Oboe.js: An approach to i/o for rest clients which is neither batch nor stream; nor SAX nor DOM

Jim Higson

2013

Contents

List of Figures

1 Abstract

A Javascript REST client library targeting both Node.js and web browsers that incorporates http streaming, pattern matching, and progressive JSON parsing, with the aim of improving performance, fault tolerance, and encouraging a greater degree of loose coupling between programs. Loose coupling is particularly considered in light of the application of Agile methodologies to SOA, providing a framework in which it is acceptable to partially restructure the JSON format in which a resource is expressed whilst maintaining compatibility with dependent systems.

A critique is made of current practice under which resources are entirely retrieved before items of interest are extracted programmatically. An alternative model is presented allowing the specification of items of interest using a declarative syntax similar to JSONPath. The identified items are then provided incrementally while the resource is still downloading.

In addition to a consideration of performance in absolute terms, the usability implications of an incremental model are also evaluated with regards to differences in user perception of performance.

2 Introduction

This dissertation does not focus on implementing software for any particular problem domain. Rather, its purpose is to encourage the REST paradigm to be viewed through a novel lens. In application this may be used to deliver tangible benefits to many common REST use cases. Although I express my thesis through programming, the contribution I hope to deliver is felt more strongly as a shift in how we *think* about http than it is a change in the underlying technology.

In the interest of developer ergonomics, REST clients have tended to style the calling of remote resources similar to the call style of the host programming language. Depending on the language, one of two schemas are followed: a synchronous style in which the http call is an expression which evaluates to the resource that was fetched; or asynchronous or monadic in which some logic is specified which may be applied to the response once it is complete. This tendency to cast REST calls using terms from the language feels quite natural; we may call a remote service without having to make any adjustment for the fact that it is remote. However, we should remember that this construct is not the only possible mapping. Importing some moderate Whorfianism (Whorf 1956)(Sapir 1958) from linguistics, we might venture to say that the programming languages we use encourage us to think in the terms that they easily support. Also UML! For any multi-packet message sent via a network some parts will arrive before others, at least approximately in-order, but whilst coding a C-inspired language whose return statements yield single, discrete values it comfortable to conceptualise the REST response as a discrete event. Perhaps better suited to representing a progressively returned value would have been the relatively unsupported Generator routine (Ralston 2000).

In most practical cases where software is being used to perform a task there is no reasonable distinction between being earlier and being quicker. Therefore, if our interest is to create fast software we should be using data at the first possible opportunity. Examining data *while* it streams rather than hold unexamined until the message ends.

The coining of the term REST represented a shift in how we think about http, away from the transfer of hypertext documents to that of arbitrary data (Fielding 2000, 407–416). It introduced no fundamentally new methods. Likewise, no genuinely new computer science techniques need be invented to realise my thesis. As a minimum, the implementation requires an http client which exposes the response whilst it is in progress and a parser which can start making sense of a response before it sees all of it. I also could not claim this thesis to be an entirely novel composition of such parts. Few ideas are genuinely new and it is often wiser to mine for solved problems then to solve again afresh. The intense competition of Web browsers to be as fast as possible has already found this solution. Load any graphics rich with images – essentially an aggregation of hypertext and images – the HTML is parsed incrementally while it is downloading and the images are requested as soon as individual <img> tags are encountered. The

browser's implementation involves a highly optimised parser created for a single task, that of displaying web pages. The new contribution of this dissertation is to provide a generic analog applicable to any problem domain.

Also progressive SVGs.[1]

2.1 REST aggregation could be faster

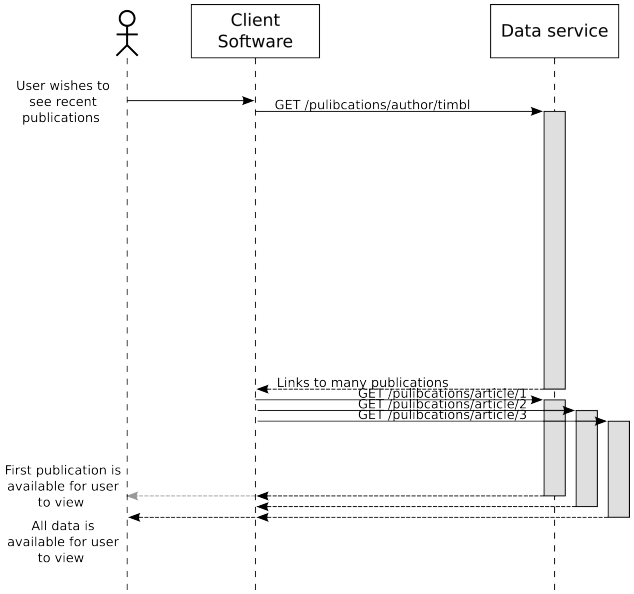

Figure 1: **Aggregation of lower-level resources exposed via REST.** The client fetches a listing of an author's publications and then the first three articles. The sequence represents the most commonly used technique in which the client does not react to the response until it is complete. In this example the second wave of requests cannot be made until the original response is complete, at which time they are issued in quick succession.

Figures 1 and 2 illustrate how a progressive REST client may without adjustments to the server be used to aggregate REST resources faster. The greatest improvement is in how early the first piece of data is able to be used. This is advantageous: firstly, progressive display in itself raises the human perception of performance (Geelhoed et al. 1995); secondly, a user wanting to scan from top to bottom may start reading the first article while waiting for the later ones to arrive; thirdly, on seeing the first content the user may notice that they have requested the wrong aggregation, allowing them to backtrack earlier.

[1] Https://github.com/jimhigson/oboe.js/blob/master/src/streamingHttp.js I can't claim superior programming ability, this version is shorter because it is not a generic solution.

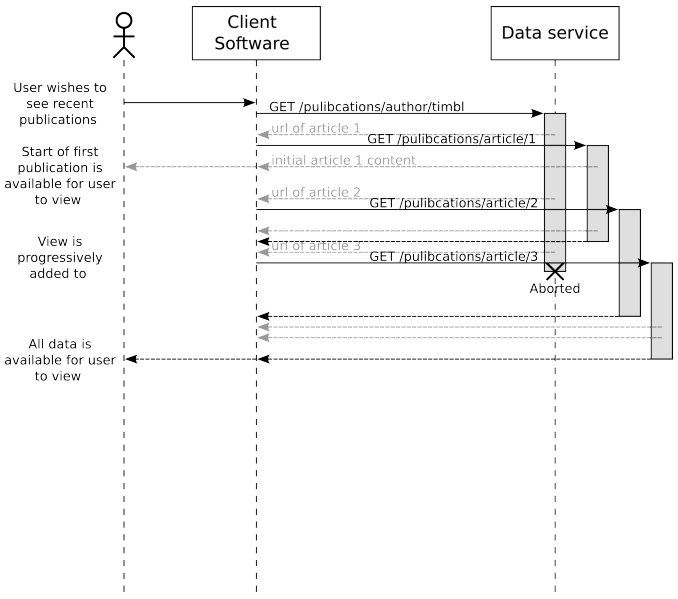

Figure 2: **Revised sequence of aggregation performed by a client capable of progressively interpreting the fetched resource.** Because UML sequence diagrams arrows draw the concept of a returned value as a one-off event rather than a continuous process, I have introduced the notation of lighter arrows illustrating fragments of an ongoing response. Each individual publication request is made at the earliest possible time, as soon as the its URL can be extracted from the publications list. Once the required data has been read from the original resource it is aborted rather than continue to download unnecessary data. This results in a moderate reduction in wait time to see all three articles but a dramatic reduction in waiting before the first content is presented. Note also how the cadence of requests is more even with four connections opened at roughly equal intervals rather than a single request followed by a rapid burst of three. Clients frequently limit the number of simultaneous connections per domain so avoiding bursts of requests is further to our advantage.

Although the label "client software" in the figures above hints at software running directly on a user's own device this is not necessarily the case, for example the client may in fact be an server-side aggregation layer. Nodes in an n-tier architecture commonly defy categorisation as 'client' or 'server' in a way which is appropriate from all frames of reference. Rather, nodes may be thought of as a client from the layer below and as a server from the layer above. A further example would be a server-side webpage generator maintaining a perceptual performance improvement by progressively writing out html using http chunked encoding. (Stefanov 2009). The demonstrated advantages hold regardless of where in the stack the 'client' is located.

2.2 Stepping outside the big-small tradeoff

Where a domain model contains a series of data, of which ranges are made available via REST, I have often seen a trade-off with regards to how much of the series each call should request. Answering this question is usually a compromise between competing concerns in which it is not simultaneously possible to addresses all concerns satisfactorily. A good example might be a Twitter's pages listing a series of tweets where the interface designers adopted a currently trending pattern (Ahuvia 2013), Infinite Scrolling. Starting from an initial page showing some finite number of tweets, upon scrolling to the bottom the next batch is automatically requested. The new batch is fetched in a json format and, once loaded, presented as html and added to the bottom of the page. Applied repeatedly this allows the user to scroll indefinitely, albeit punctuated by slightly jolting pauses while new content is loaded. To frame the big-small tradeoff we might consider the extreme choices. Firstly, requesting just one tweet per http request. By requesting the smallest possible content individual calls would complete very quickly and the pauses would be short. Taking the extreme small end the page stutters, pausing momentarily but frequently. Taking the opposite extreme, by requesting some huge number of tweets we see long periods of smooth scrolling partitioned by long waits.

I propose that my thesis may be applied used to stand down from this compromise by delivering pauses which are both infrequent and short. In the Twitter example, once we have thinking about http progressively this may be achieved quite simply by issuing large requests but instead of deferring all rendering until the request completes, render individual tweets incrementally as they are progressively parsed out of the ongoing response.

Integrate: twitter: page could update at bottom and top with same transport perhaps.

2.3 Staying fast on a fallible network

The reliability of networks that REST operates over varies widely. Considering the worst case we see mobile networks in marginal signal over which it is common for ongoing downloads to be abruptly disconnected. Existing http clients handle this kind of unexpected termination poorly. Consider the everyday situation of somebody using a smartphone browser to check their email. The use of Webmail necessitates that the communication in made via REST rather than a mail specific protocol such as IMAP. Mobile data coverage is less than network operators claim (Anon. 2011) so while travelling the signal can be expected to be lost and reestablished many times. Whilst not strictly forbidding their inspection, the web developer's standard AJAX toolkit are structured in such a way as to encourage the developer to consider partially successful messages as wholly unsuccessful. For example, the popular AJAX library jQuery automatically parses complete JSON or XML responses before handing back to the application. But on failure there is no attempt to parse or deliver the partial response. To programmers who know where to look the partial responses are retrievable as raw text but handling them is a special case, bringing-your-own-parser affair. Because of this difficulty I can only find examples of partial messages being dropped without inspection. In practice this means that for the user checking her email, even if 90% of her inbox had been retrieved she will be shown nothing. When the network is available again the application will have to download from scratch, including the 90% which it already fetched. I see much potential for improvement here.

Not every message, incomplete, is useful. Whilst of course a generic REST client cannot understand the semantics of specific messages fully enough to decide if a partially downloaded message is useful, I propose it would be an improvement if the content from incomplete responses could be handled using much the same programming as for complete responses. This follows naturally from a conceptualisation of the http response as a progressive stream of many small parts; as each part arrives it should be possible to use it without knowing if the next will be delivered successfully. This style of programming encourages thinking in terms of optimistic locking. Upon each partial delivery there is an implicit assumption that it may be acted on straight away and the next will also be successful. In cases where this assumption fails the application should be notified so that some rollback may be performed.

2.4 Agile methodologies, frequent deployments, and compatibility today with versions tomorrow

In most respects SOA architecture fits well with the fast release cycle that Agile methodologies encourage. Because in SOA we may consider that all data is local rather than global and that the components are loosely coupled and autonomous, frequent releases of any particular sub-system shouldn't pose a problem to the

correct operation of the whole. Following emergent design it should be possible for the format of resources to be realised slowly and iteratively as a greater understanding of the problem is achieved. Unfortunately in practice the ability to change is hampered by tools which encourage programming against rigidly specified formats. Working in enterprise I have often seen the release of dozens of components cancelled because of a single unit that failed to meet acceptance criteria. By allowing a tight coupling that depends on exact versions of formats, the perfect environment is created for contagion whereby the updating of any single unit may only be done as part of the updating of the whole.

An effective response to this problem would be to integrate into a REST client library the ability to use a response whilst being only loosely coupled to the *shape* of the overall message.

2.5 Deliverables

To avoid feature creep I am paring down the software deliverables to the smallest work which can we said to realise my thesis. Amongst commentators on start-up companies this is known as a *zoom-in pivot* and the work it produces should be the *Minimum Viable Product* or MVP (Reis 2011 p. ??), the guiding principle being that it is preferable to produce a little well than more badly. By focusing tightly I cannot not deliver a full stack so I am forced to implement only solutions which interoperate with existing deployments. This is advantageous; to somebody looking to improve their system small additions are easier to action than wholesale change.

To reify the vision above, a streaming client is the MVP. Because all network transmissions may be viewed though a streaming lens an explicitly streaming server is not required. Additionally, whilst http servers capable of streaming are quite common even if they are not always programmed as such, I have been unable to find any example of a streaming-capable REST client.

2.6 Criteria for success

In evaluating this project, we may say it has been a success if non-trivial improvements in speed can be made without a corresponding increase in the difficulty of programming the client. This improvement may be in terms of the absolute total time required to complete a representative task or in a user's perception of the speed in completing the task. Because applications in the target domain are much more io-bound than CPU-bound, optimisation in terms of the execution time of a algorithms will be de-emphasised unless especially egregious. The measuring of speed will include a consideration of performance degradation due to connections which are terminated early.

Additionally, I shall be looking at common ways in which the semantics of a message are expanded as a system's design emerges and commenting on the value

of loose coupling in avoiding disruption given unanticipated format changes.

