Oboe.js: An approach to I/O for REST clients which is neither batch nor stream; nor SAX nor DOM

Jim Higson

2013

Contents

1 Abstract **7**

2 Introduction **8**

2.1 How REST aggregation could be faster 9

2.2 Stepping outside the big-small tradeoff 11

2.3 Staying fast on a fallible network 12

2.4 Agile methodologies, frequent deployments, and compatibility today with versions tomorrow 12

2.5 Deliverables 13

2.6 Criteria for success 13

3 Background **15**

3.1 The web as an application platform 15

3.2 Node.js 16

3.3 Json and XML data transfer formats 19

3.4 Common patterns for connecting to REST services 20

3.5 JsonPath and XPath selector languages 22

3.6 Browser XML Http Request (XHR) 25

3.7 XHRs and streaming 26

3.8 Browser streaming frameworks 27

3.9 Parsing: SAX and Dom 28

4 Design and Reflection: **31**

4.1 Detecting types in JSON 32

4.2 Importing CSS4's explicit capturing to Oboe's JSONPath 35

4.3 Parsing the JSON Response 36

4.4 API design 36

4.5 Earlier callbacks when paths are found prior to nodes 39

4.6 Choice of streaming data transport 40

4.7 Handling transport failures 42

4.8 Oboe.js as a Micro-Library 42

5 Implementation **43**

5.1 Components of the project 43

5.2 Design for automated testing 44

5.3 Running the tests 45

5.4 Packaging to a single distributable file 47

5.5 Styles of Programming 49

5.6 Incrementally building the parsed content 50

5.7 Oboe JSONPath Implementation 53

6 Conclusion **56**

6.1 Differences in the programs written using Oboe.js 56

6.2 Benchmarking vs non-progressive REST 57

6.3 Comparative Programmer Ergonomics 59

6.4 Performance of code styles under various engines 60

6.5 Status as a micro-library 62

6.6 potential future work 62

7 Appendix i: Limits to number of simultaneous connections under various http clients **64**

8 Appendix ii: Oboe.js source code listing **65**

8.1 clarinetListenerAdaptor.js 65

8.2 events.js 66

8.3 functional.js 67

8.4 incrementalContentBuilder.js 71

8.5 instanceController.js 76

8.6 jsonPath.js 81

8.7 jsonPathSyntax.js 90

8.8 lists.js 93

8.9 pubSub.js 97

8.10 publicApi.js 98

8.11 streamingHttp-browser.js 99

8.12 streamingHttp-node.js 102

8.13 util.js 105
8.14 wire.js 107

9 Appendix iii: Benchmarking **108**
9.1 benchmarkClient.js 108
9.2 benchmarkServer.js 111

10 Bibliography **114**

List of Figures

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. . . 9

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. 10

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 15

4 *Single-threaded vs multi-threaded scheduling for a http aggregator* 17

5 **Major components of Oboe.js illustrating program flow from http transport to application callbacks.** UML facet/receptacle notation is used to show the flow of events and event names are given in capitals. For clarity events are depicted as transferring directly between publisher and subscriber but this is actually performed through an intermediary. 43

6 **The test pyramid**. Much testing is done on the low-level components of the system, less on their composed behaviours, and less still on a whole-system level. 44

7 **Relationship between various files and test libraries** *other half of sketch from notebook* 46

8 List representation of an ascent rising from leaf to root through a JSON tree. Note the special ROOT value which represents the location of the pathless root node. The ROOT value is an object, taking advantage of object uniqueness to ensure that its location is unequal to all others. 51

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

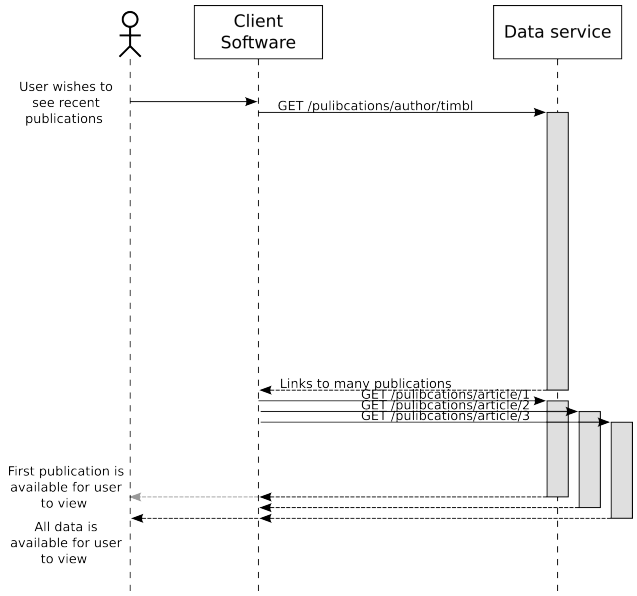

Figure 1: **Sequence diagram showing the aggregation of low-level REST resources.** A client fetches an author's publication list and then their first three articles. This sequence represents the most commonly used technique in which the client does not react to the response until it is complete. In 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.

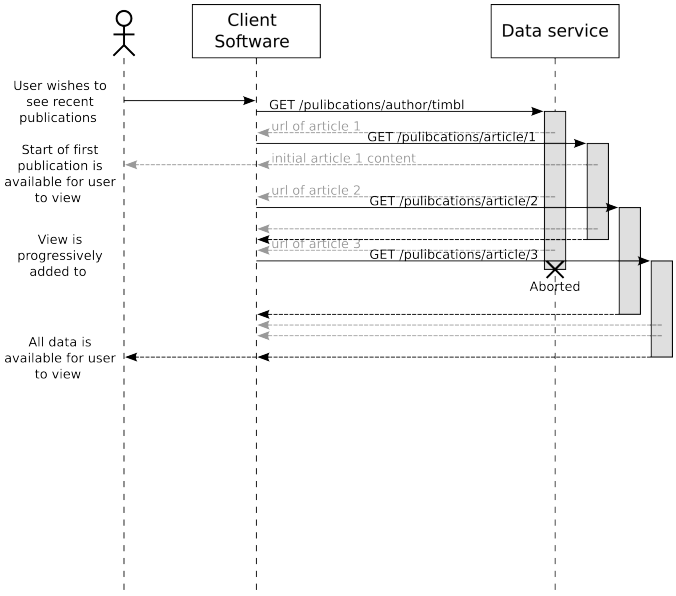

Figure 2: **Revised aggregation sequence for a client capable of progressively interpreting the resources.** Because arrows in UML sequence diagrams draw returned values as a one-off happening rather than a continuous process, I have introduced a lighter arrow notation representing fragments of an incremental 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

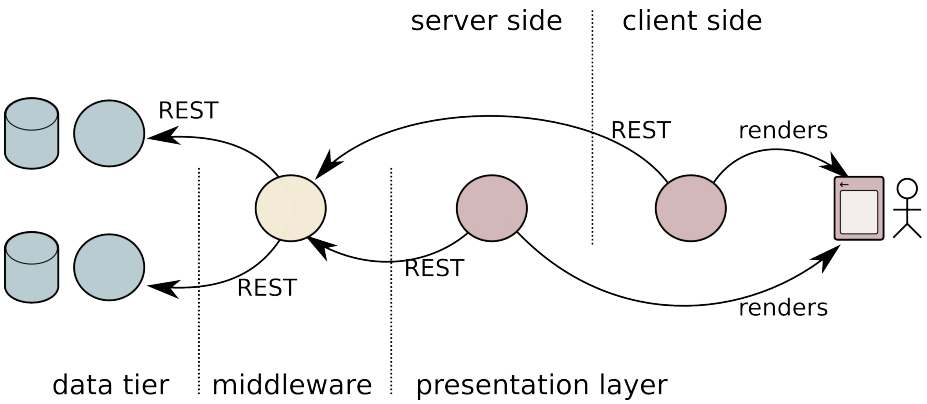

Figure 3: **Labelling nodes in an n-tier architecture**. Regardless of where a node is located, REST may be used as the means of communication. By focusing on REST clients, nodes in the middleware and presentation layer fall in our scope. Although network topology is often split about client and server 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 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.

