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

Jim Higson

2013

Contents

9 Appendix iii: Benchmarking   **109**

10 Bibliography   **115**

List of Figures

1 Abstract

A new design for http client libraries incorporating http streaming, pattern matching, and incremental parsing, with the aim of improving performance, fault tolerance, and encouraging a greater degree of loose coupling between programs. A Javascript client capable of progressively parsing JSON resources is presented targeting both Node.js and web browsers. Loose coupling is particularly considered in light of the application of Agile methodologies to REST and 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 developer ergonomics and end user perception of performance.

2 Introduction

This purpose of this dissertation is to encourage the REST paradigm to be viewed through a novel lens which 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 make is felt more strongly as a modification in how we *think* about http than as the delivery of new software.

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 a some invocation halts execution for the duration of the request before evaluating to the fetched resource; or asynchronous in which the logic is specified to be applied to a response once it is available. Languages encourage our thinking to follow the terms that they easily support(Whorf 1956). While there is some overlap, languages which promote concurrency though threading consider blocking in a single thread to be acceptable and will generally prefer the former mode whereas languages with first class functions are naturally conversant in callbacks and will prefer the latter. We should remember in programming that languages limit the patterns that we readily see (Yukihiro 2003) and that better mappings may be possible. This observation extends to graphical notations such as UML whose constructs strongly reflect the programming languages of the day. For any multi-packet message sent via a network some parts will arrive before others, at least approximately in-order, but viewed from inside a language whose statements invariably yield single, discrete values it comfortable to conceptualise the REST response as a discrete event. UML sequence diagrams contain the syntax for instantaneously delivered return values, with no corresponding notation for a resource whose data is progressively revealed.

In most practical cases where we wish to be fast in performing a task there is no reasonable distinction between acting *earlier* and being *quicker*. To create efficient software we should be using data at the first possible opportunity: examining content *while it streams* rather than holding it unexamined until it is wholly available.

While 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. Similarly building on previous ideas, no new computing techniques need be invented to realise my thesis. As a minimum it requires an http client which reveals the response whilst it is in progress and a parser which can begin to interpret that response before it sees all of it. Nor is it novel to use these preexisting parts in composition. Every current web browser already implements such a schema; load any complex webpage – essentially an aggregation of hypertext and other resources – the HTML will be parsed and displayed incrementally while it is downloading and resources such as images are requested in parallel as soon as

they are referenced. The images may themselves be presented incrementally in the case of progressive JPEGs or SVGs[1]. This incremental display is achieved through highly optimised software 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.

2.1   How REST aggregation could be faster
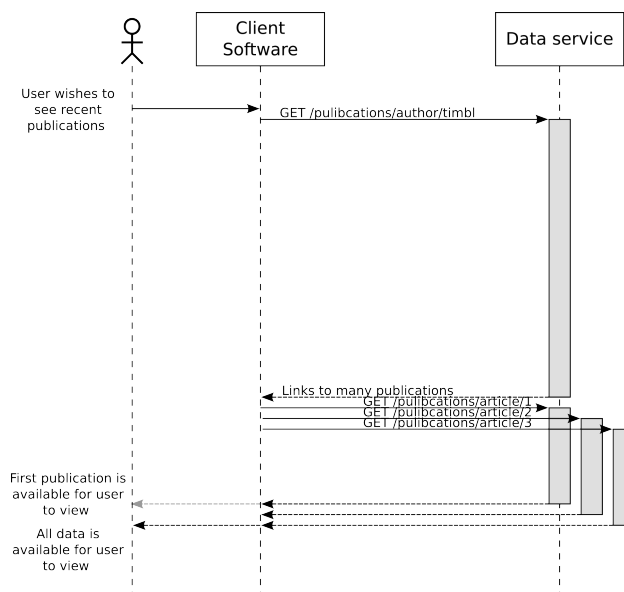
Figure 1: **Sequence diagram showing the aggregation of low-level REST resources.** A client fetches an author's publication list and then their first three articles. This sequence represents the most commonly used technique in which the client does not react to the response until it is complete. In 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 comparatively illustrate how a progressive client may, without adjustments to the server, be used to produce an aggregated resource sooner. This results in a moderate improvement in the time taken to show the complete aggregation but a dramatic improvement in the time to show the first content. The ability to present the first content as early as possible is a desirable trait

---

[1]For quite an obviously visible example of progressive SVG loading, try loading this SVG using a recent version of Google Chrome: http://upload.wikimedia.org/wikipedia/commons/0/04/Marriage_(Same-Sex_Couples)_Bill,_Second_Reading.svg For the perfectionist SVG artist, not just the final image should be considered but also the XML source order, for example in this case it would be helpful if the outline of the UK appeared first and the exploded sections last.
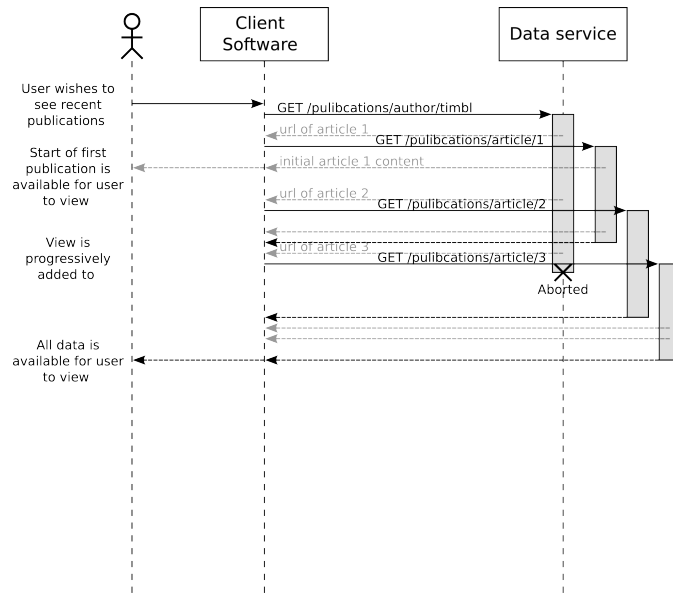
Figure 2: **Revised aggregation sequence for a client capable of progressively interpreting the resources.** Because arrows in UML sequence diagrams draw returned values as a one-off happening rather than a continuous process, I have introduced a lighter arrow notation representing fragments of an incremental response. Each request for an individual publication is made as soon as the its URL can be extracted from the publications list and once all required data has been read from the original response it is aborted rather than continue to download unnecessary data.

for system usability because it allows the user to start reading earlier and a progressively rendered display in itself increases the human perception of speed (Geelhoed et al. 1995). Note also how the cadence of requests is more steady in Figure 2 with four connections opened at roughly equal intervals rather than a single request followed by a rapid burst of three. Both clients and servers routinely limit the number of simultaneous connections per peer so avoiding bursts of requests is further to our advantage. Appendix i lists some actual limits.

Nodes in an n-tier architecture defy categorisation as 'client' or 'server' in a way which is appropriate from all frames of reference. A node might be labeled as the 'server' from the layer below and 'client' from the layer above. Although the "client software" labels in the figures above hint at something running directly on a user's own device, the same benefits apply if this layer is running remotely. If this layer were generating a web page on the server-side to be displayed by the client's browser, the perceptual speed improvements apply because of http chunked encoding (Stefanov 2009). If this layer were a remote aggregation service, starting to write out the aggregated response early provides much the same benefits so long as the client is also able to interpret it progressively and, even if it were not, the overall delivery remains faster.

2.2 Stepping outside the big-small tradeoff

Where a domain model contains data in a series with continuous ranges requestable via REST, I have often noticed a tradeoff in the client's design with regards to how much should be requested in each call. Because at any time it shows only a small window into a much larger model, the social networking site Twitter might be a good example. The Twitter interface designers adopted a popular interface pattern, Infinite Scrolling (Ahuvia 2013). Starting from an initial page showing some finite number of tweets, once the user scrolls and reaches the end of the list the next batch is automatically requested. When loaded, this new batch is converted to HTML and added to the bottom of the page. Applied repeatedly the illusion of an infinitely long page in maintained, albeit punctuated with pauses whenever new content is loaded. For the programmers working on this presentation layer there is a tradeoff between sporadically requesting many tweets, yielding long, infrequent delays and frequently requesting a little, giving an interface which stutters momentarily but often.

I propose that progressive loading could render this tradeoff unnecessary by simultaneously delivering the best of both strategies. In the Twitter example this could be achieved by making large requests but instead of deferring all rendering until the request completes, add the individual tweets to the page as they are incrementally parsed out of the ongoing response. With a streaming transport, the time taken to receive the first tweet should not vary depending on the total number that are also being sent so there is no relationship between the size of the request made and the time taken to first update the interface.

2.3 Staying fast on a fallible network

REST operates over networks whose reliability varies widely. On unreliable networks connections are abruptly dropped and in my opinion existing http clients handle unexpected terminations suboptimally. Consider the everyday situation of a person using a smartphone browser to check their email. Mobile data coverage is often weak outside of major cities (Gill 2013) so while travelling the signal will be lost and reestablished many times. The web developer's standard AJAX toolkit is structured in a way that encourages early terminated connections to be considered as wholly unsuccessful rather than as partially successful. For example, the popular AJAX library jQuery automatically parses JSON or XML responses before passing back to the application but given an early disconnection there is no attempt to hand over the partial response. To the programmer who knows where to look the partial responses are extractable as raw text but handling them involves writing a special case and is difficult because standard parsers are not amenable to incomplete markup. Because of this difficulty I can only find examples of partial messages being dropped without inspection. For the user checking her email, even if 90% of her inbox had been retrieved before her phone signal was lost, the web application will behave as if it received none and show her nothing. Later, when the network is available again the inbox will be downloaded from scratch, including the 90% which previously delivered. I see much potential for improvement here.

I propose moving away from this polarised view of successful/unsuccessful requests to one in which identifiable parts of a message are recognised as interesting in themselves, regardless of what follows, and these parts are handed back to the application as streaming occurs. 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. Should an early disconnection occur, the content delivered up to that point will have already been handled so no special case is required to salvage it. In most cases the only recovery necessary will be to make a new request for just the part that was missed. This approach is not incompatible with a problem domain where the usefulness of an earlier part is dependent on the correct delivery of the whole providing optimistic locking is used. In this case earlier parts may be used immediately but their effect rolled back should a notification of failure be received.

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

In most respects a SOA architecture fits well with the fast release cycle encouraged by Agile methodologies. 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. In allowing a design to emerge organically it should be possible for the structure of resource formats to be realised slowly and iteratively while a greater understanding of the problem is gained. Unfortunately in practice the ability to change is hampered by tools which encourage programming against rigidly specified formats. If a program is allowed to be tightly coupled to a data format it will resist changes in the programs which produce data to that format. 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 insisting on exact data formats, subsystems become tightly coupled and 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 clients the ability to use a response whilst being only loosely coupled to the *shape* of the 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, the guiding principle being that it is preferable to produce a little well than more badly. Amongst commentators on start-up companies this is known as a *zoom-in pivot* (Reis 2011 p172) and the work it produces should be the *Minimum Viable Product* or MVP (Reis 2011 p106-110). With a focus on quality I could not deliver a full stack so I am obliged to implement only solutions which interoperate with existing deployments. This is advantageous; to somebody looking to improve their system small enhancements are more inviting 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-receptive 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 considering how the semantics of a message are expanded as a system's design emerges and commenting on the value of loose coupling between data formats and the programs which act on them in avoiding disruption given unanticipated format changes.

3   Background

Figure 3: **Labelling nodes in an n-tier architecture**. Regardless of where a node is located, REST may be used as the means of communication. By focusing on REST clients, nodes in the middleware and presentation layer fall in our scope. Although network topology is often split about client and server side, for our purposes categorisation as tiers is a more meaningful distinction. According to this split the client-side presentation layer and server-side presentation layer serve the same purpose, generating mark-up based on aggregated data created in the middle tier

3.1   The web as an application platform

Application design has historically charted an undulating path pulled by competing approaches of thick and thin clients. Having evolved from a document viewing system to an application platform for all but the most specialised tasks, the web perpetuates this narrative by resisting categorisation as either mode.

While the trend is generally for more client scripting and for some sites Javascript is now requisite, there are also counter-trends. In 2012 twitter reduced load times to one fifth of their previous design by moving much of their rendering back to the server-side, commenting that "The future is coming and it looks just like the past" (Lea 2012). Under this architecture short, fast-loading pages are generated on the server-side but Javascript is also provides progressively enhancement. Although it does not generate the page anew, the Javascript must know how to create most of the interface elements so one weakness of this architecture is that much of the presentation layer logic must be expressed twice.

Despite client devices taking on responsibilities which would previously have been performed on a server, there is a limit to how much of the stack may safely be offloaded in this direction. The client-side ultimately falls under the control of the user so no important business decisions should be taken here. A banking

site should not allow loan approval to take place in the browser because for the knowledgeable user any decision would be possible. Separated from data stores by the public internet, the client is also a poor place to perform data aggregation or examine large data sets. For non-trivial applications these restrictions encourage a middle tier to execute business logic and produce aggregate data.

While REST may not be the only communications technology employed by an application architecture, for this project we should examine where the REST clients fit into the picture. REST is used to pull data from middleware for the sake of presentation regardless of where the presentation resides. Likewise, rather than connect to databases directly, for portability middlewares often communicate with a thin REST layer which wraps data stores. This suggests three uses:

- From web browser to middleware
- From server-side presentation layer to middleware
- From middleware to one or more nodes in a data tier

Fortunately, each of these contexts require a similar performance profile. The node is essentially acting as a router dealing with small messages containing only the information they requested rather than dealing with a whole model. As a part of an interactive system low latency is important whereas throughput can be increased relatively cheaply by adding more hardware. As demand for the system increases the total work required grows but the complexity of any one of these tasks does remains constant. Although serving any particular request might be done in series, the workload as a whole at these tiers consists of many independent tasks and as such is embarrassingly parallelisable.

3.2   Json and XML data transfer formats

later mention JSON 'nodes'/'paths' a lot. Good place to intro here

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.

3.3   Node.js

Node.js is a general purpose tool for executing Javascript outside of a browser. I has the aim of low-latency i/o and is used predominantly for server applications and command line tools. It is difficult to judge to what degree Javascript is a distraction from Node's principled design and to what degree the language defines the platform.

In most imperative languages the thread is the basic unit of concurrency. whereas Node presents the programmer with a single-threaded abstraction. Threads are an effective means to share parallel computation over multiple cores but are less well suited to scheduling concurrent tasks which are mostly i/o dependent. Programming threads safely with shared access to mutable objects requires great care and experience, otherwise the programmer is liable to create race conditions. Considering for example a Java http aggregator; because we wish to fetch in parallel each http request is assigned to a thread. These 'requester' tasks are computationally simple: make a request, wait for a complete response, and then participate in a Barrier to wait for the others. Each thread consumes considerable resources but during its multi-second lifespan requires only a fraction of a millisecond on the CPU. It is unlikely any two requests return at exactly the same moment so usually the threads will process in series rather than parallel anyway. Even if they do, the actual CPU time required in making an http request is so short that any concurrent processing is a pyrrhic victory. Following Node's lead, traditionally thread-based environments are beginning to embrace asynchronous, single-threaded servers. The Netty project can be though of as roughly the Java equivalent of Node.