3 Background

3.1 The web as an application platform

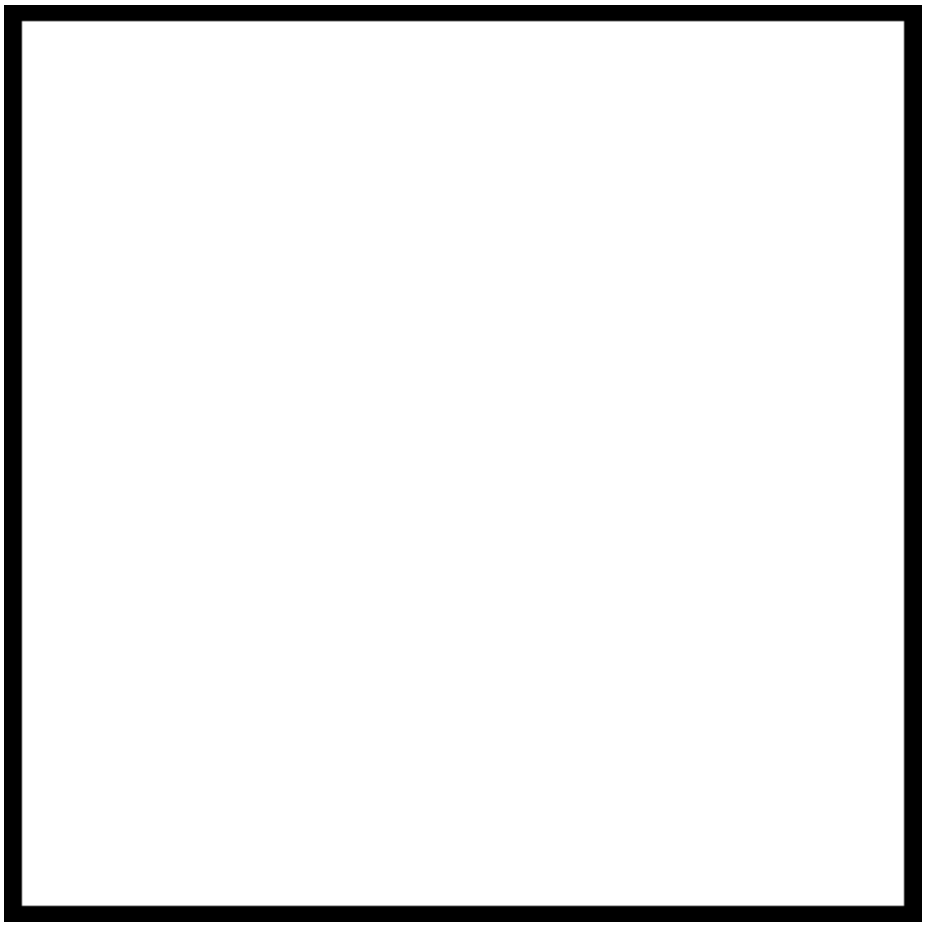

Figure 3: *A webapp running with a front end generated partially on server and partially on client side.* Ie, front-end client-side, front-end server-side, presentation layer a more meaningful distinction than

Application design, particularly regarding the presentation layer, has charted an undulating path pulled by competing patterns of thick and thin clients. Having been taken up as the platform today for all but the most specialised applications, the web continues in this fashion by resisting easy categorisation as either mode. Although born on the network, at inception the web wasn't particularly graphical and didn't tread in the steps of networked graphical technologies such as X11

in which every presentation decision was made on a remote server[2] – instead of sending fine-grained graphical instructions, a much more compact document mark-up format was used. At the same time, the markup-format was unlike like Gopher by being not totally semantic meaning that presentation layer concerns were kept partially resident on the server. At this time, whereas CGI was being used to serve documents with changeable content, it was not until 1996 with *ViaWeb* (later to become Yahoo Stores) that a user could be given pages comparable in function to the GUI interface of a desktop application. (Graham 2004 - get page number, in old dis). The interface of these early web applications comprised of pages dynamically generated on the server side, but handled statically on the client side so far as the browser was not able to be scripted to manipulate the page in any way.

The modern, client-scripted web bears a striking resemblance to NeWS. Rather than send many individual drawings, the server could send parametrised instructions to show the client *how* some item of presentation is drawn. Having received the program, the only communications required are the parameters. This mixed-model provides no lesser degree of server-side control but by using client-side rendering a much faster experience was possible than would otherwise be possible over low-speed networks (Hopkins 1994).

Today it is agreed that program architecture should separate presentation from operational logic but there is no firm consensus on where each concern should be exercised. While it feels that Javascript is becoming requisite to even display a page, there are also actions in the opposite direction, for example in 2012 twitter moved much of their rendering back to the server-side reducing load times to one fifth of their previous design, commenting "The future is coming and it looks just like the past" (Lea 2012). This model generated server-side short pages that load quick and are ready to be displayed but also sent the Javascript which would allow the display to be updated without another full server load. One weakness of this model is that the same presentational logic requires two expressions.

Like most interactive programming, client-side scripts usually suffer greater delays waiting for io than because javascript execution times present a bottleneck. Because Javascript is used for user interfaces, frame-rates are important. Single threaded so js holds up rendering. Important to return control to the browser quickly. However, once execution of each js frame of execution is no more than the monitor refresh rate, further optimisation brings zero benefit. Hence, writing extremely optimised Javascript, especially focusing on micro-optimisations that hurt code readability is a bit silly.

> The user does something, then the app responds visually with immediacy at 30 frames per second or more, and completes a task in a few hundred milliseconds. As long as an app meets this user goal, it doesn't matter how big an abstraction layer it has to go through to get to silicon. (Mullany 2013)

[2] Https://github.com/substack/http-browserify.

3.2 Node.js

Include? Node not just for servers. CLI tools etc.

Include? Compare to Erlang. Waiter model. Node restaurant much more efficient use of expensive resources.

Include? No 'task' class or type, tasks are nothing more than functions, possibly having some values implicitly wrapped up in their closure.

Include? Easy to distribute software (npm etc)

It is difficult to say to what degree Node's use of Javascript is a distraction from the system's principled design aims and to what degree it defines the technology. Paradoxically, both may be so. Javascript has proven itself very effective as the language to meet Node's design goals but this suitability is not based on Javascript's association with web browsers, although it is certainly beneficial: for the first time it is possible to program presentation logic once which is capable of running on either client or server. Being already familiar with Javascript, web programmers were the first to take up Node.js first but the project mission statement makes no reference to the web; Node's architecture is well suited to any application domain where low-latency responses to i/o is more of a concern than heavyweight computation. Web applications fit well into this niche but they are far from the only domain that does so.

In most imperative languages attempts at concurrency have focused on threaded execution, whereas Node is by design single-threaded. Threads are an effective means to speed up parallel computation but not well suited to concurrently running tasks which are mostly i/o dependent. Used for io, threads consume considerable resources while spending most of their lives waiting, occasionally punctuated with short bursts of activity. Programming Java safely with threads which share access to mutable objects requires great care and experience, otherwise the programmer is liable to create race conditions. If we consider for example a Java thread-based http aggregator; each 'requester' thread waits for seconds and then processes for milliseconds. The ratio of waiting to processing is so high that any gains achieved through actual concurrent execution of the active phase is pyrrhic. Following Node's lead, even traditionally thread-based environments such as Java are starting to embrace asynchronous, single-threaded servers with projects such as Netty.

Node manages concurrency by managing an event loop of queued tasks and expects each task never to block. Non-blocking calls are used for all io and are callback based. Unlike Erlang, Node does not swap tasks out preemptively, it always waits for tasks to complete. This means that each task must complete quickly; while this might at first seem like an onerous requirement to put on the programmer, in practice the asynchronous nature of the toolkit makes following this requirement more natural than not. Indeed, other than accidental non-terminating loops or heavy number-crunching, the lack of any blocking io whatsoever makes it rather difficult to write a node program whose tasks do

not exit quickly. This programming model of callback-based, asynchronous, non-blocking io with an event loop is already the model followed inside web browsers, which although multi-threaded in some regards, present a single-threaded virtual machine in terms of Javascript execution.

A programmer working with Node's single-thread is able to switch contexts quickly to achieve a very efficient kind of concurrency because of Javascript's support for closures. Because of closures, under Node the responsibility to explicitly store state between making an asynchronous call and receiving the callback is removed from the programmer. Closures require no new syntax, the implicit storage of this data feels so natural and inevitable that looking at the typical program it is often not obvious that the responsibility exists at all.

```
// Rather than blocking, this function relies on non-blocking io and
// schedules three tasks, each of which exit quickly, allowing this
// node instance to continue with other tasks in between.
// However sophisticated and performant this style of programming,
// to the developer it is barely more difficult than if a blocking io
// model were followed.

function makeRequest( host, path ) {

   var options = { host: host, path: path };

   http.get(options, function(response){

      // This function will be called when the response starts. The callback
      // having started listening to the response object will quickly exit
      // as Node tasks are want to do. Because of closures we are able to
      // access the path variable declared in a containing scope, even after
      // the containing scope has exited.
      console.log("The response has started for " + path);

      response.on('data', function(chunk) {

         // This function is called each time some data is received from the
         // http request

         console.log('Got some response ' + chunk);

      });
   }).on("error", function(e){

      console.log("Got error: " + e.message);
   });
```

```
    console.log("Request has been made");
}
```

3.3 Streams in Node

> Streams in node are one of the rare occasions when doing something the fast way is actually easier. SO USE THEM. not since bash has streaming been introduced into a high level language as nicely as it is in node." high level node style guide

Bash streams a powerful abstraction easily programmed for linear streaming. Node more powerful, allows a powerful streaming abstraction which is no more complex to program than a javascript webapp front end. Essentially a lower-level (and therefore more powerful) interface to streaming such as unix sockets or tcp connections.

> Node Stream API, which is the core I/O abstraction in Node.js (which is a tool for I/O) is essentially an abstract in/out interface that can handle any protocol/stream that also happens to be written in JavaScript. [http://maxogden.com/a-proposal-for-streaming-xhr.html]

Streams in node are a variant of the observer pattern and fit into a wider Node event model. Streams emit 'readable' events when they have some data to be read and 'end' events when they are finished. Apart from error handling, so far as reading is concerned, that is the extent of the API.

3.4 Web browsers hosting REST clients

Http is essentially a thinly-wrapped text response around some usually text-based (but sometimes binary) data. It may give the length of the content as a header, but is not obliged to. It supports an explicitly chunked mode, but even the non-chunked mode may be considered as a stream. For example, a program generating web pages on the server side might choose to use chunking so that the browser is better able to choose when to re-render during the progressive display of a page (Stefanov 2009) but this is optional and without these hints progressive rendering will still take place.

The requesting of http from Javascript, commonly termed AJAX, was so significant a technique in establishing the modern web application architecture that it is often taken as being a synonym for Javascript-heavy web pages. Although an acronym for Asynchronous Javascript and XML, for data services designed with delivery to client-side web applications in mind JSON is almost exclusively

preferred to XML and the term is used without regard for the data format of the response (the unpronounceable *AJAJ* never took off). During the 'browser war' years adding non-standard features was a common form of competition between authors; following this pattern Internet Explorer originally made AJAX possible by exposing Microsoft's Active X *Xml Http Request*, or XHR, object to Javascript programmers. This was widely copied as functional equivalents were added to all major browsers and the technique was eventually formalised by the W3C(van Kesteren and Jackson 2006). What followed was a period of stagnation for web browsers. HTML4 reached W3C Recommendation status in 2001 but having subsequently found several evolutionary dead ends such as XHTML, the developer community would see no major updates until HTML5 started to gather pace some ten years later. In this context the web continued to rapidly mature as an application platform and AJAX programming inevitably overtook the original XHR specification, browser vendors again adding their own proprietary extensions to compensate.

Given this backdrop of non-standard extensions and lagging standardisation, abstraction layers predictably rose in popularity. Despite a reputation Javascript being poorly standardised, as a language it is very consistently implemented. More accurately we should say that the libraries provided by the environment lack compatibility. Given an abstraction layer to gloss over considerable differences cross-browser webapp developers found little difficulty in targeting multiple platforms. The various abstraction competed on developer ergonomics with the popular jQuery and Prototype.js promoting themselves respectively as *"do more, write less"* and *"elegant APIs around the clumsy interfaces of Ajax"*. JSON being a subset of Javascript, web developers barely noticed their privileged position whereby the serialisation of their data format mapped exactly onto the basic types of their programming language. As such there was never any confusion as to which exact object structure to de-serialise to. If this seems like a small advantage, contrast with the plethora of confusing and incompatible representations of JSON output presented by the various Java JSON parsers; JSON's Object better resembles Java's Map than Object and the confusion between JSON null, Java null, and Jackson's NullNode[3] is a common cause of errors. Endowed with certainty regarding deserialisation, JSON parsers could be safely integrated directly into AJAX libraries. This provided a call style while working with remote resources so streamlined as to require hardly any additional effort.

```
jQuery.ajax('http://example.com/people.json', function( people ) {

   // The parsing of the people json into a javascript object
   // feels so natural that it is easy to forget while looking
   // at the code that it happens at all.
```

[3] Https://github.com/jimhigson/oboe.js/blob/master/src/streamingHttp.js I can't claim superior programming ability, this version is shorter because it is not a generic solution.

```
   alert('the first person is called ' + people[0].name);
});
```

Whilst simple, the above call style is built on the assumption that a response is a one-time event and no accommodation is made for a continuously delivered response. Meanwhile, the XHR2 standardisation process had started and was busy observing and specifying proprietary extensions to the original XHR1. Given an interest in streaming, the most interesting of these is the progress event:

> While the download is progressing, queue a task to fire a progress event named progress about every 50ms or for every byte received, whichever is least frequent. (van Kesteren 2012)

Prior to this addition there had been no mechanism, at least so far as the published specs to an XHR instance in a streaming fashion. However, while all major browsers currently support progress events in their most recently versions, the installed userbase of supporting browsers is unlikely to grow fast enough that this technique may be relied upon without a fallback for several years.

In fact, this is exactly how web browsers are implemented. However, this progressive use of http is hardwired into the browser engines rather than exposing an API suitable for general use and as such is treated as something of a special case specific to web browsers and has not so far seen a more general application. I wish to argue that a general application of this technique is viable and offers a worthwhile improvement over current common methods.

While until recently browsers have provided no mechanism to stream into AJAX, almost every other instance of downloading has taken advantage of streaming and progressive interpretation. This includes image formats, as the progressive PNG and JPEG; markup as progressive display of html and svg; video; and Javascript itself – script interpretation starts before the script is wholly fetched. Each of these progressive considerations is implemented as a specific-purpose mechanism internal to the browser which is not exported to Javascript and as such is not possible to repurpose.

3.5 Browser streaming frameworks

As the web's remit spread to include more applications which would previously have been native apps, to be truly 'live' many applications found the need to be able to receive real-time push events. Dozens of streaming transports have been developed sidestepping the browser's apparent limitations.

The earliest and most basic attempt was to poll by making many requests, I won't consider this approach other than to say it came with all the usually associated downsides. Despite the inadequacy of this approach, from here the

improved technique of *long polling* was invented. A client makes a request to the server side. Once the connection is open the server waits, writing nothing until a push is required. To push the server writes the message and closes the http connection; since the http response is now complete the content may be handled by the Javascript client which then immediately makes a new request, reiterating the cycle of wait and response. This approach works well where messages are infrequently pushed but where the frequency is high the limitation of one http transmission per connections requires imposes a high overhead.

Observing that while browsers lack progressive ajax, progressive html rendering is available, *push tables* achieve progressive data transfer by serialising streaming data to a HTML format. Most commonly messages are written to a table, one row per message. On the client side this table is hidden in an off-screen frame and the Javascript streaming client watches the table and reacts whenever a new row is found. In many ways an improvement over long-polling, this approach nevertheless suffers from an unnatural data format. Whilst html is a textual format so provides a degree of human-readability, html was not designed with the goal of an elegent or compact transfer of asynchronous data. Contrasted with a SOA ideal of *'plumbing on the outside'*, peeking inside the system is difficult whilst bloated and confusing formats are tasked with conveying meaning.

Both long polling and push tables are better throught of as a means to circumvent restrictions than indigene technology. A purose-built stack, *Websockets* is poised to take over, building a standardised duplex transport and API on top of http's chunked mode. While the newest browsers support websockets, most of the wild use base does not. Nor do older browsers provide a fine-grained enough interface into http in order to allow a Javascript implementation. In practice, real-world streaming libraries such as socket.io [CITE] are capable of several streaming techniques and can select the best for a given context. To the programmer debugging an application the assortment of transports only enhances the black-box mentality with regards to the underlying transports.

Whilst there is some overlap, each of the approaches above addresses a problem only tangentially related to this project's aims. Firstly, requiring a server that can write to an esoteric format feels quite anti-REST, especially given that the server is sending in a format which requires a specific, known, specialised client rather than a generic tool. In REST I have always valued how prominently the plumbing of a system is visible, so that to sample a resource all that is required is to type a URL and be presented with it in a human-comprehensible format.

Secondly, as adaptations to the context in which they were created, these frameworks realise a view of network usage in which downloading and streaming are dichotomously split, whereas I aim to realise a schema without dichotomy in which *streaming is adapted as the most effective means of downloading.* In existing common practice a wholly distinct mechanism is provided vs for data which is ongoing vs data which is finite. For example, the display of real-time stock data might start by AJAXing in historical and then separately use a websocket to maintain up-to-the-second updates. This requires the server to

support two distinct modes. However, I see no reason why a single transport could not be used for both. Such a server might start answering a request by write historic events from a database, then switch to writing out live data in the same format in response to messages from a MOM. By closing the dichotomy we would have the advantage that a single implementation is able to handle all cases.

