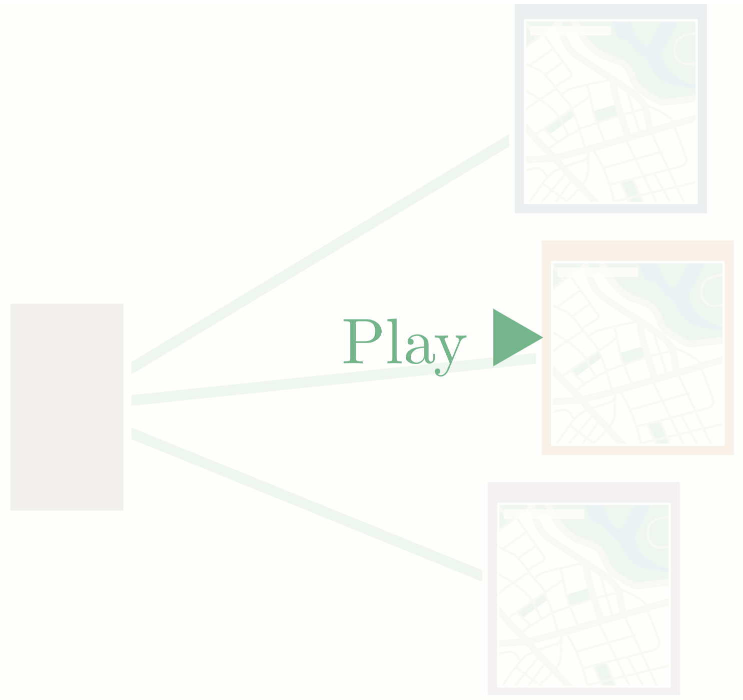
> time overall.
>
> 请求更少，虽然第一个数据被更早地处理，但需要更多的请求，这意味着更大的 http 开销和更多的时间。

Oboe.js breaks out of the tradeoff by beating both. Large resources load just as responsively as smaller ones so the developer can request more and let it stream.

Oboe.js 打败了两者，打破了权衡。大型资源的加载与小型资源一样响应迅速，因此开发人员可以请求更多资源并让它流式传输。

In the visualisation below three rival clients connect to the same server. The top client requests a lot of data, the middle a little data twice, and the bottom a lot using Oboe.js:

在下面的可视化中，三个竞争客户端连接到同一台服务器。顶部客户端请求大量数据，中间两次请求少量数据，底部使用 Oboe.js 请求大量数据：



Send historic and live data using the same transport
使用相同的传输方式发送历史数据和实时数据

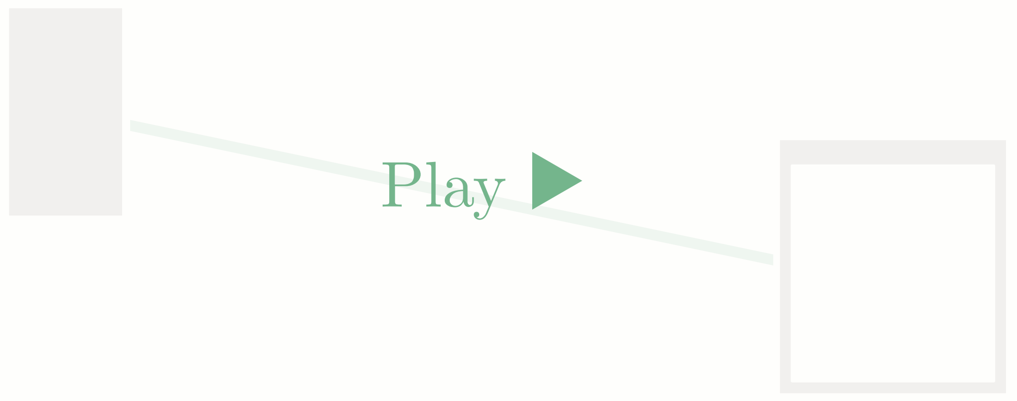
It is a common pattern in web interfaces to fetch existing data and then keep the page updated with ‘live’ events as they happen. We traditionally use two transports here but wouldn’t our day be easier if we didn’t have to program distinct cases?

这是 Web 界面中的一种常见模式，用于获取现有数据，然后在发生“实时”事件时保持页面更新。我们传统上在这里使用两种传输，但如果我们不必

对不同的案例进行编程，我们的日子会不会更轻松?

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:

在下面的示例中，消息服务器有意写入一个永远不会完成的 JSON 响应。它首先将现有消息作为一个块写出，然后在新消息出现时继续写出它们。“旧”和“新”数据之间的唯一区别是时间:



Publish cacheable, streamed content
发布可缓存的流式内容

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.

