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

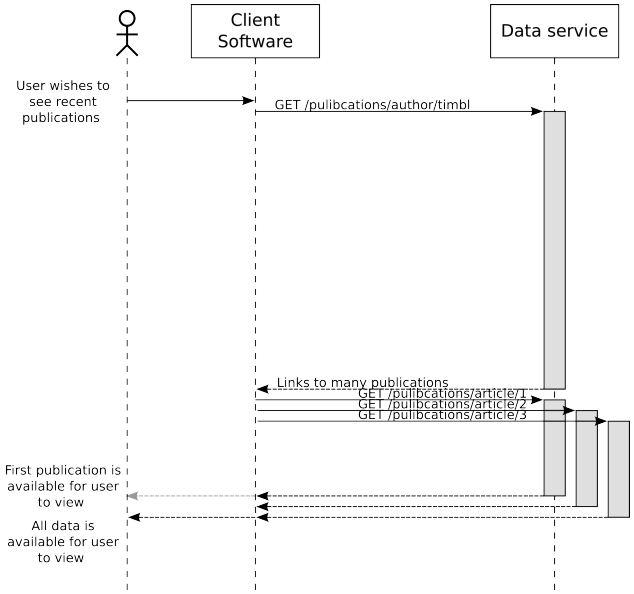

Figure 1: **Aggregation of lower-level resources exposed via REST.** The client fetches a listing of an author's publications and then the first three articles. The sequence represents the most commonly used technique in which the client does not react to the response until it is complete. In this example 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] See http://jackson.codehaus.org/1.0.1/javadoc/org/codehaus/jackson/node/NullNode.html.

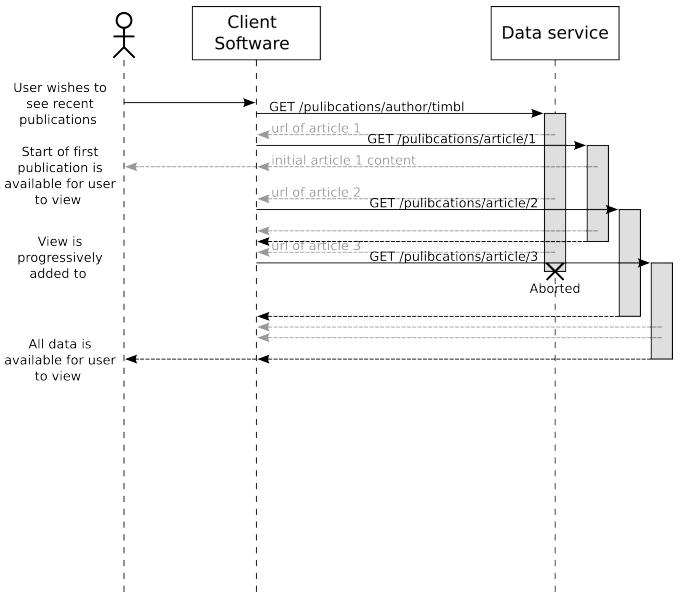

Figure 2: **Revised sequence of aggregation performed by a client capable of progressively interpreting the fetched resource.** Because UML sequence diagrams arrows draw the concept of a returned value as a one-off event rather than a continuous process, I have introduced the notation of lighter arrows 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

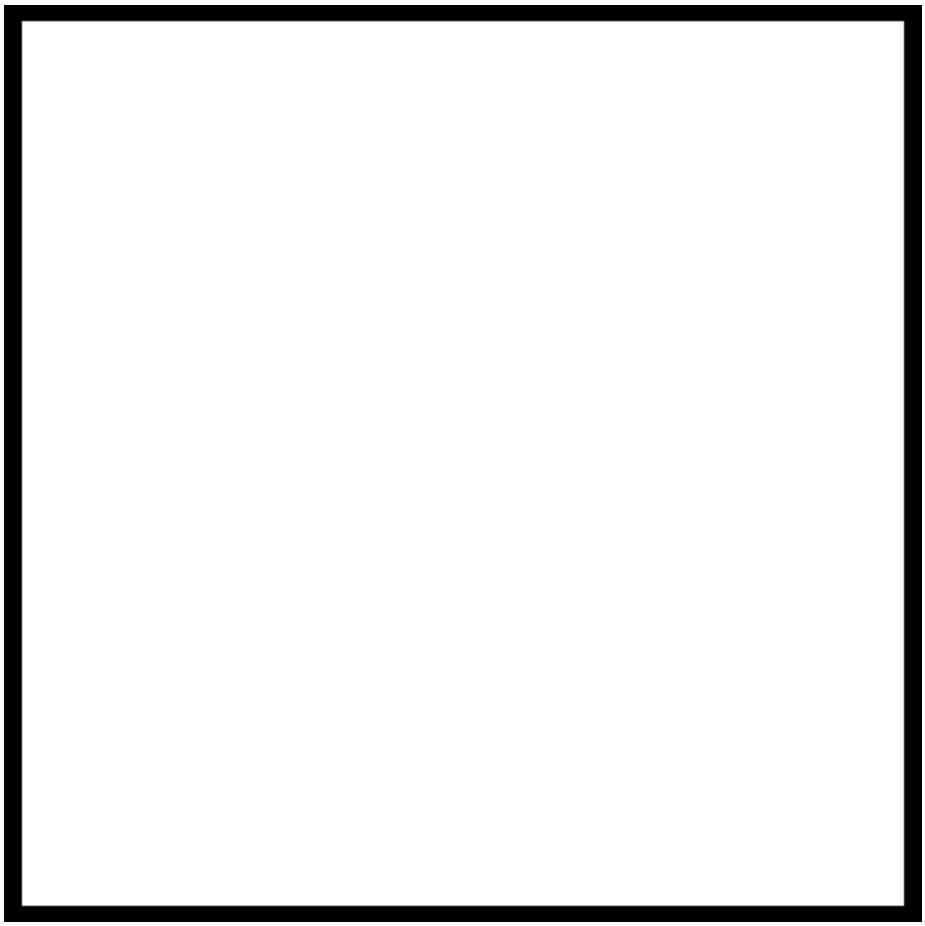

Figure 3: *A webapp running with a front end generated partially on server and partially on client side.* Ie, front-end client-side, front-end server-side, presentation layer a more meaningful distinction than

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

[2] Rather confusingly, X11 would call the *server* the *client* but I use terms here by their more cannonical meaning such that the client is the machine the user is actually interacting with.

> doesn't matter how big an abstraction layer it has to go through to get to silicon. (Mullany 2013)

3.2 Node.js

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.

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 ) {
```

[3]See http://jackson.codehaus.org/1.0.1/javadoc/org/codehaus/jackson/node/NullNode.html.

```
    // 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.

    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.