Node builds on a model of event-based, asynchronous i/o that was established by Javascript execution in web browsers. Although Javascript in a browser
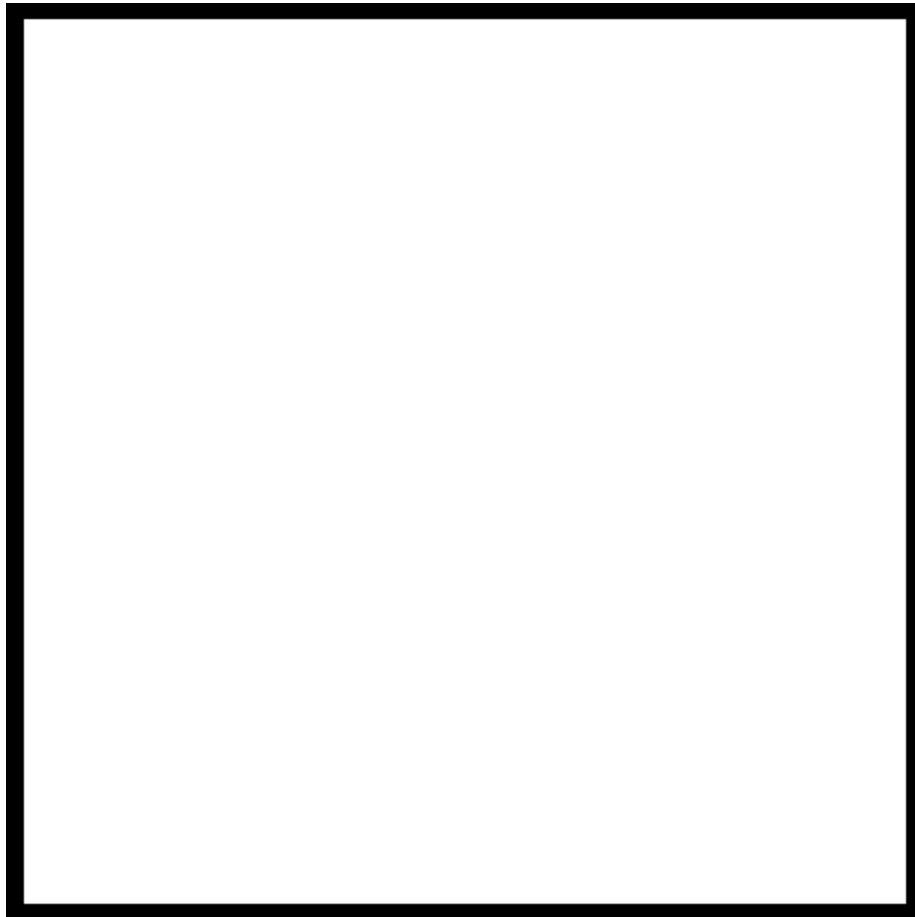
Figure 4: *Single-threaded vs multi-threaded scheduling for a http aggregator*

may be performing multiple tasks simultaneously, for example requesting several resources from the server side, it does so from within a single-threaded virtual machine. Node similarly facilitates concurrency by managing an event loop of queued tasks and providing exclusively non-blocking i/o. Unlike Erlang, Node does not swap tasks out preemptively, it always waits for tasks to complete before moving onto the next. This means that each task must complete quickly to avoid holding up others. *Prima facie* this might seem like an onerous requirement to put on the programmer but in practice with only non-blocking i/o each task naturally exits quickly without any special effort. Accidental non-terminating loops or heavy number-crunching aside, with no reason for a task to wait it is difficult to write a node program where the tasks do not complete quickly.

Each task in node is simply a Javascript function. Node is able to swap its single Javascript thread between these tasks efficiently while providing the programmer with an intuitive interface because of closures. Utilising closures, the responsibility of maintaining state between issuing an asynchronous call and receiving the callback is removed from the programmer by folding it invisibly into the language. This implicit data store requires no syntax and feels so natural and inevitable that it is often not obvious that the responsibility exists at all.

Consider the example below. The code schedules three tasks, each of which are very short and exit quickly allowing Node to finely interlace them between other concurrent concerns. The `on` method is used to attach functions as listeners to streams. However sophisticated and performant this style of programming, to the developer it is hardly more difficult an expression than if a blocking io model were followed. It is certainly easier to get right than synchronising mutable objects for sharing between threads.

```javascript
function printResourceToConsole(url) {

    http.get(url)
        .on('response', function(response){

            // This function will be called when the response starts.
            // It logs to the console, adds a listener and quickly exits.

            // Because it is captured by a closure we are able to reference
            // the url parameter after the scope that declared it has finished.
            console.log("The response has started for " + path);

            response.on('data', function(chunk) {
                // This function is called each time some data is received from the
                // http request. In this example we write the response to the console
                // and quickly exit.
                console.log('Got some response ' + chunk);

            }).on('end', function(){
```

```
            console.log('The response is complete');
        })

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

        console.log("There was an error: " + e.message);
    });
    console.log("The request has been made");
}
```

> "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." (Ogden 2012)

In Node i/o is performed through a unified streaming interface regardless of the source. The streams follow a publisher-subscriber pattern fitting comfortably with the wider event-driven model. Although the abstraction provided by streams is quite a thin layer on top of the host system's socket, it forms a powerful and intuitive interface. For many tasks it is preferable to program in a 'plumbing' style by joining one stream's output to another's input. In the example below a resource from the internet is written to the local filesystem.

```
http.get(url)
    .on('response', function(response){
        response.pipe(fs.createWriteStream(pathToFile));
    });
```

3.4   Browser XML Http Request (XHR)

Making http requests from Javascript, commonly termed AJAX, was so significant in establishing the modern web architecture that it is sometimes used synonymously with Javascript-rich web applications. Although AJAX is an acronym for **A**synchronous **J**avascript (**a**nd) **X**ML, this reflects the early millennial enthusiasm for XML as the one true data format and in practice any textual format may be transferred. Today JSON is generally preferred, especially for delivery to client-side web applications. During the 'browser war' years web browsers competed by adding non-standard features; Internet Explorer made AJAX possible in 2000 by exposing Microsoft's Active X *Xml Http Request* (XHR) class to the Javascript sandbox. This was widely copied and near equivalents were added to all major browsers. In 2006 the interface was eventually formalised by the W3C (van Kesteren and Jackson 2006). XHR's slow progresss to standardisation reflected a period of general stagnation for web standards. HTML4 reached Recommendation status in 2001 but having subsequently found

20

several evolutionary dead ends such as XHTML, there would be no major updates until HTML5 started to gather pace some ten years later.

Despite a reputation for being poorly standardised, as a language Javascript enjoys consistent implementation. More accurately we would say that browser APIs exposed to Javascript lack compatibility. Given this backdrop of vendor extensions and lagging standardisation, abstraction layers predictably rose in popularity. Various abstractions competed primarily on developer ergonomics with the popular jQuery and Prototype.js libraries promoting themselves as *"do more, write less"* and *"elegant APIs around the clumsy interfaces of Ajax"*. Written against the unadorned browser, Javascript applications read as a maze of platform-detection and special cases. Once applications were built using Javascript abstractions over the underlying browser differences, they could be written purposefully and were able to express more complex ideas without becoming incomprehensible.

JSON, itself a subset of Javascript, emerged as the main format output by REST end points when requesting via AJAX. Javascript programmers occupy a privileged position whereby their serialisation format maps exactly onto the inbuilt types of their programming language. As such there is never any confusion regarding which object structure to de-serialise to. Should this advantage seem insubstantial, contrast with the plethora of confusing and incompatible representations of JSON that are output by the various Java parsers: JSON's Object better resembles Java's Map interface than Java Objects, creating linguistic difficulties, and the confusion between JSON null, Java null, and Jackson's NullNode[2] is a common cause of errors. Emboldened by certainty regarding deserialisation, AJAX libraries directly integrated JSON parsers, providing a call style for working with remote resources so streamlined as to require hardly any additional effort.

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

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

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

3.5   XHRs and streaming

Browser abstraction layers brought an improvement in expressivity to web application programming but were ultimately limited to supporting the lowest

---

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

common denominator of the available browser abilities. At the time that the call style above was developed the most popular browser gave no means of access to partial responses. Inevitably, it draws a conceptualisation of the response as a one-time event with no accommodation offered for progressively delivered data.

The followup standard, XHR2 is now at Working Draft stage. Given ambitions to build a streaming REST client, of greatest interest 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)

Presently this event is supported by all but legacy browsers.

The historic lack of streaming allowed by XHR stands incongruously with the browser as a platform which has long used streaming to precipitate almost every other remote resource. Progressive image formats, html, svg, video and Javascript itself (script interpretation starts before the script is fully loaded) are all examples of this.

3.6 Browser streaming frameworks

The web's remit is increasingly widening to encompass scenarios which would have previously been the domain of native applications. In order to use live data many current webapps employ frameworks which push soft real-time events to the client side. In comparison to the XHR2 progress event, this form of streaming has a different but overlapping purpose. Whereas XHR2 enables downloads to be viewed as short-lived streams but does not otherwise disrupt the sequence of http's request-response model, streaming frameworks facilitate an entirely different sequence, that of perpetual data. Consider a webmail interface; initially the user's inbox is downloaded via REST and a streaming download might be used to speed its display. Regardless of if the response is interpreted progressively, this inbox download is a standard REST call and shares little in common with the push events which follow to provide instant notification as new messages arrive.

Push tables sidestep the browser's absent data streaming abilities by leaning on a resource that it can stream: progressive html. From the client a page containing a table is hidden in an off-screen iframe. This table is served from a a page that never completes, fed by a connection that never closes. When the server wishes to push a message to the client it writes a new row in this table which is then noticed by Javascript monitoring the iframe on the client. More recently, **Websockets** is a new standard that builds a standardised streaming transport on top of http's chunked mode. Websockets requires browser implementation and cannot be retrofitted to older browsers through Javascript. Websockets are

a promising technology but for the time being patchy support means it cannot be used without a suitable fallback.

These frameworks do not interoperate at all with REST. Because the resources they serve never complete they may not be read by a standard REST client. Unlike REST they also are not amenable to standard http mechanics such as caching. A server which writes to an esoteric format requiring a specific, known, specialised client also feels quite anti-REST, especially when we consider that the format design reflects the nature of the transport more so than the resource. This form of streaming is not, however, entirely alien to a SOA mindset. The data formats, while not designed primarily for human readability are nontheless text based and a person may take a peek inside the system's plumbing simply by observing the traffic at a particular URL. In the case of push-tables, an actual table of the event's properties may be viewed from a browser as the messages are streamed.

3.7   Parsing: SAX and Dom

In the XML domain 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].

```javascript
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 for 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 5: *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

```java
// 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[^ 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:

```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:

```xml
<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.
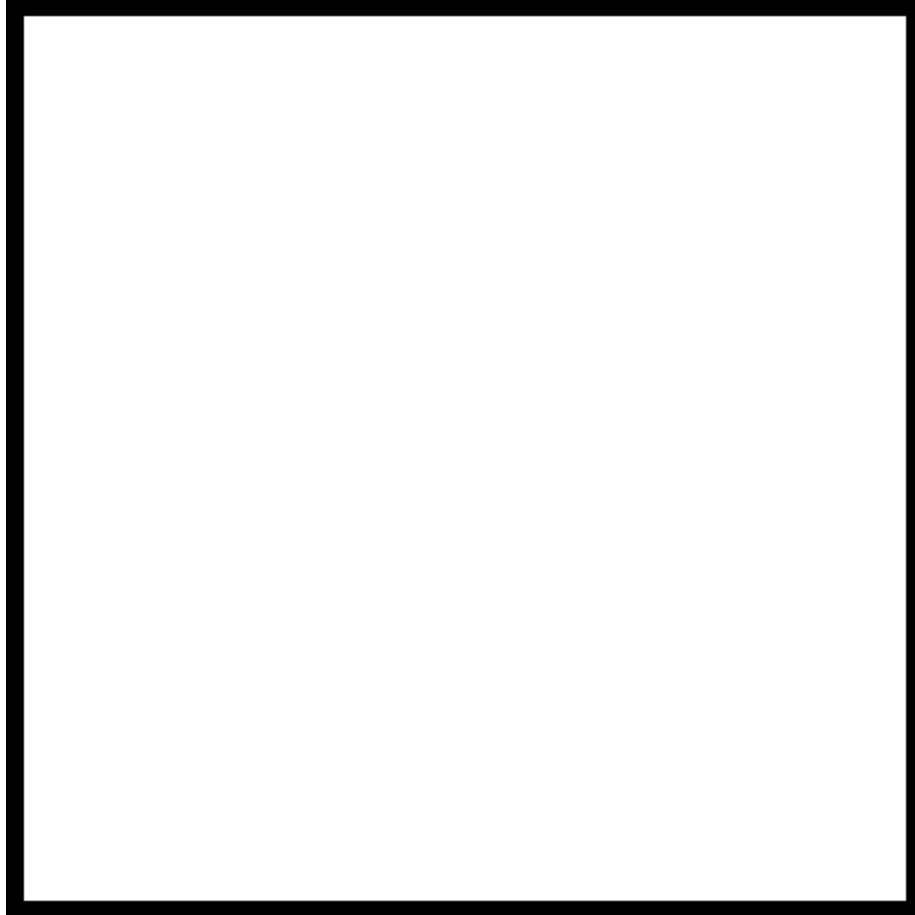
Figure 6: Relationship between the main players in the JS testing landscape. JSTD, Karma, Jasmine, NodeUnit, jasmine-node, Browsers

4   Design and Reflection:

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

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

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

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

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

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

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

4.1   JSONPath and types

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

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

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

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

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

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

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

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

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

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

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

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

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

In the below example typing is trickier still.