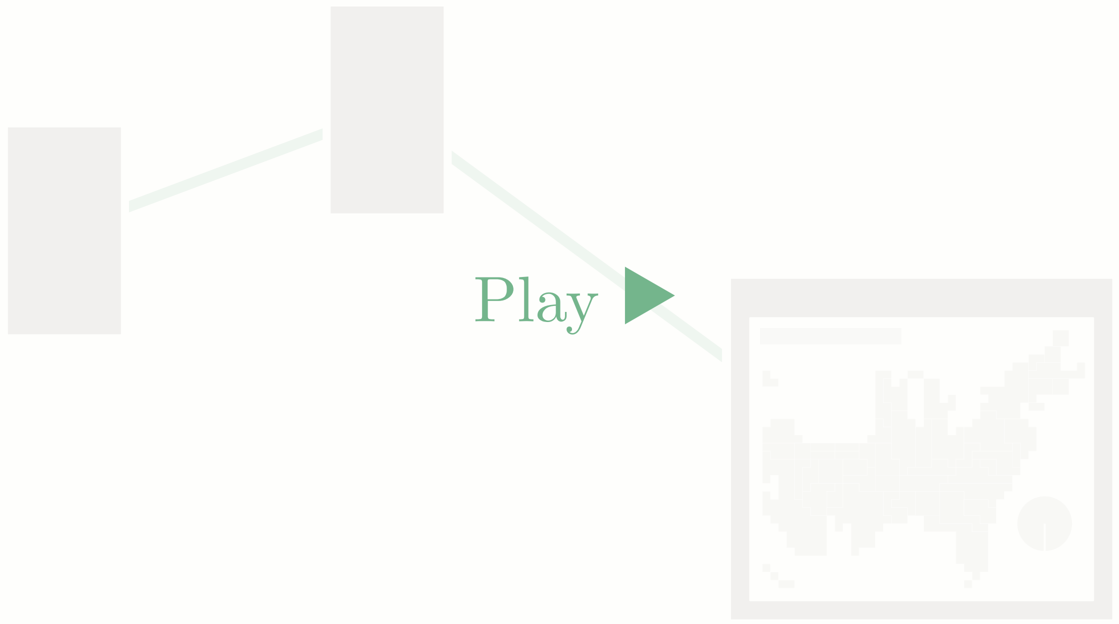
上面我们有一个服务，其中的响应故意永远不会完成。在这里，我们将考虑一种略有不同的情况：JSON 流式传输以反映实时事件但最终结束。

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.

大多数流式 HTTP 技术（如 Websockets）有意避免缓存和代理。Oboe.js 是不同的；通过采用基于 REST 的流式传输方法，它仍然与 HTTP 中介兼容，并且可以利用缓存来更好地分发内容。

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.

下面的可视化是基于从维基百科获取的图表，并模拟了每个州在 2012 年美国总统大选中宣布的结果。时间被加速，以至于小时被压缩成秒。



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.

这不适用于所有用例。在事后实时数据不再有趣的情况下，Websockets 仍然是更好的选择。基于 REST 的可缓存流最适用于实时数据不特定于单个用户并且随着时间的推移仍然很有趣的情况。

What downsides?

Because it is a pure Javascript parser, Oboe.js requires more CPU time than JSON.parse. Oboe.js works marginally more slowly for small messages that load very quickly but for most real-world cases using i/o effectively beats optimising CPU time.

因为它是一个纯 Javascript 解析器，Oboe.js 比 JSON.parse 需要更多的 CPU 时间。对于加载速度非常快的小消息，Oboe.js 的工作速度稍微慢一些，但对于大多数实际情况，使用 i/o 有效地胜过优化 CPU 时间。

SAX parsers require less memory than Oboe's pattern-based parsing model because they do not build up a parse tree. See Oboe.js vs SAX vs DOM.

SAX 解析器比 Oboe 的基于模式的解析模型需要更少的内存，因为它们不构建解析树。请参阅 Oboe.js 与 SAX 与 DOM。

If in doubt, benchmark, but don't forget to use the real internet, including mobile, and think about perceptual performance.

如果有疑问，请进行基准测试，但不要忘记使用真实的互联网，包括移动设备，并考虑感知性能。

Improve this page 改进此页面

Is this page clear enough? Typo-free? Could you improve it?

这个页面够清楚吗？无错字？你能改进一下吗？

Please fork the Oboe.js website repo and make a pull request. The markdown source is here.

请分叉 Oboe.js 网站 repo 并提出拉取请求。降价源在这里。