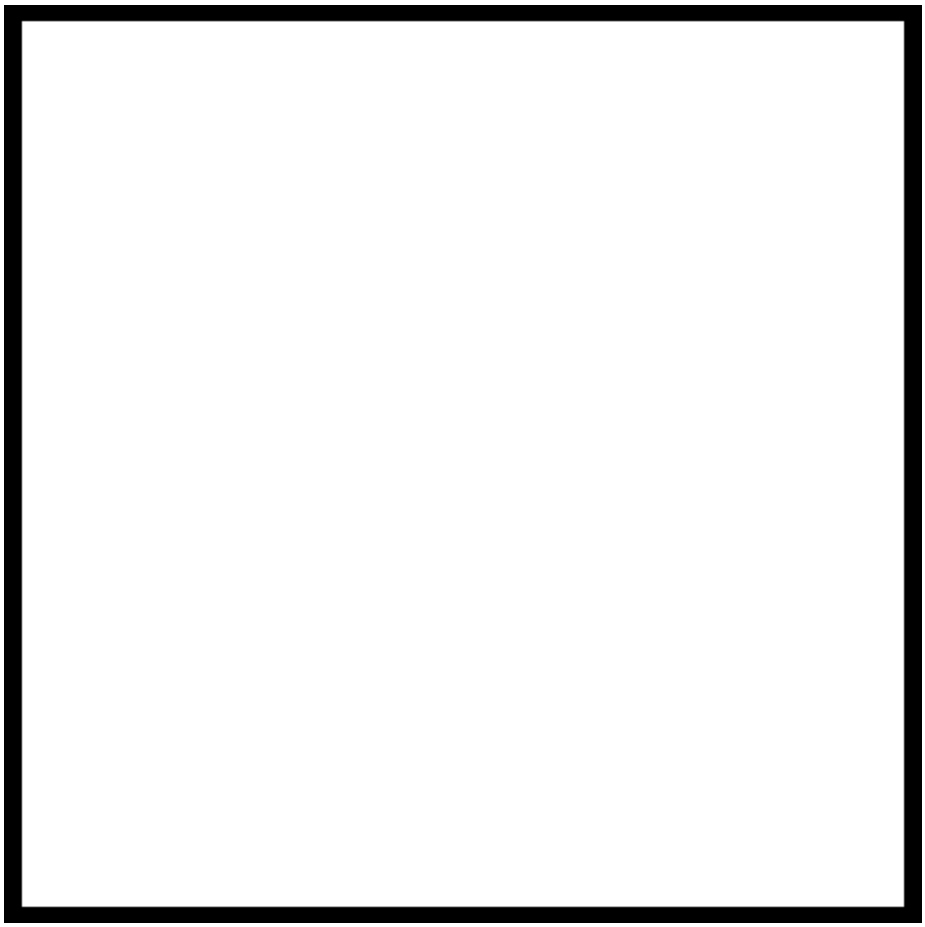

Figure 4: *Single-threaded vs multi-threaded scheduling for a http aggregator*

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 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.

```
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));
   });
```

3.3 Json and XML data transfer formats

Both XML and JSON are text based, tree shaped data formats with human and machine readability. One of the design goals of XML was to simplify SGML to the point that a graduate student could implement a full parser in a week (Eberhart and Fischer 2002 p287). Continuing this arc of simpler data formats, JSON "The fat-free alternative to XML(Douglas 2009)" isolates Javascript's syntax for literal values into a stand-alone serialisation language. For the graduate tackling JSON parsing the task is simpler still, being expressible as fifteen context free grammars.

Whereas XML's design can be traced to document formats, JSON's lineage is in a programming language. From these roots isn't surprising that JSON maps more directly to the metamodel that most programmers think in. XML parsers produce Elements, Text, Attributes, ProcessingInstruction which require extra translation before they are convenient to use inside a programming language. Because JSON already closely resembles how a programmer would construct a runtime model of their data, fewer steps are required before using the deserialised form in a given programming language. The JSON nodes: *strings*, *numbers*,

objects and *arrays* will in many cases map directly onto their language types and, for loosely typed languages at least, the parser output bears enough similarity to domain model objects that it may be used directly without any further transformation.

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

Both JSON and XML are used to serialise to and from orderless constructs but but while serialised to text, an ordered list of characters, the nodes are inevitably encountered according to some serialisation order. There is no rule forbidding serialisation to JSON or XML attributes in an order-significant way but in general the order is considered to not be significant in the serialised format's model. In the example above, the people objects would probably have been written out to represent either a class with two public properties or a hash map. On receiving this data the text would be demarshalled into similar structures and that the data found an ordered expression during transport would be quickly forgotten. However, when viewing a document through a streaming and interpreting documents while still incomplete this detail cannot be ignored as a concern relating only to the accidents of transfer. If nodes were interpreted based on their first field in the example above Walter would find a different handling than the other two. Because the serialisation will contain items which are written to follow an indeterminate order it will be important to ensure that, despite the streaming, the REST client does not encourage programming in a way that depends on the order that these fields are received.

3.4 Common patterns for connecting to REST services

For languages such as Javascript or Clojure with a loosely-typed representation of objects as generic key-value pairs, when a JSON REST resource is received, the output from the parser resembles the normal object types closely enough that it is acceptable to use it directly throughout the program. For XML this is not the case and some marshaling is required. In more strongly typed OO languages such as Java or C#, JSON's relatively freeform, classless objects are less convenient. For the example JSON from the previous section to be smoothly consumed, instantiating instances of a domain model Person class with methods such as `getName()` and `getTown()` would be preferable, representing the remote resource's objects no differently than if they had originated locally. Automatic marshaling generalises this process by providing a two-way mapping between

the domain model and its serialisation, either completely automatically or based on a declarative specification. It is common in strongly typed languages for REST client libraries to automatically demarshal as part of receiving a fetched rest response. From the programmer's vantage it is as if the domain objects themselves had been fetched. Adding an additional layer, another common design pattern intended to give a degree of isolation between remote resources and the local domain model is to demarshal automatically only so far as *Data Transfer Objects* (DTOs). DTOs are instances of classes which implement no logic other than storage, and from these DTOs the domain model objects may be programmatically instantiated. DTOs are more necessary when using XML. For reading JSON resources we might say that the JSON objects *are* the DTOs.

The degree of marshaling that is used generally changes only the types of the entities that the REST client library hands over to the application developer without affecting the overall structure of the message. Regardless of the exact types given, having received the response model the developer will usually start by addressing the pertinent parts of the response by drilling down into the structures using assessor operators from the programming language itself.

```
// Java example - programmatic approach to domain model interrogation

// The methods used to drill down to desired components
// are all getters: getPeople, getName, and getTown.

void handleResponse( RestResponse response ) {

   for( Person p : response.getPeople() ) {
      addPersonToDb( p.getName(), p.getTown() );
   }
}

// equivalent Javascript - the programming follows the same basic
// process. This time using Javascript's dot operator.

function handleResponse( response ){

   response.people.forEach( function( person ){
      addPersonToDb( p.name, p.town );
   });
}
```

One weakness in this means of drilling down is that the code making the inspection is quite tightly coupled to the precise structure of the thing that it is inspecting. Taking the above example, if the resource being fetched were later refactored such that the town concept were refactored into a fuller address with a town-county-country tuple, the code addressing the structure would also

have to change just to continue to do the same thing. Although this kind of drill-down programming is commonly practiced and not generally recognised as a code smell, requiring knock-on changes when an unrelated system is refactored seems to me as undesirable here as it would be anywhere else.