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

```
        }
    }
}
```

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

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

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

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

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

4.2   JSONPath improving stability over upgrades

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

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

37
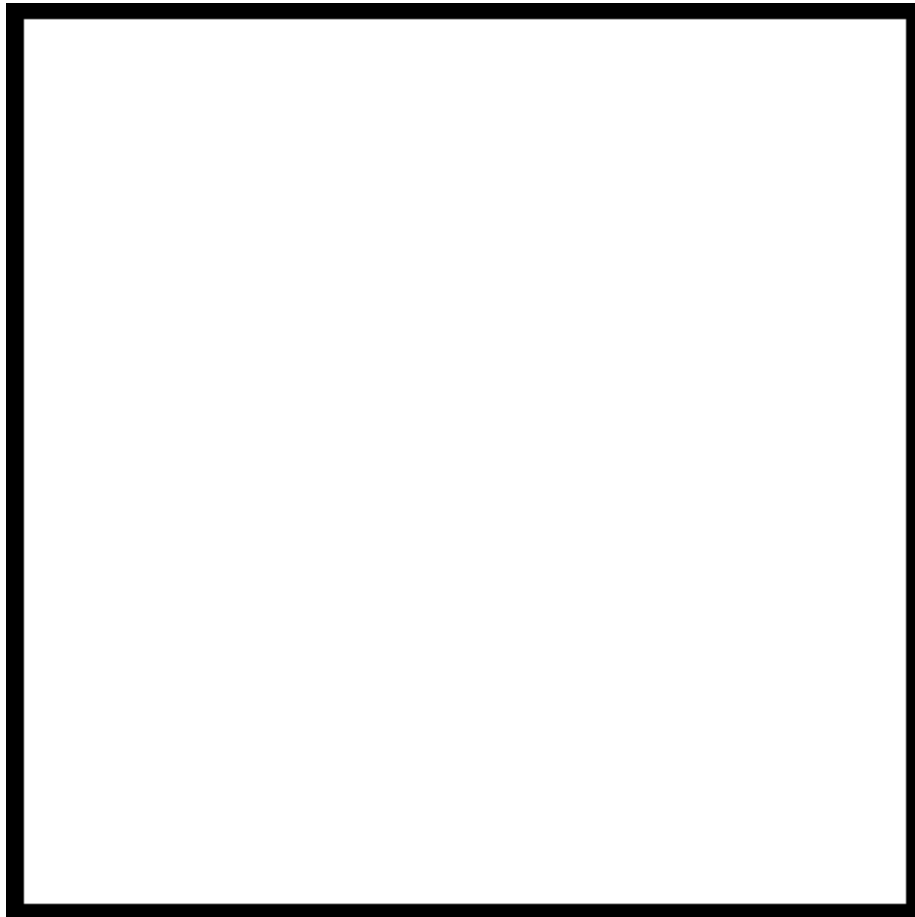
Figure 7: extended json rest service that still works - maybe do a table instead

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

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

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

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

4.3 Importing CSS4's explicit capturing to Oboe's JSON-Path

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

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

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

4.4 Parsing the JSON Response

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

4.5 API design

API allows body to be given as Object and converts into JSON because it is anticipated that REST services which emmit JSON will also accept it

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

Expose single global.

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

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

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

While for simple web applications usage is much as above,
In real world usage on more complex apps jQuery.ajax is often injected into the scope of the code which wants to use it. Easier stubbing so that tests don't have to make actual AJAX calls.

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

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

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

Taking on this style,

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

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

```
oboe('resources/someJson.json')
   .node({
```

41

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

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

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

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

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

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

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

```
oboe('resources/someJson.json')
   .on( 'node', 'medalWinners.*', function(person, path, ancestors) {
      console.log( person.name + ' won the ' + lastOf(path) + ' medal' );
   });
```

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

4.6 Earlier callbacks when paths are matched

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

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

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

4.7   Oboe.js as a Micro-Library

Http traffic, especially sending entropy-sparse text formats is often gzipped at point of sending in order to deliver it more quickly, so in measuring a download footprint it usually makes more sense to compare post-gzipping. A Javascript library qualifies as being *micro* if it is delivered in 5k or less, 5120 bytes. Micro-libraries also tend to follow the ethos that it is better for a developer to gather together several tiny libraries than one that uses a one-size-fits-all approach, perhaps echoing the unix command line tradition of small programs which each do do exactly one thing. Javascript Micro-libraries are listed at,[3] which includes this project. Oboe.js feels on the edge of what is possible to elegantly do as a micro-library so while the limit is somewhat arbitrary, for the sake of adoption smaller is better and keeping below this limit whilst writing readable code is an interesting challenge. As well as being a small library, in the spirit of a micro-library a project should impose as few restrictions as possible on its use and be designed to be completely agnostic as to which other libraries or programming styles that it is used with.

4.8   Choice of streaming data transport

Considering longpoll, push-tables and websockets. . .

I find that it is not necessary to take this dichotomous view of streaming.

Whilst there is some overlap, each of the approaches above addresses a problem only tangentially related to this project's aims. Firstly,

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.

---

[3]Http://microjs.com/.

44

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.

4.9   Handling transport failures

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

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

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

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

4.10   Fallback support on less-capable platforms

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

Because of differences in the capabilities in browsers, providing a streaming REST client is not possible on all browsers. If this were possible, it would not have been necessary to invent push pages or long polling. Specifically, none but the most recent versions of Internet Explorer provide any way to access an AJAX response before it is complete. I have taken the design decision that it is

ok to degrade on these platforms so long as the programmer developing with Oboe.js does not have to make special cases for these platforms. Likewise, nor should the REST service need be aware of the client, disallowing detecting client capabilities and switching transport strategy. Requiring branching on either side places extra responsibilities on the programmer which they would not otherwise be required to consider whilst viewing REST through a non-streaming lens.

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

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

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

5 Implementation

5.1 Components of the project
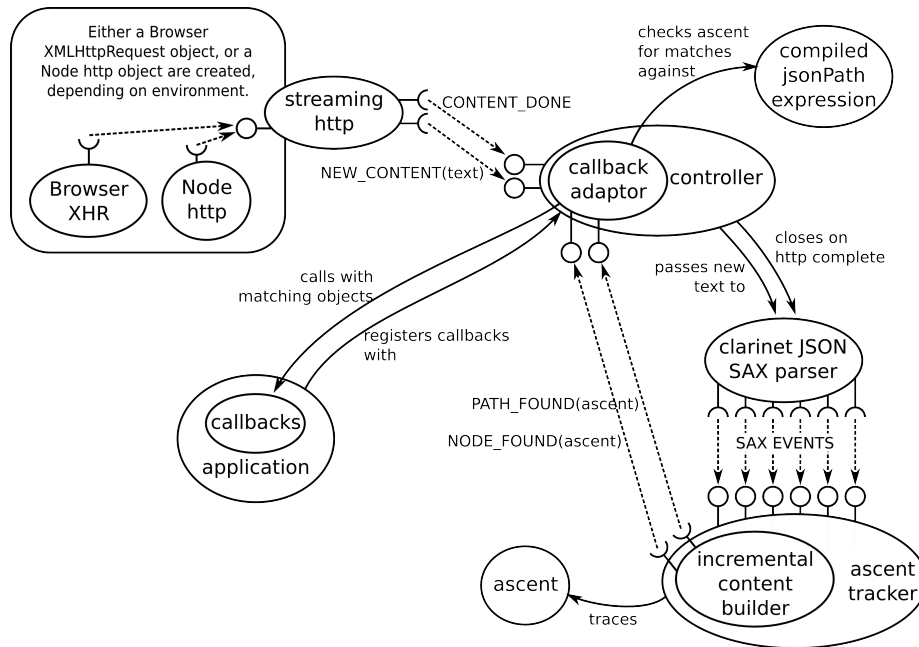
Figure 8: **Major components that make up Oboe.js** illustrating program flow from http transport to registered callbacks. Every component is not shown here. Particularly, components whose responsibility it is to initialise the oboe instance but have no role once it is running are omitted. UML facet/receptacle notation is used to show the flow of events with event names in capitals.

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

Internally, communication between components is facilitated by an event bus which is local to to Oboe instance. Most components interact solely by picking up events, processing them and publishing further events in response. Essentially, Oboe's architecture resembles a fairly linear pipeline visiting a series of units, starting with http data and sometimes ending with callbacks being notified. This use of an event bus is a variation on the Observer pattern which removes the need for each unit to obtain a reference to the previous one so that it may observe it, giving a highly decoupled shape to the library. Once everything is wired into the bus very little central control is required and the larger behaviours

emerge as the consequence of this interaction between finer ones. One downside is perhaps that a central event bus does not lend itself to a UML class diagram, giving a diagram shape with an event bus as a central hub and everything else hanging off it as spokes.

5.2   Automated testing

Automated testing improves what can be written, not just making what is written more reliable. Tests deal with the problem of "irreducible complexity" - when a program is made out of parts whose correct behaviour cannot be observed without all of the program. Allows smaller units to be verified before verifying the whole.
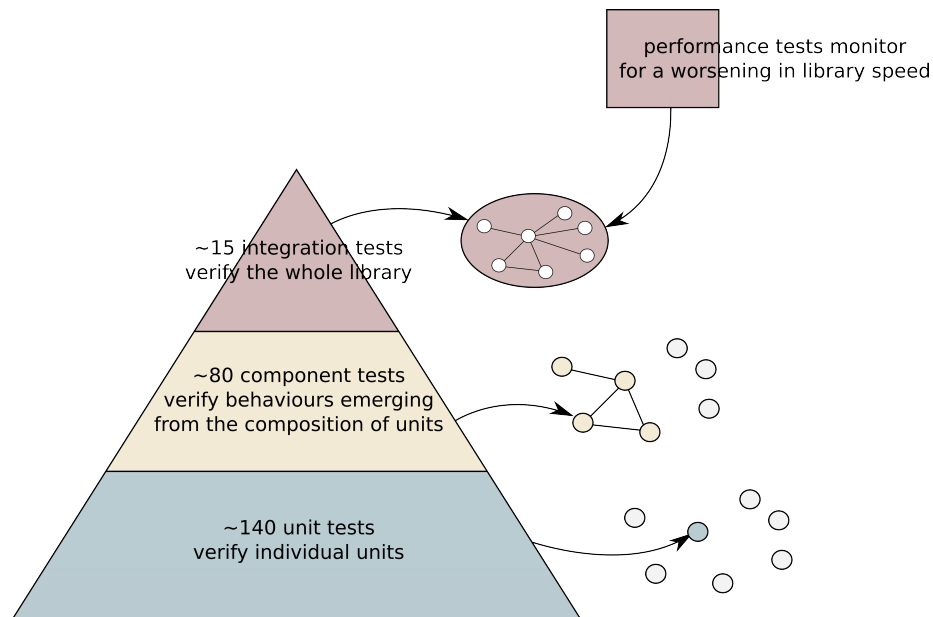
Figure 9: **The test pyramid**. Relying on the assumption that verification of small parts provides a solid base from which to compose system-level behaviours. A Lot of testing is done on the low-level components of the system, less on the component level and less still on a whole-system level where only smoke tests are provided.

The testing itself is a non-trivial undertaking with 80% of code written for this project being test specifications. Based on the idea that a correct system must be built from individually correct units, the majority of the specifications are unit tests, putting each unit under the microscope and describing the correct behaviour as completely as possible. Component tests zoom out from examining individual components to focus on their correct composition, falsifying only the

http traffic. To avoid testing implementation details the component tests do not look at the means of coupling between the code units but rather check for the behaviours which should emerge as a consequence of their composition. At the apex of the test pyramid are a small number of integration tests. These verify Oboe as a black box without any knowledge of, or access to the internals, using only the APIs which are exposed to application programmers. When running the integration tests a REST service is first spun up so that correctness of the whole library may be examined against an actual server.

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

Dual-implementation of same interface for streamingHttp might be considered polymorphism, but a function not a class and both are never loaded at run time.

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

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

5.3   Running the tests

The Grunt task runner was used to automate routine tasks such as executing the tests and building. Unit and component tests run automatically whenever a source file changes. As well as being correct execution, the project is required to not surpass a certain size so the built size is also checked. As a small, tightly focused project the majority of programming is refactoring already working code. Running tests on save provides quick feedback so that mistakes are found as soon as they are made. Agile practitioners emphasise the importance of tests that execute quickly (Martin 2008, T9), the 220 unit and component tests run in less than a second so discovering mistakes is near instant. If the "content of any medium is always another medium" (McLuhan 1964 p8), we might say that the content of programming is the program that is realised by its execution. A person working in arts and crafts sees the thing as they work but a programmer will usually not see the execution simultaneously as they program. Conway observed that an artisan works by transform-in-place "start with the working material in place and you step by step transform it into its final form" whereas software is created through intermediate proxies, and attempts to close this gap by merging programming with the results of programming (Conway 2004 side8-9). When we bring together the medium and the message the cost of small experimentation is very low and I feel that programming becomes more explorative and expressive.

The integration tests are not run on save because they intentionally simulate slow transfers and take some time to run. The integration tests are used as a final check against built code before a branch in git can be merged into the master. Once the code has been packaged for distribution the internals are no longer visible the integration tests which are coded against the public API are the only runnable tests. While these tests don't individually test every component, they are designed to exercise the whole codebase so that a mistake in any component will be visible through them. Grunt executes the build, including starting up the test REST services that give the integration tests something to fetch.
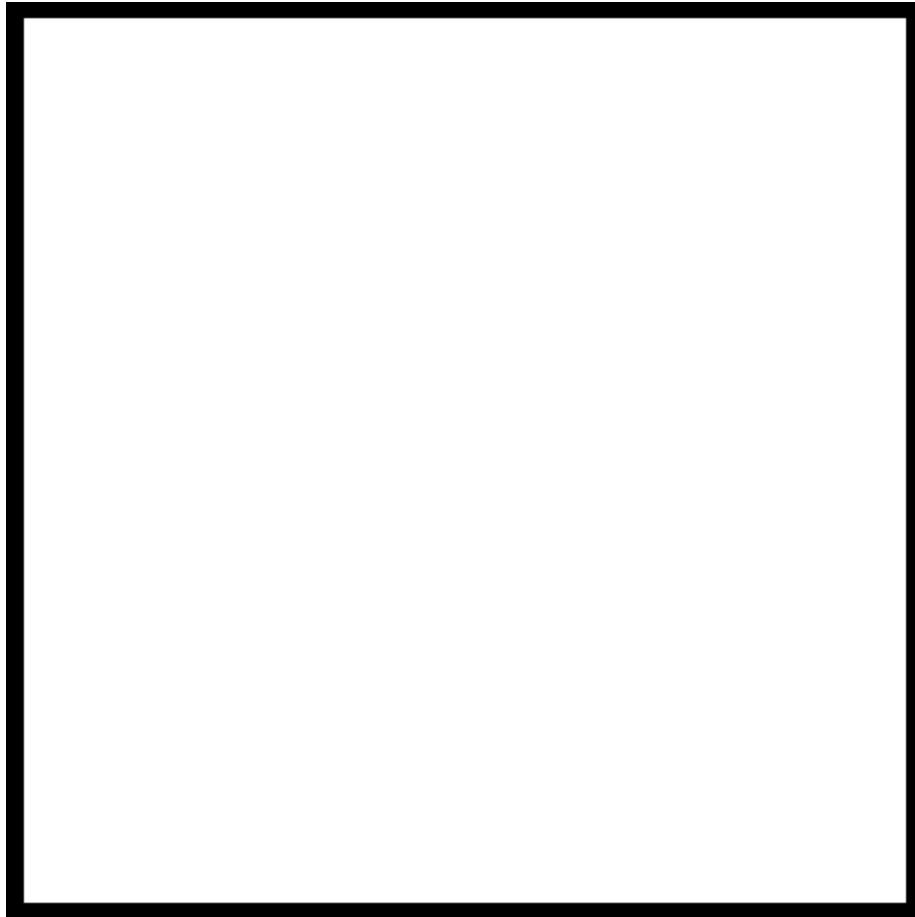
Figure 10: **Relationship between various files and test libraries** *other half of sketch from notebook*