It shouldn't be a surprise that a dichotomous implementation of streaming, where a streaming transport is used only for live events is incompatible with http caching. If an event is streamed when it is new, but then when it is old made available for download, http caching between the two requests is impossible. However, where a single mode is used for both live and historic events the transport is wholly compatible with http caching.

If we take streaming as a technique to achieve efficient downloading, not only for the transfer of forever-ongoing data, none of these approaches are particularly satisfactory.

3.6 Json and XML

Although AJAX started as a means to transfer XML, today JSON "The fat-free alternative to XML(Douglas 2009)" is the more popular serialisation format. The goals of XML were to simplify SGML to the point that a graduate student would be able to implement a parser in a week [@javaxml p ???]. For the student tackling JSON a few hours with a parser generator should surfice, being expressable in 15 CFGs. Indeed, because JSON is a strict subset of Javascript, in many cases the Javascript programmer requires no parser at all. Unimpeeded by SGML's roots as a document format, JSON provides a much more direct analogue to the metamodel of a canonical modern programming language with entities such as *string*, *number*, *object* and *array*. By closely mirroring a programmer's metamodel, visualising a mapping between a domain model and it's serialised objects becomes trivial.

```
{
   people: [
      {name: 'John', town:'Oxford'},
      {name: 'Jack', town:'Bristol'}
   ]
}
```

This close resemblance to the model of the programming in some cases causes fast-changing formats.

Like XML attributes, as a serialised text format, JSON objects have an order but are almost always parsed to and from orderless maps meaning that the order of the keys/value pairings as seen in the stream usually follows no defined order.

No rule in the format would forbid representing of an ordered map in an ordered way but most tools on receiving such a message would ignore the ordering.

(MINE SOA assignment). Also the diagram.

3.7 Parsing: SAX and Dom

In the XML world two standard parser models exist, SAX and DOM, with DOM far the more popular. DOM performs a parse as a single evaluation, on the request of the programmer, returning an object model representing the whole of the document. At this level of abstraction the details of the markup are only distant concern. Conversely, SAX parsers are probably better considered as tokenisers, providing a very low-level event driven interface in line with the Observer pattern to notify the programmer of syntax as it is seen. Each element's opening and closing tag is noted

This presents poor developer ergonomics by requiring that the programmer implement the recording of state with regard to the nodes that they have seen. For programmers using SAX, a conversion to their domain objects is usually implemented imperatively. This programming tends to be difficult to read and programmed once per usage rather than assembled as the combination of reusable parts. For this reason SAX is usually reserved for fringe cases where messages are very large or memory unusually scarce.

DOM isn't just a parser, it is also a cross-language defined interface for manipulating the XML in real time, for example to change the contents of a web page in order to provide some interactivity. In JSON world, DOM-style parser not referring to the DOM spec, or what browser makers would mean. Rather, borrowing from the XML world to mean a parser which requires the whole file to be loaded.

Suppose we want to extract the name of the first person. Given a DOM parser this is very easy:

```
function nameOfFirstPerson( myJsonString ) {

   // Extracting an interesting part from JSON-serialised data is
   // relatively easy given a DOM-style parser. Unfortunately this
   // forbids any kind of progressive consideration of the data.
   // All recent browsers provide a JSON parser as standard.

   var document = JSON.parse( myJsonString );
   return document.people[0].name; // that was easy!
}
```

Contrast with the programming below which uses the clarinet JSON SAX parser. To prove that I'm not exaggerating the case, see published usages at [Clarinet demos].

```
function nameOfFirstPerson( myJsonString, callbackFunction ){

   // The equivalent logic, expressed in the most natural way
   // fora s JSON SAX parser is longer and much more
   // difficult to read. The developer pays a high price for
   // progressive parsing.

   var clarinet = clarinet.parser(),

       // with a SAX parser it is the developer's responsibility
       // to track where in the document the cursor currently is,
       // requiring several variables to maintain.
       inPeopleArray = false,
       inPersonObject = false,
       inNameAttribute = false,
       found = false;

   clarinet.onopenarray = function(){
      // for brevity we'll cheat by assuming there is only one
      // array in the document. In practice this would be overly
      // brittle.

      inPeopleArray = true;
   };

   clarinet.onclosearray = function(){
      inPeopleArray = false;
   };

   clarinet.onopenobject = function(){
      inPersonObject = inPeopleArray;
   };

   clarinet.oncloseobject = function(){
      inPersonObject = false;
   };

   clarinet.onkey = function(key){
      inNameAttribute = ( inPeopleObject && key == 'name');
   };

   clarinet.onvalue = function(value){
      if( !found && inNameAttribute ) {
         // finally!
         callbackFunction( value );
         found = true;
```

```
    }
  };

  clarinet.write(myJsonString);
}
```

As we can see above, SAX's low-level semantics require a lengthy expression and for the programmer to maintain state regarding the position in the document – usually recording the ancestors seen on the descent from the root to the current node – in order to identify the interesting parts. This order of the code is also quite unintuitive; generally event handlers will cover multiple unrelated concerns and each concern will span multiple event handlers. This lends to programming in which separate concerns are not separately expressed in the code.

3.8 Common patterns when connecting to REST services

Marshaling provides two-way mapping between a domain model and a serialisation as JSON or XML, either completely automatically or based on a declarative specification. To handle a fetched rest response it is common to automatically demarshal it so that the application may make use of the response from inside its own model, no differently from objects assembled in any other way. From the perspective of the programmer it is as if the domain objects themselves had been fetched. Another common design pattern, intended to give a degree of isolation between concerns, is to demarshal automatically only so far as Data Transfer Objects (DTOs), instances of classes which implement no logic other than storage, and from there programmatically instantiate the domain model objects. Going one step further, for JSON resources sent to loosely-typed languages with a native representation of objects as generic key-value pairs such as Javascript or Clojure, the marshaling step is often skipped: the output from the parser so closely resembles the language's built-in types that it is simplest to use it directly. Depending on the programming style adopted we might say that the JSON parser's output *is* the DTO and create domain model objects based on it, or that no further instantiation is necessary.

Ultimately the degree of marshaling that is used changes only the level of abstraction of the resource that the REST client library hands over to the application developer. Regardless of the exact form of the response model, the developer will usually programmatically extract one or more parts from it via calls in the programming language itself. For example, on receiving a resource de-marshaled to domain objects, a Java developer will inspect it by calling a series of getters in order to narrow down to the interesting parts. This is not to say that the whole of the message might not in some way be interesting, only that by using it certain parts will need to be identified as distinct areas of concern.

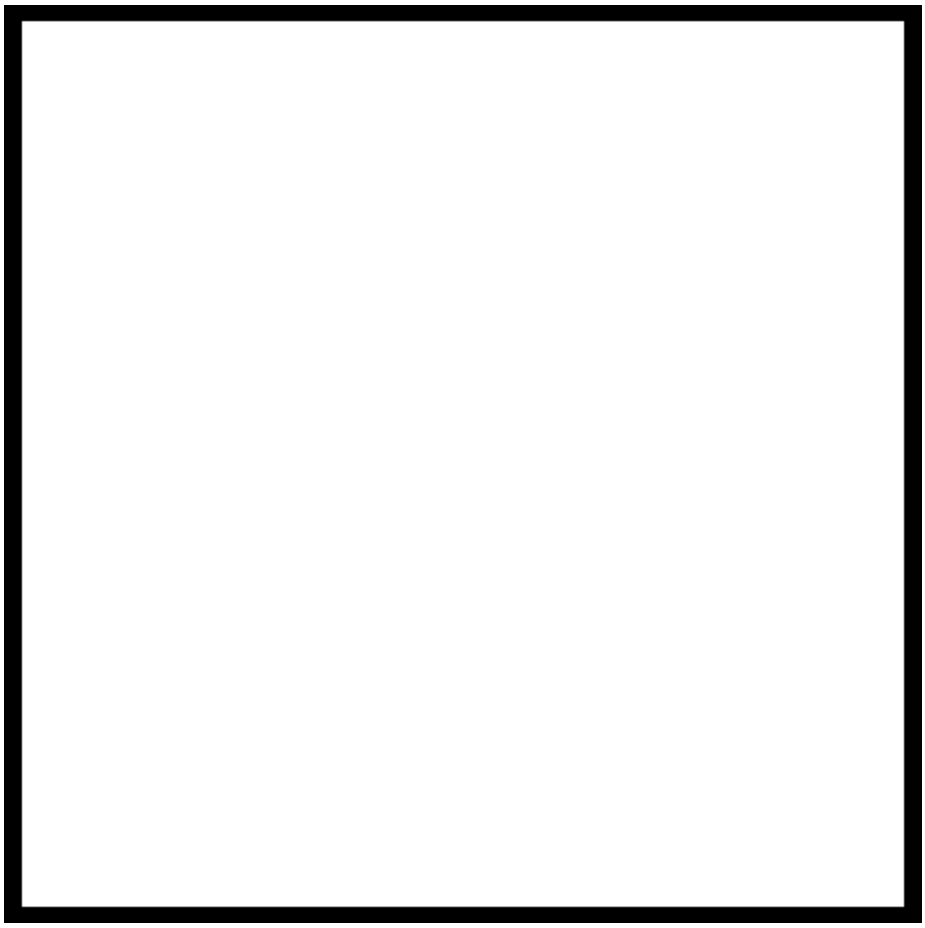

Figure 4: *Degrees of automatic marshaling.* From marshaling directly to domain objects, DTOs, using parser output as a DTO, or using objects directly. Distinguish work done by library vs application programmer's domain

```
// An example programmatic approach to a domain model interrogation
// under Java; upon receiving a list of people, each person's name
// is added to a database. The methods used to drill down to the
// pertinent components of the response are all getters: getPeople,
// getGivenName, and getSurname.
void handleResponse( RestResponse response ) {

   for( Person p : response.getPeople() ) {
      addNameToDb( p.getGivenName(), p.getSurname() );
   }
}


// Although in this Javascript example the objects passed to the handler
// remain in the form given by the JSON parser, containing no domain-specific
// getters, the programming represents a different expression of the same
// basic process.
function handleResponse( response ){

   response.people.forEach( function( person ){
      addNameToDb( p.givenName, p.surname );
   });
}
```

Because it is applied directly to the metamodel of the language[ˆ It could be argued that getters aren't a part of the metamodel of Java itself, but they form such a common pattern that it is a part], this extraction has become such a natural component of a workflow that it maye be used while thinking of it as wholly unremarkable. In the examples above we are interacting with the model in the way that the language makes the most easy to conceptualise. However se should consider that, however subtly embedded, the technique is an invented construct and only one of the possible formulations which might have been drawn.

One weakness of this inspection model is that, once much code is written to interrogate models in this way, the interface of the model becomes increasingly expensive to change as the code making the inspections becomes more tightly coupled with the thing that it is inspecting. Taking the above example, if the model were later refactored such that the concepts of firstName and surName were pulled from the Person class into an extracted Name class, because the inspection relies on a sequence of calls made directly into domain objects, the code making the query would also have to change. Whilst following the object oriented principle of encapsulation of data, such that the caller does not have to concern themselves with the data structures hidden behind the getter, there is no such abstraction for when the structure itself changes. Given an Agile environment where the shape of data is refactored regularly, this would be a problem when programming against any kind of resource; for example, if change

of objects formats propagates knock-on changes where ever the object is used it is very difficult to commit small diffs to the VCS which make incremental changes to a tightly focused area of the system. A method of programming which truly embraced extreme programming would allow constant change without disparate, barely related parts having to be modified in parallel when structural refactoring occurs. The coupling is all the more acute where the format of the item being inspected is defined by an independently maintained service.

contagion problem

Extraneous changes dilute the changelog, making it less easily defined by code changes which are intrinsically linked to the actual change in the logic being expressed by the program, and therefore to the thinking behind the change and the reason for the change.

3.9 JsonPath and XPath

Both the above difficulty in identifying the interesting parts of a message whilst using a streaming parser and the problem with tight coupling of programmatic drilling down to REST formats leads me to search for areas where this problem has already been solved.

In the domain of markup languages there are associated query languages such as XPATH whose coupling is loose enough that their expressions may continue to function after the exact shape of a message is refactored. While observing this is nothing more radical than using the query languages in more-or-less they were intended, their employment is not the most natural coming from a programming context in which the application developer's responsibilities usually start where the demarshaler's end. Consider the following XML:

```
<people>
   <person>
      <givenName>...</givenName>
      <familyName>Bond</familyName>
   </person>
</people>
```

The XPath //person[0]//surname//text() would continue to identify the correct part of the resource without being updated after the xml analogue of the above Java Name refactor:

```
<people>
   <person>
      <name>
         <givenName>...</givenName>
         <familyName>Bond</familyName>
```

```
      </name>
   </person>
</people>
```

Luckily in JSON there exists already an attempt at an equivalent named Jsonpath. JsonPath closely resembles the javascript code which would select the same nodes. Not a real spec.

```
// an in-memory person with a multi-line address:
let person = {
   name: {givenName:'', familyName:''},
   address: [
      "line1",
      "line2",
      "line3"
   ]
}


// in javascript we can get line two of the address as such:
let address = person.address[2]

// the equivalent jsonpath expression is identical:
let jsonPath = "person.address[2]"

// although jsonpath also allows ancestor relationships which are not
// expressible quite so neatly as basic Javascript:
let jsonPath2 = "person..given"
```

Xpath is able to express identifiers which often survive refactoring because XML represents a tree, hence we can consider relationships between entities to be that of contains/contained in (also siblings?). In application of XML, in the languages that we build on top of XML, it is very natural to consider all elements to belong to their ancestors. Examples are myriad, for example consider a word count in a book written in DOCBook format - it should be calculable without knowing if the book is split into chapters or not since this is a concept internal to the organisation of the book itself nd not something that a querier is likely to find interesting - if this must be considered the structure acts as barrier to information rather than enabling the information's delivery. Therefore, in many cases the exact location of a piece of information is not as important as a more general location of x being in some way under y.

This may not always hold. A slightly contrived example might be if we were representing a model of partial knowledge:

```
<people>
   <person>
      <name>
         <isNot><surname>Bond</surname></isNot>
      </name>
   </person>
</people>
```

The typical use pattern of XPath or JSONPath is to search for nodes once the whole serialisation has been parsed into a DOM-style model. JSONPath implementation only allows for search-type usage: https://code.google.com/p/jsonpath/To examine a whole document for the list of nodes that match a jsonpath expression the whole of the tree is required. But to evaluate if a single node matches an expression, only the *path of the descent from the root to that node* is required – the same state as a programmer usually maintains whilst employing a SAX parser. This is possible because JSONPath does not have a way to express the relationship with sibling nodes, only ancestors and decedents.

One limitation of the JSONPath language is that it is not possible to construct an 'containing' expression. CSS4 allows this in a way that is likely to become familiar to web developers over the next five years or so.

3.10 Testing

By the commonjs spec, test directory should be called 'test' (http://wiki.commonjs.org/wiki/Packages/1.0#Package_Directory_Layout) doesn't matter for my project since not using commonjs, but might as well stick to the convention.

How TDD helps How can fit into methodology

- JSTD
- NodeUnit
- Karma
- Jasmine

Initially started with jstestdriver but found it difficult. Karma started because engineers working on the Angular project in Google were "struggling a lot with jstd": http://www.youtube.com/watch?v=MVw8N3hTfCI - jstd is a google project Even Jstd's authors seems to be disowning it slightly. Describe what was once its main mode of operation as now being for stress testing of jstd itself only. Problems: browsers become unresponsive. Generally unreliable, has to be restarted frequently.