In *the Red Queen's race* it took "all the running you can do, to keep in the same place". Ideally as a programmer I'd like to expend effort to make my code to do something new, or to perform something that it already did better, not so that it keeps the same. Following an object oriented encapsulation of data such that a caller does not have to concern themselves with the data structures behind an interface, the internal implementation may be changed without disruptions to the rest of the code base. However when the structure of the inter-object composition is revised, isolation from the changes is less often recognised as a desirable trait. A method of programming which truly embraced extreme programming would allow structural refactoring to occur without disparate, parts having to be modified in parallel.

Extraneous changes also dilute a VCS changelog, making it less easily to later follow a narrative of code changes which are intrinsic to the difference in logic expressed by the program, and therefore harder to later understand the thinking behind the change and the reason for the change.

3.5 JsonPath and XPath selector languages

The problem of drilling down to pertinent fragments of a message without tightly coupling to the format could be somewhat solved if instead of programmatically descending step-by-step, a language were used which allows the right amount of specificity regarding which parts to select. For markup languages there are associated query languages whose coupling is loose enough that not every node that is descended through must be specified. The best known is XPATH but there is also JSONPath, a JSON equivalent (Goessner 2007).

As far as possible the JSONPath language follows the javascript to descend into the same sub-tree.

```
// in Javascript we can get the town of the second person as:
let town = subject.people[2].town

// the equivalent JSONPath expression is identical:
let townSelector = "people[2].town"

// We would be wise not to write overly-specific selectors.
// JSONPath also provides an ancestor relationship not found in Javascript:
let betterTownSelector = "people[2]..town"
```

Consider the resource below:

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

The JSONPath `people.*..town` against the above JSON format would continue to select correctly after a refactor to the JSON below:

```
{
   people: [
      {  name: 'John',
         address:{town:'Oxford', county:'Oxon', country:'uk'}
      },
      {  name: 'Jack',
         address:{town:'Bristol', county:'Bristol', country:'uk'}
      }
      {  address:{
            town:'Cambridge', county:'Cambridgeshire',
            country:'uk'
         },
         name: 'Walter'
      }
   ]
}
```

Maintaining compatibility with unanticipated format revisions through selector languages is easier with JSON than XML. The XML metamodel contains overlapping representations of equivalent entities which a refactored format is liable to switch between. Each XML element has two distinct lists of child nodes, attribute children and node list children; from one perspective attributes are child nodes of their parent element but they can alternatively be considered as data stored in the element. Because of this classification ambiguity an XML document doesn't form a single, correct n-way tree. Because of the difference in expressivity between attributes which may only be strings and child nodes which allow recursive structure, this is a common refactor when a more detailed mapping is required and a scalar value is upgraded to be compound. XPath selectors written in the most natural way do not track this change.

```
<people>
   <person name="John" town="Oxford"></person>
</people>
```

The XPath `//person@town` matches the XML above but because of the refactor from attribute to child element fails to match the revised version below.

```
<people>
   <person>
      <name>
         John
      </name>
      <address>
         <town>Oxford</town> <county>Oxon</county>
      </address>
   </person>
</people>
```

Reflecting its dual purpose for marking up for documents or data, XML also invites ambiguous interpretation of the whitespace between tags. Whitespace is usually meaningful for documents but ignorable for data. Strictly, whitespace text nodes are a part of the document but in practice many tree walkers discard them as insignificant. In the XML above the `<person>` element may be enumerated as either the first or second child of `<people>` depending on whether the whitespace before it is considered. Likewise, the text inside `<name>` might be `'John'` or `'(newline)(tab)(tab)John'`. Inherited from JSON's programming language ancestry, the space between tokens is never significant.

Programming against a changing service is always going to be a moving target, but it is easier to miss with XPATH than with JSON. In JSON each nodes has only one, unambiguous set of children so the metamodel does not present the format author with a selection from logical equivalents that are addressed through different mechanisms. If a scalar value is updated to a compound only the node itself changes, the addressing of the node is unaffected.

Generally in descriptive hierarchical data there is a trend for ancestorship to denote the same relationship between concepts regardless of the number of intermediate generations. In the example above, `town` transitioned from child to grandchild of `person` without disturbing the implicit 'lives in' relationship. In JSONPath the `..` operator provides matching through zero or more generations, unperturbed when extra levels are added. Of course, this trend will not hold for every conceivable way of building message semantics because it is possible that an intermediate node on the path from ancestor to descendant will change the nature of the expressed relationship. A slightly contrived example might be if we expanded our model to contain fuzzy knowledge:

```
<people>
   <person>
      <name>
         <isProbably>Bob</isProbably>
```

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

Considering the general case, it will not be possible to track all possible service refactors safely. By necessity a resource consumer should limit their ambitions to tracking ontology additions which do not change the meaning of the existing concepts. In practice integration testing against the beta version of a service will be necessary to be pre-warned of upcoming, incompatible changes. If an incompatibility is found the ability to then create an expression which is compatible with with a present and known future version remains a valuable tool because it decouples service consumer and provider update schedules, removing the need for the client to march perfectly in sync with the service.

3.6 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 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 is today 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);
});
```

3.7 XHRs and streaming

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

The historic lack of streaming for data fetched using XHR stands incongruously with the browser as a platform in which almost every other remote resource is interpreted progressively. Examples include progressive image formats, html, svg, video, and Javascript itself (script interpretation starts before the script is fully loaded).

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

The progress event is supported by the latest version of all major browsers. However, Internet Explorer only added support recently with version 10 and there is a significant user base remaining on versions 8 and 9.

3.8 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.

3.9 Parsing: SAX and Dom

From the XML world two standard parser types exist, SAX and DOM, with DOM by far the more popular. Although the terms originate in XML, both styles of parsers are also available for JSON. DOM performs a parse as a single evaluation and returns a single object model representing the whole of the document. Conversely, SAX parsers are probably better considered as tokenisers, providing a very low-level event driven interface following the Observer pattern that notifies the programmer of each token separately as it is found. From DOM's level of abstraction the markup syntax is a distant concern whereas for SAX each element's opening and closing tag is noted so the developer may not put the data's serialisation aside. SAX has the advantages that it may read a document progressively and has lower memory requirements because it does not store the parsed tree. Correspondingly, it it popular for embedded systems on limited hardware which need to handle documents larger than the available RAM.

Suppose we have some json representing people and want to extract the name of the first person. Given a DOM parser this may be written quite succinctly:

```
function nameOfFirstPerson( myJsonString ) {

   // All recent browsers provide JSON.parse as standard.

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

To contrast, the equivalent below uses SAX, expressed in the most natural way for the technology.[3]

```
function nameOfFirstPerson( myJsonString, callbackFunction ){


   var clarinet = clarinet.parser(),

       // With a SAX parser it is the developer's responsibility
       // to track where in the document the cursor currently is,
       // Several variables are used to maintain this state.
       inPeopleArray = false,
       inPersonObject = false,
       inNameAttribute = false,
       found = false;
```

[3] For an example closer to the real world see https://github.com/dscape/clarinet/blob/master/samples/twitter.js.

```
    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);
}
```

The developer pays a high price for progressive parsing, the SAX version is considerably longer and more difficult to read. SAX's low-level semantics require a lengthy expression and push the responsibility of maintaining state regarding the current position in the document and the nodes that have previously been seen onto the programmer. This maintenance of state tends to programmed once per usage rather than assembled as the composition of reusable parts. I find the order of the code under SAX quite unintuitive; event handlers cover multiple unrelated cases and each concern spans multiple handlers. This lends to a style of programming in which separate concerns do not find separate expression in the code. It is also notable that, unlike DOM, as the depth of the document

being interpreted increases, the length of the programming required to interpret it also increases, mandating more state be stored and an increased number of cases be covered per event handler.

While SAX addresses many of the problems raised in this dissertation, I find the unfriendly developer ergonomics pose too high a barrier to its adoption for all but fringe uses.

4 Design and Reflection:

Using a combination of techniques from the previous chapter I propose that it is possible to combine the desirable properties from SAX and DOM parsers into a REST client library which allows streaming but is also convenient to program.