5.4   Packaging as a single, distributable file
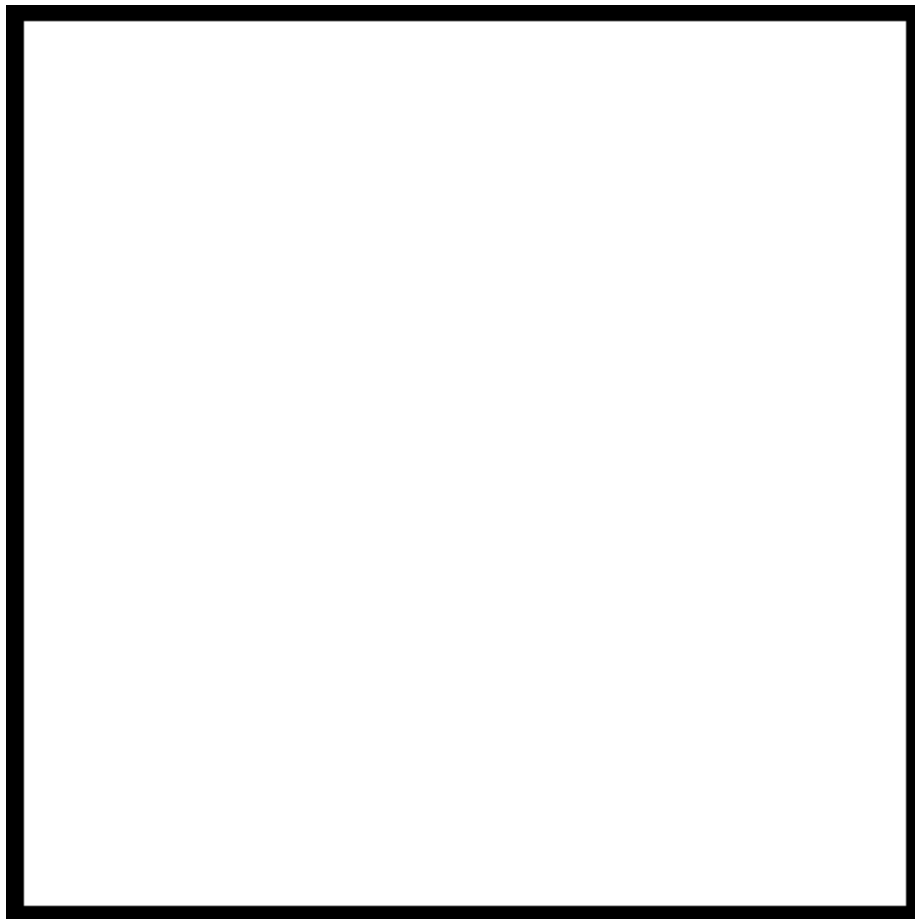
Figure 11: **Packaging of many javascript files into multiple single-file packages.** The packages are individually targeted at different execution contexts, either browsers or node *get from notebook, split sketch diagram in half*

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

Javascript files are interpreted in series by the browser so load-time dependencies must precede dependants. Unsurprisingly, separate files once concatenated following the same order as delivered to the browser will load more quickly but are functionally equivalent, at least barring syntax errors. Several tools exist to automate this stage of the build process, incorporating a topological sort of the dependency digraph in order to find a working concatenation order.

Early in this project I chose *Require.js* although I later moved on because it was too heavyweight. Javascript as a language doesn't have an import statement. Require contributes the importing ability to Javascript from inside the language sandbox as the `require` function, a standard asynchronous call. Calls to `require` AJAX in and execute the imported source, returning any exported symbols by a callback. For non-trivial applications this mode is intended mostly for debugging; because a network hop is involved the protocol is chatty and slowed by highly latent calls between modules. For efficient delivery Require also has the `optimise` command which concatenates into a single file by using static analysis to deduce a workable source order. Because `require` may appear anywhere in the source, this in the general case is of course undecidable so Require falls back to lazy loading. In practice undecidability isn't a problem because imports are generally not subject to branching. In larger webapps lazy loading speeding up the initial page load and is actually an advantage. The technique of *Asynchronous Module Definition* (AMD) intentionally imports rarely-loaded modules in response to events. By resisting the static analysis the units will not be downloaded until they are needed.

AMD is mostly of interest to web applications with a central hub but also some rarely used parts. Oboe does not fit this profile: everybody who uses it will use all of the library. Regardless, I hoped to use `optimise` to generate my combined Javascript file. Even after optimisation, Require's design necessitates that calls to `require` stay in the code and that the require.js run-time component is available to handle these calls. For a micro-library a ???k overhead was too large to accommodate. Overall, Require seems more suited to developing stand-alone applications than programming libraries.

Having abandoned Require, I decided to pick up the simplest tool which could possibly work. With only 15 source files and a fairly sparse dependency graph finding a working order on paper wasn't a daunting task. Combined with a Grunt analogue to the unix `cat` command I quickly had a working build process. I adjusted each Javascript file to, when loaded directly, place its API in the global namespace, then post-concatenation wrapped the combined in a single function, converting the APIs inside the function from global to the scope of that function, thereby hiding the implementation for code outside of Oboe.

For future consideration there is Browserify. This library reverses the 'browser first' image of Javascript by converting applications targeted at Node into a single file efficiently packaged for delivery to a web browser, conceptually making Node the primary environment for Javascript and adapting browser execution to match. Significantly, require leaves no trace of itself in the concatenated

Javascript other than Adaptors presenting browser APIs as the Node equivalents. Browserify's http adaptor[4] is complete but more verbose compared to Oboe's version[5].

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

5.5   Styles of Programming

The implementation of Oboe is mixed paradigm. Events flow throughout the whole library but in terms of code style the components are a mix of procedural, functional and object-oriented programming. Object orientation is used only to wrap the library in an Object-oriented public API and as a tuple-like store for multiple values. Constructors are not used, nor is there any inheritance or notable polymorphism. Closures, not objects, are used as the primary means of data storage and hiding. Many of the entities painted in figure 8 map onto no single, addressable language construct and exist only as a set of event handlers trapped inside the same closure, taking advantage of the fact that their reachability from some event emitter prevents required parameters from being garbage collected. From outside the closure hidden values are not only private as would be seen in an OO model, they are inherently unaddressable. Although only sparingly OO, the high-level design's componentisation hasn't departed from how it might be implemented in an OO metamodel and Object Oriented design patterns remain influential despite being only loosely followed.

Because of the pressures on code size I decided not to use a general purpose functional library and instead create my own with only the parts that I need; see functional.js. Functional programming in Javascript is known to be slower than other styles, particularly under Firefox because it lacks Lambda Lifting and other similar optimisations (Guo 2013). Considering to what degree performance concerns should dissuade us from a functional style, we may consider the library's execution context. Because of the single-threaded model any application's

---

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

[5]Https://github.com/jimhigson/oboe.js/blob/master/src/streamingXhr.js This version is shorter mostly because it is not a generic solution.

Javascript execution is in between frames serving concurrent concerns so to minimise the impact on latency for the other tasks it is important that no task occupies the CPU for very long. On the browser about 16ms is a fair maximum, allowing painting to occur at 60 frames per second. In Node there is no hard limit but any CPU-hogging task degrades the responsiveness of other responses. Context switching imposes a very low overhead and responsive sharing generally proffers many small frames over a few larger ones. In any case, server-side tasks especially are more often i/o bound than CPU bound. Oboe's progressive design naturally splits tasks which would otherwise be performed in a single frame over many. For example, parsing and marshaling. Although the overall computation may be higher, the total performance of the system should be improved.

Javascript is of course an imperative language but over many iterations Oboe has tended towards a declarative style. In incrementalContentBuilder.js programming was initially stateful and procedural, reading like the instructions to perform a task. Over many refactors the flavour of the code has changed, the reading now tending towards a description of desired behaviour.

5.6   Incrementally building up the content

As shown in figure 8, there is an incremental content builder and ascent tracer which handle the output from the Clarinet JSON SAX parser. Taken together, these might be considered a variant of the Adaptor pattern, providing to the controller a simpler interface than is presented by Clarinet. However, this is not the model implementation of the pattern; the adapted interface is even-driven rather than call-driven: we receive six kinds of event and in response emmit from a narrower vocabulary of two.

To evaluate JSONPath expressions the controller requires a path to the current JSON node, the node itself, and any ancestor nodes. This is delivered by the incremental content builder as the payload of the NODE_FOUND and PATH_FOUND events. For each Clarinet event the builder provides a corresponding function which, working from the current path, returns the next path after the event has been applied. For example, the `objectopen` and `arrayopen` events move the current node deeper in the document and are handled by adding new items to the path, whereas for `closeobject` and `closearray` we remove one. Over the course of parsing a complete JSON file the path will in this way be manipulated to visit every node, allowing each to be tested against the registered JSONPath expressions. Internally, the builder's event handlers are declared as the combination of a smaller number of basic reusable handler parts. Oboe is largely unconcerned regarding a JSON node's type so given that several of the Clarinet events differ only by the type of the nodes they announce, Oboe is able to generify their handling by composing from a common pool of handler-parts. Picking up `openobject` and `openarray` events, both fall through to the same 'nodeFound', differing only in a parameter. Similarly, consider the `value` event which is fired when Clarinet encounters a String or Number. Because primitive

nodes are always leaves the builder regards this as a node which instantaneously starts and ends, handled programmatically as the functional composition of the `nodeFound` and `curNodeFinished`. The reuse of smaller instructions to build up larger ones is perhaps slightly reminiscent of CISC CPU design in which micro-instructions are combined to implement the chip's advertised interface.

Although the builder functions are stateless, ultimately the state regarding the current path needs to be stored between clarinet calls. This is handled by the ascent tracker. This tiny component merely serves as a holder for this data, starting from an empty path it passes the path to each builder function and stores the result to be given to the next one.
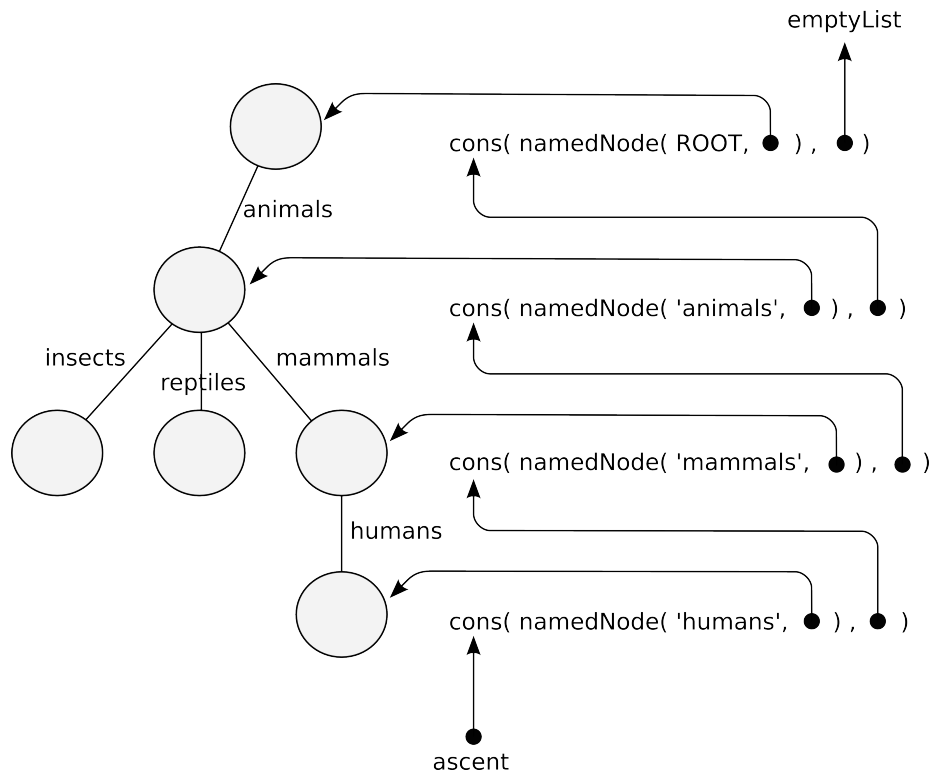
Figure 12: List representation of an ascent from leaf to root of a JSON tree. Note the special ROOT token which represents the path mapping to the root node (of course nothing maps to the root) - this is an object, taking advantage of object identity to ensure that the token is unequal to anything but itself. This list form is built up by the incremental content builder and is the format that compiled JSONPath expressions test against for matches

The path of the current node is maintained as a singly linked list, with each list element holding the field name and the node and the node itself, see figure 12. The list is arranged with the JSON root at the far end and the current node at

the head. As we traverse the JSON the current node is appended and removed many times whereas the root is immutable. This ordering was chosen because it is computationally very efficient since all updates to the list are at the head. Each link in the list is immutable, enforced by newer Javascript engines as frozen objects.[6]

Linked lists were chosen in preference to the more conventional approach of using native Javascript Arrays for several reasons. Firstly, I find this area of the program more easy to test and debug given immutable data structures. Handling native Arrays without mutating would be very expensive because on each new path the array would have to be copied rather than edited in-place. Unpicking a stack trace is easier if I know that every value revealed is the value that has always occupied that space because I don't have to think four-dimensionally projecting my mind forwards and back in time to different values that were there when the variable was used. The lack of side effects means I can try explore new commands in the debugger's CLI without worrying about breaking the execution of the program. Most Javascript virtual machines are also quite poor at array growing and shrinking so for collections whose size changes often are outperformed by linked lists. Finally, this is a very convenient format for the JSONPath engine to perform matching on as will be discussed in the next section. The Javascript file lists.js implements the list functions: `cons`, `head`, `tail`, `map`, `foldR`, `all`.

Because it is more common to quote paths as descents rather than ascent, on the boundary to the outside world Oboe reverses the order and, because Javascript programmers will not be familiar with this structure, converts to arrays.

5.7   Oboe JSONPath Implementation

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

---

[6]See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\T1\textbackslash{}_Objects/Object/freeze. Although older engines don't provide any ability to create immutable objects at run-time, we can be fairly certain that the code does not mutate these objects or the tests would fail when run in environments which are able to enforce this.

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

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

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

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

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

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

The usage profile for JSONPath expressions in Oboe is to be compiled once and then evaluated many times, once for each node encountered while parsing the JSON. Because matching is performed perhaps hundreds of times per file the most pressing performance consideration is for matching to execute quickly, the time required to compile is relatively unimportant. Oboe's JSONPath design contrasts with JSONPath's reference implementation which, because it provides a first order function, freshly reinterprets the JSONPath string each time it is invoked.