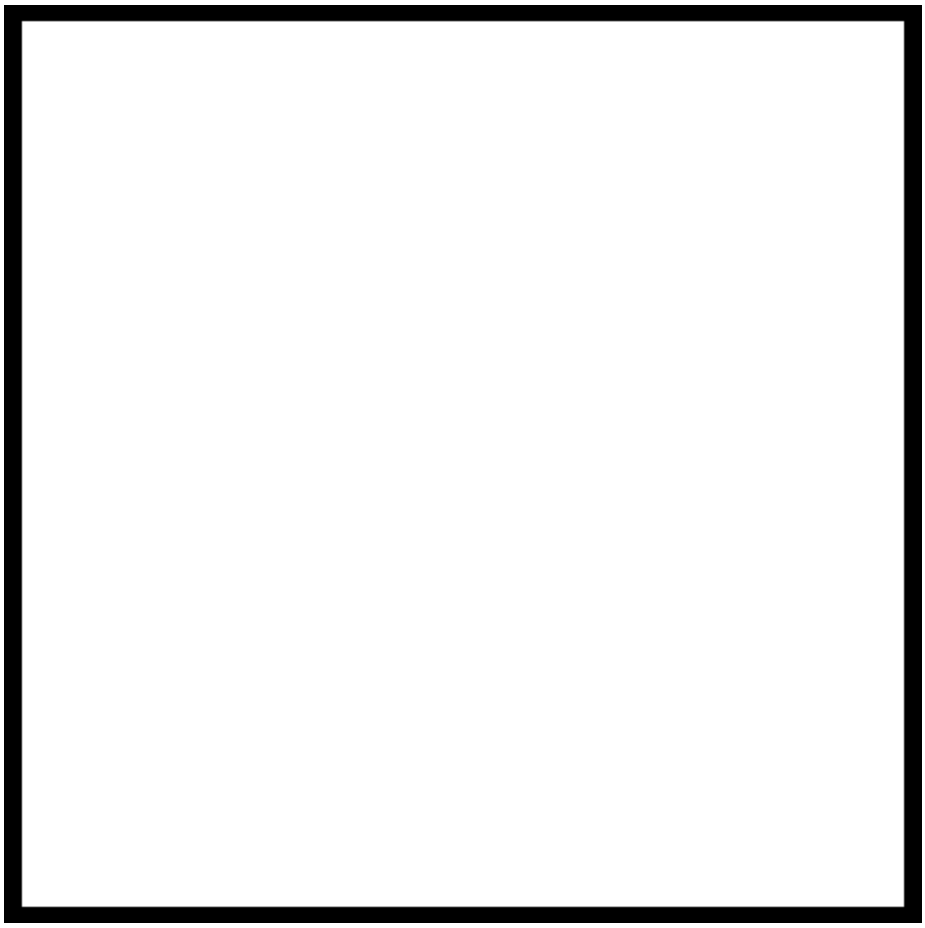

Figure 4: *Degrees of automatic marshaling.* From marshaling directly to domain objects, DTOs, using parser output as a DTO, or using objects directly. Distinguish work done by library vs application programmer's domain

```
// 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.

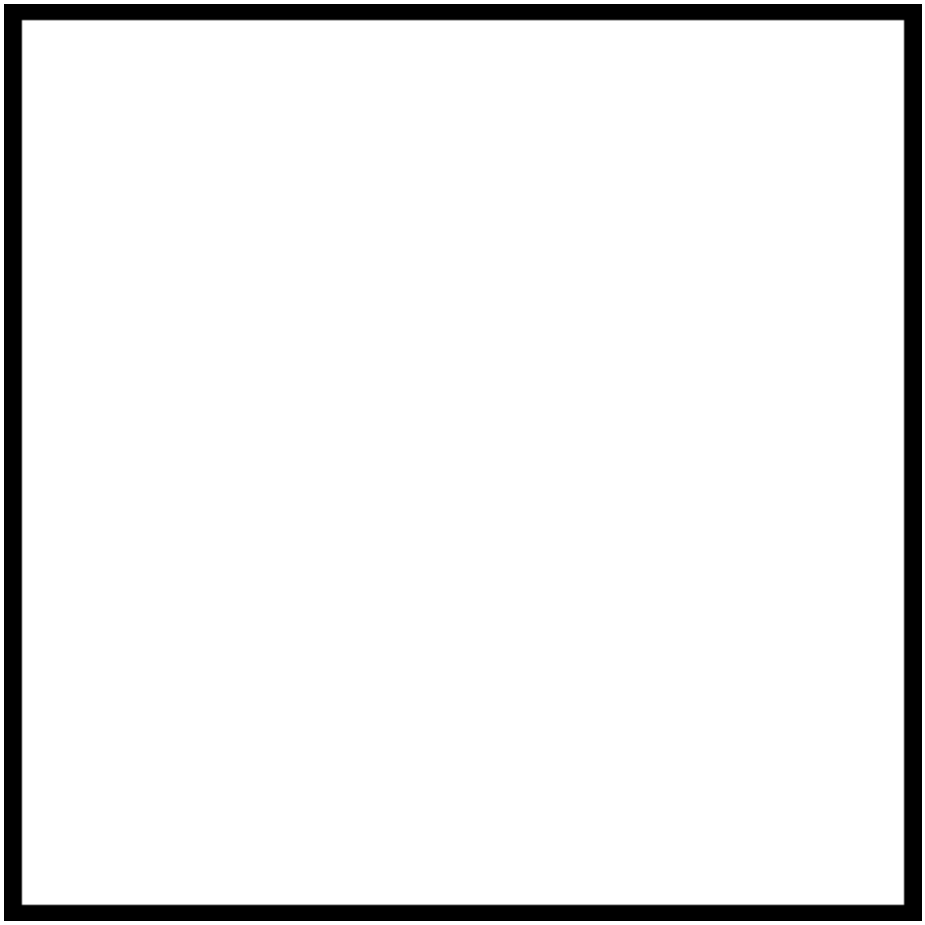

Figure 5: Relationship between the main players in the JS testing landscape. JSTD, Karma, Jasmine, NodeUnit, jasmine-node, Browsers

4 Application and Reflection

What a Micro-library is. What motivates the trend? This library has a fairly small set of functionality, it isn't a general purpose do-everything library like jQuery so its size will be looked at more critically if it is too large. Micro library is the current gold standard for compactness. Still, have a lot to do in not very much code.

Relationship between type of a node and its purpose in the document. 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.

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).

Why SAX is dumb: As a principle, the programmer should only have to handle the cases which are interesting to them, not wade manually through a haystack in search of a needle, which means the library should provide an expressive way of associating the nodes of interest with their targetted callbacks.

4.1 High-level design

A feature set which is minimal but contain no obvious omissions.

Under the heading [Anatomy of a SOA client] I deconstructed the way in which programming logic is often used to identify the parts of a model which are currently interesting and started to look at some declarative ways in which these parts can be obtained.

Turn this model inside out. Instead of the programmer finding the parts they want as a part of the general logic of the program, declaratively define the interesting parts and have these parts delivered to the language logic. Once we make the shift to thinking in this way, it is no longer necessary to have the whole resource locally before the interesting sub-parts are delivered.

Focus on replacing ajax, rather than streaming. In older browsers, getting the whole message at once is no worse than it is now.

4.1.1 stability over upgrades

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

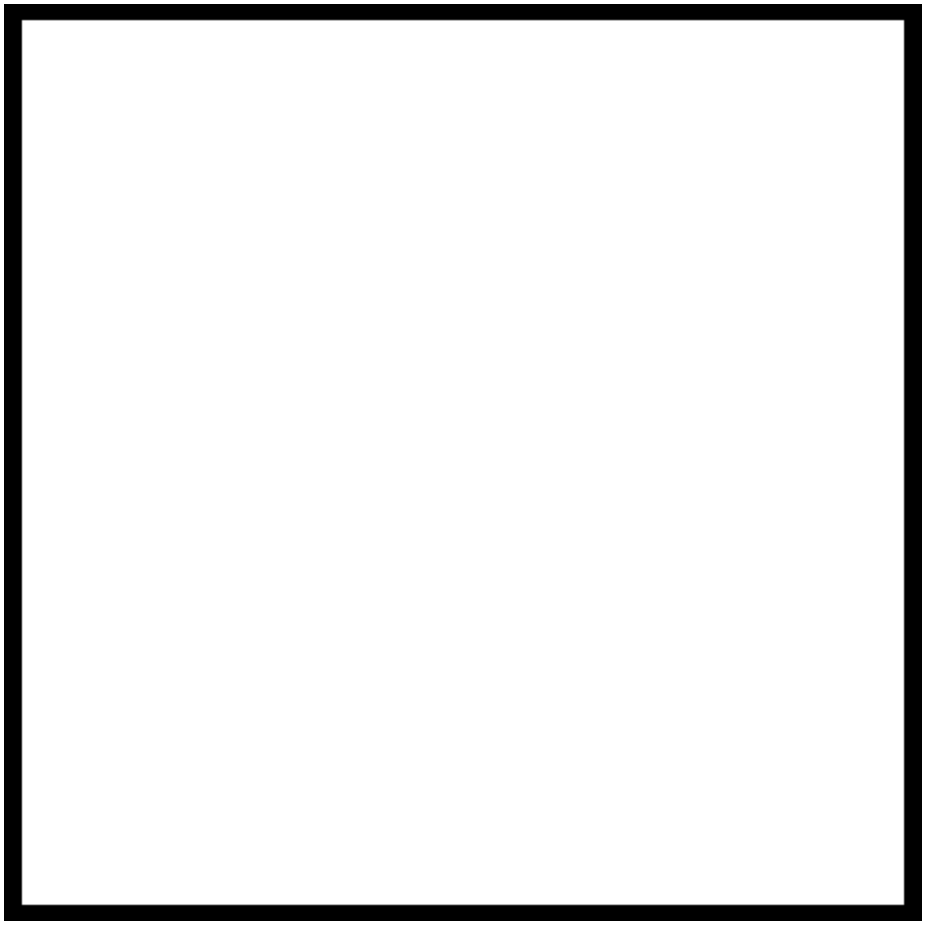

Figure 6: Over several hops of aggregation, the benefits of finding the interesting parts early

4.1.2 suitability for databases

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

4.2 resume on failure

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 can be used to request any contiguous part of a response rather than the whole. Common in download managers but not REST clients. This ability can be used to. Why not this one. Resume on a higher semantic level.

4.3 overall design philosophy and breaking out of big/small tradeoff

In which a callback call is received not just when the whole resource is downloaded but for every interesting part which is seen while the transfer is ongoing. 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

Best of both modes

Granularity: only need read as far as necessary. Services could be designed to write the big picture first. Alternatively, where resources link to one another, can stop reading at the link. Eg, looking for a person's publications, start with an index of people but no need to read whole list of people.

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

4.4 high-level choice of technologies

can justify why js as:

Most widely deployable.

Node: asynchronous model built into language already, no 'concurrent' library needed. Closures convenient for picking up again where left off.

Node programs often so asynchronous and callback based they become unclear in structure. Promises approach to avoid pyramid-shaped code and callback spaghetti.