By observing the flow of data streams through a SAX parser we can say that the REST workflow is more efficient if we do not wait until we have everything before we start using the parts that we do have. However, the SAX model presents poor developer ergonomics because it is not usually convenient to think on the level of abstraction that it presents: that of markup tokens. Using SAX, a programmer may only operate on a convenient abstraction after inferring it from a lengthy series of callbacks. In terms of ease of use, DOM is generally preferred because it provides the resource whole and in a convenient form. My design aims to duplicate this convenience and combine it with progressive interpretation by removing one restriction: that the node which is given is always the document root. From a hierarchical markup such as XML or JSON, when read in order, the sub-trees are fully known before we fully know their parent tree. We may select pertinent parts of a document and deliver them as fully-formed entities as soon as they are known, without waiting for the remainder of the document to arrive.

By my design, identifying the interesting parts of a document before it is complete involves turning the established model for drilling-down inside-out. Under asynchronous I/O the programmer's callback traditionally receives the whole resource and then, inside the callback, locates the sub-parts that are required for a particular task. Inverting this process, I propose extracting the locating logic currently found inside the callback and using it to decide when the callback should be used. The callback will receive complete fragments from the response once they have been selected according to this logic.

I will be implementing using the Javascript language because it has good support for non-blocking I/O and covers both contexts where this project will be most useful: in-browser programming and server programming. Focusing on the MVP, I will only be implementing the parsing of one mark-up language. Although this technique could be applied to any text-based, tree-shaped markup, I find that JSON best meets my goals because it is widely supported, easy to parse, and because it defines a single n-way tree, is amenable to selectors which span multiple format versions.

JSONPath is especially applicable to node selection as a document is read because it specifies only constraints on paths and 'contains' relationships. Because of the top-down serialisation order, on encountering any node in a serialised JSON stream, I will have already seen enough of the prior document to know its full path. JSONPath would not be so amenable if it expressed sibling relationships because there is no similar guarantee of having seen other nodes on the same level when any particular node is encountered. A new implementation of the

language is required because the existing JSONPath library is implemented only as a means to search through already gathered objects and is too narrow in applicability to be useful in our context.

Not all of the JSONPath language is well suited when we consider we are selecting specifically inside a REST resource. Given this context it is likely that we will not be examining a full model but rather a subset that we requested and was assembled on our behalf according to the parameters that we supplied. We can expect to be interested in all of the content so search-style selections such as 'books costing less than X' are less useful than queries which identify nodes because of their type and position such as 'all books in the discount set', or, because we know we are examining `/books/discount`, simply 'all books'. In creating a new JSONPath implementation I have chosen to follow the existing language somewhat loosely, thereby specialising the matching and avoiding unnecessary code. It is difficult to anticipate what the real-world matching requirements will be but if I deliver now the 20% of possible features that I'm reasonably sure will be used for 80% of tasks, for the time being any functionality which is not covered may be implemented inside the callbacks themselves and later added to the selection language. For example, somebody wishing to filter on the price of books might use branching to further select inside their callback. I anticipate that the selections will frequently involve types so it is useful to analyse the nature of type imposition with regards to JSON.

4.1 Detecting types in JSON

JSON markup describes only a few basic types. On a certain level this is also true for XML – most nodes are either of type Element or Text. However, the XML metamodel provides tagnames; essentially, a built-in type system for subclassifying the elements. JSON has no similar notion of types beyond the basic constructs: array, object, string, number. To understand data written in JSON's largely typeless model it is often useful if we think in terms of a more complex type system. This imposition of type is the responsibility of the observer rather than of the observed. The reader of a document is free to choose the taxonomy they will use to interpret it and this decision will vary depending on the purposes of the reader. The required specificity of taxonomy differs by the level of involvement in a field. Whereas 'watch' may be a reasonable type for most data consumers, to a horologist it is likely to be unsatisfactory without further sub-types. To serve disparate purposes, the JSONPath variant provided for node selection will have no inbuilt concept of type, the aim being to support programmers in creating their own.

```
<!--  XML leaves no doubt as to the labels we give to an Element's type.
      Although we might further interpret, this is a 'person' -->
<person  name='...' gender="male"
         age="45" height="175cm" profession="architect">
</person>
```

```
/*    JSON meanwhile provides no built-in type concept.
      This node's type might be 'thing', 'animal', 'human', 'male', 'man',
      'architect', 'artist' or any other of many overlapping impositions
      depending on our reason for examining this data */
{  "name":"...", "gender":"male", "age":"45"
   "height":"172cm" "profession":"architect">
}
```

In the absence of node typing beyond categorisation as objects, arrays and various primitives, the key immediately mapping to an object is often taken as a loose marker of its type. In the below example we may impose the the type 'address' prior to examining the contents because of the field name in the parent node.

```
{
   "name": ""
,  "residence": {
      "address": [
         "47", "Cloud street", "Dreamytown"
      ]
   }
,  "employer": {
      "name": "Mega ultra-corp"
   ,  "address":[
         "Floor 2", "The Offices", "Alvediston", "Wiltshire"
      ]
   }
}
```

This means of imposing type is simply expressed in JSONPath. The selector `address` would match all nodes whose parent maps to them via an address key.

As a loosely typed language, Javascript gives no protection against lists which store disparate types but by sensible convention this is avoided. Likewise, in JSON, although type is a loose concept, the items in a collection will generally be of the same type. From here follows a sister convention illustrated in the example below, whereby each item from an array is typed according to the key in the grandparent node which maps to the array.

```
{
   "residences": {
      "addresses": [
         ["10", "Downing street", "London"]
      ,  ["Chequers Court", "Ellesborough, "Buckinghamshire"]
      ,  ["Beach Hut", "Secret Island", "Bahamas"]
```

```
      ]
   }
}
```

In the above JSON, `addresses.*` would correctly identify the addresses. The pluralisation of field names such as 'address' becoming 'addresses' is common when marshaling from OO languages because the JSON keys are based on getters whose name typically reflects their cardinality; `public Address getAddress()` or `public List<Address> getAddresses()`. This may pose a problem in some cases and it would be interesting in future to investigate a system such as Ruby on Rails that natively understands English pluralisation. I considered introducing unions as an easy way to cover this situation, allowing expressions resembling `address|addresses.*` but decided that it is simpler if this problem is solves outside of the JSONPath language if the programmer registers two selection specifications against the same handler function.

In the below example types may not be easily inferred from ancestor keys.

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

Here, the keys which map onto addresses are named by the relationship between the parent and child nodes rather than by the type of the child. The type classification problem could be solved using an ontology with 'address' subtypes 'residence', 'premises', and 'office' but this solution feels quite heavyweight for a simple selection language. I chose instead to import the idea of *duck typing* from Python programing, as named in a 2000 usenet discussion:

> 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 (Martelli 2000)

A 'duck-definition' for the above JSON would be any object which has number, street, and town properties. We take an individualistic approach by deriving type from the node in itself rather than the situation in which it occurs. Because I find this selection technique simple and powerful I decided to add it to my JSONPath variant. As discussed in section 3.5, JSONPath's syntax is designed to resemble the equivalent Javascript accessors, but Javascript has no syntax for a value-free list of object keys. The closest available notation is for object literals so I created a duck-type syntax derived from this by omitting the values, quotation marks, and commas. The address type described above would be written as `{number street town}`. Field order is insignificant so `{a b}` and `{b a}` are equivalent.