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

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

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

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

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

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

By verifying the tokens through their own unit tests it is simpler to thoroughly specify the tokenisation, producing simpler failure messages than if it were done through the full JSONPath engine. We might consider the unit test layer of the pyramid (figure 9) is further split into two sub-layers. Arguably, the upper of these sub-layer is not a unit test because it is verifying two units together. There is some redundancy with the tokens being tested both individually and as full expressions. I maintain that this is the best approach regardless because

---

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

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

stubbing out the tokenizer functions would be a considerable effort and would not improve the rigor of the JSONPath specification.
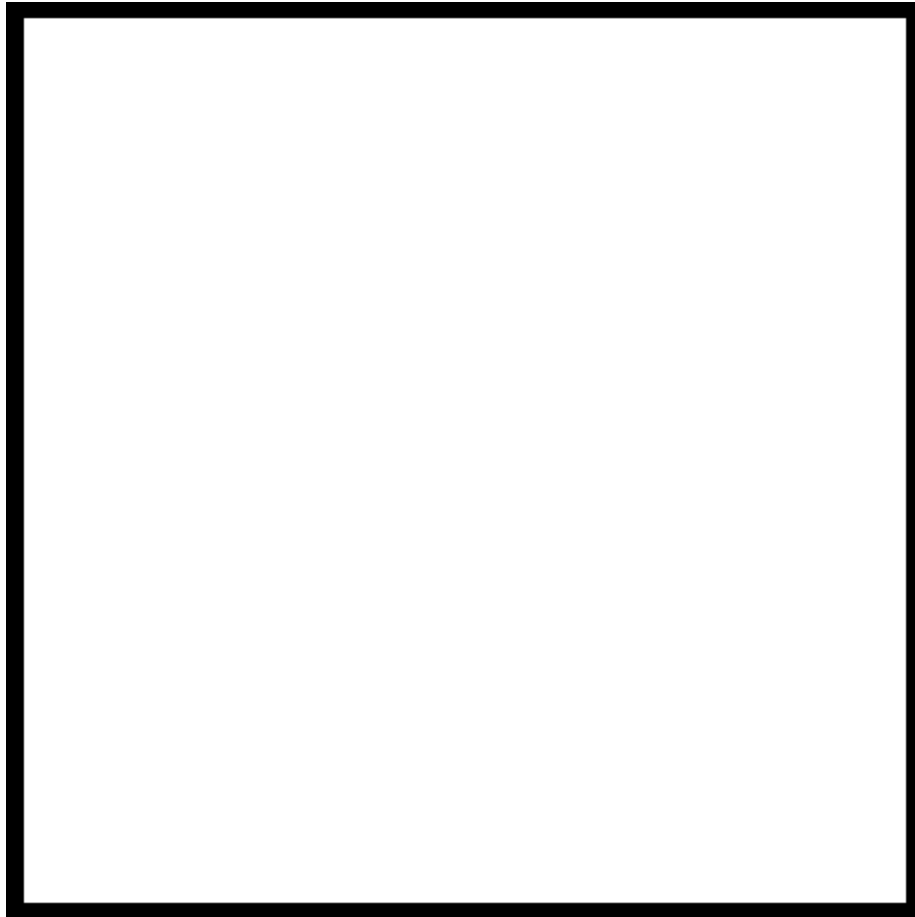
Figure 13: Some kind of diagram showing jsonPath expressions and functions partially completed to link back to the previous function. Include the statementExpr pointing to the last clause

6 Conclusion

6.1 Benchmarking vs non-progressive REST

I feel it is important to experimentally answer the question, *is this actually any faster?*. To do this I have created a small benchmarking suite that runs under Node.js. I chose Node because it at its code is a very basic platform which I feel it gives a more repeatable environment than modern browsers which at during the tests could be performing any number of background tasks. These tests may be seen in the `benchmark` folder of the project. Node also has the advantage in measuring the memory of a running process is not swamped by the memory taken up by the browser itself.

One of the proposed advantages of progressive REST is an improved user experience because of earlier, more progressive interface rendering and a perceptual improvement in speed. I am not focusing on this area for benchmarking because it would be much more difficult to measure, involving human participants. While I can't provide numbers on the perceptual improvements, I have created sites using Oboe and the improvement in responsiveness over slower networks is large enough to be obvious.

The benchmark mimics a relational database-backed REST service. Relational databases serve data to a cursor one tuple at a time. The simulated service writes out twenty tuples as JSON objects, one every ten milliseconds. To simulate network slowness, Apple's *Network Line Conditioner* was used. I chose the named presets "3G, Average Case" and "Cable modem" to represent poor and good networks respectively.[12] Each test involves two node processes, one acting as the client and one as the server, with data transfer between them via normal http.

Memory was measured using Node's built in memory reporting tool, `process.memoryusage()` and the maximum figure returned on each run was taken

Each object in the returned JSON contains a URL to a further resource. Each further resource is fetched and parsed. The aggregation is complete when we have them all.

Strategy	Network	First output (ms)	Total time (ms)	Max. Memory (Mb)
Oboe.js	Good	40	804	6.2
Oboe.js	Poor	60	1,526	6.2
JSON.parse (DOM)	Good	984	1,064	9,0
JSON.parse (DOM)	Poor	2550	2,609	8.9

---

[12]Http://mattgemmell.com/2011/07/25/network-link-conditioner-in-lion/.

Clarinet (SAX)	Good	34	781	5.5
Clarinet (SAX)	Poor	52	1,510	5.5

Vs Json.parse shows a dramatic improvement over first output of about 96% and a smaller but significant improvement of about 40% in time required to complete the task. Oboe's performance in terms of time is about 15% slower than Clarinet; since Oboe is built on Clarinet it could not be faster but I had hoped for these results to be closer.

As expected, in this simulation of real-world usage, the extra computation compared to JSON.parse which is needed by Oboe's more involved algorithms or Clarinet's less efficient parsing[13] have been dwarfed by better i/o management. Reacting earlier using slower handlers has been shown to be faster overall than reacting later with quicker ones. I believe that this vindicates a focus on efficient management of i/o over faster algorithms. I believe that much programming takes a "Hurry up and wait" approach by concentrating overly on optimal computation rather than optimal i/o management.

There is an unexpected improvement vs JSON.parse in terms of memory usage. It is not clear why this would be but it may be attributable to the json fetching library used to simplify the JSON.parse tests having a large dependency tree. As expected, Clarinet shows the largest improvements in terms of memory usage. For very large JSON I would expect Clarinet's memory usage to remain roughly constant whilst the two approaches rise linearly with the size of the resource.

6.2 Comparative Programmer Ergonomics

For each of the benchmarks above the code was laid out in the most natural way for the strategy under test.

Strategy	Code Required (lines)	Code required (chars)
Oboe.js	3	64
JSON.parse	5	102
Clarinet (SAX)	30	lots!

Oboe was the shortest:

```
oboe(DB_URL).node('{id url}.url', function(url){
```

---

[13] Http://writings.nunojob.com/2011/12/clarinet-sax-based-evented-streaming-json-parser-in-javascript-for-the-browser-and-nodejs.html.

```
    oboe(url).node('name', function(name){

        console.log(name);
    });
});
```

Non-progressive parsing was slightly longer, requiring in addition a loop, an if statement, and programmatically selecting specific parts of the results:

```
// JSON.parse. The code is shortened and simplified by get-json from NPM:
// https://npmjs.org/package/get-json

getJson(DB_URL, function(err, records) {

    records.data.forEach( function( record ){

        if( record.url ) {

            getJson(record.url, function(err, record) {

                console.log(record.name);
            });
        }
    });
});
```

The JSON.parse version is very closely coupled with the format that it is handling. We can see this in the fragments `records.data`, `record.url`, `record.name` which expects to find sub-trees at very specific locations in the JSON. The code might be said to contain a description of the format that it is for. Conversely, the Oboe version describes the format only so far as is needed to identify the parts that it is interested in; the remainder of the format could change and the code would continue to work. As well as being simpler to program against than the previous simplest mode, I believe this demonstrates a greater tolerance to changing formats.

The Clarinet version of the code may be seen in appendex (??). This version is greater in verbosity and obfuscation. I don't think a person could look at this source and understand what is being parsed without thinking about it for a long time. Parameter names such as 'key' or 'value' must be chosen by the position of the token in the markup, prior to understanding the semantics it represents. By contrast, Oboe and JSON.parse both allow names to be given by the meaning of the token.

6.3 Performance of code styles under various engines

The 15% overhead of Oboe vs Clarinet suggests Oboe might be computationally expensive. With very fast networks the extra computation might outweigh a more efficient i/o strategy.

The file `test/specs/oboe.performance.spec.js` contains a simple benchmark. This test registeres a very complex JSONPath expression which intentionally uses all of the language and fetches a JSON file containing 100 objects, each with 8 String properties against . Correspondingly the expression is evaluated just over 800 times and 100 matches are found. Although real http is used, it is kept within the localhost. The results below are averaged from ten runs. The tests executed on a Macbook Air, except for Chrome Mobile which was tested on an iPhone 5. Tests requiring Microsoft Windows were performed inside a virtual machine.

Curl is a simple download to stdout from the shell and is included as a control run to provide a baseline.

Platform	Total Time	Throughput (nodes per ms)
Curl (control)	42ms	*n/a*
Node.js v0.10.1	172ms	4.67
Chrome 30.0.1599 (Mac OS X 10.7.5)	202ms	3.98
Safari 6.0.5 (Mac OS X 10.7.5)	231ms	3.48
IE 10.0.0 (Windows 8)	349ms	2.30
Chrome Mobile iOS 30.0.1599 (iOS 7.0.2)	431ms	1.86
Firefox 24.0.0 (Mac OS X 10.7)	547ms	1.47
IE 8.0.0 (Windows XP)	3,048ms	0.26

We can see that Firefox is much slower than other modern browsers despite its SpiderMonkey Javascript engine being normally quite fast. This is probably explicable in part by SpiderMonkey's just-in-time compiler being poor at optimising functional Javascript (Guo 2013). Because the JSON nodes are not of a common type the related callsites are not monomorphic which Firefox also optimises poorly (Guo 2013). When the test was repeated using a simpler JSONPath expression Firefox showed by far the largest improvement indicating that on this platform the functional pattern matching is the bottleneck.

Of these results I find only the very low performance on old versions of Internet Explorer concerning, almost certainly degrading user experience more than it is improved. It might be reasonable to conclude that for complex use cases

Oboe is currently not unsuited to legacy platforms. Since this platform cannot progressively interpret an XHR response, if performance on legacy platforms becomes a serious concern one option might be to create a non-progressive library with the same API which could be selectively delivered to those platforms in place of the main version.

Nonetheless, in its current form Oboe may slow down the total time when working over the very fastest connections.

For an imperative language coded in a functional style the compiler may not optimise as effectively as if a functional language was used. This is especially the case under a highly dynamic language in which everything, even the built-in constructs are mutable. I think Javascript was a good choice of language given it is already well adopted and allows the targeting of server and client side with only minimal effort, giving a very large number of applications with the potential to adopt Oboe. However, there are obvious inefficiencies such as the the descent and ancestor arrays which are always created to be handed to application callbacks but that I anticipate will be predominantly ignored. The design of Oboe is very amicable to implementation under a functional language and it would be interesting to see the results.

6.4   Status as a micro-library

Built versions of Oboe as delivered reside in the project's `dist` folder. The file `oboe-browser.min.js` is the minified version which should be sent to browsers gzipped. After gzip is applied this file comes to 4966 bytes; close to but comfortably under the 5120 limit. At roughly the size as a very small image, the size of Oboe should not discourage adoption.

6.5   potential future work

There is nothing about Oboe which precludes working with other tree-shaped format. If there is demand, An XML/XPATH version seems like an obvious expansion. Currently Oboe only operates on http traffic. While this restriction is reasonable in a Browser context, under Node it is unnecessarily limiting and should be lifted by allowing arbitrary streams to be read.

Oboe stores all items that are parsed from the JSON it receives, resulting in a memory use which is as high as a DOM parser. These are kept in order to be able to provide a match to any possible JSONPath expression. However, in most cases memory would be saved if the parsed content were only stored so far as is needed to provide matches against the JSONPath expressions which have actually been registered. For typical use cases I expect this would allow the non-storage of large branches. Likewise, the current implementation takes a rather brute force approach when examining node for pattern matches: check every registered JSONPath expression against every node and path that are

found in the JSON. For many expressions we are able to know there is no possibility of matching a JSON tree, either because we have already matched or because the the current node's ancestors already mandate failure. A more sophisticated programme might disregard provably unsatisfiable handlers for the duration of a subtree. Either of these changes would involve some rather difficult programming and because matching is fast enough I think brute force is the best approach for the time being.

During JSONPath matching much of the computation is repeated. For example, matching the expression `b.*` against many children of a common parent will repeat the same test, checking if the parent's name is 'b', for each child node. Because the JSONPath matching is stateless, recursive and side-effect free there is a potential to cut out repeated computation by using a functional cache. This would reduce the overall amount of computation needed for JSONPath expressions with common substrings to their left side or nodes with a common ancestry. Current Javascript implementations make it difficult to manage a functional cache, or caches in general, from inside the language itself because there is no way to occupy only the unused memory. Weak references are proposed in ECMAScript 6 but currently only experimentally supported[14]. For future development they would be ideal.

The nodes which Oboe hands to callbacks are mutable meaning that potentially the correct workings of the library could be broken if the containing application carelessly alters them. Newer implementations of Javascript allows a whole object to be made immutable, or just certain properties via an immutability decorator and the `defineProperty` method. This would probably be an improvement.

---

[14]At time of writing, Firefox is the only engine supporting WeakHashMap by default. In Chome it is implemented but not available to Javascript unless explicitly enabled by a browser flag. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\T1\textbackslash{}_Objects/WeakMap retrieved 11th October 2013.