JSTD, as a Java program, is difficult to start via Grunt. Also an issue that Grunt post-dates Karma by enough that JSTD doesn't have the attention of the Grunt community.

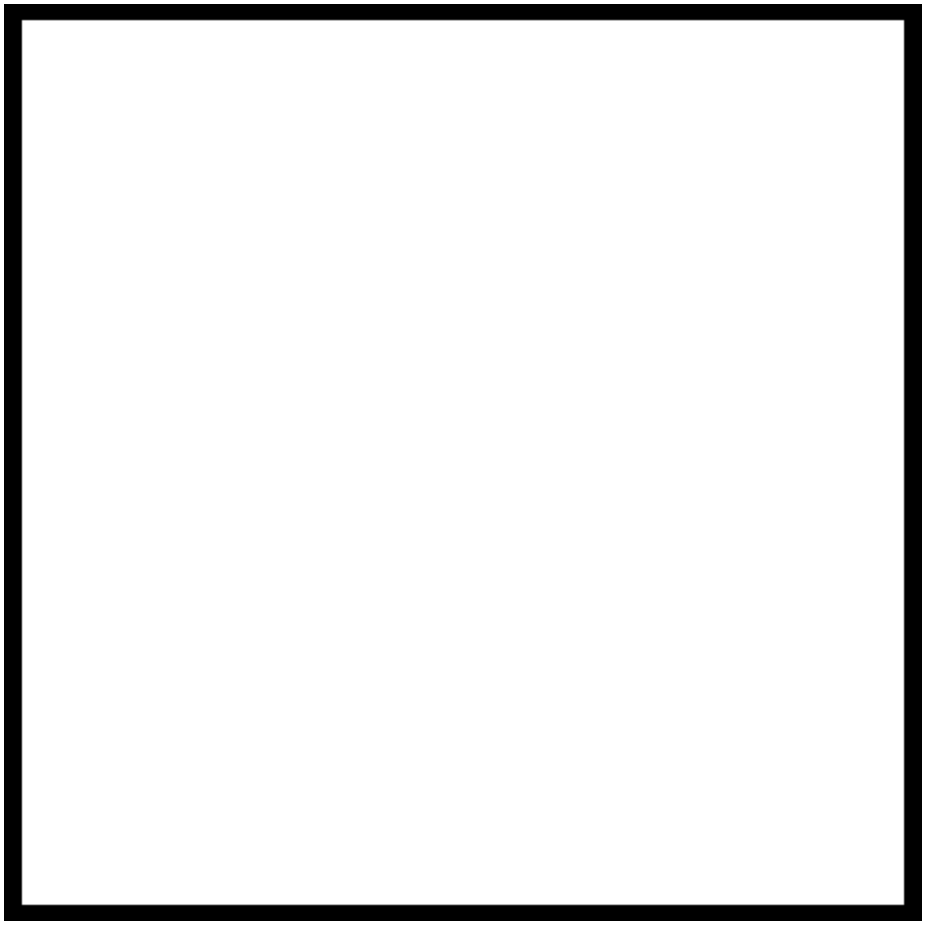

Figure 5: Relationship between the main players in the JS testing landscape. JSTD, Karma, Jasmine, NodeUnit, jasmine-node, Browsers

4 Design and Reflection:

Using a combination of the techniques investigated in the previous chapter, I propose that a simple design is possible which makes REST clients more efficient whilst being no more difficult to program. Although simple, this model fits poorly with established vocabulary, requiring a transport that sits *somewhere between 'stream' and 'download'* and a parsing strategy which *takes elements from SAX and DOM* but follows neither model.

Implementation in Javascript gives me the widest deployment options, covering client-side browser programming, server programming, use in command line tools, or any other usage. This context dictates a design which is non-blocking, asynchronous and callback based. While influenced by the language, the model of REST client proposed here is not limited to Javascript or web usage and I intent to comment briefly also on the applicability to other platforms. Likewise, I have also chosen to focus on JSON although I will also be commenting on the parallel applicability of these ideas to XML.

From DOM we may observe that as a programmer, using a resource is simpler when a parsed entity is passed whole to a single callback, rather than the SAX model which requires the programmer to infer the entity from a lengthy series of callbacks. From observing SAX parsers or progressive HTML rendering, we can say that http is more efficient if we no not wait until we have everything before we start using the parts that we do have. DOM parsers pass a fully parsed node to registered callbacks, whole and ready to use, invariably at the root of the parsed document. From the vantage of the library's user, my thesis duplicates this convenience but removes one restriction; that the node which is passed must be the root. Because the mark-up formats we are dealing with are hierarchical and serialised depth-first it is possible to fully parse any sub-tree without fully knowing the parent node. From these observations we may program a new kind of REST client which is as performant as SAX but as easy to program as DOM.

To follow this progressive-but-complete model, identifying the interesting parts of a document involves turning the traditional model for drilling down inside-out. Traditionally the programmer's callback receives the document then inside that callback drills down to locate the parts that they are interested in. Instead I propose taking the drilling-down logic out from inside the callback and instead wrap the callback in it. This means that the callback receives selected parts of the response which the library has already drilled down to on behalf of the programmer.

Whilst JSONPath's existing implementation is only implemented for searching over already gathered objects, this kind of searching is just one application for the query language. I find that this is a very suitable declarative language to use to specify the parts of a response that a developer would like to drill-down to given the context of a document whose parse is in progress. JSONPath is especially applicable because it specifies only 'contained-in/contains' type

relationships. On encountering any node in a serialised JSON stream, because of the depth-first serialisation order I will always have previously seen its ancestors. Hence, having written a suitably flexible JSONPath expression compiler such that it does not require a complete document, I will have enough information to evaluate any expression against any node at the time when it is first identified in the document. Because XML is also written depth-first, the same logic would apply to an XPath/XML variant of this project.

The definition of 'interesting' will be generic and accommodating enough so as to apply to any data domain and allow any granularity of interest, from large object to individual datums. With just a few lines of programming

4.1 JSONPath and types

Given its use to identify interesting parts of a document, not all of the published JSONPath spec is useful. Parts of a document will be considered interesting because of their type, position, or both. This contrasts with filter-type queries such as 'books costing less than X'. Examining REST responses it is likely we will not be explicitly searching through a full model but rather selecting from a resource subset that the programmer requested, assembled on their behalf using their parameters so we can expect the developer to be interested in most of the content. In creating a new JSONPath implementation, I have chosen to follow the published spec only loosely, thereby avoiding writing unnecessary code. This is especially the case, as in the books example above whereby a user of the library could easily add the filter in the callback itself. Following the principle of writing less, better I feel it is better to deliver only the features I am reasonably certain will be well used but keep open the ability to add more later should it be required.

JSON markup describes only a few basic types. On a certain level this is also true for XML – most nodes are of either type Elements or Text. However, the XML metamodel provides tagnames, essentially a built-in Element sub-typing mechanism. Floating above this distinction, a reader abstracting over the details of the markup may forget that a node is an Element instance and describe it as an instance of its tagname, without considering that the tagname is a sub-type of Element. JSON comes with no such built-in type description language. On top of JSON's largely typeless model we often place a concept of type. Drawing parallels with the physical world, this imposition of type is the responsibility of the observer, rather than of the observed. A document reader has a free choice of the taxonomy they will use to impose type on the parts of the document, and this decision will vary depending on the purpose of the reader. The specificity required of a taxonomy differs by the level of involvement in a field, whereas 'watch' may be a reasonable type to most data consumers, to a horologist it is unlikely to be satisfactory without further sub-types. In the scope of this dissertation, since selecting on type is desirable, my JSONPath variant must be able to distinguish types at various levels of specificity; whilst my selection

language will have no inbuilt concept of type, the aim is to support programmers in creating their own.

integrate with above or discard, maybe move to compatibility with future versions Relationship between type of a node and its purpose in the document (or, perhaps, the purpose the reader wishes to put it to). Purpose is often obvious from a combination of URL and type so can disregard the place in the document. This structure may be carefully designed but ultimately a looser interpretation of the structure can be safer.

```
<!--  XML leaves no doubt as to the labels we give to the types
      of the nodes. This is a 'person' -->
<person  name='...' gender="male"
         age="45" height="175cm" profession="architect">
</person>

/*    JSON meanwhile provides no such concrete concept. This node's
      type might be 'thing', 'animal', 'human', 'man', 'architect',
      'artist' or any other of many overlapping impositions depending
      on what purpose the document it is read for */
{  "name":"...", "gender":"male", "age":"45"
   "height":"175cm" "profession":"architect">
}
```

In the absence of node typing beyond the categorisation as objects, arrays and various primitive types, the key immediately mapping to the object is often taken as a lose concept of the type of the object. Quite fortunately, rather than because of a well considered object design, this tends to play well with automatically marshaling of domain objects expressed in a Java-style OO language because there is a strong tendency for field names – and by extension, 'get' methods – to be named after the *type* of the field, the name of the type also serving as a rough summary of the relationship between two objects. See figure **??** below.

In the below example, we impose the the type 'address' because of the parent node's field name. Other than this, these are standard arrays of strings:

```
{
   name: '...'
,  residence: {
      address: [
         '...', '...', '...'
      ]
   }
,  employer: {
      name: '...'
   ,  address :[
```

```
         '...', '...', '...'
      ]
   }
}
```

Although, being loosely typed, in Javascript there is no protection against using arrays to contain disparate object, by sensible convention the items will usually be of some common type. Likewise in JSON, although type is a loose concept, on some level the elements of an array will generally be of the same type. This allows a sister convention seen in the below example, whereby each of a list of items are typed according to the key in the grandparent node which maps to the array.

```
{
   residences: {
      addresses: [
         ['Townhouse', 'Underground street', 'Far away town']
      ,  ['Beach Hut', 'Secret Island', 'Bahamas']
      ]
   }
}
```

The pluralisation of 'address' to 'addresses' above may be a problem to a reader wishing to detect address nodes. I considered introducing an 'or' syntax for this situation, resembling `address|addresses.*` but instead decided this problem, while related to type, is simpler to solve outside of the JSONPath language. A programmer may simply use two JSONPaths mapping to the same callback function.

In the below example typing is trickier still.

```
{
   name: '...'
,  residence: {
      number:'...', street:'...', town:'...'
   }
,  employer:{
      name: '...'
   ,  premises:[
         { number:'...', street:'...', town:'...' }
      ,  { number:'...', street:'...', town:'...' }
      ,  { number:'...', street:'...', town:'...' }
      ]
   ,  registeredOffice:{
         number:'...', street:'...', town:'...'
```

```
      }
   }
}
```

The properties holding addresses are named by the relationship between the parent and child nodes rather than the type of the child. There are two ways we may be able to select objects out as addresses. Firstly, because of an ontology which subtypes 'residence', 'premises', and 'office' as places with addresses. More simply, we may import the idea of duck typing from Python programing.

> In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.

Discussion of typing in Python language, 2000. https://groups.google.com/forum/?hl=en#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ

A 'duck-definition' of address might be any object which has a number, street and town. That is to say, type is individualistically communicated by the object itself rather than by examining the relationships described by its containing ancestors. JSONPath comes with no such expressivity but I find this idea so simple and useful that I have decided to create one. The JSONPath language is designed to resemble programmatic Javascript access but Javascript has no syntax for a list of value-free properties. The closest available is the object literal format; my duck-type syntax is a simplification with values and commas omitted. In the case of the addresses a duck-type expression would be written as `{number street town}`. Generally, when identifying items of a type from a document it makes sense if the type expression is contravariant so that sub-types are also selected. If we consider that we create a sub-duck-type when we add to a list of required fields and super-duck-types when we remove them, we have a non-tree shaped type space with root type `{}` which matches any object. Therefore, the fields specified need not be an exhaustive list of the object's properties.

The various means of discerning type which are constructable need not be used exclusively. For example, `aaa{bbb ccc}` is a valid construction combining duck typing and the relationship with the parent object.

4.2 JSONPath improving stability over upgrades

need to look at this an check doesn't duplicate rest of diss.

- Use of `..` over `.`
- Keep this short. Might not need diagram if time presses.

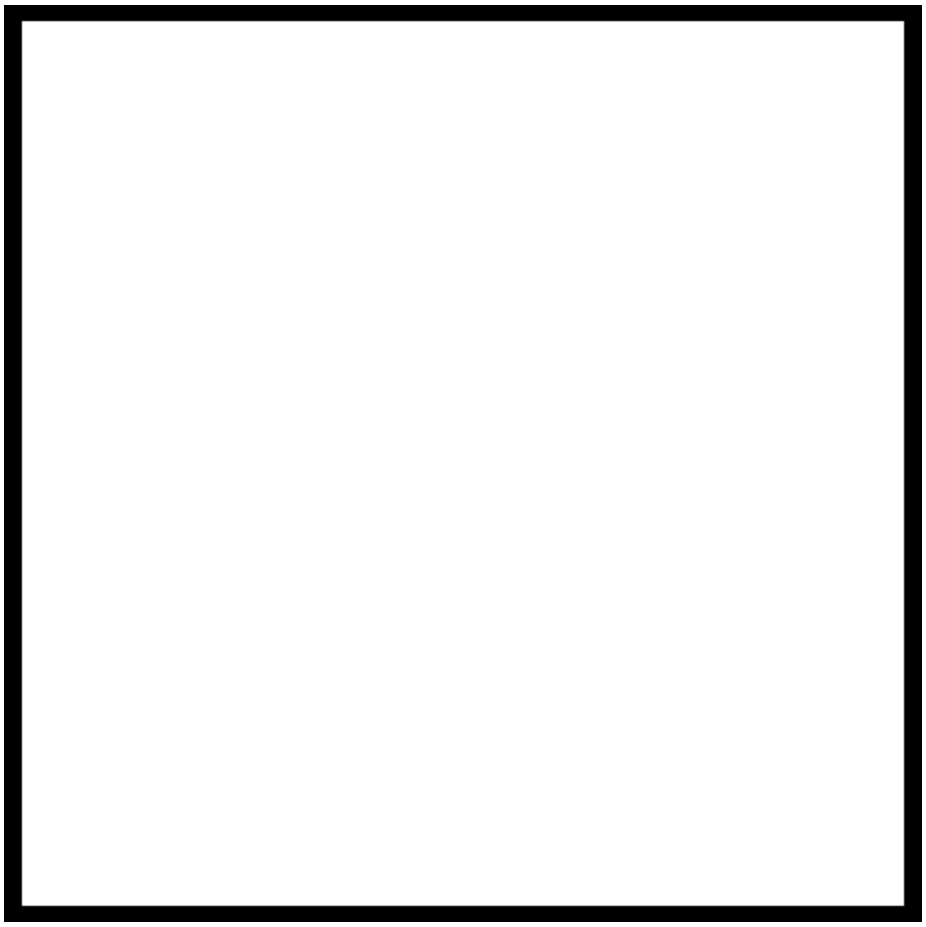

Figure 6: extended json rest service that still works - maybe do a table instead

Programming to identify a certain interesting part of a resource today should with a high probability still work when applied to future releases.

Requires some discipline on behalf of the service provider: Upgrade by adding of semantics only most of the time rather than changing existing semantics.

Adding of semantics should could include adding new fields to objects (which could themselves contain large sub-trees) or a "push-down" refactor in which what was a root node is pushed down a level by being suspended from a new parent.

why JSONPath-like syntax allows upgrading message semantics without causing problems [SOA] how to guarantee non-breakages? could publish 'supported queries' that are guaranteed to work

4.3 Importing CSS4 selector capturing to Oboe JSONPath

Sometimes when downloading a collection of items it is less useful to be given each element individually than being kept up to date as the collection is expanded. Certain Javascript libraries such as d3.js and Angular interface more naturally with arrays of items than individual entities. To allow integration with these libraries, on receiving an array of items it is useful to be repeatedly passed the same containing array whenever a new element is concatenated onto it.