It is difficult to generalise but when selecting items from a document I believe it will often be useful if nodes which are covariant with the given type are also matched. We may consider that there is a root duck type `{}` which matches any node, that we create a sub-duck-type if we add to the list of required fields, and a super-duck-type if we remove from it. Because in OOP extended classes may add new fields, this idea of the attribute list expanding for a sub-type applies neatly to JSON REST resources marshaled from an OO representation. In implementation, to conform to a duck-type a node must have all of the required fields but could also have any others.

limitations: why must be last term

4.2 Importing CSS4's explicit capturing to Oboe's JSONPath

JSONPath naturally expresses a 'contained in' relationship using the dot notation but no provision is made for the inverse 'containing' relationship. *Cascading Style Sheets*, CSS, the web's styling language, has historically shared this restriction but a proposal for extended selectors which is currently at Editor's Draft stage (Etemad and Atkins 2013) introduces an elegant solution. Rather than add an explicit 'containing' relationship, the draft observes that previously CSS has always selected the element conforming to the right-most of the selector terms, allowing only the deepest mentioned element to be styled. This restriction is lifted by allowing terms to be prefixed with `$` in order to make them explicitly capturing; a selector without an explicit capturing term continues to work as before. The CSS selector `form.important input.mandatory` selects mandatory inputs inside important forms but `$form.important input.mandatory` selects important forms with mandatory fields.

The new css4 capturing technique will be adapted for Oboe JSONPath. By duplicating a syntax which the majority of web developers should become familiar with over the next few years I hope that Oboe's learning curve can be made a little more gradual. Taking on this feature, the selector `person.$address.town` would identify an address node with a town child, or `$people.{name, dob}`

would provide the people array repeatedly whenever a new person is added to it. Javascript frameworks such as d3.js and Angular are designed to work with whole models as they change. Consequently, the interface they present converses more fluently with collections than individual entities. If we are downloading data to use with these libraries it is more convenient if we use explicit capturing so that we are notified whenever the collection is expanded and can pass it on.

4.3 Parsing the JSON Response

While SAX parsers provide an unappealing interface to application developers, as a starting point to handle low-level parsing in higher-level libraries they work very well (most XML DOM parsers are built in this way). The pre-existing Clarinet project is well tested, liberally licenced, and compact, meeting our needs perfectly. The name of this project, Oboe.js, was chosen in tribute to the value delivered by Clarinet.

4.4 API design

Everything that Oboe is designed to do can already be achieved by combining a SAX parser with imperatively coded node selection. This has not been adopted widely because it requires verbose, difficult programming in a style which is unfamiliar to most programmers. With this in mind it is a high priority to design a public API for Oboe which is concise, simple, and resembles other commonly used tools. If Oboe's API is made similar to common tools, a lesser modification should be required to switch existing projects to streaming http.

For some common use cases it should be possible to create an API which is a close enough equivalent to popular tools that it can be used as a direct drop-in replacement. Although used in this way no progressive loading would be enacted, when refactoring towards a goal the first step is often to create a new expression of the same logic (Martin 2008, 212). By giving basic support for non-progressive downloading, the door is open for apps to incrementally refactor towards a progressive expression. Allowing adoption as a series of small, easily manageable steps rather than a single leap is especially helpful for teams working under Scrum because all work must fit within a fairly short timeframe.

jQuery is by far the most popular library for AJAX today. The basic call style for making an AJAX GET request is as follows:

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

```
      console.log( "the request failed" );
   });
```

While jQuery is callback-based and internally event driven, the public API it exposes does not wrap asynchronously retrieved content in event objects. Event type is expressed by the name of the method used to add the listener. These names, `done` and `fail`, follow generic phrasing and are common to every functionality that jQuery provides asynchronously. Promoting brevity, the methods are chainable so that several listeners may be added from one statement. Although Javascript supports exception throwing, a fail event is used. Exceptions are not applicable to non-blocking I/O because at the time of the failure the call which provoked the exception will have already been popped from the call stack.

`jQuery.ajax` is overloaded and the parameter may be an object, allowing more information to be given:

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

This pattern of passing arguments as an object literal is common in Javascript for functions which take a large number of arguments, particularly if some are optional. This avoids having to pad unprovided optional arguments in the middle of the list with null values and, because the purpose of the values is apparent from the callee, also avoids an anti-pattern where a callsite can only be understood after counting the position of the arguments.

Taking on this style while extending it to cover events for progressive parsing, we arrive at the following Oboe public API:

```
oboe("resources/people.json")
   .node( "person.name", function(name, path, ancestors) {
      console.log("There is somebody called " + name);
   })
   .done( function( wholeJson ) {
      console.log("That is everyone!");
   })
   .fail( function() {
      console.log("Actually, the download failed. Please forget " +
                  "the people I just told you about");
   });
```

In jQuery only one `done` handler is usually added to a request; the whole content is always given so there is only one thing to receive. Under Oboe there will

usually be several separately selected areas of interest inside a JSON document so I anticipate that typically multiple handlers will be added. A shortcut style is provided for adding several selector/handler pairs at a time:

```
oboe("resources/people.json")
   .node({
      "person.name": function(personName, path, ancestors) {
         console.log("Let me tell you about " + name + "...");
      },
      "person.address.town": function(townName, path, ancestors) {
         console.log("they live in " + townName);
      }
   });
```

Note the `path` and `ancestors` parameters in the examples above. These provide additional information regarding the context in which the identified node was found. Consider the following JSON:

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

In this JSON we may extract the runners using the pattern `{name time}` or `medalWinners.*` but nodes alone are insufficient because their location communicates information which is as important as their content. The `path` parameter provides the location as an array of strings plotting a descent from the JSON root to the found node. For example, Bolt has path `['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 in any case has no ancestors, the nodes in the ancestor list will have been only partially parsed.

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

      console.log( person.name + " won the " + lastOf(path) + " medal "
         + "with a time of " + person.time );
   });
```

Being loosely typed, Javascript would not enforce that ternary callbacks are used as selection handlers. Given that before a callback is made the application programmer must have provided a JSONPath selector for the locations in the document she is interested in, for most JSON formats the content alone is sufficient. The API design orders the callback parameters so that in most common cases a unary or binary function can be given.

Using Node.js the code style is more obviously event-based. Listeners are normally added using an `.on` method and the event name is a string 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) {

      console.log( "Well done " + person.name );
   });
```

While allowing both styles creates an API which is larger than it needs to be, creating a library which is targeted at both the client and server side is a ballance between self-consistency spanning environments and consistency *with* the environment, 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 work with the other without difficulty. Implementating the duplicative parts of the API should require only a minimal degree of extra coding because they may be expressed in common using partial completion. Because `'!'` is the JSONPath for the root of the document, for some callback `c`, `.done(c)` is a equal to `.node('!', c)`. Likewise, `.node` is easily expressible as a partial completion of `.on` with `'node'`. Below a thin interface layer may share a commonb implementation.

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

4.5 Earlier callbacks when paths are found prior to nodes

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 can potentially trigger notifications at two points: when it is first addressed and when it is complete. The API facilitates this by providing a `path` event following much the same pattern as `node`.

```
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:
    gui.showMedalTable();
})
.node( "medalWinners.*", function(person, path) {
    let metal = lastOf(path);
    gui.addPersonToMedalTable(person, metal);
})
.fail( function(){
    // That didn"t work. Revert!
    gui.hideMedalTable();
});
```

Implementing path notifications requires little extra code, requiring JSONPath expressions can be evaluated when items are found in addition to when they are completed.

4.6 Choice of streaming data transport

As discussed in section 3.8, current techniques to provide streaming over http encourage a dichotomous split of traffic as either stream or download. I find that this split is not necessary and that streaming may be used as the most effective means of downloading. Streaming services implemented using push pages or websockets are not REST. Under these frameworks a stream has a URL address but data in the stream is not addressable. This is similar to STREST, the *Service Trampled REST* anti-pattern (Cragg 2006), in which http URLs are viewed as locating endpoints for services rather than the actual resources. Being unaddressable, the data in the stream is also uncacheable: an event which is streamed live cannot later, when it is historic, be retrieved from a cache which was populated by the stream. These frameworks use http as the underlying transport but I find they do not follow http's principled design. Because of these concerns, in the browser I will only be supporting downloading using XHR.