7 Appendix i: Limits to number of simultaneous connections under various http clients

http Client	connection limit per server
Firefox	6
Internet Explorer	4
Chrome / Chromium	32 sockets per proxy 6 sockets per destination host 256 sockets per process

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

http://msdn.microsoft.com/de-de/magazine/ee330731.aspx#http11_max_con

http://dev.chromium.org/developers/design-documents/network-stack#TOC-Connection-Management

8 Appendix ii: Oboe.js source code listing

8.1 clarinetListenerAdaptor.js

```javascript
/**
 * A bridge used to assign stateless functions to listen to clarinet.
 *
 * As well as the parameter from clarinet, each callback will also be passed
 * the result of the last callback.
 *
 * This may also be used to clear all listeners by assigning zero handlers:
 *
 *    clarinetListenerAdaptor( clarinet, {} )
 */
function clarinetListenerAdaptor(clarinetParser, handlers){

   var state;

   clarinet.EVENTS.forEach(function(eventName){

      var handlerFunction = handlers[eventName];

      clarinetParser['on'+eventName] = handlerFunction &&
                                   function(param) {
                                       state = handlerFunction( state, param);
                                   };
   });
}
```

8.2 events.js

```javascript
/**
 * This file declares some constants to use as names for event types.
 */

var // NODE_FOUND, PATH_FOUND and ERROR_EVENT feature
    // in the public API via .on('node', ...) or .on('path', ...)
    // so these events are strings
    NODE_FOUND   = 'node',
    PATH_FOUND   = 'path',

    // these events are never exported so are kept as
    // the smallest possible representation, numbers:
    _S = 0,
    ERROR_EVENT   = _S++,
    ROOT_FOUND    = _S++,
    NEW_CONTENT = _S++,
    END_OF_CONTENT = _S++,
    ABORTING = _S++;
```

8.3   functional.js

```javascript
/**
 * Partially complete a function.
 *
 * Eg:
 *     var add3 = partialComplete( function add(a,b){return a+b}, [3] );
 *
 *     add3(4) // gives 7
 */
var partialComplete = varArgs(function( fn, boundArgs ) {

    return varArgs(function( callArgs ) {

        return fn.apply(this, boundArgs.concat(callArgs));
    });
}),


/**
 * Compose zero or more functions:
 *
 *     compose(f1, f2, f3)(x) = f1(f2(f3(x))))
 *
 * The last (inner-most) function may take more than one parameter:
 *
 *     compose(f1, f2, f3)(x,y) = f1(f2(f3(x,y))))
 */
  compose = varArgs(function(fns) {

    var fnsList = arrayAsList(fns);

    function next(params, curFn) {
       return [apply(params, curFn)];
    }

    return varArgs(function(startParams){

        return foldR(next, startParams, fnsList)[0];
    });
}),

/**
 * Call a list of functions with the same args until one returns a
 * truthy result. Similar to the || operator.
 *
```

```
 * So:
 *      lazyUnion([f1,f2,f3 ... fn])( p1, p2 ... pn )
 *
 * Is equivalent to:
 *      apply([p1, p2 ... pn], f1) ||
 *      apply([p1, p2 ... pn], f2) ||
 *      apply([p1, p2 ... pn], f3) ... apply(fn, [p1, p2 ... pn])
 *
 * @returns the first return value that is given that is truthy.
 */
   lazyUnion = varArgs(function(fns) {

      return varArgs(function(params){

         var maybeValue;

         for (var i = 0; i < len(fns); i++) {

            maybeValue = apply(params, fns[i]);

            if( maybeValue ) {
               return maybeValue;
            }
         }
      });
   });

/**
 * This file declares various pieces of functional programming.
 *
 * This isn't a general purpose functional library, to keep things small it
 * has just the parts useful for Oboe.js.
 */


/**
 * Call a single function with the given arguments array.
 * Basically, a functional-style version of the OO-style Function#apply for
 * when we don't care about the context ('this') of the call.
 *
 * The order of arguments allows partial completion of the arguments array
 */
function apply(args, fn) {
   return fn.apply(undefined, args);
}
```

```
/**
 * Define variable argument functions but cut out all that tedious messing about
 * with the arguments object. Delivers the variable-length part of the arguments
 * list as an array.
 *
 * Eg:
 *
 * var myFunction = varArgs(
 *     function( fixedArgument, otherFixedArgument, variableNumberOfArguments ){
 *         console.log( variableNumberOfArguments );
 *     }
 * )
 *
 * myFunction('a', 'b', 1, 2, 3); // logs [1,2,3]
 *
 * var myOtherFunction = varArgs(function( variableNumberOfArguments ){
 *     console.log( variableNumberOfArguments );
 * })
 *
 * myFunction(1, 2, 3); // logs [1,2,3]
 *
 */
function varArgs(fn){

    var numberOfFixedArguments = fn.length -1;

    return function(){

        var numberOfVaraibleArguments = arguments.length - numberOfFixedArguments,

            argumentsToFunction = Array.prototype.slice.call(arguments);

        // remove the last n element from the array and append it onto the end of
        // itself as a sub-array
        argumentsToFunction.push(
            argumentsToFunction.splice(numberOfFixedArguments, numberOfVaraibleArguments)
        );

        return fn.apply( this, argumentsToFunction );
    }
}


/**
 * Swap the order of parameters to a binary function
 *
```

```
 * A bit like this flip: http://zvon.org/other/haskell/Outputprelude/flip_f.html
 */
function flip(fn){
   return function(a, b){
      return fn(b,a);
   }
}



/**
 * Create a function which is the intersection of two other functions.
 *
 * Like the && operator, if the first is truthy, the second is never called,
 * otherwise the return value from the second is returned.
 */
function lazyIntersection(fn1, fn2) {

   return function (param) {

      return fn1(param) && fn2(param);
   };
}
```

8.4   incrementalContentBuilder.js

```
/**
 * This file provides various listeners which can be used to build up
 * a changing ascent based on the callbacks provided by Clarinet. It listens
 * to the low-level events from Clarinet and fires higher-level ones.
 *
 * The building up is stateless so to track a JSON file
 * clarinetListenerAdaptor.js is required to store the ascent state
 * between calls.
 */


var keyOf = attr('key');
var nodeOf = attr('node');


/**
 * A special value to use in the path list to represent the path 'to' a root
 * object (which doesn't really have any path). This prevents the need for
 * special-casing detection of the root object and allows it to be treated
 * like any other object. We might think of this as being similar to the
 * 'unnamed root' domain ".", eg if I go to
 * http://en.wikipedia.org./wiki/En/Main_page the dot after 'org' deliminates
 * the unnamed root of the DNS.
 *
 * This is kept as an object to take advantage that in Javascript's OO objects
 * are guaranteed to be distinct, therefore no other object can possibly clash
 * with this one. Strings, numbers etc provide no such guarantee.
 **/
var ROOT_PATH = {};


/**
 * Create a new set of handlers for clarinet's events, bound to the fire
 * function given.
 */
function incrementalContentBuilder( fire) {


   function arrayIndicesAreKeys( possiblyInconsistentAscent, newDeepestNode) {

      /* for values in arrays we aren't pre-warned of the coming paths
         (Clarinet gives no call to onkey like it does for values in objects)
         so if we are in an array we need to create this path ourselves. The
         key will be len(parentNode) because array keys are always sequential
```

```
        numbers. */

   var parentNode = nodeOf( head( possiblyInconsistentAscent));

   return      isOfType( Array, parentNode)
         ?
            pathFound(  possiblyInconsistentAscent,
                        len(parentNode),
                        newDeepestNode
            )
         :
            // nothing needed, return unchanged
            possiblyInconsistentAscent
         ;
}

function nodeFound( ascent, newDeepestNode ) {

   if( !ascent ) {
      // we discovered the root node,
      fire( ROOT_FOUND, newDeepestNode);

      return pathFound( ascent, ROOT_PATH, newDeepestNode);
   }

   // we discovered a non-root node

   var arrayConsistentAscent  = arrayIndicesAreKeys( ascent, newDeepestNode),
       ancestorBranches        = tail( arrayConsistentAscent),
       previouslyUnmappedName = keyOf( head( arrayConsistentAscent));

   appendBuiltContent(
      ancestorBranches,
      previouslyUnmappedName,
      newDeepestNode
   );

   return cons(
            namedNode( previouslyUnmappedName, newDeepestNode ),
            ancestorBranches
   );
}


/**
 * Add a new value to the object we are building up to represent the
```

75

```javascript
  * parsed JSON
 */
function appendBuiltContent( ancestorBranches, key, node ){

   nodeOf( head( ancestorBranches))[key] = node;
}


/**
 * Get a new key->node mapping
 *
 * @param {String|Number} key
 * @param {Object|Array|String|Number|null} node a value found in the json
 */
function namedNode(key, node) {
   return {key:key, node:node};
}


/**
 * For when we find a new key in the json.
 *
 * @param {String|Number|Object} newDeepestName the key. If we are in an
 *    array will be a number, otherwise a string. May take the special
 *    value ROOT_PATH if the root node has just been found
 *
 * @param {String|Number|Object|Array|Null|undefined} [maybeNewDeepestNode]
 *    usually this won't be known so can be undefined. Can't use null
 *    to represent unknown because null is a valid value in JSON
 **/
function pathFound(ascent, newDeepestName, maybeNewDeepestNode) {

   if( ascent ) { // if not root

      // If we have the key but (unless adding to an array) no known value
      // yet. Put that key in the output but against no defined value:
      appendBuiltContent( ascent, newDeepestName, maybeNewDeepestNode );
   }

   var ascentWithNewPath = cons(
                              namedNode( newDeepestName,
                                         maybeNewDeepestNode),
                              ascent
                           );

   fire( PATH_FOUND, ascentWithNewPath);

   return ascentWithNewPath;
```

```
}


/**
 * For when the current node ends
 */
function curNodeFinished( ascent ) {

   fire( NODE_FOUND, ascent);

   // pop the complete node and its path off the list:
   return tail( ascent);
}

return {

   openobject : function (ascent, firstKey) {

      var ascentAfterNodeFound = nodeFound(ascent, {});

      /* It is a perculiarity of Clarinet that for non-empty objects it
         gives the first key with the openobject event instead of
         in a subsequent key event.

         firstKey could be the empty string in a JSON object like
         {'':'foo'} which is technically valid.

         So can't check with !firstKey, have to see if has any
         defined value. */
      return defined(firstKey)
      ?
         /* We know the first key of the newly parsed object. Notify that
            path has been found but don't put firstKey permanently onto
            pathList yet because we haven't identified what is at that key
            yet. Give null as the value because we haven't seen that far
            into the json yet */
         pathFound(ascentAfterNodeFound, firstKey)
      :
         ascentAfterNodeFound
      ;
   },

   openarray: function (ascent) {
      return nodeFound(ascent, []);
   },
```

```javascript
        // called by Clarinet when keys are found in objects
        key: pathFound,

        /* Emitted by Clarinet when primitive values are found, ie Strings,
           Numbers, and null.
           Because these are always leaves in the JSON, we find and finish the
           node in one step, expressed as functional composition: */
        value: compose( curNodeFinished, nodeFound ),

        // we make no distinction in how we handle object and arrays closing.
        // For both, interpret as the end of the current node.
        closeobject: curNodeFinished,
        closearray: curNodeFinished
    };
}
```

8.5 instanceController.js

```
/**
 * This file implements a light-touch central controller for an instance
 * of Oboe which provides the methods used for interacting with the instance
 * from the calling app.
 */


function instanceController(fire, on, clarinetParser, contentBuilderHandlers) {

    var oboeApi, rootNode;

    // when the root node is found grap a reference to it for later
    on(ROOT_FOUND, function(root) {
        rootNode = root;
    });

    on(NEW_CONTENT,
        function (nextDrip) {
            // callback for when a bit more data arrives from the streaming XHR

            try {

                clarinetParser.write(nextDrip);
            } catch(e) {
                /* we don't have to do anything here because we always assign
                    a .onerror to clarinet which will have already been called
                    by the time this exception is thrown. */
            }
        }
    );

    /* At the end of the http content close the clarinet parser.
       This will provide an error if the total content provided was not
       valid json, ie if not all arrays, objects and Strings closed properly */
    on(END_OF_CONTENT, clarinetParser.close.bind(clarinetParser));


    /* If we abort this Oboe's request stop listening to the clarinet parser.
       This prevents more tokens being found after we abort in the case where
       we aborted during processing of an already filled buffer. */
    on( ABORTING, function() {
        clarinetListenerAdaptor(clarinetParser, {});
    });
```

```
clarinetListenerAdaptor(clarinetParser, contentBuilderHandlers);

// react to errors by putting them on the event bus
clarinetParser.onerror = function(e) {
   fire(ERROR_EVENT, e);

   // note: don't close clarinet here because if it was not expecting
   // end of the json it will throw an error
};

function addPathOrNodeCallback( eventId, pattern, callback ) {

   var matchesJsonPath = jsonPathCompiler( pattern );

   // Add a new callback adaptor to the eventBus.
   // This listener first checks that he pattern matches then if it does,
   // passes it onto the callback.
   on( eventId, function( ascent ){

      var maybeMatchingMapping = matchesJsonPath( ascent );

      /* Possible values for maybeMatchingMapping are now:

         false:
            we did not match

         an object/array/string/number/null:
            we matched and have the node that matched.
            Because nulls are valid json values this can be null.

         undefined:
            we matched but don't have the matching node yet.
            ie, we know there is an upcoming node that matches but we
            can't say anything else about it.
      */
      if( maybeMatchingMapping !== false ) {

         notifyCallback(callback, maybeMatchingMapping, ascent);
      }
   });
}

function notifyCallback(callback, matchingMapping, ascent) {
   /*
      We're now calling back to outside of oboe where the Lisp-style
      lists that we are using internally will not be recognised
```

```
        so convert to standard arrays.

        Also, reverse the order because it is more common to list paths
        "root to leaf" than "leaf to root"
     */

     var descent      = reverseList(ascent),

        // To make a path, strip off the last item which is the special
        // ROOT_PATH token for the 'path' to the root node
        path         = listAsArray(tail(map(keyOf,descent))),
        ancestors  = listAsArray(map(nodeOf, descent));

     try{

        callback( nodeOf(matchingMapping), path, ancestors );
     }catch(e)  {

        // An error occured during the callback, publish it on the event bus
        fire(ERROR_EVENT, e);
     }
}

/**
 * Add several listeners at a time, from a map
 */
function addListenersMap(eventId, listenerMap) {

   for( var pattern in listenerMap ) {
      addPathOrNodeCallback(eventId, pattern, listenerMap[pattern]);
   }
}

/**
 * implementation behind .onPath() and .onNode()
 */
function addNodeOrPathListenerApi( eventId, jsonPathOrListenerMap,
                                   callback, callbackContext ){

   if( isString(jsonPathOrListenerMap) ) {
      addPathOrNodeCallback(
         eventId,
         jsonPathOrListenerMap,
         callback.bind(callbackContext||oboeApi)
      );
   } else {
```

```
            addListenersMap(eventId, jsonPathOrListenerMap);
         }

         return this; // chaining
      }

      /**
       * Construct and return the public API of the Oboe instance to be
       * returned to the calling application
       */
      return oboeApi = {
         path  :   partialComplete(addNodeOrPathListenerApi, PATH_FOUND),
         node  :   partialComplete(addNodeOrPathListenerApi, NODE_FOUND),
         on    :   addNodeOrPathListenerApi,
         fail  :   partialComplete(on, ERROR_EVENT),
         done  :   partialComplete(addNodeOrPathListenerApi, NODE_FOUND, '!'),
         abort :   partialComplete(fire, ABORTING),
         root  :   function rootNodeFunctor() {
                       return rootNode;
                   }
      };
   }
```