```
// example of pyramid code
```

In comparison to typical Tomcat-style threading model. Threaded model is powerful for genuine parallel computation but Wasteful of resources where the tasks are more io-bound than cpu-bound. Resources consumed by threads while doing nothing but waiting.

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

functional, pure functional possible [FPR] but not as nicely as in a pure functional language, ie function caches although can be implemented, not universal on all functions.

easy to distribute softare (npm etc)

4.5 principles of a loosely coupled reader

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

Requires a small amount of 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. See 7

(CITE: re-read citations from SOA)

4.6 Design of the jsonpath parser

NB: This consideration of type in json could be in the Background section.

Xml comes with a strong concept of the *type* of an element, the tag name is taken as a more immediate fundamental property of the thing than the attributes. For example, in automatic json-Java object demarshallers, the tag name is always mapped to the Java class. In JSON, other than the base types common to most languages (array, object, string etc) there is no further concept of type. If we wish to build a further understanding of the type of the objects then the realtionship with the parent object, expressed by the attribute name, is more likely to indicate the type. A second approach is to use duck typing in which the relationship of the object to its ancestors is not examined but the properties of the object are used instead to communicate an enhanced concept of type. For example, we might say that any object with an isbn and a title is a book.

Duck typing is of course a much looser concept than an XML document's tag names and collisions are possible where objects co-incidentally share property

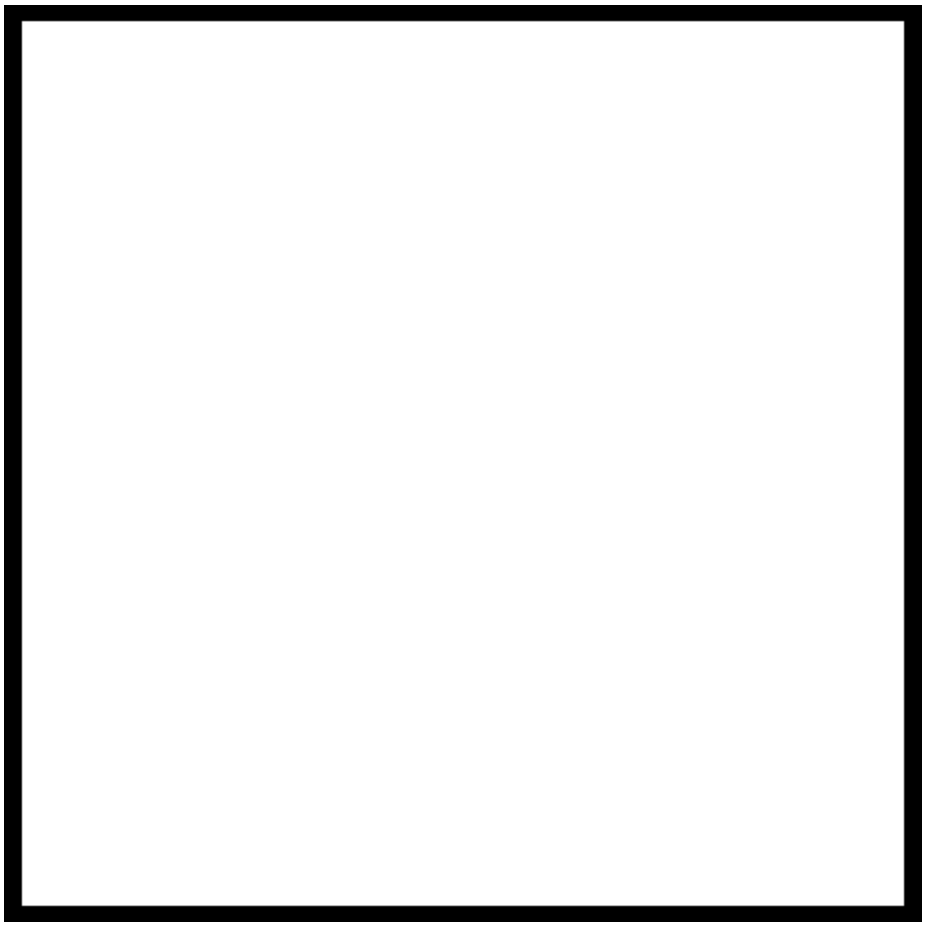

Figure 7: extended json rest service that still works - maybe do a table instead

names. In practice however, I find the looseness a strength more often than a weakness. Under a tag-based marshalling from an OO language, sub-types are assigned a new tag name and as a consumer of the document, the 'isa' relationship between a 'class' tagname and it's 'sub-tabname' may be difficult to track. It is likely that if I'm unaware of this, I'm not interested in the extended capabilities of the subclass and would rather just continue to recieve the base superclass capabilities as before. Under duck typing this is easy - becase the data consumer lists the

A third injection of type into json comes in the form of taking the first property of an object as being the tagname. Unsatisfactory, objects have an order while serialised as json but once deserialised typically have no further order. Clarinet.js seems to follow this pattern, notifying of new objects only once the first property's key is known so that it may be used to infer type. Can't be used with a general-purpose JSON writer tool, nor any JSON writer tool that reads from common objects.

Design not just for now, design to be stable over future iterations of the software. Agile etc.

Why an existing jsonPath implmentation couldn't be used: need to add new features and need to be able to check against a path expressed as a stack of nodes.

More important to efficiently detect or efficiently compile the patterns?

As discussed in section ???, Sax is difficult to program and not widely used.

First way to identify an interesting thing is by its location in the document. In the absense of node typing beyond the categorisation as objects, arrays and various primative 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 stong 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 8 below.

By sensible convention, even in a serialisation format with only a loose definition of lists, lists contain only items of the same type. This gives way to a sister convention, that for lists of items, the key immediately linking to the

Essentially two ways to identify an interesting node - by location (covered by existing jsonpath)

Why duck typing is desirable in absense of genuine types in the json standard (ala tag names in XML). or by a loose concept of type which is not well supported by existing jsonpath spec.

Compare duck typing to the tag name in

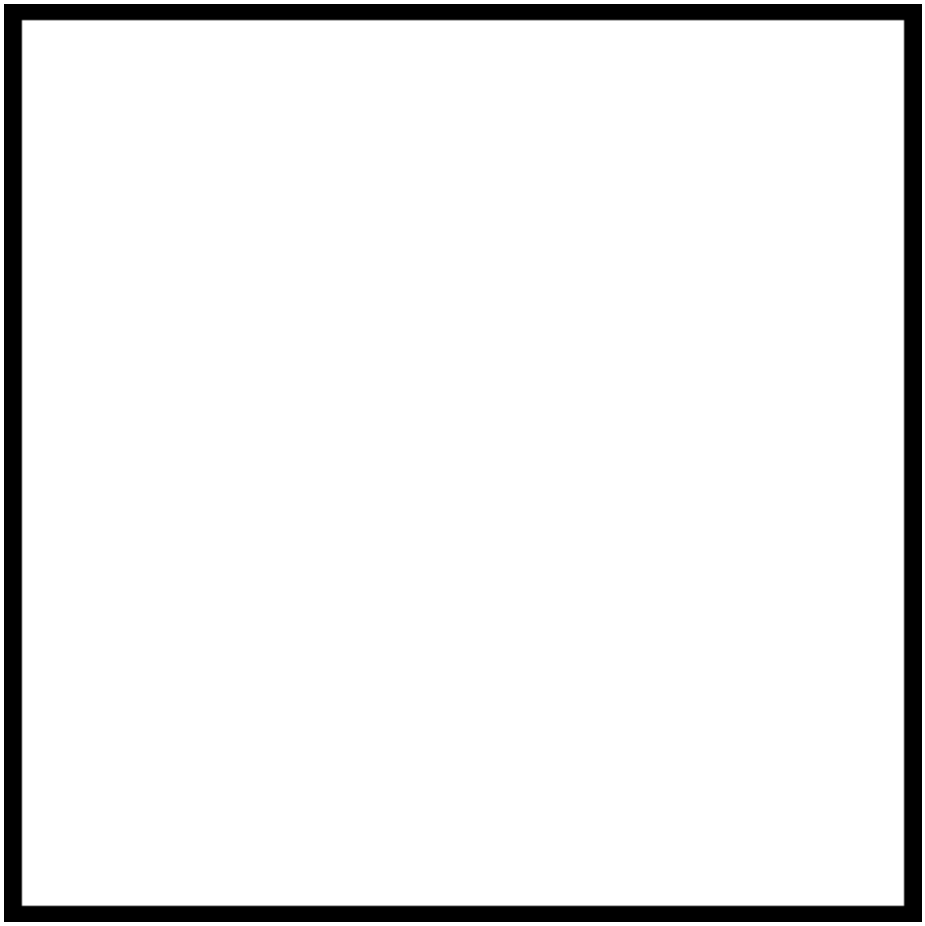

Figure 8: UML class diagram showing a person class in relationship with an address class. In implementation as Java the 'hasAddress' relationship would typically be reified as a getAddress method. This co-incidence of object type and the name of the field referring to the type lends itself well to the tendency for the immediate key before an object to be taken as the type when Java models are marshaled into json

To extend JsonPath to support a concise expression of duck typing, I chose a syntax which is similar to fields in jsonFormat:

```
// the curly braces are my extension to jsonpath"
let jsonPath = jsonPathCompiler("{name, address, email}");

// the above jsonPath expression would match this object in json expression and
// like all json path expressions the pattern is quite similar to the object that
// it matches. The object below matches because it contains all the fields listed
// in between the curly braces in the above json path expresson.

let matchingObject = {
   "name": "...",
   "address": "...",
   "email": "...:
}

jsonPath(matchingObject); // evaluates to true
```

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)

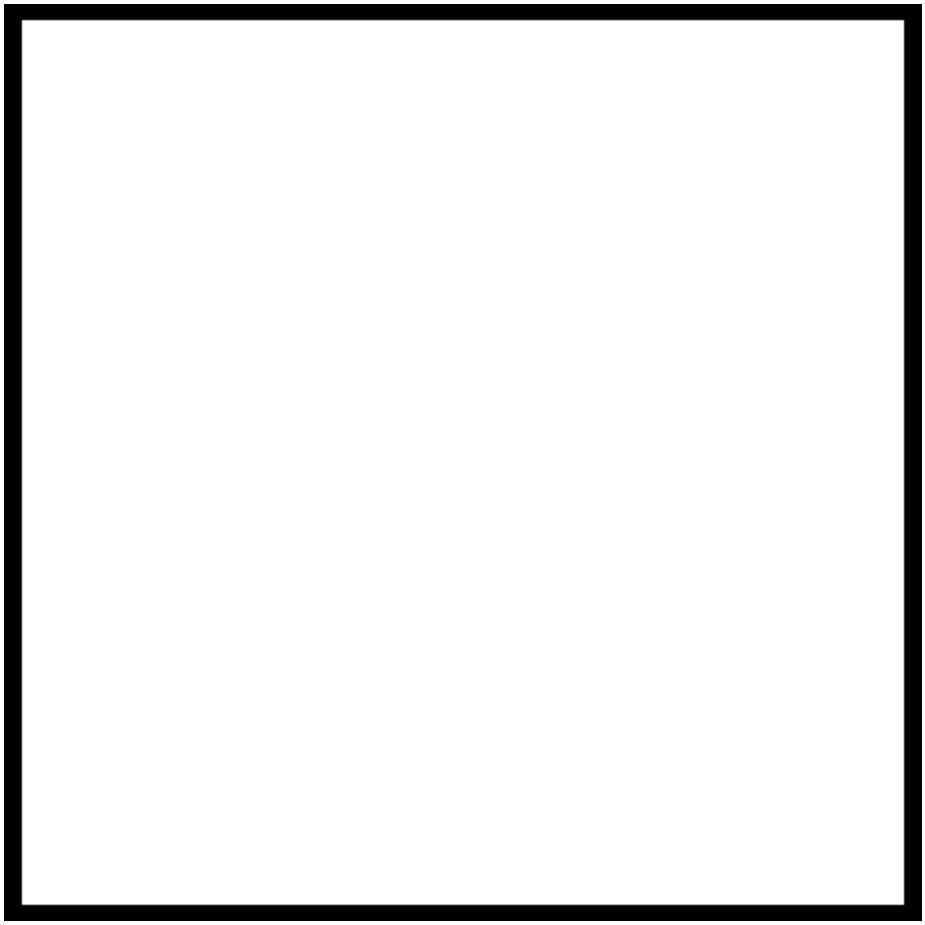

Figure 9: Diagram showing why list is more memory efficient - multiple handles into same structure with different starts, contrast with same as an array

In parser, can't use 'y' flag to the regualr expression engine which would allow much more elegant matching. Only alternative is cumersome: to slice the string and match all tokens with regexes starting with '^' in order to track the current location. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\T1\textbackslash{}_Expressions]

4.7 Incrementally building up the content

Like SAX, calls from clarinet are entirely 'context free'. Ie, am told that there is a new object but without the preceding calls the root object is indistinguishable from a deeply nested object. 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.

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

Explain match starting from end of candidate path

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?

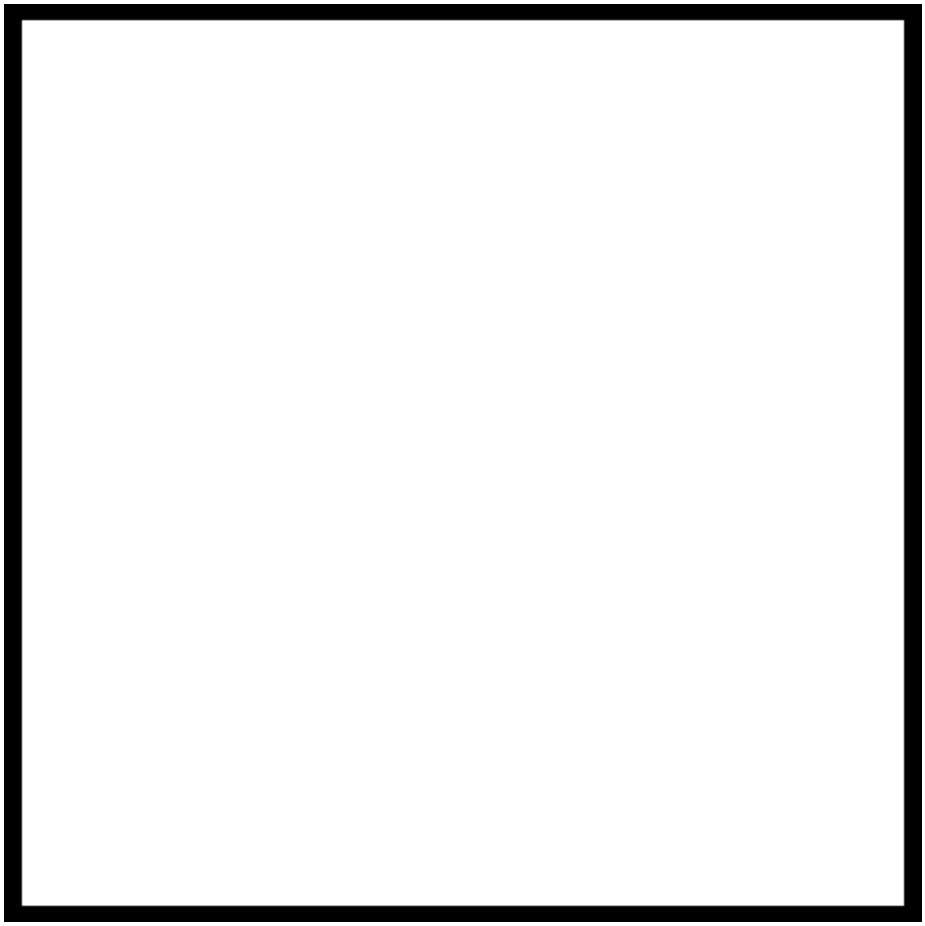