Although I am designing Oboe as a client for ordinary REST resources and not focusing on the library a means to receive live events, it is interesting to speculate if Oboe could be used as a REST-compatible bridge to unify live and static data. Consider a REST service which gives the results per-constituency for a UK general election. Requesting historic results, the data is delivered in JSON format much as usual. Requesting the results for the current year on the night of the election, an incomplete JSON with the constituencies known so far would be immediately sent, followed by the remainder sent individually as the results are called. When all results are known the JSON would finally close leaving a complete resource. A few days later, somebody wishing to fetch the results would use the *same url for the historic data as was used on the night for the live data*. This is possible because the URL refers only to the data that is required, not to whether it is current or historic. Because it eventually formed a

complete http response, the data that was streamed is not incompatible with http caching and a cache which saw the data when it was live could store it as usual and later serve it as historic. More sophisticated intermediate caches sitting on the network between client and service recognise when a new request has the same url as an already ongoing request, serve the response received so far, and then continue by giving both inbound requests the content as it arrives from the already established outbound request. Hence, the resource would be cacheable even while the election results are streaming. An application developer programming with Oboe would not have to handle live and historic data as separate cases because the node and path events they receive are the same. Without branching, the code which displays results as they are announced would automatically be able to show historic data.

Taking this idea one step further, Oboe might be used for infinite data which intentionally never completes. In principle this is not incompatible with http caching although more research would have to be done into how current caches handle requests which do not finish. A REST service which serves infinite resources would have to confirm that it is delivering to a streaming client, perhaps with a request header. Otherwise, if a non-streaming REST client were to use the service it would try to get 'all' of the data and never complete its task.

Supporting only XHR as a transport unfortunately means that on older browsers which do not fire progress events (see section 3.7) a progressive conceptualisation of the data transfer is not possible. I will not be using streaming workarounds such as push tables because this would create a client which is unable to connect to the majority of REST services. Degrading gracefully, the best compatible behaviour is to wait until the document completes and then interpret the whole content as if it were streamed. Because nothing is done until the request is complete the callbacks will be fired later than on a more capable platform but will have the same content and be in the same order. By reverting to non-progressive AJAX on legacy platforms application authors will not have to write special cases and the performance should be no worse than with a traditional ajax library such as jQuery. On legacy browsers Oboe could not be used to receive live data because nothing can be read before the request finishes. In the election results example, no constituencies would be shown until they had all been called.

Node's standard http library provides a view of the response as a standard ReadableStream so there will be no problems programming to a progressive interpretation of http. In Node all streams provide a common API regardless of their origin so there is no reason not to allow arbitrary streams to be read. Although Oboe is intended primarily as a REST client, under Node it will be capable of reading data from any source. Oboe might be used to read from a local file, an ftp server, a cryptography source, or the process's standard input.

4.7 Handling transport failures

Oboe cannot know the correct behaviour when a connection is lost so this decision is left to the containing application. Generally on request failure one of two behaviours are expected: if the actions performed in response to data so far remains valid in the absence of a full transmission their effects will be kept and a new request made for just the missed part; alternatively, if all the data is required for the actions to be valid, the application should take an optimistic locking approach and perform a rollback.

4.8 Oboe.js as a Micro-Library

Http traffic is often compressed using gzip so that it transfers more quickly, particularly for entropy-sparse text formats such as Javascript. Measuring a library's download footprint it usually makes more sense to compare post-compression. For the sake of adoption smaller is better because site creators are sensitive to the download size of their sites. Javascript micro-libraries are listed at microjs.com, which includes this project, a library qualifies as being *micro* if it is delivered in 5kb or less, 5120 bytes. Micro-libraries tend to follow the ethos that it is better for an application developer to gather together several tiny libraries than find one with a one-size-fits-all approach, perhaps echoing the unix command line tradition for small programs which each do do exactly one thing. 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 be agnostic as to which other libraries or programming styles it will be combined with. Oboe feels on the edge of what is possible to elegantly do as a micro-library so while the limit is somewhat arbitrary, and keeping below this limit whilst writing readable code provides an interesting extra challenge.

5 Implementation

5.1 Components of the project

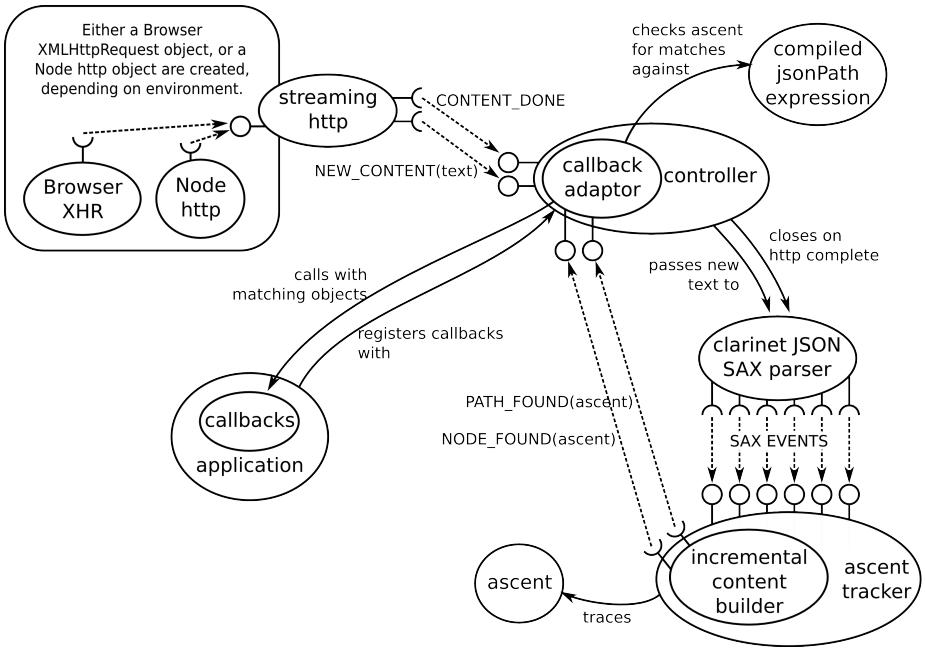

Figure 5: **Major components of Oboe.js illustrating program flow from http transport to application callbacks.** UML facet/receptacle notation is used to show the flow of events and event names are given in capitals. For clarity events are depicted as transferring directly between publisher and subscriber but this is actually performed through an intermediary.

Oboe's architecture describes a fairly linear pipeline visiting a small number of tasks between receiving http content and notifying application callbacks. The internal componentisation is designed primarily so that automated testing can provide a high degree of confidence regarding the correct working of the library. A local event bus facilitates communication inside the Oboe instance and most components interact solely by using this bus; receiving events, processing them, and publishing further events in response. The use of an event bus is a variation on the Observer pattern which removes the need for each unit to locate specific other units before it may listen to their events, giving a highly decoupled shape to the library in which each part knows the events it requires but not who publishes them. Once everything is wired into the bus no central control is required and the larger behaviours emerge as the consequence of interaction between finer ones.