8.6 jsonPath.js

```
/**
 * The jsonPath evaluator compiler used for Oboe.js.
 *
 * One function is exposed. This function takes a String JSONPath spec and
 * returns a function to test candidate ascents for matches.
 *
 *  String jsonPath -> (List ascent) -> Boolean/Object
 *
 * This file is coded in a pure functional style. That is, no function has
 * side effects, every function evaluates to the same value for the same
 * arguments and no variables are reassigned.
 */
// the call to jsonPathSyntax injects the token syntaxes that are needed
// inside the compiler
var jsonPathCompiler = jsonPathSyntax(function (pathNodeSyntax,
                                                doubleDotSyntax,
                                                dotSyntax,
                                                bangSyntax,
                                                emptySyntax ) {

   var CAPTURING_INDEX = 1;
   var NAME_INDEX = 2;
   var FIELD_LIST_INDEX = 3;

   var headKey = compose(keyOf, head);

   /**
    * Create an evaluator function for a named path node, expressed in the
    * JSONPath like:
    *    foo
    *    ["bar"]
    *    [2]
    */
   function nameClause(previousExpr, detection ) {

      var name = detection[NAME_INDEX],

          matchesName = ( !name || name == '*' )
                           ?  always
                           :  function(ascent){return headKey(ascent) == name};


      return lazyIntersection(matchesName, previousExpr);
   }
```

```
/**
 * Create an evaluator function for a a duck-typed node, expressed like:
 *
 *    {spin, taste, colour}
 *    .particle{spin, taste, colour}
 *    *{spin, taste, colour}
 */
function duckTypeClause(previousExpr, detection) {

   var fieldListStr = detection[FIELD_LIST_INDEX];

   if (!fieldListStr)
      return previousExpr; // don't wrap at all, return given expr as-is

   var hasAllrequiredFields = partialComplete(
                                 hasAllProperties,
                                 arrayAsList(fieldListStr.split(/\W+/))
                              ),

      isMatch =  compose(
                    hasAllrequiredFields,
                    nodeOf,
                    head
                 );

   return lazyIntersection(isMatch, previousExpr);
}

/**
 * Expression for $, returns the evaluator function
 */
function capture( previousExpr, detection ) {

   // extract meaning from the detection
   var capturing = !!detection[CAPTURING_INDEX];

   if (!capturing)
      return previousExpr; // don't wrap at all, return given expr as-is

   return lazyIntersection(previousExpr, head);

}

/**
 * Create an evaluator function that moves onto the next item on the
```

```
 * lists. This function is the place where the logic to move up a
 * level in the ascent exists.
 *
 * Eg, for JSONPath ".foo" we need skip1(nameClause(always, [,'foo']))
 */
function skip1(previousExpr) {


   if( previousExpr == always ) {
      /* If there is no previous expression this consume command
         is at the start of the jsonPath.
         Since JSONPath specifies what we'd like to find but not
         necessarily everything leading down to it, when running
         out of JSONPath to check against we default to true */
      return always;
   }

   /** return true if the ascent we have contains only the JSON root,
    *  false otherwise
    */
   function notAtRoot(ascent){
      return headKey(ascent) != ROOT_PATH;
   }

   return lazyIntersection(
            /* If we're already at the root but there are more
               expressions to satisfy, can't consume any more. No match.

               This check is why none of the other exprs have to be able
               to handle empty lists; skip1 is the only evaluator that
               moves onto the next token and it refuses to do so once it
               reaches the last item in the list. */
            notAtRoot,

            /* We are not at the root of the ascent yet.
               Move to the next level of the ascent by handing only
               the tail to the previous expression */
            compose(previousExpr, tail)
   );

}

/**
 * Create an evaluator function for the .. (double dot) token. Consumes
 * zero or more levels of the ascent, the fewest that are required to find
 * a match when given to previousExpr.
```

```
 */
function skipMany(previousExpr) {

   if( previousExpr == always ) {
      /* If there is no previous expression this consume command
         is at the start of the jsonPath.
         Since JSONPath specifies what we'd like to find but not
         necessarily everything leading down to it, when running
         out of JSONPath to check against we default to true */
      return always;
   }

   var
      // In JSONPath .. is equivalent to !.. so if .. reaches the root
      // the match has succeeded. Ie, we might write ..foo or !..foo
      // and both should match identically.
      terminalCaseWhenArrivingAtRoot = rootExpr(),
      terminalCaseWhenPreviousExpressionIsSatisfied = previousExpr,
      recursiveCase = skip1(skipManyInner),

      cases = lazyUnion(
                  terminalCaseWhenArrivingAtRoot
               ,  terminalCaseWhenPreviousExpressionIsSatisfied
               ,  recursiveCase
               );

   function skipManyInner(ascent) {

      if( !ascent ) {
         // have gone past the start, not a match:
         return false;
      }

      return cases(ascent);
   }

   return skipManyInner;
}

/**
 * Generate an evaluator for ! - matches only the root element of the json
 * and ignores any previous expressions since nothing may precede !.
 */
function rootExpr() {

   return function(ascent){
```

86

```
        return headKey(ascent) == ROOT_PATH;
    };
}

/**
 * Generate a statement wrapper to sit around the outermost
 * clause evaluator.
 *
 * Handles the case where the capturing is implicit because the JSONPath
 * did not contain a '$' by returning the last node.
 */
function statementExpr(lastClause) {

    return function(ascent) {

        // kick off the evaluation by passing through to the last clause
        var exprMatch = lastClause(ascent);

        return exprMatch === true ? head(ascent) : exprMatch;
    };
}

/**
 * For when a token has been found in the JSONPath input.
 * Compiles the parser for that token and returns in combination with the
 * parser already generated.
 *
 * @param {Function} exprs   a list of the clause evaluator generators for
 *                           the token that was found
 * @param {Function} parserGeneratedSoFar the parser already found
 * @param {Array} detection the match given by the regex engine when
 *                           the feature was found
 */
function expressionsReader( exprs, parserGeneratedSoFar, detection ) {

    // if exprs is zero-length foldR will pass back the
    // parserGeneratedSoFar as-is so we don't need to treat
    // this as a special case

    return   foldR(
                function( parserGeneratedSoFar, expr ){

                    return expr(parserGeneratedSoFar, detection);
                },
                parserGeneratedSoFar,
                exprs
```

```
                    );

        }

        /**
         *  If jsonPath matches the given detector function, creates a function which
         *  evaluates against every clause in the clauseEvaluatorGenerators. The
         *  created function is propagated to the onSuccess function, along with
         *  the remaining unparsed JSONPath substring.
         *
         *  The intended use is to create a clauseMatcher by filling in
         *  the first two arguments, thus providing a function that knows
         *  some syntax to match and what kind of generator to create if it
         *  finds it. The parameter list once completed is:
         *
         *    (jsonPath, parserGeneratedSoFar, onSuccess)
         *
         *  onSuccess may be compileJsonPathToFunction, to recursively continue
         *  parsing after finding a match or returnFoundParser to stop here.
         */
        function generateClauseReaderIfTokenFound (

                            tokenDetector, clauseEvaluatorGenerators,

                            jsonPath, parserGeneratedSoFar, onSuccess) {

           var detected = tokenDetector(jsonPath);

           if(detected) {
              var compiledParser = expressionsReader(
                                clauseEvaluatorGenerators,
                                parserGeneratedSoFar,
                                detected
                          ),

                  remainingUnparsedJsonPath = jsonPath.substr(len(detected[0]));

              return onSuccess(remainingUnparsedJsonPath, compiledParser);
           }
        }

        /**
         * Partially completes generateClauseReaderIfTokenFound above.
         */
        function clauseMatcher(tokenDetector, exprs) {
```

88

```
    return   partialComplete(
                generateClauseReaderIfTokenFound,
                tokenDetector,
                exprs
            );
}


/**
 * clauseForJsonPath is a function which attempts to match against
 * several clause matchers in order until one matches. If non match the
 * jsonPath expression is invalid and an error is thrown.
 *
 * The parameter list is the same as a single clauseMatcher:
 *
 *    (jsonPath, parserGeneratedSoFar, onSuccess)
 */
var clauseForJsonPath = lazyUnion(

   clauseMatcher(pathNodeSyntax    , list( capture,
                                           duckTypeClause,
                                           nameClause,
                                           skip1 ))

,  clauseMatcher(doubleDotSyntax  , list( skipMany))

   // dot is a separator only (like whitespace in other languages) but
   // rather than make it a special case, use an empty list of
   // expressions when this token is found
,  clauseMatcher(dotSyntax         , list() )

,  clauseMatcher(bangSyntax        , list( capture,
                                           rootExpr))

,  clauseMatcher(emptySyntax       , list( statementExpr))

,  function (jsonPath) {
      throw Error('"' + jsonPath + '" could not be tokenised')
   }
);


/**
 * One of two possible values for the onSuccess argument of
 * generateClauseReaderIfTokenFound.
 *
 * When this function is used, generateClauseReaderIfTokenFound simply
```

```
 * returns the compiledParser that it made, regardless of if there is
 * any remaining jsonPath to be compiled.
 */
function returnFoundParser(_remainingJsonPath, compiledParser){
   return compiledParser
}

/**
 * Recursively compile a JSONPath expression.
 *
 * This function serves as one of two possible values for the onSuccess
 * argument of generateClauseReaderIfTokenFound, meaning continue to
 * recursively compile. Otherwise, returnFoundParser is given and
 * compilation terminates.
 */
function compileJsonPathToFunction( uncompiledJsonPath,
                                    parserGeneratedSoFar ) {

   /**
    * On finding a match, if there is remaining text to be compiled
    * we want to either continue parsing using a recursive call to
    * compileJsonPathToFunction. Otherwise, we want to stop and return
    * the parser that we have found so far.
    */
   var onFind =      uncompiledJsonPath
                ?  compileJsonPathToFunction
                :  returnFoundParser;

   return    clauseForJsonPath(
             uncompiledJsonPath,
             parserGeneratedSoFar,
             onFind
          );
}

/**
 * This is the function that we expose to the rest of the library.
 */
return function(jsonPath){

   try {
      // Kick off the recursive parsing of the jsonPath
      return compileJsonPathToFunction(jsonPath, always);

   } catch( e ) {
      throw Error( 'Could not compile "' + jsonPath +
```

```
                      '" because ' + e.message
            );
        }
    }

});
```

8.7 jsonPathSyntax.js

```
var jsonPathSyntax = (function() {

   var

   /**
    * Export a regular expression as a simple function by exposing just
    * the Regex#exec. This allows regex tests to be used under the same
    * interface as differently implemented tests, or for a user of the
    * tests to not concern themselves with their implementation as regular
    * expressions.
    *
    * This could also be expressed point-free as:
    *    Function.prototype.bind.bind(RegExp.prototype.exec),
    *
    * But that's far too confusing! (and not even smaller once minified
    * and gzipped)
    */
      regexDescriptor = function regexDescriptor(regex) {
            return regex.exec.bind(regex);
      }

   /**
    * Join several regular expressions and express as a function.
    * This allows the token patterns to reuse component regular expressions
    * instead of being expressed in full using huge and confusing regular
    * expressions.
    */
   ,   jsonPathClause = varArgs(function( componentRegexes ) {

            // The regular expressions all start with ^ because we
            // only want to find matches at the start of the
            // JSONPath fragment we are inspecting
            componentRegexes.unshift(/^/);

            return    regexDescriptor(
                        RegExp(
                            componentRegexes.map(attr('source')).join('')
                        )
                      );
      })

   ,   possiblyCapturing =               /(\$?)/
   ,   namedNode =                       /([\w-_]+|\*)/
   ,   namePlaceholder =                 /()/
```

```
,    nodeInArrayNotation =          /\["([^"]+)"\]/
,    numberedNodeInArrayNotation = /\[(\d+|\*)\]/
,    fieldList =                   /{([\w ]*?)}/
,    optionalFieldList =           /(?:{([\w ]*?)})?/


     //   foo or *
,    jsonPathNamedNodeInObjectNotation  = jsonPathClause(
                                             possiblyCapturing,
                                             namedNode,
                                             optionalFieldList
                                           )


     //   ["foo"]
,    jsonPathNamedNodeInArrayNotation   = jsonPathClause(
                                             possiblyCapturing,
                                             nodeInArrayNotation,
                                             optionalFieldList
                                           )


     //   [2] or [*]
,    jsonPathNumberedNodeInArrayNotation = jsonPathClause(
                                             possiblyCapturing,
                                             numberedNodeInArrayNotation,
                                             optionalFieldList
                                           )


     //   {a b c}
,    jsonPathPureDuckTyping             = jsonPathClause(
                                             possiblyCapturing,
                                             namePlaceholder,
                                             fieldList
                                           )


     //   ..
,    jsonPathDoubleDot                  = jsonPathClause(/\.\./)


     //   .
,    jsonPathDot                        = jsonPathClause(/\./)


     //   !
,    jsonPathBang                       = jsonPathClause(
                                             possiblyCapturing,
                                             /!/
                                           )
```

93

```
         //    nada!
     ,   emptyString                              = jsonPathClause(/$/)


     ;



     /* We export only a single function. When called, this function injects
        into another function the descriptors from above.
      */
     return function (fn){
        return fn(
           lazyUnion(
              jsonPathNamedNodeInObjectNotation
           ,  jsonPathNamedNodeInArrayNotation
           ,  jsonPathNumberedNodeInArrayNotation
           ,  jsonPathPureDuckTyping
           )
        ,  jsonPathDoubleDot
        ,  jsonPathDot
        ,  jsonPathBang
        ,  emptyString
        );
     };

}());
```

8.8   lists.js

```javascript
/**
 * Like cons in Lisp
 */
function cons(x, xs) {

    /* Internally lists are linked 2-element Javascript arrays.

       So that lists are all immutable we Object.freeze in newer
       Javascript runtimes.

       In older engines freeze should have been polyfilled as the
       identity function. */
    return Object.freeze([x,xs]);
}

/**
 * The empty list
 */
var emptyList = null,

/**
 * Get the head of a list.
 *
 * Ie, head(cons(a,b)) = a
 */
    head = attr(0),

/**
 * Get the tail of a list.
 *
 * Ie, head(cons(a,b)) = a
 */
    tail = attr(1);


/**
 * Converts an array to a list
 *
 *    asList([a,b,c])
 *
 * is equivalent to:
 *
 *    cons(a, cons(b, cons(c, emptyList)))
 **/
```

```javascript
function arrayAsList(inputArray){

   return reverseList(
      inputArray.reduce(
         flip(cons),
         emptyList
      )
   );
}

/**
 * A varargs version of arrayAsList. Works a bit like list
 * in LISP.
 *
 *     list(a,b,c)
 *
 * is equivalent to:
 *
 *     cons(a, cons(b, cons(c, emptyList)))
 */
var list = varArgs(arrayAsList);

/**
 * Convert a list back to a js native array
 */
function listAsArray(list){

   return foldR( function(arraySoFar, listItem){

      arraySoFar.unshift(listItem);
      return arraySoFar;

   }, [], list );

}

/**
 * Map a function over a list
 */
function map(fn, list) {

   return list
            ? cons(fn(head(list)), map(fn,tail(list)))
            : emptyList
            ;
}
```

```javascript
/**
 * foldR implementation. Reduce a list down to a single value.
 *
 * @pram {Function} fn      (rightEval, curVal) -> result
 */
function foldR(fn, startValue, list) {

   return list
            ? fn(foldR(fn, startValue, tail(list)), head(list))
            : startValue
            ;
}


/**
 * Returns true if the given function holds for every item in
 * the list, false otherwise
 */
function all(fn, list) {

   return !list ||
          fn(head(list)) && all(fn, tail(list));
}


/**
 * Apply a function to every item in a list
 *
 * This doesn't make any sense if we're doing pure functional because
 * it doesn't return anything. Hence, this is only really useful if fn
 * has side-effects.
 */
function each(fn, list) {

   if( list ){
      fn(head(list));
      each(fn, tail(list));
   }
}


/**
 * Reverse the order of a list
 */
function reverseList(list){

   // js re-implementation of 3rd solution from:
   //    http://www.haskell.org/haskellwiki/99_questions/Solutions/5
```

```javascript
function reverseInner( list, reversedAlready ) {
    if( !list ) {
        return reversedAlready;
    }

    return reverseInner(tail(list), cons(head(list), reversedAlready))
}

return reverseInner(list, emptyList);
}
```