Expressing a 'contained in' relationship comes naturally to JSONPath, but no provision is made for a 'containing' relationship. Cascading Style Sheets, or CSS, the web's styling language has long shared this restriction but a recent proposal, currently at Editor's Draft stage (Etemad and Atkins 2013) provides an elegant means to cover this gap. Rather than add an explicit 'containing' relationship, the css4 proposal observes that css selectors have previously only allowed selection of the right-most of the terms given, allowing only the deepest element mentioned to be selected. This restriction is removed by allowing terms may be prefixed with `$` in order to make them capturing: in the absence of an explicitly capturing term the right-most continues to capture. Whereas `form.important input.mandatory` selects for styling mandatory inputs inside important forms, `$form.important input.mandatory` selects important forms with mandatory fields.

Importing the CSS4 dollar into Oboe's JSONPath should make it much easier to integrate with libraries which treat arrays as their basic unit of operation and uses a syntax which the majority of web developers are likely to be familiar with over the next few years.

4.4 Parsing the JSON Response

While SAX parsers provide an unfriendly interface to application developers, as a starting point for higher-level parsers they work very well (in fact, most XML DOM parsers are made in this way). The pre-existing project Clarinet is well tested, liberally licenced and compact, meeting the goals of this project perfectly. In fact, the name of this project, Oboe.js, was chosen in tribute to the value delivered by Clarinet.

4.5 API design

In designing the API developer ergonomics are the top priority. This is especially pertinent given that the library does nothing that can't be done with existing tools such as JSON SAX parsers but that those tools are not used because they require too much effort to form a part of most developers' everyday toolkit.

Expose single global.

To pursue good ergonomics, I will study successful libraries and, where appropriate, copy their APIs. We may assume that the existing libraries have already over time come to refined solutions to similar problems. Working in a style similar to existing libraries also makes the library easier to learn. Lastly, if we create a library which functions similarly enough to existing tools it should be easy to modify an existing project to adopt it. In the most common use cases, it should be possible to create a library with a close functional equivalence that can be used as a direct drop-in replacement. Used in this way, no progressive loading would be done but it opens the door for the project taking up the library to be refactored towards a progressive model over time. By imitating existing APIs we allow adoption as a series of small, easily manageable steps rather than a single leap. This is especially helpful for teams wishing to adopt this project working under Scrum because all tasks must be self-contained and fit within a fairly short timeframe.

jQuery's basic call style for making an AJAX GET request follows:

```
jQuery.ajax("resources/shortMessage.txt")
   .done(function( text ) {
      console.log( 'Got the text: ' + text );
   }).
   .fail(function(data) {
      console.log( 'the request failed' );
   });
```

While for simple web applications usage is much as above,
In real world usage on more complex apps jQuery.ajax is often injected into the

scope of the code which wants to use it. Easier stubbing so that tests don't have to make actual AJAX calls.

While certainly callback-based, the jQuery is somewhat implicit in being event-based. There are no event names separate from the methods which add the listeners and there are no event objects, preferring to pass the content directly. The names used to add the events (done, fail) are also generic, used for all asynchronous requests. The methods are chainable which allows several listeners to be added in one statement.

By method overloading, if the request requires more information than the parameter to `jQuery.ajax` may be an object. This pattern of accepting function parameters as an object is a common in Javascript for functions that take a large number of optional arguments because it makes understanding the purpose of each argument easier to understand from the callsite than if the meaning depended on the position in a linear arguments list and the gaps filled in with nulls.

```
jQuery.ajax({ url:"resources/shortMessage.txt",
              accepts: "text/plain",
              headers: { 'X-MY-COOKIE': '123ABC' }
           });
```

Taking on this style,

```
oboe('resources/someJson.json')
   .node( 'person.name', function(name, path, ancestors) {
      console.log("got a name " + name);
   })
   .done( function( wholeJson ) {
      console.log('got everything');
   })
   .fail( function() {
      console.log('actually, the download failed. Forget the' +
                  ' people I just told you about');
   });
```

Because I foresee several patterns being added for most types of JSON documents, a shortcut format is also available for adding multiple patterns in a single call by using the patterns as the keys and the callbacks as the values in a key/value mapping:

```
oboe('resources/someJson.json')
   .node({
      'person.name': function(personName, path, ancestors) {
```

```
      console.log("let me tell you about " + name);
   },
   'person.address.town': function(townName, path, ancestors) {
      console.log("they live in " + townName);
   }
});
```

Note the path and ancestors parameters in the examples above. Most of the time giving the callback the matching content is enough to be able to act but it is easy to imagine cases where a wider context matters. Consider this JSON:

```
{
   "event": "mens 100m",
   "date": "5 Aug 2012",
   "medalWinners": {
      "gold":     {"name": 'Bolt',    "time": "9.63s"},
      "silver":   {"name": 'Blake',   "time": "9.75s"},
      "bronze":   {"name": 'Gatlin',  "time": "9.79s"}
   }
}
```

Here we can extract the runners by the patterns such as `{name time}` or `medalWinners.*` but clearly the location of the node in the document is interesting as well as the context. The `path` parameter provides this information by way of an array of strings plotting the descent from the JSON root to the match, for example `['medalWinners', 'gold']`. Similarly, the `ancestors` array is a list of the ancestors starting at the immediate parent of the found node and ending with the JSON root node. For all but the root node (which has no ancestors anyway) the nodes in this list will be only partially parsed. Being untyped, Javascript does not enforce the arity of the callback. Because much of the time only the content itself is needed, the API design orders the callback parameters to take advantage of the loose typing so that a unary function taking only the content may be given.

For the widest context currently available, the whole document as it has been parsed so far may be accessed using the `.root` method. Since `.root` relates to the oboe instance itself rather than the callback per-say, it can be accessed from any code with a reference to the oboe object.

`http://nodejs.org/docs/latest/api/events.html#events_emitter_on_event_listener`

In node.js the code style is more obviously event-based. Listeners are added via a `.on` method with a string event name given as the first argument. Adopting this style, my API design for oboe.js also allows events to be added as:

```
oboe('resources/someJson.json')
   .on( 'node', 'medalWinners.*', function(person, path, ancestors) {
```

```
      console.log( person.name + ' won the ' + lastOf(path) + ' medal' );
   });
```

While allowing both styles uncountably creates an API which is larger than it needs to be, creating a library which is targeted at both the client and server side, I hope this will help adoption by either camp. The Two styles are similar enough that a person familiar with one should be able to pick up the other without difficulty. In implementation a duplicative API should require only a minimal degree of extra coding because these parts may be expressed in common and their scope reduced using partial completion. Because `'!'` is the JSONPath for the root of the document, for some callback c, `.done(c)` is a synonym for `.node('!', c)` and therefore below a thin interface layer may share an implementation. Likewise, `.node` is easily expressible as a partial completion of `.on` with `'node'`.

4.6 Earlier callbacks when paths are matched

Following with the project's aim of giving callbacks as early as possible, sometimes useful work can be done when a node is known to exist but before we have the contents of the node. This means that each node found in a JSON document has the potential to trigger notifications at two points: when it is first discovered and when it is complete. The API facilitates this by providing a `path` callback following much the same pattern as the `node` callback.

```
oboe('events.json')
   .path( 'medalWinners', function() {
      // We don't know the winners yet but we know we have some so let's
      // start drawing the table already:
      interface.showMedalTable();
   })
   .node( 'medalWinners.*', function(person, path) {
      interface.addPersonToMedalTable(person, lastOf(path));
   })
   .fail( function(){
      // That didn't work. Revert!
      interface.hideMedalTable();
   });
```

In implementation providing path notifications is a simple matter of allowing the evaluation of the json path expressions when items are pushed to the stack of current nodes in addition to when they are popped.

4.7 Oboe.js as a Micro-Library

Http traffic, especially sending entropy-sparse text formats is often gzipped at point of sending in order to deliver it more quickly, so in measuring a download footprint it usually makes more sense to compare post-gzipping. A Javascript library qualifies as being *micro* if it is delivered in 5k or less, 5120 bytes. Micro-libraries also tend to follow the ethos that it is better for a developer to gather together several tiny libraries than one that uses a one-size-fits-all approach, perhaps echoing the unix command line tradition of small programs which each do do exactly one thing. Javascript Micro-libraries are listed at,[4] which includes this project. Oboe.js feels on the edge of what is possible to elegantly do as a micro-library so while the limit is somewhat arbitrary, for the sake of adoption smaller is better and keeping below this limit whilst writing readable code is an interesting challenge. As well as being a small library, in the spirit of a micro-library a project should impose as few restrictions as possible on its use and be designed to be completely agnostic as to which other libraries or programming styles that it is used with.

4.8 Handling transport failures

Oboe should allow requests to fail while the response is being received without necessarily losing the part that was successfully received.

Researching error handing, I considered the option of automatically resuming failed requests without intervention from the containing application. Http 1.1 provides a mechanism for Byte Serving via the `Accepts-Ranges` header [http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.5] which is used to request any contiguous fragment of a resource – in our case, the part that we missed when the download failed. Having examined this option I came to the conclusion that it would encourage brittle systems because it assumes two requests to the same URL will give byte-wise equal responses.

A deeper problem is that Oboe cannot know the correct behaviour when a request fails so this is better left to the containing applications. Generally on request failure, two behaviours may be anticipated. If the actions performed in response to data received up to time of failure remain valid in the absence of a full transmission, their effects may be kept and a URL may be constructed to request just the lost part. Alternatively, under optimistic locking, the application developer may choose to perform rollback. In either case, responding to errors beyond informing the calling application is outside of Oboe's scope.

IO errors in a non-blocking system cannot be handled via exception throwing because the call which will later cause an error will no longer be on the stack at the time that the error occurs. Error-events will be used instead.

[4] Https://github.com/substack/http-browserify.

4.9 Fallback support on less-capable platforms

something about market share and link to figures in an appendix?

Because of differences in the capabilities in browsers, providing a streaming REST client is not possible on all browsers. If this were possible, it would not have been necessary to invent push pages or long polling. Specifically, none but the most recent versions of Internet Explorer provide any way to access an AJAX response before it is complete. I have taken the design decision that it is ok to degrade on these platforms so long as the programmer developing with Oboe.js does not have to make special cases for these platforms. Likewise, nor should the REST service need be aware of the client, disallowing detecting client capabilities and switching transport strategy. Requiring branching on either side places extra responsibilities on the programmer which they would not otherwise be required to consider whilst viewing REST through a non-streaming lens.

Given that streaming is not possible on older platforms, I must considering the best experience that is possible. We may imagine a situation in which the whole download completes followed by all listeners being notified from a single Javascript frame of execution. While not progressive in any way, this situation is essentially standard REST plus JSONPath routing and no less performant than if more traditional libraries were used. I find this satisfactory: for the majority of users the experience is improved and for the others it is made no worse, resulting in a net overall benefit.

In the Javascript language itself interoperability is very rarely an issue. Javascript's model of prototypical inheritance allows changes to be made to the browser's libraries on the fly; as soon as a prototype is changed all instances of the type reflect the change even if they has already been created (source). Because the base types that come with the browser are essentially global, changing them for the use of a single codebase is generally deprecated because of the possibility of collisions. However, this technique is often used to retrofit new standards onto older platforms. For example, the Functional-style Array iteration methods remove the need to write C-style for loops and are defined in the ECMAScript 5 specification http://www.jimmycuadra.com/posts/ecmascript-5-array-methods - all of these methods are implementable in pure Javascript. There exist several mature pure Javascript projects for browsers which lack native support, licenced to allow inclusion in this project (CITE ONE). While I am constrained in the ability to accept streaming AJAX in older browsers, there is no such restriction on my ability to express my thesis in a more modern, functional style of Javascript.

Node is highly capable, with no shortcomings that will make Oboe.js difficult to implement. It does, however use its own stream API rather than emulate the browser API so will require platform-specific programming inside the library. This abstraction will be hidden from the library user so will not require any special programming on their part.

5 Implementation

5.1 Components of the project

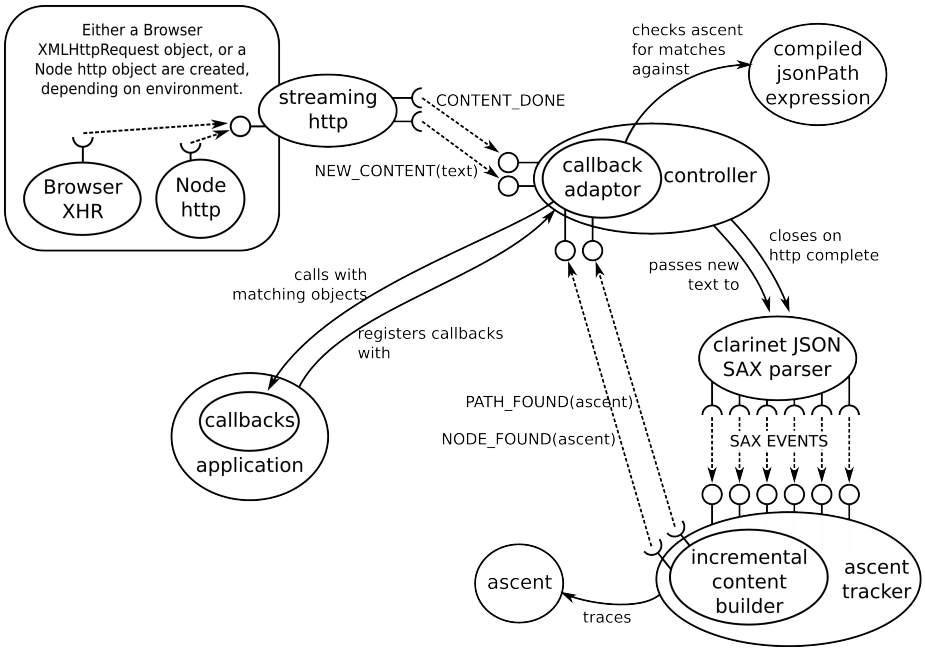

Figure 7: **Major components that make up Oboe.js** showing flow from http transport to registered callbacks. Every component is not shown here. Particularly, components whose responsibility it is to initialise the oboe instance but have no role once it is running are omitted. UML facet/receptacle notation is used to show the flow of events with event names in capitals.

Oboe's architecture has been designed to so that I may have as much confidence as possible regarding the correct working of the library through automated testing. Designing a system to be amenable to testing in this case meant splitting into many co-operating parts each with an easily specified remit.

Although there is limited encapsulation to follow an OO-style arrangement, data is hidden. Outside of Oboe, only a restricted public API is exposed. Not only are the internals inaccessible, they are unaddressable. With no references attached to any external data structures, most data exists only as captured within closures.

Internally, communication between components is facilitated by an event bus which is local to to Oboe instance, with most components interacting solely by picking up events, processing them and publishing further events in response. Essentially, Oboe's architecture resembles a fairly linear pipeline visiting a series of units, starting with http data and sometimes ending with callbacks being

notified. This use of an event bus is a variation on the Observer pattern which removes the need for each unit to obtain a reference to the previous one so that it may observe it, giving a highly decoupled shape to the library. Once everything is wired into the bus very little central control is required and the larger behaviours emerge as the consequence of this interaction between finer ones. One downside is perhaps that a central event bus does not lend itself to a UML class diagram, giving a diagram shape with an event bus as a central hub and everything else hanging off it as spokes.