5.2 Design for automated testing

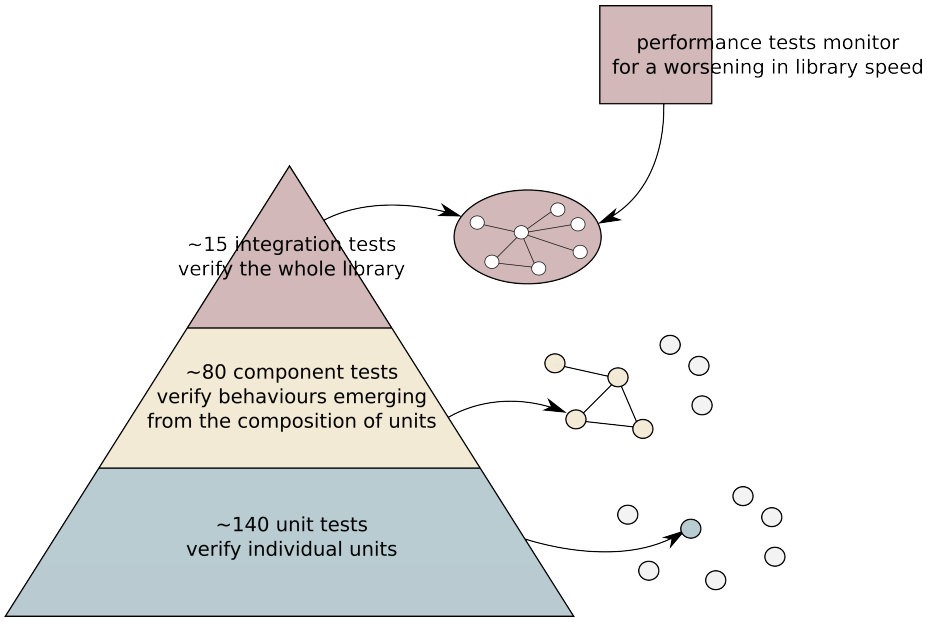

Figure 6: **The test pyramid**. Much testing is done on the low-level components of the system, less on their composed behaviours, and less still on a whole-system level.

80% of the code written for this project is test specification. Because the correct behaviour of a composition requires the correct behaviour of its components, the majority are *unit tests*. The general style of a unit test is to plug the item under test into a mock event bus and check that when it receives input events the expected output events are consequently published.

The *Component tests* step back from examining individual components to a position where their behaviour as a composition may be examined. Because the compositions are quite simple there are fewer component tests than unit tests. The component tests do not take account of *how* the composition is drawn and predominantly examine the behaviour of the library through its public API. One exception is that the streamingXHR component is switched for a stub so that http traffic can be simulated.

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 the same API as is exposed to application programmers. These tests are the most expensive to write but a small number are necessary in order to verify that Oboe works correctly end-to-end. Without access to the internals http traffic cannot be faked so before these tests can be performed a corresponding REST

service is started. This test service is written using Node and returns known content progressively according to predefined timings, somewhat emulating a slow internet connection. The integration tests particularly verify behaviours where platform differences could cause inconsistencies. For example, the test url `/tenSlowNumbers` writes out the first ten natural numbers as a JSON array at a rate of two per second. The test registers a JSONPath selector that matches the numbers against a callback that aborts the http request on seeing the fifth. The correct behaviour is to get no sixth callback, even when running on a platform lacking support for XHR2 and all ten will have already been downloaded.

Confidently black-box testing a stateful unit is difficult. Because of side-effects and hidden state we do not know if the same call will later give a different behaviour. Building up the parse result from SAX events is a fairly complex process which cannot be implemented efficiently as stateless Javascript. To promote testability the state is delegated to a simple state-storing unit. The intricate logic may then be expressed as a separately tested set of side-effect free functions which transition between one state and the next. Although proof of correctness is impossible, for whichever results the functions give while under test, uninfluenced by state I can be confident that they will always yield the same response given the same future events. The separate unit maintaining the state has exactly one responsibility, to hold the parse result between function calls, and is trivial to test. This approach slightly breaks with the object oriented principle of encapsulation by hiding state behind the logic which acts on it but I feel that the departure is justified by the more testable codebase.

To enhance testability Oboe has also embraced dependency injection. Components do not instantiate their dependencies but rather rely on them being passed in by an inversion of control container during the wiring phase. For example, the network component which hides browser differences does not know how to create the underlying XHR that it adapts. Undoubtedly, by not instantiating its own transport this component presents a less friendly interface: it's data source is no longer a hidden implementation detail but exposed as a part of the it's API at the responsibility of the caller. I feel this is mitigated by the interface being purely internal. Dependency injection in this case allows the tests to be written more simply because it is easy to substitute the real XHR for a stub. Unit tests should test exactly one unit, were the streaming http object to create its own transport, the XHR would also be under test, plus whichever external service it connects to. Because Javascript allows redefinition of built in types the stubbing could have potentially also be done by overwriting the XHR constructor to return a mock. However this is to be avoided as it opens up the possibility of changes to the environment leaking between test cases.

5.3 Running the tests

The Grunt task runner is used to automate routine tasks such as executing the tests and building, configured so that the unit and component tests run

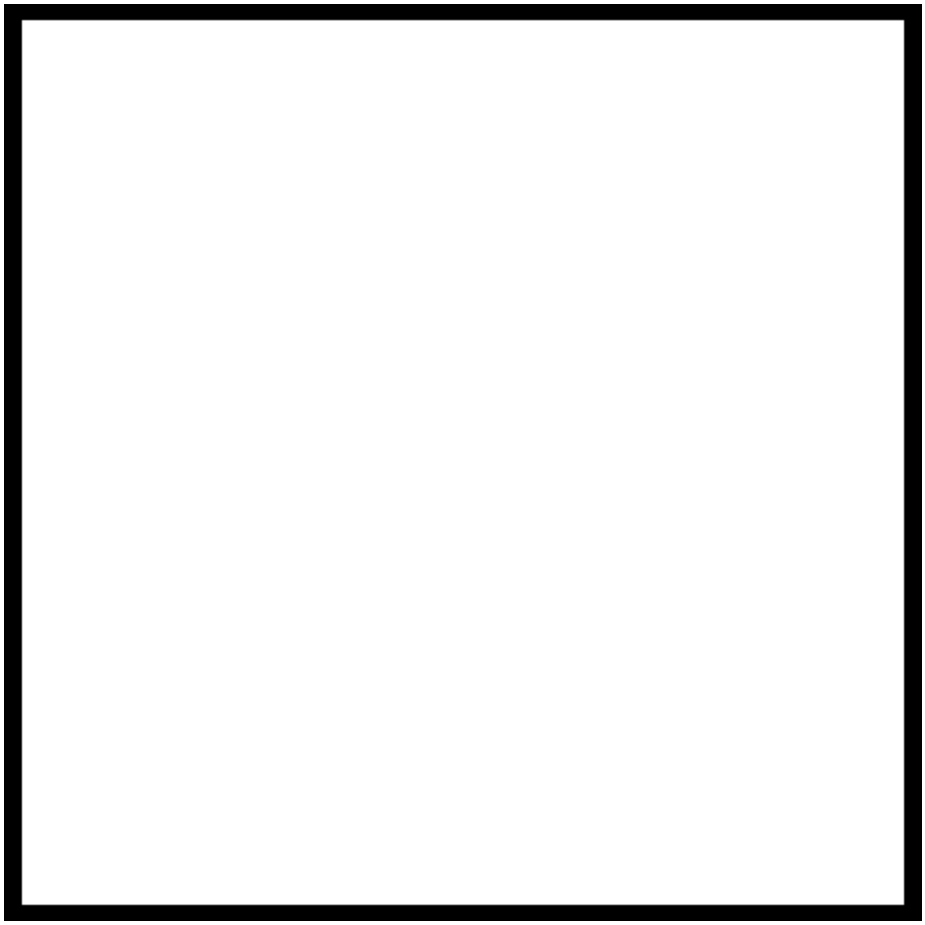

Figure 7: **Relationship between various files and test libraries** *other half of sketch from notebook*

automatically whenever a change is made to a source file or specification. As well as executing correctly, the project is required not to surpass a certain size so this also checked on every save. Because Oboe is a small, tightly focused project the majority of the programming time is spent refactoring already working code. Running tests on save provides quick feedback so that mistakes are found before my mind has moved on to the next context. Agile practitioners emphasise the importance of tests that execute quickly (Martin 2008 p.314:T9) – Oboe's 220 unit and component tests run in less than a second so discovering programming mistakes is almost instant. If the "content of any medium is always another medium" (McLuhan 1964 p.8), we might say that the content of programming is the process that is realised by its execution. A person working in a physical medium sees the thing they are making but the programmer does usually not see their program's execution simultaneously as they create. Conway notes 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," but software is created through intermediate proxies. He attempts to close this gap by merging programming with the results of programming (Conway 2004 pp.8-9). I feel that if we bring together the medium and the message by viewing the result of code while we write it, we can build as a series of small, iterative, correct steps and programming can be more explorative and expressive. Running the tests subtly, automatically hundreds of times per day builds isn't merely convenient, this build process makes me a better programmer.