Figure 10: Show a call into a compiled jsonPath to explain coming from incrementalParsedContent with two lists, ie the paths and the objects and how they relate to each other. Can use links to show that object list contains objects that contain others on the list. Aubergine etc example might be a good one

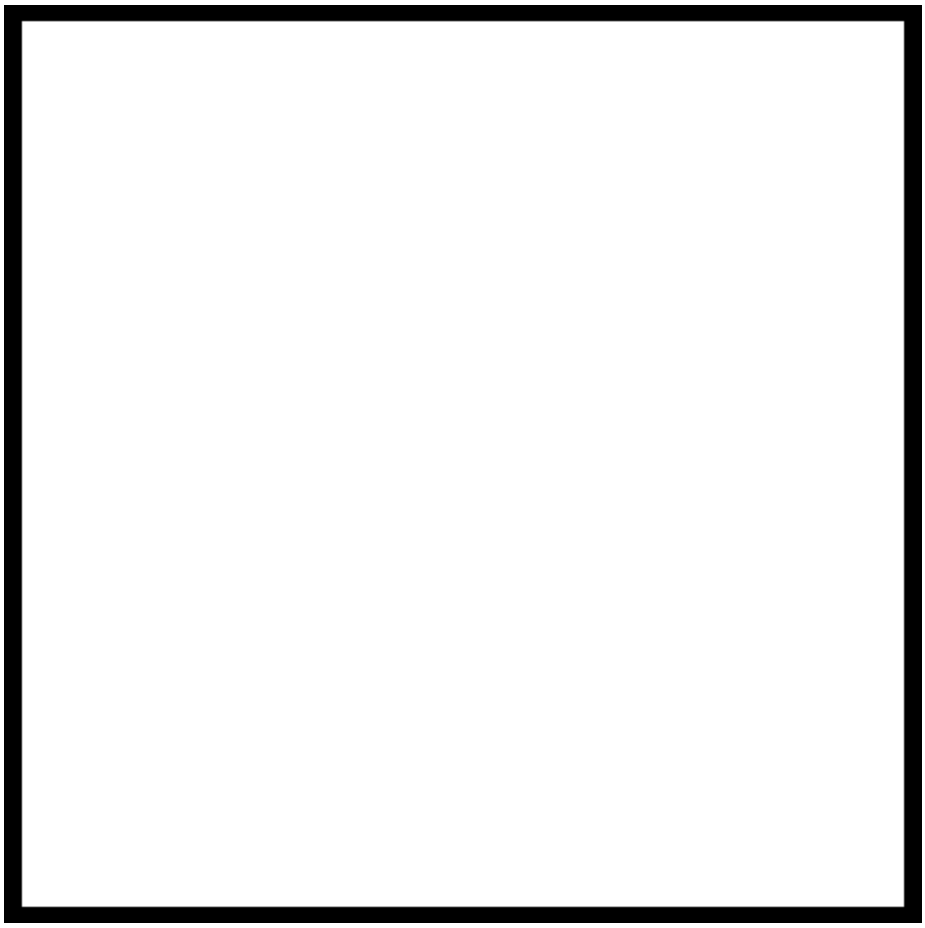

Figure 11: Some kind of diagram showing jsonPath expressions and functions partially completed to link back to the previous function. Include the statementExpr pointing to the last clause

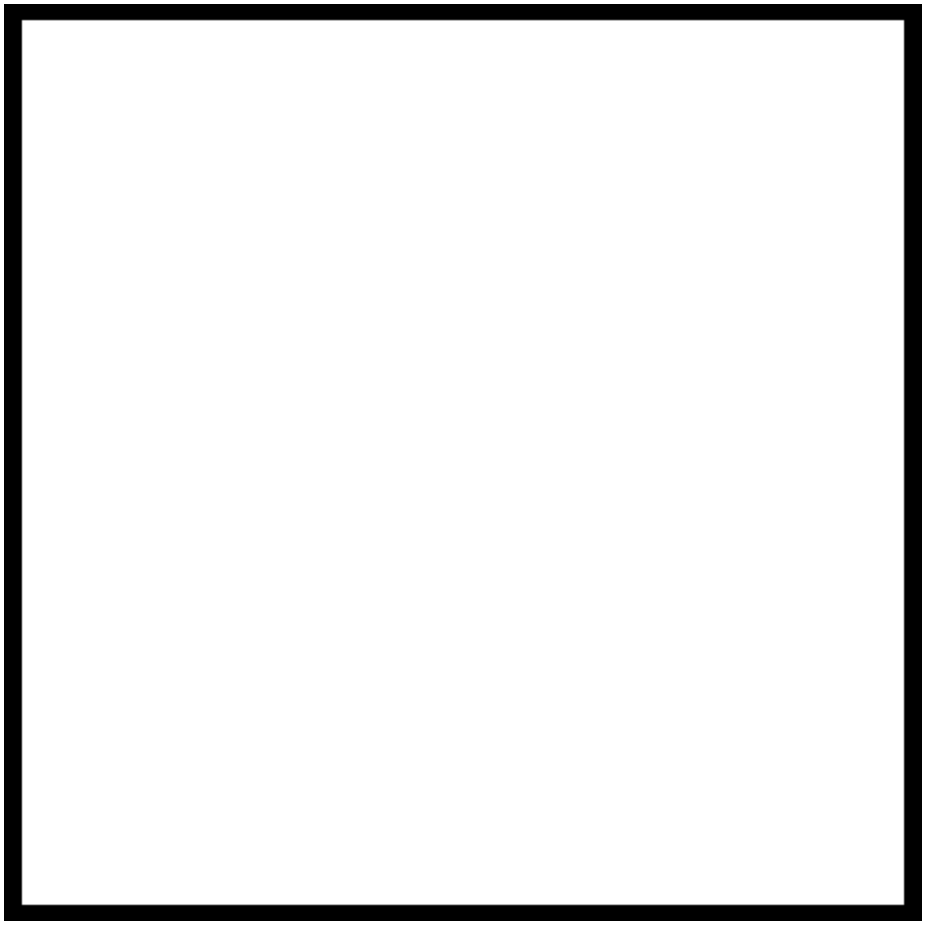

Figure 12: Overall design of Oboe.js. Nodes in the diagram represent division of control so far that it has been split into different files.

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

4.7.1 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:

- 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

4.8 styles of programming

The code presented is the result of the development many prior versions, it has never been rewritten in the sense of starting again. Nontheless, every part has been complely renewed several times. I am reviewing only the final version. Git promotes regular commits, there have been more than 500.

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

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.

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.

Different ways to do currying below:

Partial completion is implemented using the language itself, not provided by the language.

Why would we choose 1 over the other? First simpler from caller side, second more flexible. Intuitive to call as a single call and can call self more easily.

In same cases, first form makes it easier to communicate that the completion comes in two parts, for example:

```
namedNodeExpr(previousExpr, capturing, name, pathStack, nodeStack, stackIndex )
```

There is a construction part (first 3 args) and a usage part (last three). Comsume many can only be constructed to ues consume 1 in second style because may refer to its own paritally completed version.

In first case, can avoid this: `consume1( partialComplete(consumeMany, previousExpr, undefined, undefined), undefined, undefined, pathStack, nodeStack, stackIndex);` because function factory can have optional arguments so don't have to give all of them

Function factory easier to debug. 'Step in' works. With partialCompletion have an awkward proxy function that breaks the programmer's train of thought as stepping through the code.

Why it is important to consider the frame of mind of the coder (CITEME: Hackers and Painters) and not just the elegance of the possible language expressions.

If implementing own functional caching, functional cache allows two levels of caching. Problematic though, for example no way to clear out the cache if memory becomes scarce.

Functional programming tends to lend better to minification than OO-style because of untyped record objects (can have any keys).

Lack of consistency in coding (don't write too much, leave to the conclusion)

Final consideration of coding: packaging up each unit to export a minimal interface. * Why minimal interfaces are better for minification

4.8.1 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,

4.9 JS code style

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.

Criticisms of Node. Esp from Erlang etc devs. Pyramid code and promises.

Although the streams themselves are stateful, because they are based on callbacks it is entirely possible to use them from a component of a javascript program which is wholly stateless.

4.9.1 functions over constructors

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.

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)

4.10 targeting node and the browser

Node+browser To use Node.js and

Need to build an abstraction layer over xhr/xhr2/node. Can only work for packets in-order, for out-of-order packets something else happens.

Use best of the capabilities of each.

4.11 Packaging the library as a single distributable file

- One file for browser and node is common.
- say how this is done

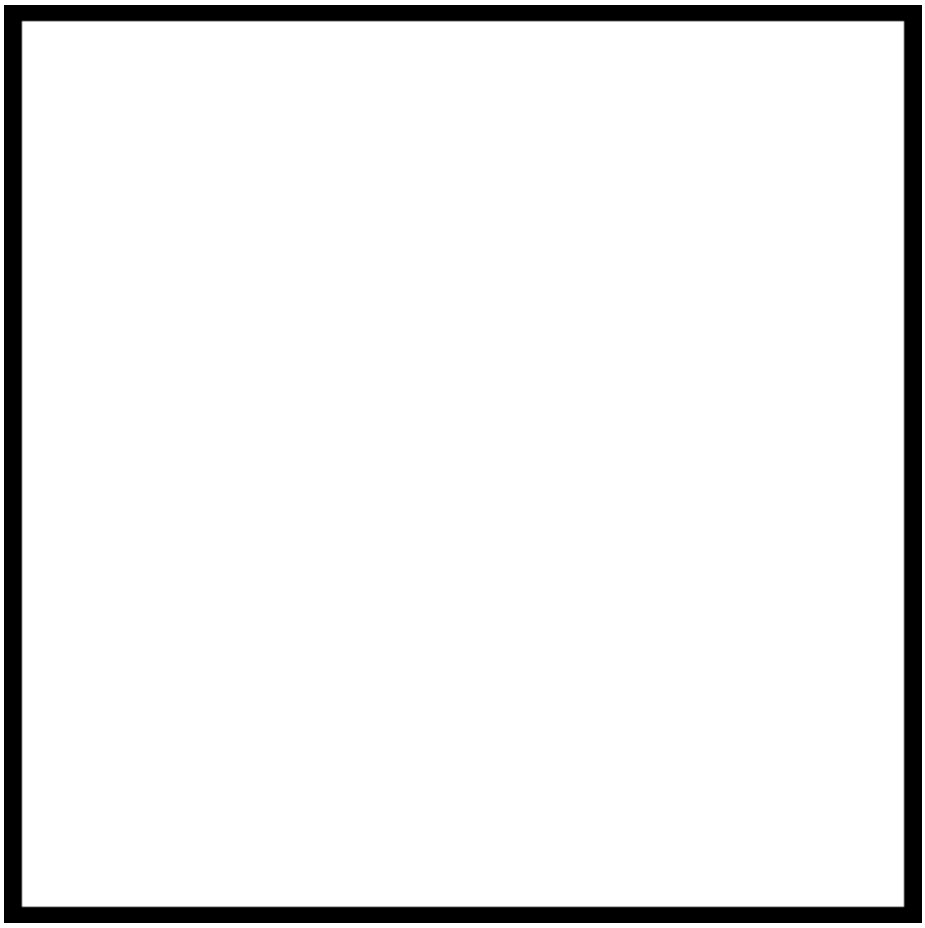

Figure 13: packaging of many javascript files into multiple single-file packages. The packages are individually targeted at different execution contexts, either browsers or node *get from notebook, split sketch diagram in half*

- why not doing this (adds bloat, inhibits micro-lib)
- extra challenges
- http adaptor is different
- packaging is different
- two distributable files, for node minification is not important so don't do to help debugging.

Composition of several source files into a distributable binary-like text file

Why distributed javascript is more like a binary than a source file. Licencing implications? Would be (maybe) under GPL. Not so under BSD.

Inherent hiding by wrapping in a scope.

Names of functions and variable names which are provably not possible to reference are lost for the sake of reduction of size of the source.

Packaging for node or browser. No need to minify for node but concatenation still done for ease of inclusion in projects