5.2 Automated testing

Automated testing improves what can be written, not just making what is written more reliable. Tests deal with the problem of "irreducible complexity" - when a program is made out of parts whose correct behaviour cannot be observed without all of the program. Allows smaller units to be verified before verifying the whole.

The testing itself is a non-trivial undertaking with 80% of code written for this project being test specifications. Based on the idea that a correct system must be built from individually correct units, the majority of the specifications are unit tests, putting each unit under the microscope and describing the correct behaviour as completely as possible. Component tests zoom out from examining individual components to focus on their correct composition, falsifying only the http traffic. To avoid testing implementation details the component tests do not look at the means of coupling between the code units but rather check for the behaviours which should emerge as a consequence of their composition. At the apex of the test pyramid are a small number of integration tests. These tests check all of Oboe, automatically spinning up a REST service so that the correctness of the whole library may be examined against an actual server.

The desire to be amenable to testing influences the boundaries on which the application splits into components. Black-box unit testing of a stateful unit is difficult; because of side-effects it may later react differently to the same calls. For this reason where state is required it is stored in very simple state-storing units with intricate program logic removed. The logic may then be separately expressed as functions which map from one state to the next. Although comprehensive coverage is of course impossible and tests are inevitably incomplete, for whatever results the functions give while under test, uninfluenced by state I can be sure that they will continue to give in any future situation. The separate unit holding the state is trivial to test, having exactly one responsibility: to store the result of a function call and later pass that result to the next function. This approach clearly breaks with object oriented style encapsulation by not hiding data behind the logic which acts on them but I feel the departure is worthwhile for the greater certainty it allows over the correct functioning of the program.

Largely for the sake of testing Oboe has also embraced dependency injection. This means that components do not create the further components that they

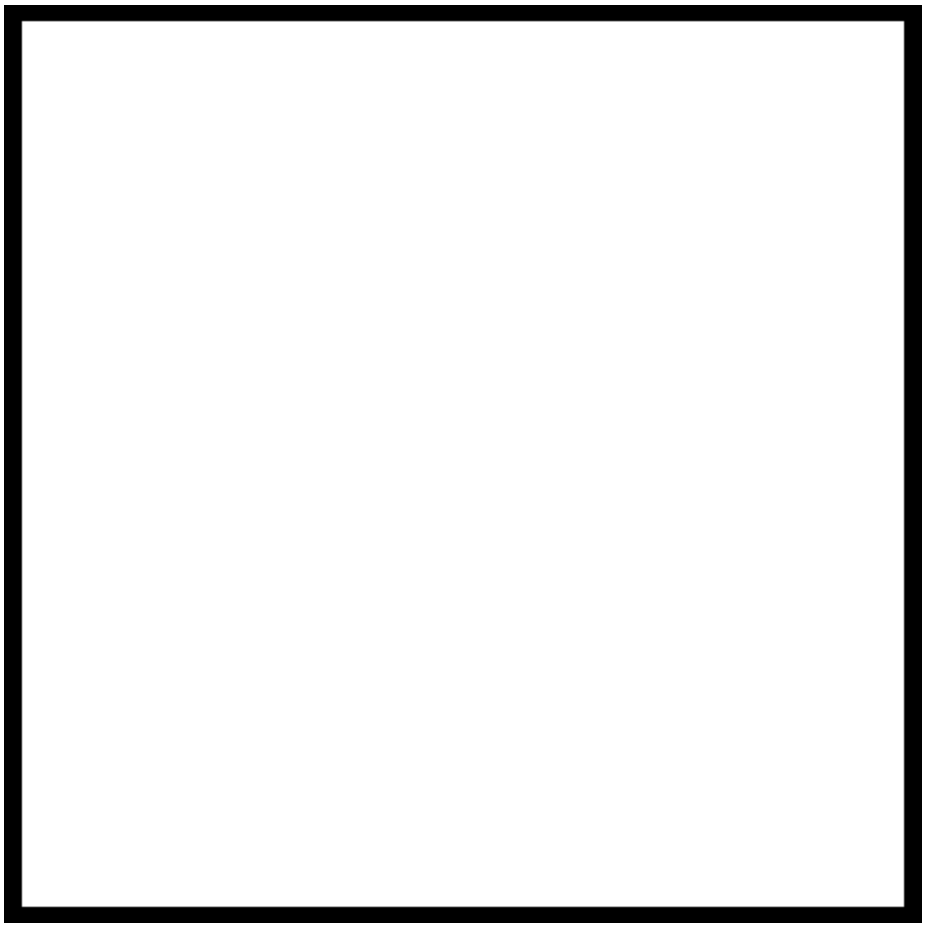

Figure 8: The testing pyramid is a common concept, relying on the assumption that verification of small parts provides a solid base from which to compose system-level behaviours. A Lot of testing is done on the low-level components of the system, less on the component level and less still on a whole-system level where only smoke tests are provided.

require but rather rely on them being provided by an external wiring. The file `wire.js` performs the actual injection. One such example is the streamingHttp component which hides various incompatible http implementations by publishing their downloaded content progressively via the event bus. This unit does not know how to create the underlying browser XHR which it hides. Undoubtedly, by not instantiating its own dependencies a it presents a less friendly interface, although this is mitigated somewhat by the interface being purely internal, the objects it depends on are no longer a hidden implementation detail but exposed as a part of the component's API. The advantage of dependency injection here is that unit testing is much simpler. Unit tests should test exactly one behaviour of one unit. Were the streaming http object to create its own transport, that part would also be under test, plus whichever external service that it connects to. Because Javascript allows redefinition of built in types, this could be avoided by overwriting the XHR constructor to return a mock but modifying the built in types for tests opens up the possibilities of changes leaking between cases. Dependency injection allows a much simpler test style because it is trivial to inject a stub in place of the XHR.

Integration tests run against a node service which returns known content according to known timings, somewhat emulating downloading via a slow internet connection. For example, the url `/tenSlowNumbers` writes out a JSON array of the first ten natural numbers at a rate of one per second, while `/echoBackHeaders` writes back the http headers that it received as a JSON object. The test specifications which use these services interact with Oboe through the public API alone as an application author would and try some tricky cases. For example, requesting ten numbers but registering a listener against the fifth and aborting the request on seeing it. The correct behaviour is to get no callback for the sixth, even when running on platforms where the http is buffered so that all ten will have already been downloaded.

5.3 Running the tests

The Grunt task runner was used to automate routine tasks such as executing the tests and building. Unit and component tests run automatically whenever a source file changes. As well as being correct execution, the project is required to not surpass a certain size so the built size is also checked. As a small, tightly focused project the majority of programming is refactoring already working code. Running tests on save provides quick feedback so that mistakes are found as soon as they are made. Agile practitioners emphasise the importance of tests that execute quickly (Martin 2008, T9), the 220 unit and component tests run in less than a second so discovering mistakes is near instant. If the "content of any medium is always another medium" (McLuhan 1964 p8), we might say that the content of programming is the program that is realised by its execution. A person working in arts and crafts sees the thing as they work but a programmer will usually not see the execution simultaneously as they program. Conway observed

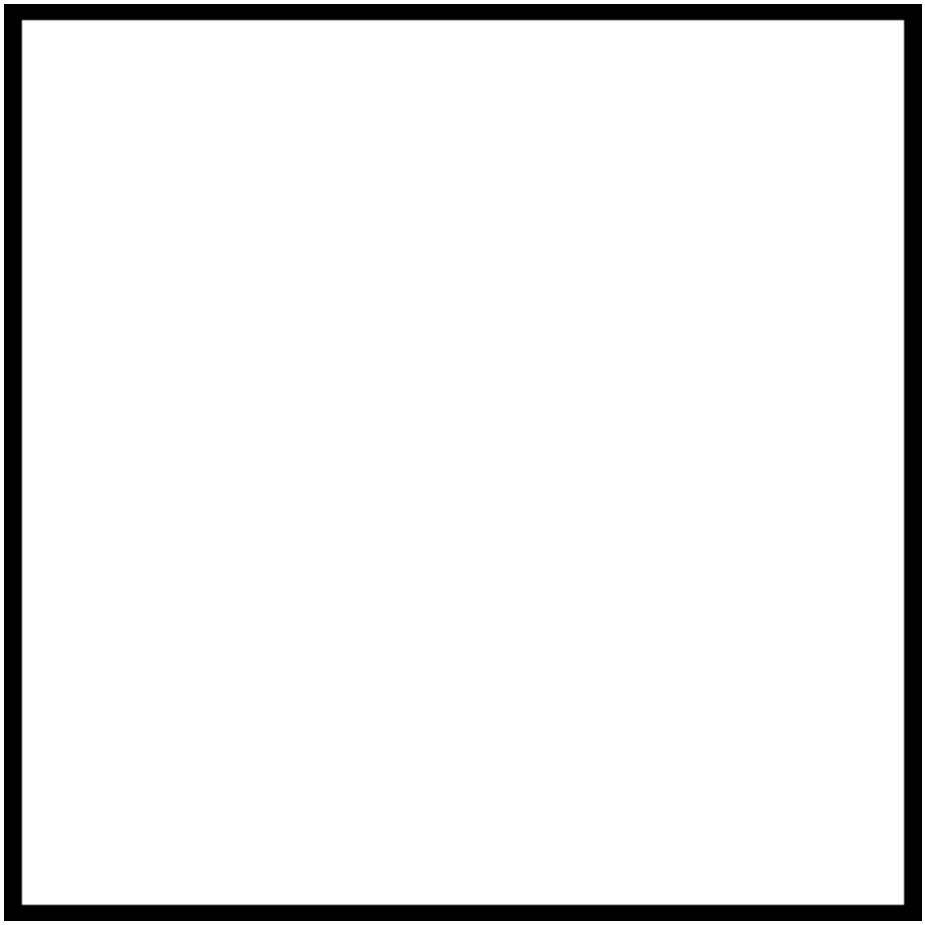

Figure 9: **Relationship between various files and test libraries** *other half of sketch from notebook*

that an artisan works by transform-in-place "start with the working material in place and you step by step transform it into its final form" whereas software is created through intermediate proxies, and attempts to close this gap by merging programming with the results of programming (Conway 2004 side8-9). When we bring together the medium and the message the cost of small experimentation is very low and I feel that programming becomes more explorative and expressive.

The integration tests are not run on save because they intentionally simulate slow transfers and take some time to run. The integration tests are used as a final check against built code before a branch in git can be merged into the master. Once the code has been packaged for distribution the internals are no longer visible the integration tests which are coded against the public API are the only runnable tests. While these tests don't individually test every component, they are designed to exercise the whole codebase so that a mistake in any component will be visible through them. Grunt executes the build, including starting up the test REST services that give the integration tests something to fetch.

5.4 Packaging as a single distributable file

As an interpreted language, Javascript may of course be ran directly without any prior compilation. While running the same code as I see in the editor is convenient while programming, it is much less so for distribution. Although the languages imposes no compulsory build phase, in practice one is necessary. Dependency managers have not yet become standard for client-side web development (although Bower is looking good) so most files are manually downloaded. For a developer wishing to include my library in their own project a single file is much more convenient. Should they not have a build process of their own, a single file is also much faster to transfer to their users, mostly because of the cost of establishing connections and the http overhead.

Javascript files are interpreted in series by the browser so dependencies must come before dependants, tricks for circular dependencies notwithstanding. Unsurprisingly, files concatenated in the same order as they would be delivered to the browser will be functionally equivalent to the same files delivered separately. Several tools are available to automate this stage of the build process, including the topological sort of the dependency digraph that finds a working concatenation order.

Early in this project I chose Require.js although it was later abandoned because I found it to be too heavyweight. Javascript as a language doesn't have an import statement so Require implements importing in Javascript itself as a normally executable function. When running raw source, this function AJAXes in the imported source but Require also has an 'optimise mode' which uses static analysis to deduce a workable source order and concatenates into a single file. This of course is impossible in the general case and if, for example, the import is subject to branching require falls back to lazy loading, fetching only when

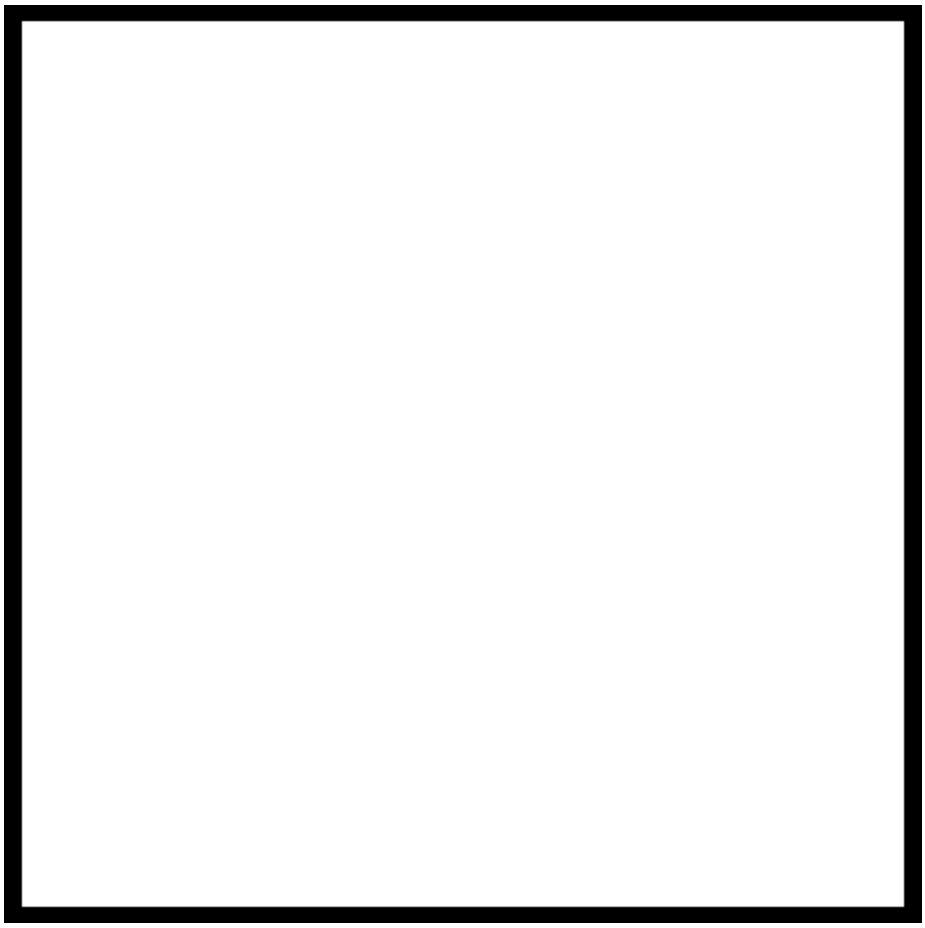

Figure 10: **Packaging of many javascript files into multiple single-file packages.** The packages are individually targeted at different execution contexts, either browsers or node *get from notebook, split sketch diagram in half*

needed. In practice undecidability isn't an issue because importing is typically unconditional and may even be an advantage in larger projects. To speed up initial loading of larger web applications, *Asynchronous Module Definition* (AMD) imports rarely-loaded functionality in response to events, resisting static analysis and so downloading the code only as it is needed.

I hoped to use Require's `optimise` to generate my distributable Javascript library. However, require's structure necessitates that calls to the importing functions stay in the code and that the require.js run-time component is available to facilitate these calls. For a small project I found this constant-size overhead too large in relation to the rest of the project. Require also prefers to be used as the sole means of loading scripts meaning that Oboe would not fit naturally into web applications not already using require. Overall, Require seems more suited to developing whole applications than programming libraries.