8.9   pubSub.js

```javascript
/**
 * Isn't this the cutest little pub-sub you've ever seen?
 *
 * Does not allow unsubscription because is never needed inside Oboe.
 * Instead, when an Oboe instance is finished the whole of it should be
 * available for GC'ing.
 */
function pubSub(){

    var listeners = {};

    return {

        on:function( eventId, fn ) {

            listeners[eventId] = cons(fn, listeners[eventId]);

            return this; // chaining
        },

        fire:function ( eventId, event ) {

            each(
                partialComplete( apply, [event || undefined] ),
                listeners[eventId]
            );
        }
    };
}
```

8.10   publicApi.js

```javascript
// export public API
var oboe = apiMethod('GET');
oboe.doGet    = oboe;
oboe.doDelete = apiMethod('DELETE');
oboe.doPost   = apiMethod('POST', true);
oboe.doPut    = apiMethod('PUT', true);

function apiMethod(httpMethodName, mayHaveRequestBody) {

   return function(firstArg){

      if (isString(firstArg)) {

         // parameters specified as arguments
         //
         //   if (mayHaveContext == true) method signature is:
         //      .doMethod( url, content )
         //
         //   else it is:
         //      .doMethod( url )
         //
         return wire(
                httpMethodName,
                firstArg,                              // url
                mayHaveRequestBody && arguments[1]     // body
         );
      } else {

         // method signature is:
         //    .doMethod({url:u, body:b, complete:c, headers:{...}})

         return wire(
                httpMethodName,
                firstArg.url,
                firstArg.body,
                firstArg.headers
         );
      }
   };
}
```

8.11 streamingHttp-browser.js

```javascript
function httpTransport(){
   return new XMLHttpRequest();
}

/**
 * A wrapper around the browser XmlHttpRequest object that raises an
 * event whenever a new part of the response is available.
 *
 * In older browsers progressive reading is impossible so all the
 * content is given in a single call. For newer ones several events
 * should be raised, allowing progressive interpretation of the response.
 *
 * @param {Function} fire a function to pass events to when something happens
 * @param {Function} on a function to use to subscribe to events
 * @param {XMLHttpRequest} xhr the xhr to use as the transport. Under normal
 *          operation, will have been created using httpTransport() above
 *          but for tests a stub can be provided instead.
 * @param {String} method one of 'GET' 'POST' 'PUT' 'DELETE'
 * @param {String} url the url to make a request to
 * @param {String|Object} data some content to be sent with the request.
 *                        Only valid if method is POST or PUT.
 * @param {Object} [headers] the http request headers to send
 */
function streamingHttp(fire, on, xhr, method, url, data, headers) {

   var numberOfCharsAlreadyGivenToCallback = 0;

   // When an ABORTING message is put on the event bus abort
   // the ajax request
   on( ABORTING, function(){

      // if we keep the onreadystatechange while aborting the XHR gives
      // a callback like a successful call so first remove this listener
      // by assigning null:
      xhr.onreadystatechange = null;

      xhr.abort();
   });

   /** Given a value from the user to send as the request body, return in
    *  a form that is suitable to sending over the wire. Returns either a
    *  string, or null.
    */
   function validatedRequestBody( body ) {
```

```
   if( !body )
      return null;

   return isString(body)? body: JSON.stringify(body);
}


/**
 * Handle input from the underlying xhr: either a state change,
 * the progress event or the request being complete.
 */
function handleProgress() {

   var textSoFar = xhr.responseText,
      newText = textSoFar.substr(numberOfCharsAlreadyGivenToCallback);


   /* Raise the event for new text.

      On older browsers, the new text is the whole response.
      On newer/better ones, the fragment part that we got since
      last progress. */

   if( newText ) {
      fire( NEW_CONTENT, newText );
   }

   numberOfCharsAlreadyGivenToCallback = len(textSoFar);
}


if('onprogress' in xhr){  // detect browser support for progressive delivery
   xhr.onprogress = handleProgress;
}

xhr.onreadystatechange = function() {

   if(xhr.readyState == 4 ) {

      // is this a 2xx http code?
      var sucessful = String(xhr.status)[0] == 2;

      if( sucessful ) {
         // In Chrome 29 (not 28) no onprogress is fired when a response
         // is complete before the onload. We need to always do handleInput
         // in case we get the load but have not had a final progress event.
         // This looks like a bug and may change in future but let's take
```

```
                // the safest approach and assume we might not have received a
                // progress event for each part of the response
                handleProgress();

                fire( END_OF_CONTENT );
            } else {

                fire( ERROR_EVENT, xhr.status );
            }
        }
    };

    try{

        xhr.open(method, url, true);

        for( var headerName in headers ){
            xhr.setRequestHeader(headerName, headers[headerName]);
        }

        xhr.send(validatedRequestBody(data));

    } catch( e ) {
        // To keep a consistent interface with Node, we can't fire an event here.
        // Node's streaming http adaptor receives the error as an asynchronous
        // event rather than as an exception. If we fired now, the Oboe user
        // has had no chance to add a .fail listener so there is no way
        // the event could be useful. For both these reasons defer the
        // firing to the next JS frame.
        window.setTimeout(partialComplete(fire, ERROR_EVENT, e), 0);
    }
}
```

8.12 streamingHttp-node.js

```javascript
function httpTransport(){
   return require('http');
}


/**
 * A wrapper around the browser XmlHttpRequest object that raises an
 * event whenever a new part of the response is available.
 *
 * In older browsers progressive reading is impossible so all the
 * content is given in a single call. For newer ones several events
 * should be raised, allowing progressive interpretation of the response.
 *
 * @param {Function} fire a function to pass events to when something happens
 * @param {Function} on a function to use to subscribe to events
 * @param {XMLHttpRequest} http the http implementation to use as the transport. Under norm
 *          operation, will have been created using httpTransport() above
 *          and therefore be Node's http
 *          but for tests a stub may be provided instead.
 * @param {String} method one of 'GET' 'POST' 'PUT' 'DELETE'
 * @param {String} url the url to make a request to
 * @param {String|Object} data some content to be sent with the request.
 *                         Only valid if method is POST or PUT.
 * @param {Object} [headers] the http request headers to send
 */
function streamingHttp(fire, on, http, method, url, data, headers) {

   if( !url.match(/http:\/\//) ) {
      url = 'http://' + url;
   }

   var parsedUrl = require('url').parse(url);

   var req = http.request({
      hostname: parsedUrl.hostname,
      port: parsedUrl.port,
      path: parsedUrl.pathname,
      method: method,
      headers: headers
   }, function(res) {

      var statusCode = res.statusCode,
          sucessful = String(statusCode)[0] == 2;

      if( sucessful ) {
```

104

```javascript
        res.on('data', function (chunk) {

            fire( NEW_CONTENT, chunk.toString() );
        });

        res.on('end', function() {

            fire( END_OF_CONTENT );
        });

    } else {

        fire( ERROR_EVENT, statusCode );
    }
});

req.on('error', function(e) {
    fire( ERROR_EVENT, e );
});

on( ABORTING, function(){
    req.abort();
});

if( data ) {
    var body = isString(data)? data: JSON.stringify(data);
    req.write(body);
}

req.end();

}
```

8.13 util.js

```javascript
/**
 * This file defines some loosely associated syntactic sugar for
 * Javascript programming
 */


/**
 * Returns true if the given candidate is of type T
 */
function isOfType(T, maybeSomething){
   return maybeSomething && maybeSomething.constructor === T;
}
function pluck(key, object){
   return object[key];
}

var attr = partialComplete(partialComplete, pluck),
    len = attr('length'),
    isString = partialComplete(isOfType, String);

/**
 * I don't like saying this:
 *
 *    foo !=== undefined
 *
 * because of the double-negative. I find this:
 *
 *    defined(foo)
 *
 * easier to read.
 */
function defined( value ) {
   return value !== undefined;
}

function always(){return true}

/**
 * Returns true if object o has a key named like every property in
 * the properties array. Will give false if any are missing, or if o
 * is not an object.
 */
function hasAllProperties(fieldList, o) {
```

```
    return      (o instanceof Object)
         &&
            all(function (field) {
               return (field in o);
            }, fieldList);
}
```

8.14   wire.js

```
/**
 * This file sits just behind the API which is used to attain a new
 * Oboe instance. It creates the new components that are required
 * and introduces them to each other.
 */

function wire (httpMethodName, url, body, headers){

   var eventBus = pubSub();

   streamingHttp( eventBus.fire, eventBus.on,
                  httpTransport(),
                  httpMethodName, url, body, headers );

   return instanceController(
                  eventBus.fire, eventBus.on,
                  clarinet.parser(),
                  incrementalContentBuilder(eventBus.fire)
   );
}
```

9   Appendix iii: Benchmarking

9.1   benchmarkClient.js

```
/* call this script from the command line with first argument either
    oboe, jsonParse, or clarinet.

    This script won't time the events, I'm using 'time' on the command line
    to keep things simple.
 */

require('color');

var DB_URL = 'http://localhost:4444/db';


function aggregateWithOboe() {

   var oboe = require('../dist/oboe-node.js');

   oboe(DB_URL).node('{id url}.url', function(url){

      oboe(url).node('name', function(name){

         console.log(name);
         this.abort();
         console.log( process.memoryUsage().heapUsed );
      });
   });
}

function aggregateWithJsonParse() {

   var getJson = require('get-json');

   getJson(DB_URL, function(err, records) {

      records.data.forEach( function( record ){

         var url = record.url;

         getJson(url, function(err, record) {
            console.log(record.name);
            console.log( process.memoryUsage().heapUsed );
```

```
        });
      });

    });

  }


function aggregateWithClarinet() {

    var clarinet = require('clarinet');
    var http = require('http');
    var outerClarinetStream = clarinet.createStream();
    var outerKey;

    var outerRequest = http.request(DB_URL, function(res) {

        res.pipe(outerClarinetStream);
    });

    outerClarinetStream = clarinet.createStream();

    outerRequest.end();

    outerClarinetStream.on('openobject', function( keyName ){
        if( keyName ) {
            outerKey = keyName;
        }
    });

    outerClarinetStream.on('key', function(keyName){
        outerKey = keyName;
    });

    outerClarinetStream.on('value', function(value){
        if( outerKey == 'url' ) {
            innerRequest(value)
        }
    });


    function innerRequest(url) {

        var innerRequest = http.request(url, function(res) {

            res.pipe(innerClarinetStream);
```

110

```javascript
        });

        var innerClarinetStream = clarinet.createStream();

        innerRequest.end();

        var innerKey;

        innerClarinetStream.on('openobject', function( keyName ){
            if( keyName ) {
                innerKey = keyName;
            }
        });

        innerClarinetStream.on('key', function(keyName){
            innerKey = keyName;
        });

        innerClarinetStream.on('value', function(value){
            if( innerKey == 'name' ) {
                console.log( value )
                console.log( process.memoryUsage().heapUsed );
            }
        });
    }
}

var strategies = {
    oboe:       aggregateWithOboe,
    jsonParse:  aggregateWithJsonParse,
    clarinet:   aggregateWithClarinet
}

var strategyName = process.argv[2];

// use any of the above three strategies depending on a command line argument:
console.log('benchmarking strategy', strategyName);

strategies[strategyName]();
```

9.2 benchmarkServer.js

```javascript
/**
 */

"use strict";

var PORT = 4444;

var TIME_BETWEEN_RECORDS = 15;
// 80 records but only every other one has a URL:
var NUMBER_OF_RECORDS = 80;

function sendJsonHeaders(res) {
   var JSON_MIME_TYPE = "application/octet-stream";
   res.setHeader("Content-Type", JSON_MIME_TYPE);
   res.writeHead(200);
}

function serveItemList(_req, res) {

   console.log('slow number server: send simulated database data');

   res.write('{"data": [');

   var i = 0;

   var inervalId = setInterval(function () {

      if( i % 2 == 0 ) {

         res.write(JSON.stringify({
            "id": i,
            "url": "http://localhost:4444/item/" + i
         }));
      } else {
         res.write(JSON.stringify({
            "id": i
         }));
      }

      if (i == NUMBER_OF_RECORDS) {

         res.end(']}');

         clearInterval(inervalId);
```

```javascript
            console.log('db server: finished writing to stream');
        } else {
            res.write(',');
        }

        i++;

    }, TIME_BETWEEN_RECORDS);
}

function serveItem(req, res){

    var id = req.params.id;

    console.log('will output fake record with id', id);

    setTimeout(function(){
        // the items served are all the same except for the id field.
        // this is realistic looking but randomly generated object fro
        // <project>/test/json/oneHundredrecords.json
        res.end(JSON.stringify({
            "id" : id,
            "url": "http://localhost:4444/item/" + id,
            "guid": "046447ee-da78-478c-b518-b612111942a5",
            "picture": "http://placehold.it/32x32",
            "age": 37,
            "name": "Humanoid robot number " + id,
            "company": "Robotomic",
            "phone": "806-587-2379",
            "email": "payton@robotomic.com"
        }));

    }, TIME_BETWEEN_RECORDS);

}

function routing() {
    var Router = require('node-simple-router'),
        router = Router();

    router.get( '/db',         serveItemList);
    router.get( '/item/:id',   serveItem);

    return router;
}
```

```
var server = require('http').createServer(routing()).listen(PORT);

console.log('Benchmark server started on port', String(PORT));
```

10 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.

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

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

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

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

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

Gill, Brendan. 2013. "OpenSignal."

Guo, shu-yu. 2013. "Two Reasons Functional Style Is Slow in SpiderMonkey." http://rfrn.org//~shu/2013/03/20/two-reasons-functional-style-is-slow-in-spidermonkey.html.

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

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

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

Ogden, Max. 2012. "Streaming XHR." http://maxogden.com/a-proposal-for-streaming-xhr.html.

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

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.

Yukihiro, Matsumoto. 2003. "The Power and Philosophy of Ruby." http://www.rubyist.net/~matz/slides/oscon2003/index.html.

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/.