```
typical pattern for packaging to work in either a node.js server or a web browser
```

Packaging for use in frameworks.

- Many frameworks already come with a wrapper arround the browser's inbuilt ajax capabilities
- they don't add to the capabilities but present a nicer interface
- I'm not doing it but others are ** browser-packaged version should be use agnostic and therefore amenable to packaging in this way

Why uglify

- Covers whole language, not just a well-advised subset.
- Closure compiler works over a subset of javascript rather than the whole language.

Why not require. Bits on what rq is can go into B&R section. *Some of this can move into 3_Background.md*

- What it is
- Why so popular
- Why a loader is necessary - js doesn't come with an import statement
- How it can be done in the language itself without an import statement
- Meant more for AMD than for single-load code

- Situations AMD is good for - large site, most visitors don't need all the code loaded
- Depends on run-time component to be loaded even after code has been optimised
- Small compatible versions exist that just do loading (almond)

- Why ultimately not suitable for a library like this - would require user to use Require before adopting it.

Browserify is closer.

- Why it is better for some projects
- Very nearly meets my needs
- But http-compatability (https://github.com/substack/http-browserify), while complete enough, isn't compact enough to not push project over micro-library size

Testing post-packaging for small set of smoke tests. Can't test everything, only through public API.

Uglify. Why not Google Closure Compiler.

4.12 automated testing

How automated testing improves what can be written, not just making what is written more reliable.

TDD drives development by influencing the design - good design is taken as that which is amenable to testing rather than which describes the problem domain accurately or solves a problem with minimum resources. Amenable to testing often means split into many co-operating parts so that each part may be tested via a simple test.

Bt encourageing splitting into co-operating objects, TDD to a certain degree is anti-encapsulation. The public object that was extracted as a new concern from a larger object now needs public methods whereas before nothing was exposed.

Jstd can serve example files but need to write out slowly which it has no concept of. Customistation is via configuration rather than by plug-in, but even if it were, the threading model is not suitable to create this kind of timed output.

Tests include an extremely large file twentyThousandRecords.js to test under stress

Why jstd's built in proxy isn't sufficient. An example of a typical Java webserver, features thread-based mutlithreading in which threads wait for a while response to be received.

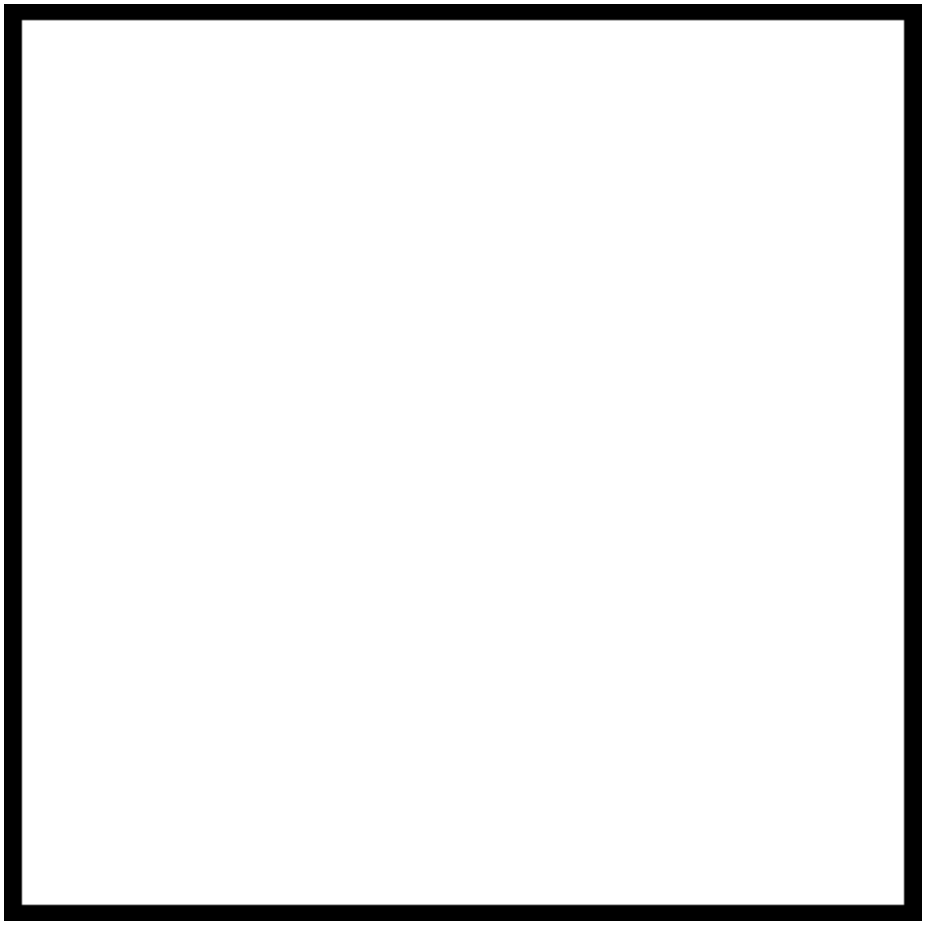

Figure 14: Relationship between various files and test libraries *other half of sketch from notebook*

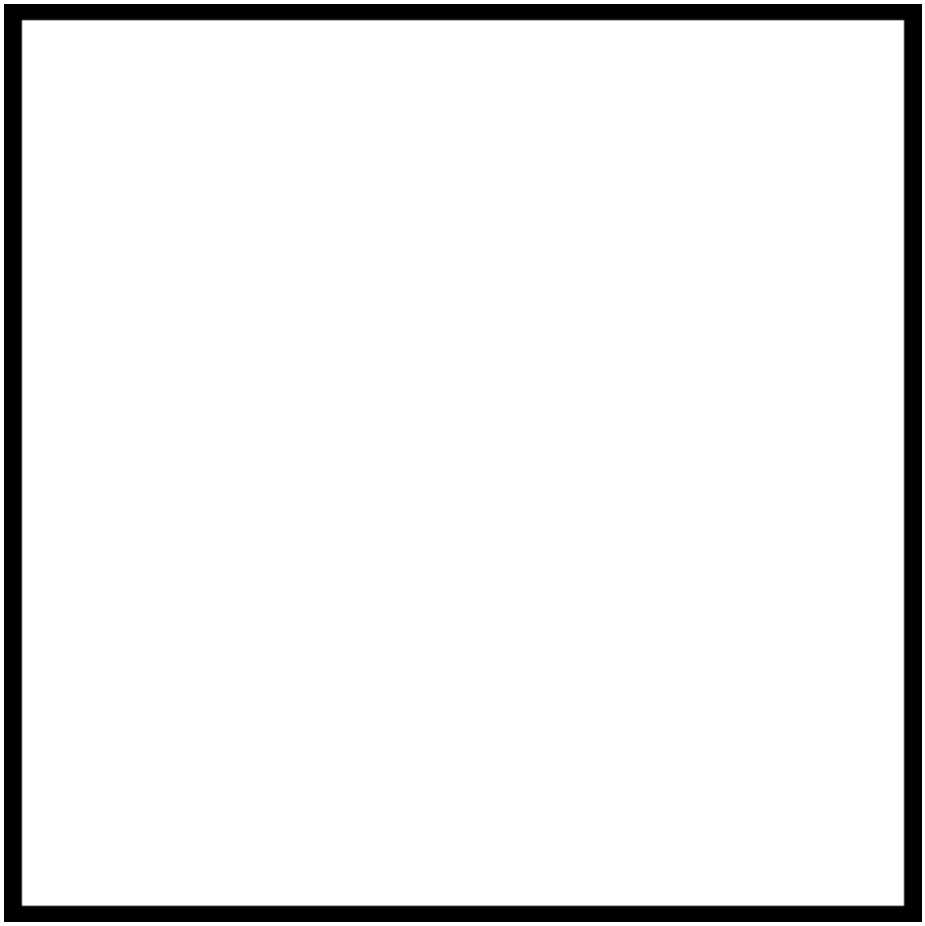

Figure 15: The testing pyramid is a common concept, relying on the assumption that verification of small parts provides a solid base from which to compose system-level behaviours. A Lot of testing is done on the low-level components of the system, whereas for the high-level tests only smoke tests are provided.

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.

Conversely, automated testing allows us to write incomprehensible code by making us into more powerful programmers, it is possible building up layers of complexity one very small part at a time that we couldn't write in a simple stage. Clarity > cleverness but cleverness has its place as well (intriducing new concepts)

Testing via node to give something to test against - slowserver. Proxy. JSTD not up to task. Shows how useful node is as a 'network glue'. The same as C was once described as a 'thin glue' [http://www.catb.org/esr/writings/taoup/html/ch04s03.html]. Transparent proxy is about 20 lines. Transparent enough to fool JSTD into thinking it is connecting directly to its server.

Node comes with very little built in (not even http) but relies on libraries written in the language itself to do everything. Could implement own http on top of sockets if wanted rather than using the provided one.

The test pyramid concept 15 fits in well with the hiding that is provided. Under the testing pyramid only very high level behaviours are tested as ??? tests. While this is a lucky co-incidence, it is also an unavoidable restriction. Once compiled into a single source file, the individual components are hidden, callable only from withing their closure. Hence, it would not be possible to test the composed parts individually post-concatenation into a single javascript file, not even via a workarround for data hiding such as found in Java's reflection. Whereas in Java the protection is a means of protecting otherwise addressable resources, once a function is trapped inside a javascript closure without external exposure it is not just protected but, appearing in no namespaces, inherently unreferenceable.

TDD fits well into an object pattern because the software is well composed into separate parts. The objects are almost tangible in their distinction as separate encapsulated entities. However, the multi-paradigm style of my implementation draws much fainter borders over the implementation's landscape.

Approach has been to the test the intricate code, then for wiring don't have tests to check that things are plumbed together correctly, rather rely on this being obvious enough to be detected via a smoke test.

A good test should be able to go unchanged as the source under test is refactored. Indeed, the test will be how we know that the code under test still works as intended. Experince tells me that testing that A listens to B (ie that the controller wires the jsonbuilder up to clarinet) produces the kind of test that 'follows the code arround' meaning that because it is testing implementation details rather than behaviours, whenever the implementation is updated the tests have to be updated too.

By testing individual tokens are correct and the use of those tokens as a wider expression, am testing the same thing twice. Arguably, redundant effort. But

may simply be easier to write in that way - software is written by a human in a certain order and if we take a bottom-up approach to some of that design, each layer is easier to create if we first know the layers that it sits on are sound. Writing complex regular expressions is still programming and it is more difficult to test them completely when wrapped in rather a lot more logic than directly. For example, a regex which matches "{a,b}" or "{a}" but not "{a,}" is not trivial.

Can test less exhaustively on higher levels if lower ones are well tested, testing where it is easier to do whilst giving good guarantees.

Genuine data hiding gets in the way sometimes. Eg, token regexes are built from the combination of smaller regualar expressions for clarity (long regular expressions are concise but hard to read), and then wrapped in functions (why? - explain to generify interface) before being exposed. Because the components are hidden in a scope, they are not addressable by the tests and therefore cannot be directly tested. Reluctantly

One dilemma in implementing the testing is how far to test the more generic sections of the codebase as generic components. A purist approach to TDD would say

Could implement a resume function for if transmission stops halfway