Having abandoned require rather than look for another sophisticated means to perform concatenation I decided to pick up the simplest tool which could possibly work, a Grunt module which works like the unix `cat` command. With only 15 source Javascript files manually finding a working order by drawing a graph on paper isn't a daunting task. As new files are added it is simple to find a place to insert them into the list. I adjusted each Javascript file to, when loaded directly, place its API in the global namespace, then post-concatenation wrapped the combined in a single function, converting the APIs inside the function from global to the scope of that function, thereby hiding the implementation for code outside of Oboe.

For future consideration there is Browserify. This library reverses the 'browser first' image of Javascript by converting applications targeted at Node into a single file efficiently packaged for delivery to a web browser, conceptually making Node the primary environment for Javascript and adapting browser execution to match. Significantly, require leaves no trace of itself in the concatenated Javascript other than Adaptors presenting browser APIs as the Node equivalents. Browserify's http adaptor[5] is complete but more verbose compared to Oboe's version[6].

As well as combining into a single file, Javascript source can made significantly smaller by removing comments and reducing inaccessible tokens to a single character. For Oboe the popular library *Uglify* is used for minification. Uglify performs only surface optimisations, operating on the AST level but concentrating mostly on compact syntax. I also considered Google's Closure compiler. Closure resembles a traditional compiler optimiser by leveraging a deeper understanding to search for smaller representations, unfortunately at the cost of safety. Decidability in highly dynamic languages is often impossible and Closure operates on a well-advised subset of Javascript, delivering no reasonable guarantee of equivalence when code is not written as the Closure authors expected.

[5] Https://github.com/substack/http-browserify.

[6] Https://github.com/jimhigson/oboe.js/blob/master/src/streamingHttp.js I can't claim superior programming ability, this version is shorter because it is not a generic solution.

Integration tests should catch any such failures but for the time being I have a limited appetite for a workflow which forces me to be suspicious of the project's build process.

5.5 Styles of Programming

Built own functional libs and list library (compose, fold etc)

Programming is finished when each line reads as a statement of fact rather than the means of making the statement so.

How doing data hiding in JS without copying an OO concept of data hiding.

Interestingly, the mixed paradigm design hasn't changed the top-level design very much from how it'd be as a pure OO project (IoC, decorators, event filters, pub/sub etc).

The code presented is the result of the development many prior versions, it has never been rewritten in the sense of starting again. Nonetheless, every part has been complely renewed several times. I am reviewing only the final version. Git promotes regular commits, there have been more than 1000.

some of it is pure functional (jsonPath, controller) ie, only semantically different from a Haskell programme others, syntactically functional but stateful to fit in with expected APIs etc

The style of implementation of the generator of functions corresponding to json path expressions is reminiscent of a traditional parser generator, although rather than generating source, functions are dynamically composed. Reflecting on this, parser gens only went to source to break out of the ability to compose the expressive power of the language itself from inside the language itself. With a functional approach, assembly from very small pieces gives a similar level of expressivity as writing the logic out as source code.

Why could implement Function#partial via prototype. Why not going to. Is a shame. However, are using prototype for minimal set of polyfills. Not general purpose.

Javascript: not the greatest for 'final' elegant presentation of programming. Does allow 'messy' first drafts which can be refactored into beautiful code. Ie, can write stateful and refactor in small steps towards being stateless. An awareness of beautiful languages lets us know the right direction to go in. An ugly language lets us find something easy to write that works to get us started. Allows a very sketchy program to be written, little more than a programming scratchpad.

Without strict typing, hard to know if program is correct without running it. In theory (decidability) and in practice (often find errors through running and finding errors thrown). Echo FPR: once compiling, good typing tends to give a reasonable sureness that the code is correct.

5.5.1 Performance implications of functional javascript

V8 and other modern JS engines are often said to be 'near-native' speed, meaning it runs at close to the speed of a similarly coded C program. However, this relies on the programmer also coding in the style of a C programmer, for example with only mono-morphic callsites and without a functional style. Once either of those programming techniques is taken up performance drops rapidly [http://rfrn.org/~shu/2013/03/20/two-reasons-functional-style-is-slow-in-spidermonkey.html] 9571 ms vs 504 ms. When used in a functional style, not 'near-native' in the sense that not close to the performance gained by compiling a well designed functional language to natively executable code. Depends on style coded in, comparison to native somewhat takes C as the description of the operation of an idealised CPU rather than an abstract machine capable of executing on an actual CPU.

(perhaps move to background, or hint at it, eg "although there are still some performance implications involved in a functional style, javascript may be used in a non-pure functional style") - with link to here

The performance degradation, even with a self-hosted forEach, is due to the JIT's inability to efficiently inline both the closures passed to forEach

Lambda Lifting, currently not implemented in SpiderMonkey or V8: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.4346

The transformations to enable the above criteria are tedious and are surely the purview of the compiler. All that's needed are brave compiler hackers

JS is much faster with "monomorphic call sites"

However, js execution time is not much of a problem,

5.5.2 Preferring functions over constructors (subsume into above section?)

What constructors are in js. Any function, but usually an uppercase initial char indicates that it is intended to be used as a constructor.

Inheritence is constructed using the language itself. While this is more flexible and allows each project to define a bespoke version of inherience to suit their particular needs or preferences, it also hampers portability more than an 'extends' keyword would.

> So far, the JavaScript community has not agreed on a common inheritance library (which would help tooling and code portability) and it is doubtful that that will ever happen. That means, we're stuck with constructors under ECMAScript 5. http://www.2ality.com/2013/07/defending-constructors.html

Functions can be like Factories, gives me the flexability to chagne how something is created but by exposing a constructor are stuck with using 'new' to create an instance of exactly one type. 'new' is inconsistent invocation with rest of language.

Dart has 'factory' constructors which are called like constructors but act like factory functions: (http://www.dartlang.org/docs/dart-up-and-running/contents/ch02.html#ch02-constructor-factory)

5.6 Incrementally building up the content

As shown in figure 7, there is an incremental content builder and ascent tracer which handle the output from the Clarinet JSON SAX parser. Taken together, these might be considered as a variant of the Adaptor pattern, providing to the controller a simpler interface than is presented by Clarinet. However, this is not a model implementation of the pattern; it is even-driven rather than call-driven: we receive six kinds of event and in response emmit a smaller vocabulary of two, creating an adaptor over pushes rather that pulls.

There are some peculiarities of Clarinet, these are kept as local as possible. Such as the field name given with the open object and internally normalises this by handling as if it were two events.

5.6.1 What JP matching requires.

To perform matching on JSONPath expressions, the controller requires the path from the root of the document to the current node is required, this is provided in the NODE_FOUND and PATH_FOUND events emitted from the incremental content builder. For each Clarinet event this builder provides a corresponding function which takes the current path and returns the path after the event has been applied. For example, the `objectopen` and `arrayopen` events add new items to the path, whereas `closeobject` and `closearray` remove them. Over the course of the parse of the whole JSON file the path in this way will visit every node, allowing each to be tested against the registered JSONPath expressions. Internally, the builder's event handlers are created from the combination of a smaller number of more basic reusable handlers; Oboe is largely indiscriminate regarding the type of nodes found in the JSON whereas Clarinet's events often differ only in the type of node that was encountered. This reuse of smaller instructions to build up larger ones is slightly reminiscent of the use of composed micro-instructions in CISC CPU design to implement the chip's higher-level instructions. Consider the `value` event which is fired when Clarinet encounters a String or Number, because primitive nodes are always leaves the internal handling is as a node which instantaneously starts and ends, expressed programmatically as the functional composition of the `nodeFound` and `curNodeFinished` handlers.

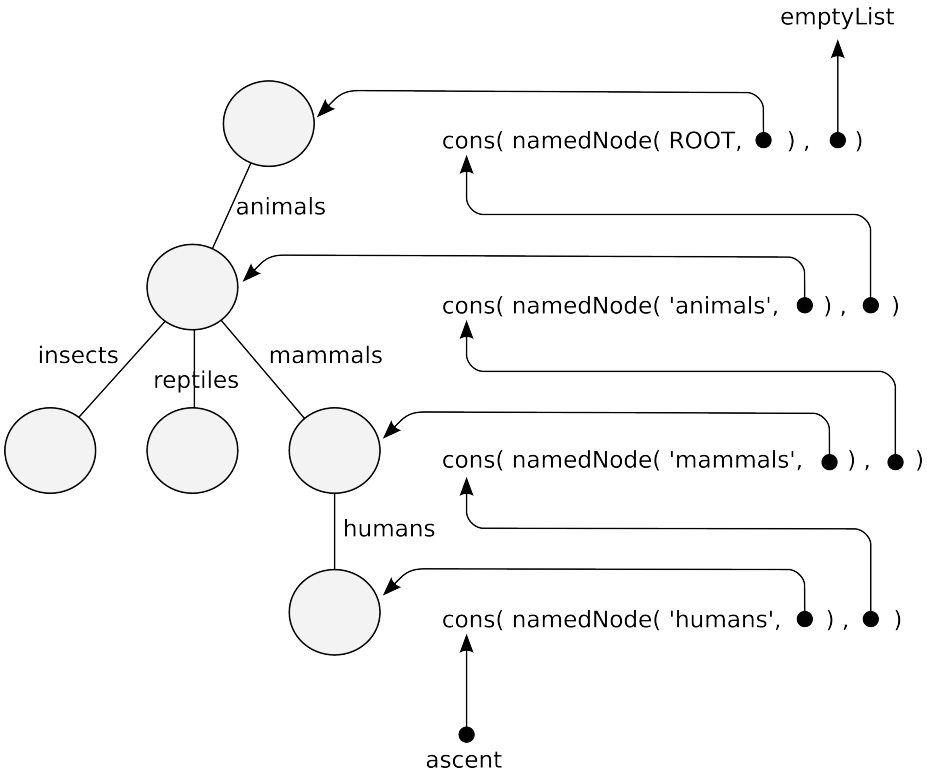

Figure 11: List representation of an ascent from leaf to root of a JSON tree

Although the builder functions are stateless, ultimately the state regarding the current path needs to be stored between clarinet calls. This is handled by the ascent tracker. This tiny component merely serves as a holder for this data, starting from an empty path it passes the path to each builder function and stores the result to be given to the next one.

The representation of the path to the current node is stored as a singly linked list, defined recursively as a series of immutable items. See figure 11. The list is ordered with the JSON root at the far end of the tail and the current node at the head. This order was chosen because as we traverse the JSON the current node is appended and removed many times whereas the root is immutable; all updates to the path are at the head end where it is computationally very cheap.

Each list element holds the field name which referenced the node in the parent object to the node and the node itself.

singly linked list: recursive data structure where each element is a head and tail.

Each list item holds

JS poor at Arrays which grow/shrink. Also not very functional if arrays are modified and highly inefficient to use them in an immutable style and copy on every mutation. Random access to middle not required, only at one end.

Good for testing against JSONPath since

Changed into normal array before handing to outside world.

Although paths are normally represented starting

Why upside down (ascent, not descent). Changes are cheaper at head of list than tail. Head changes as move through doc, tail doesn't.

5.6.2 What Clarinet provides.

Because Clarinet is a SAX parser, the calls that I receive from it are entirely context free; it is my responsibility to build this context.

Luckily, it should be easy to see that building up this context is a simple matter of maintaining a stack describing the descent from the root node to the current node.

5.6.3 Bridging between the two. The split.

Single piece of state: the ascent.

jsonPath parser gets the output from the incrementalParsedContent, minimally routed there by the controller.

On first attempt at ICB, had two stacks, both arrays, plus reference to current node, current key and root node. After refactorings, just one list was enough. Why single-argument functions are helpful (composition etc)

Stateless makes using a debugger easier - can look back in stack trace and because of no reassignment, can see the whole, unchanged state of the parent call. What the params are now are what they always have been, no chance of reassignment (some code style guides recommend not to reassign parameters but imperative languages generally do not forbid it) No Side effects: can type expressions into debugger to see evaluation without risk of changing program execution.

A refactoring was used to separate logic and state:

- Take stateful code
- Refactor until there is just one stateful item
- This means that that item is reassigned rather than mutated
- Make stateless by making all functions take and return an instance of that item
- Replace all assignment of the single stateful var with a return statement
- Create a simple, separate stateful controller that just updates the state to that returned from the calls

Very testable code because stateless - once correct for params under test, will always be correct. Nowhere for bad data to hide in the program.

How do notifications fit into this?

By going to List-style, enforced that functions fail when not able to give an answer. Js default is to return the special 'undefined' value. Why this ensured more robustness but also sometimes took more code to write, ie couldn't just do if(tail(foo)) if foo could be empty but most of the time that would be correct

Explain why Haskel/lisp style lists are used rather than arrays

- In parser clauses, lots of 'do this then go to the next function with the rest'.
- Normal arrays extremely inefficient to make a copy with one item popped off the start
- Link to FastList on github
- For sake of micro-library, implemented tiny list code with very bare needed
- Alternative (first impl) was to pass an index around
- But clause fns don't really care about indexes, they care about top of the list.
- Slight advantage to index: allows going past the start for the root path (which doesn't have any index) instead, have to use a special value to keep node and path list of the same length

- Special token for root, takes advantage of object identity to make certain that cannot clash with something from the json. Better than '**root**' or similar which could clash. String in js not considered distinct, any two strings with identical character sequences are indistinguishable.

Anti-list: nothing is quite so small when making a mircro-library as using the types built into the language, coming as they are for zero bytes.

- For recognisably with existing code, use lists internally but transform into array on the boundary between Oboe.js and the outside world (at same time, strip off special 'root path' token)

5.7 Oboe JSONPath Implementation

Not surprisingly given its importance, the JSONPath implementation is one of the most refactored and considered parts of the Oboe codebase. Like many small languages, on the first commit it was little more than a series of regular expressions[7] but has slowly evolved into a featureful and efficient implementation[8]. The extent of the rewriting was possible because the correct behaviour is well defined by test specifications[9].

The JSONPath compiler exposes a single higher-order function to the rest of Oboe. This function takes a JSONPath as a String and, proving it is a valid expression, returns a function which tests for matches to the JSONPath. Both the compiler and the functions that it generates benefit from being stateless. The type of the compiler, expressed as Haskell syntax would be:

```
String -> Ascent -> JsonPathMatchResult
```

The match result is either a failure to match, or a hit, with the node that matched. In the case of path matching, the node may currently be unknown. If the pattern has a clause prefixed with `$`, the node matching that clause is captured and returned as the result. Otherwise, the last clause is implicitly capturing.

The usage profile for JSONPath expressions in Oboe is to be compiled once and then evaluated many times, once for each node encountered while parsing the JSON. Because matching is performed perhaps hundreds of times per file the

[7] JSONPath compiler from the first commit can be found at line 159 here: https://github.com/jimhigson/oboe.js/blob/a17db7accc3a371853a2a0fd755153b10994c91e/src/main/progressive.js#L159.

[8] For contrast, the current source can be found at https://github.com/jimhigson/oboe.js/blob/master/src/jsonPath.js.

[9] The current tests are viewable at https://github.com/jimhigson/oboe.js/blob/master/test/specs/jsonPath.unit.spec.js and https://github.com/jimhigson/oboe.js/blob/master/test/specs/jsonPathTokens.unit.spec.js.

most pressing performance consideration is for matching to execute quickly, the time required to compile is relatively unimportant. Oboe's JSONPath design contrasts with JSONPath's reference implementation which, because it provides a first order function, freshly reinterprets the JSONPath string each time it is invoked.

The compilation is performed by recursively by examining the left-most side of the string for a JSONPath clause. For each kind of clause there is a function which matches ascents against that clause, for example by checking the name field. By partial completion this function is specialised to match against one particular name. Once a clause function is generated, compilation recurs by passing to itself the remaining unparsed portion of the JSONPath string. This continues until it is called with a zero-length JSONPath. On each recursive call the clause function is wrapped in the result from the next recursive call, resulting ultimately in a linked series of clause functions. When evaluated against an ascent, each clause functions examines the head of the ascent and passes the ascent onto the next function if it passes. A special clause functions, `skip1` is used for the `.` syntax and places no condition on the head of the ascent but passes on to the next clause only the tail, thus moving evaluation of the ascent one node up the parsed JSON tree. Similarly, there is a `skipMany` which maps onto the `..` syntax and recursively consumes nodes until it can find a match in the next clause.

JsonPath implementation allows the compilation of complex expressions into an executable form, but each part implementing the executable form is locally simple. By using recursion, assembling the simple functions into a more function expressing a more complex rule also follows as being locally simple but gaining a usefully sophisticated behaviour through composition of simple parts. Each recursive call of the parser identifies one token for non-empty input and then recursively digests the rest.

As an example, the pattern `!.$person..{height tShirtSize}` once compiled to a Javascript functional representation would roughly resemble this:

```
statementExpr(             // wrapper, added when JSONPath is zero-length
   duckTypeClause(         // token 6, {height tShirtSize}
      skipMany(            // token 5, '..'
         capture(          // token 4, css4-style '$' notation
            nameClause(    // token 3, 'person'
               skip1(      // token 2, '.'
                  rootExpr // token 1, '!' at start of JSONPath expression
               )
            'person' )
         )
   ), ['height', 'tShirtSize'])
)
```

Since I am only using a side-effect free subset of Javascript for this segment of Oboe it would be safe to use a functional cache. As well as saving time by avoiding repeated execution, this could potentially also save memory because where two JSONPath strings contain a common start they could share the inner parts of their functional expression. Although Javascript doesn't come with functional caching, it can be added using the language itself.[10] I suspect, however, that hashing the parameters might be slower than performing the matching. Although the parameters are all immutable and could in theory be hashed by object identity, in practice there is no way to access an object id from inside the language so any hash of a node parsed out of JSON would have to walk the entire subtree rooted from that node.

The JSONPath tokenisation is split out into its own file and separately tested. The tokenisation implementation is based on regular expressions, they are the simplest form able to express the clause patterns. The regular expressions are hidden to the outside the tokenizer and only functions are exposed to the main body of the compiler. The regular expressions all start with ^ so that they only match at the head of the string. A more elegant alternative is the 'y'[11] flag but as of now this lacks wide browser support.

By verifying the tokens through their own unit tests it is simpler to thoroughly specify the tokenisation, producing simpler failure messages than if it were done through the full JSONPath engine. We might consider the unit test layer of the pyramid (figure 8) is further split into two sub-layers. Arguably, the upper of these sub-layer is not a unit test because it is verifying two units together. There is some redundancy with the tokens being tested both individually and as full expressions. I maintain that this is the best approach regardless because stubbing out the tokenizer functions would be a considerable effort and would not improve the rigor of the JSONPath specification.

5.8 Callback and mutability Problem

Stateful controller very easy to test - only 1 function.

Javascript provides no way to decalre an object with 'cohorts' who are allowed to change it whereas others cannot - vars may be hidden via use of scope and closures (CITE: crockford) but attributes are either mutable or immutable.

Why this is a problem.

- bugs likely to be attributied to oboe because they'll be in a future *frame of execution*. But user error.

Potential solutions:

[10] Probably the best known example being `memoize` from Underscore.js: http://underscorejs.org/#memoize.

[11] Https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions.

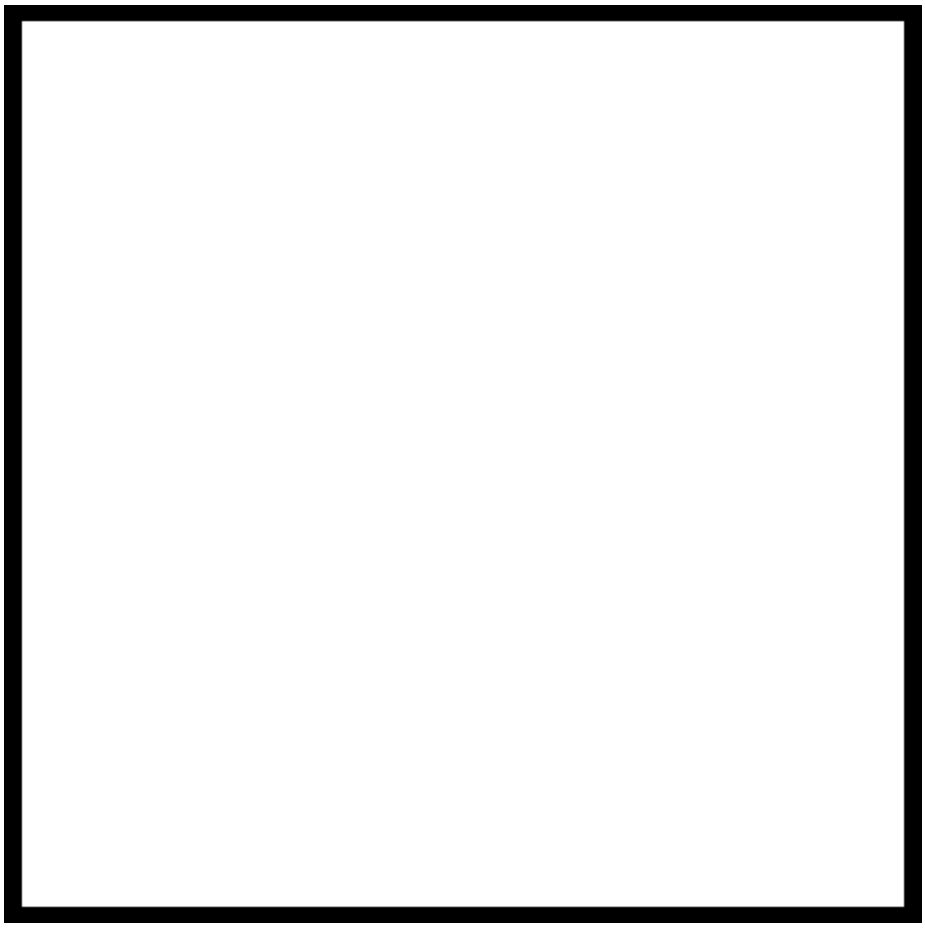

Figure 12: Some kind of diagram showing jsonPath expressions and functions partially completed to link back to the previous function. Include the statementExpr pointing to the last clause

- full functional-style immutability. Don't change the objects, just have a function that returns a new one with one extra property. Problem - language not optimised for this. A lot of copying. Still doesn't stop callback receiver from changing the state of hte object given. (CITE: optimisations other languages use)
- immutable wrappers.
- defensive cloning
- defining getter properties

6 Conclusion

Doing things faster vs doing things earlier. “Hurry up and wait” approach to optimisation.

6.1 weaknesses

implementation keeps ‘unreachable’ listeners difficult decidability/proof type problem to get completely right but could cover most of the easy cases

Parse time for large files spread out over a long time. Reaction to parsed content spread out over a long time, for example de-marshalling to domain objects. For UX may be preferable to have many small delays rather than one large one.

Doesn’t support all of jsonpath. Not a strict subset of the language.

Rest client as a library is passing mutable objects to the caller. too inefficient to re-create a new map/array every time an item is not as efficient in immutability as list head-tail type storage

An immutability wrapper might be possible with defineProperty. Can’t casually overwrite via assignment but still possible to do defineProperty again.

Would benefit from a stateless language where everything is stateless at all times to avoid having to program defensively.

Aborting http request may not stop processing on the server. Why this is perhaps desirable - transactions, leaving resources in a half-complete state.

6.2 Suitability for databases (really just an inline asside)

Databases offer data one row at a time, not as a big lump.

6.3 Development methodology

Did it help?

Switched several times. Could have started with winning side? Tension between choosing latest and greatest (promising much) or old established solution alraedy experienced with but with known problems. Judging if problems will become too much of a hinderence and underestimating the flaws. JSTD was yesterday’s latest and greatest but Karma genuinely is great. In end, right solution was found despite not being found in most direct way.

Packaging was a lot of work but has delivered the most concise possible library.

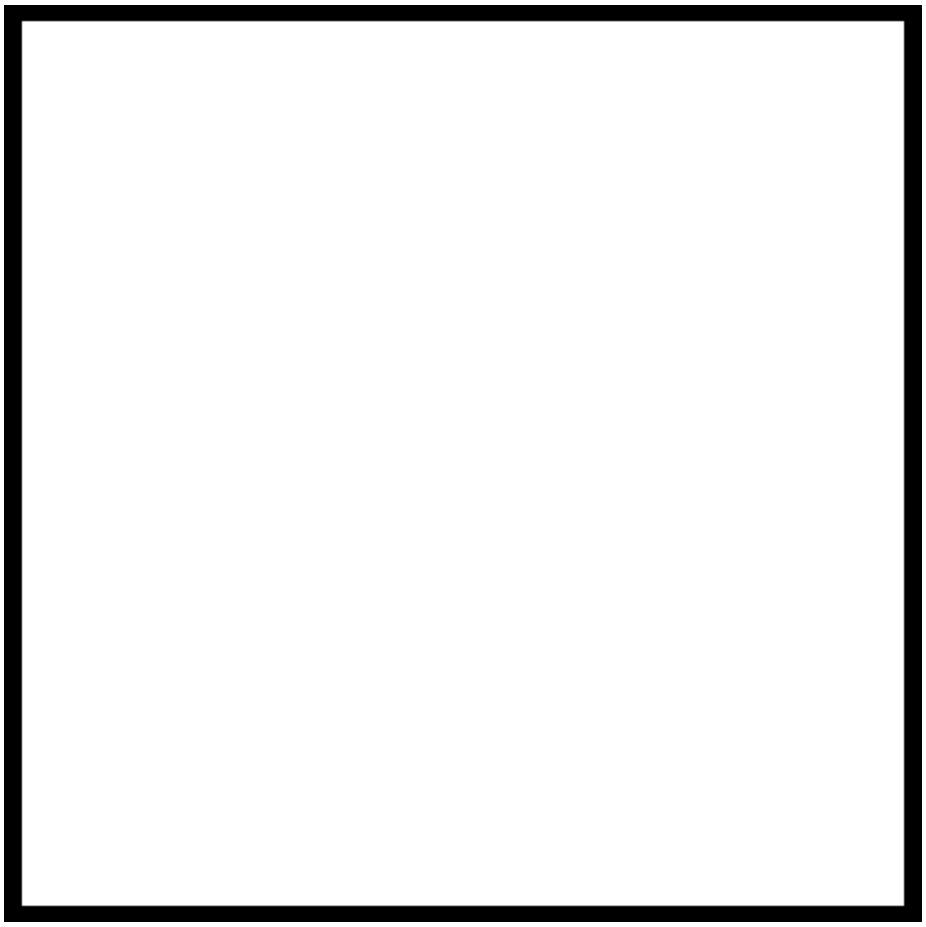

Figure 13: A pie chart showing the sizes of the various parts of the codebase

6.4 Size

Comment on the size of the libraray

6.5 Handling invalid input

Invalid jsonpaths made from otherwise valid clauses (for example two roots) perhaps could fail early, at compile time. Instead, get a jsonPath that couldn't match anything. Invalid syntax is picked up.

Same pattern could be extended to XML. Or any tree-based format. Text is easier but no reason why not binary applications.

Not particularly useful reading from local files.

Does not save memory over DOM parsing since the same DOM tree is built. May slightly increase memory usage by utilising memory earlier that would otherwise be dept dormant until the whole transmission is received but worst case more often a concern than mean.

Implementation in a purely functional language with lazy evaluation: could it mean that only the necessary parts are computed? Could I have implemented the same in javascript?

Would be nice to: * discard patterns that can't match any further parts of the tree * discard branches of the tree that can't match any patterns * just over the parsing of branches of the tree that provably can't match any of the patterns

6.6 Comparative usages

Interesting article from Clarinet: http://writings.nunojob.com/2011/12/clarinet-sax-based-evented-streaming-json-parser-in-javascript-for-the-browser-and-nodejs.html

In terms of syntax: compare to SAX (clarinet) for getting the same job done. Draw examples from github project README. Or from reimplementing Clarinet's examples.

Consider:

- Difficulty to program
- Ease of reading the program / clarity of code
- Resources consumed
- Performance (time) taken
- about the same. Can react equally quickly to io in progress, both largely io bound.
- Is earlier really faster?

6.7 Community reaction

Built into Dojo Followers on Github Being posted in forums (hopefully also listed on blogs) No homepage as of yet other than the Github page

6.8 Possible future work

Do da XMLs

6.9 Benchmarking

Complex JSONPAth tested against JSON with approx 2,000 nodes, finding 100 matches. Real http, full stack Oboe.

Browser	Time
Firefox 24.0.0 (Mac OS X 10.7)	547ms
Chrome 30.0.1599 (Mac OS X 10.7.5)	237ms
Chrome Mobile iOS 30.0.1599 (iOS 7.0.2)	431ms
Safari 6.0.5 (Mac OS X 10.7.5)	231ms
IE 8.0.0 (Windows XP)	3048ms

7 Appendix

8 Bibliography

Ahuvia, Yogev. 2013. "Design Patterns: Infinite Scrolling: Let's Get To The Bottom Of This http://uxdesign.smashingmagazine.com/2013/05/03/infinite-scrolling-get-bottom/." Smashing Magazine.

Anon. 2011. "3G mobile data network crowd-sourcing survey." BBC News.

Conway, Mel. 2004. *Humanizing Application Building: An Anthropological Perspective*. http://melconway.com/Home/pdf/humanize.pdf.

Douglas, Crockford. 2009. "JSON: The fat-free alternative to XML." http://json.org.

Etemad, Elika J, and Tab Atkins. 2013. "Selectors Level 4." http://dev.w3.org/csswg/selectors4/.

Fielding, R. T. 2000. "Principled design of the modern Web architecture."

Geelhoed, Erik, Peter Toft, Suzanne Roberts, and Patrick Hyland. 1995. "To influence Time Perception." Hewlett Packard Labs. http://www.sigchi.org/chi95/proceedings/shortppr/egd_bdy.htm.

Graham, Paul. 2004. *The Other Road Ahead*. O'Reilly and Associates.

Hopkins, Don. 1994. *The X-Windows Disaster*. Hungry Minds.

Lea, Tom. 2012. "Improving performance on twitter.com." {[}https://blog.twitter.com/2012/improving-performance-twittercom{]}.

Martin, Robert "Uncle Bob." 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*.

McLuhan, Marshall. 1964. *Understanding Media: The Extensions of Man*.

Mullany, Michael. 2013. "5 Myths About Mobile Web Performance." http://www.sencha.com/blog/5-myths-about-mobile-web-performance.

Ralston, Anthony. 2000. "Encyclopedia of Computer Science." Nature Pub. Group.

Reis, Eric. 2011. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business Publishing.

Sapir, E. 1958. "Culture, Language and Personality (ed. D. G. Mandelbaum)." Berkeley, CA: University of California Press.

Stefanov, Stoyan. 2009. "Progressive rendering via multiple flushes." http://www.phpied.com/progressive-rendering-via-multiple-flushes/.

Whorf, B. L. 1956. "Language, Thought and Reality (ed. J. B. Carroll)." Cambridge, MA: MIT Press.

van Kesteren, Anne. 2012. “XMLHttpRequest Level 2 Working Draft.” http://www.w3.org/TR/XMLHttpRequest2/#make-progress-notifications.

van Kesteren, Anne, and Dean Jackson. 2006. “The XMLHttpRequest Object.” http://www.w3.org/TR/2006/WD-XMLHttpRequest-20060405/.