```
.onError( error ) {
   this.resume();
}
```

4.13 Inversion of Control

Aim of creating a micro-library rules out building in a general-purpose IoC library.

However, can still follow the general principles.

Why the Observer pattern (cite: des patterns) lends itself well to MVC and inversion of control.

What the central controller does; acts as a plumber connecting the various parts up. Since oboe is predominantly event/stream based, once wired up little intervention is needed from the controller. Ie, A knows how to listen for ??? events but is unintested who fired them.

4.14 support for older browsers

Still works as well as non-progressive json Could be used for content that is inherently streaming (wouldn't make sense without streaming)

4.14.1 polyfilling

The decline of bad browsers. Incompatibility less of a concern than it was.

Node doesn't require, built on v8.

http://www.jimmycuadra.com/posts/ecmascript-5-array-methods Unlike the new methods discussed in the first two parts, the methods here are all reproducible using JavaScript itself. Native implementations are simply faster and more convenient. Having a uniform API for these operations also promotes their usage, making code clearer when shared between developers.

Even when only used once, preferable to polyfill as a generic solution rather than offer a one-time implementation because it better splits the intention of the logic being presented from the mechanisms that that logic sits on and, by providing abstraction, elucidates the code.

4.15 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 imutability 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.

5 Conclusion

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

5.1 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.

5.2 Size

Comment on the size of the libraray

5.3 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

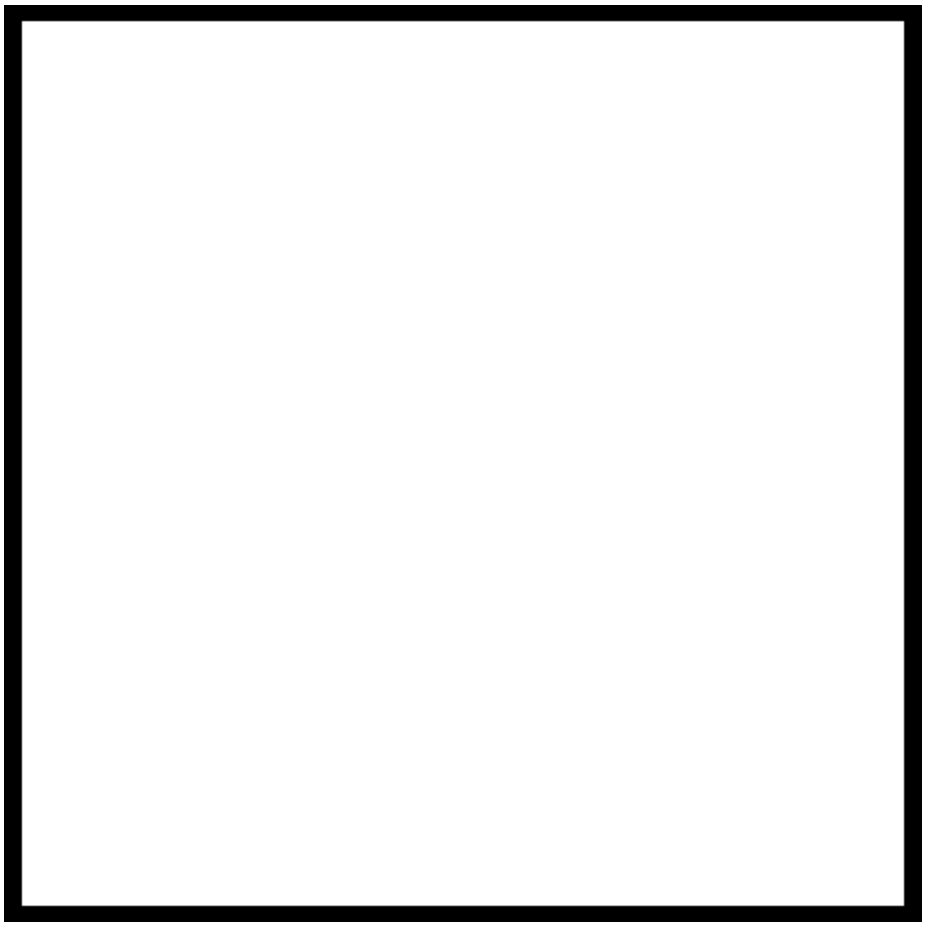

Figure 16: A pie chart showing the sizes of the various parts of the codebase

5.4 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?

5.5 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 Appendix

7 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.

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

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{]}.

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/.