Integration tests are not run on save. They intentionally simulate a slow network so they take some time to run and I'd already have started the next micro-task by the time they complete. Oboe is version controlled using git and hosted on github. The integration tests are used as the final check before a branch in git is merged into the master.

5.4 Packaging to a single distributable file

As an interpreted language Javascript may be run without any prior compilation. Directly running the files that are open in the editor is convenient while programming but, unless a project is written as a single file, in practice some build phase is required to create an easily distributable form. Dependency managers have not yet become standard for client-side web development so dependant libraries are usually manually downloaded. For a developer wishing to include my library in their own project a single file is much more convenient than the multi-file raw source. If they are not using a similar build process on their site, a single file is also faster to transfer to their users, mostly because the http overhead is of constant size per resource.

Javascript files are interpreted in series by the browser so load-time dependencies must precede dependants. If several valid Javascript files are concatenated in the same order as delivered to the browser, the joined version is functionally equivalent to the individual files. This is a common technique so that code can

be written and debugged as many files but distributed as one. Several tools exist to automate this stage of the build process that topologically sort the dependency graph before concatenation in order to find a suitable script order.

Early in the project I chose *Require.js* for this task. Javascript as a language doesn't have an import statement. Require contributes the importing ability to Javascript from inside the language itself by providing an asynchronous `require` function. Calls to `require` AJAX in and execute the imported source, passing any exported items to the given callback. For non-trivial applications loading each dependency individually over AJAX is intended only for debugging because making so many requests is slow. For efficient delivery Require also has the `optimise` command which concatenates an application into a single file by using static analysis to deduce a workable source order. Because the `require` function may be called from anywhere, this is undecidable in the general case so Require falls back to lazy loading. In practice this isn't a problem because imports are generally not subject to branching. For larger webapps lazy loading is a feature because it speeds up the initial page load. The technique of *Asynchronous Module Definition* (AMD) intentionally imports rarely-loaded modules in response to events; by resisting static analysis the dependant Javascript will not be downloaded until it is needed. AMD is mostly of interest to applications with a central hub but also some rarely used parts. For example, most visits to online banking will not need to create standing orders so it is better if this part is loaded on-demand rather than increase the initial page load time.

I hoped to use Require's `optimise` to automate the creation of a combined Javascript file for Oboe. Oboe would not benefit from AMD because everybody who uses it will use all of the library but using Require to find a working source order would save having to manually implement one. Unfortunately this was not feasible. Even after optimisation, Require's design necessitates that calls to the `require` function are left in the code and that the Require run-time component is available to handle them. At more than 5k gzipped this would have more than doubled Oboe's download footprint.

After removing Require I decided to pick up the simplest tool which could possibly work. With about 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 and a distributable library requiring no run-time dependency management to be loaded.

For future consideration there is Browserify. This library reverses the 'browser first' Javascript mindset by viewing Node as the primary target for Javascript development and adapting the browser environment to match. Browserify converts applications written for Node into a single file packaged for delivery to a web browser. Significantly, other than Adaptors wrapping the browser APIs and presenting their features as if they were the Node equivalents, Browserify

leaves no trace of itself in the final Javascript. Additionally, the http adaptor[4] is capable of using XHRs as a streaming source when used with supporting browsers.

After combining into a single file Javascript source can be made significantly smaller by *minification* techniques such as reducing scoped symbols to a single character or deleting the comments. For Oboe the popular minifier library *Uglify* was chosen. Uglify performs only surface optimisations, concentrating mostly on producing compact syntax by manipulating the code's abstract syntax tree. I also considered Google's *Closure Compiler* which resembles a traditional optimiser by leveraging a deeper understanding of the code semantics. Unfortunately, proving equivalence in highly dynamic languages is often impossible and Closure Compiler is only safe given a well-advised subset of Javascript. It delivers no reasonable guarantee of equivalence if code is not written as the Closure team expected. Integration tests would catch any such failures but for the time being I decided that even given the micro-library limits, a slightly larger file is a worthwhile tradeoff for a safer build process

5.5 Styles of Programming

Oboe does not follow any single paradigm and is written as a mix of procedural, functional and object-oriented programming styles. Classical object orientation is used only so far as the library exposes an Object-oriented public API. Although Javascript supports them, classes and constructors are not used, nor is there any inheritance or notable polymorphism. Closures form the primary means of data storage and hiding. Most entities do not give a Javascript object on instantiation, they are constructed as a set of event handlers with access to shared values from a common closure. As inner-functions of the same containing function, the handlers share access to variables from the containing scope. From outside the closure the values are not only protected as private as would be seen in an OO model, they are inherently unaddressable.

Although not following an established object orientated metamodel, the high-level componentisation hasn't departed very far from what I would make were I following that style and OO design patterns have influenced their layout considerably. If we wished to think in terms of the OO paradigm we might say that values trapped inside closures are private attributes and that the handlers registered on the event bus are public methods. In this regard, the high-level internal design of Oboe can be discussed using the terms from a more standard object oriented metamodel.

Even where it creates a larger deliverable library I have generally preferred writing as short functions which are combined to form longer ones. Writing shorter functions reduces the size of the minimum testable unit which, because each test can test a very small unit of functionality, encourages very simple unit

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

tests. Because the tests are simple there is less room for unanticipated cases to hide. Due to pressures on code size I decided not to use a general purpose functional library and created my own with only the parts that are needed. See functional.js (Appendix p.67). Functional programming in Javascript is known to be slower than other styles, particularly in Firefox which lacks optimisations such as Lambda Lifting (Guo 2013). I do not think this should be a major problem. Because of its single-threaded execution model, in the browser any Javascript is ran during script execution frames, interlaced with frames for other concurrent concerns. To minimise the impact on other concerns such as rendering it is important that no task occupies the CPU for very long. Since most monitors refresh at 60Hz, about 16ms is a fair target for the maximum duration of a script frame. In Node no limit can be implied from a display but any CPU-hogging task degrades the responsiveness of concurrent concerns. Switching tasks is cheap so sharing the CPU well generally prefers many small execution frames over a few larger ones. Whether running in a browser or server, the bottleneck is more often I/O than processing speed; providing no task contiguously holds the CPU for an unusually long time an application can usually be considered fast enough. Oboe's progressive model favours sharing because it naturally splits the work over many execution frames which by a non-progressive mode would be performed during a single frame. Although the overall CPU time will be higher, Oboe should share better with other concerns and because of better I/O management the total system performance should be improved.

5.6 Incrementally building the parsed content

As shown in figure 5 on page 43, there is an *incremental content builder* and *ascent tracer* which handle SAX events from the Clarinet JSON parser. By presenting to the controller a simpler interface than is provided by Clarinet, taken together these might be considered as an Adaptor pattern, albeit modified to be even-driven rather than call-driven: we receive six event types and in response emit from a vocabulary of two, `NODE_FOUND` and `PATH_FOUND`. The events from Clarinet are low level, reporting the sequence of tokens in the markup; those emitted are at a much higher level of abstraction, reporting the JSON nodes and paths as they are discovered. Testing a JSONPath expression for a match against any particular node requires the node itself, the path to the node, and the ancestor nodes. For each newly found item in the JSON this information is delivered as the payload of the two event types emitted by the content builder. When the callback adaptors receive these events they have the information required to test registered patterns for matches and notify application callbacks if required.

The path to the current node is maintained as a singly linked list in which each item holds the node and the field name that links to the node from its parent. Each link in the list is immutable, enforced in newer Javascript engines by using

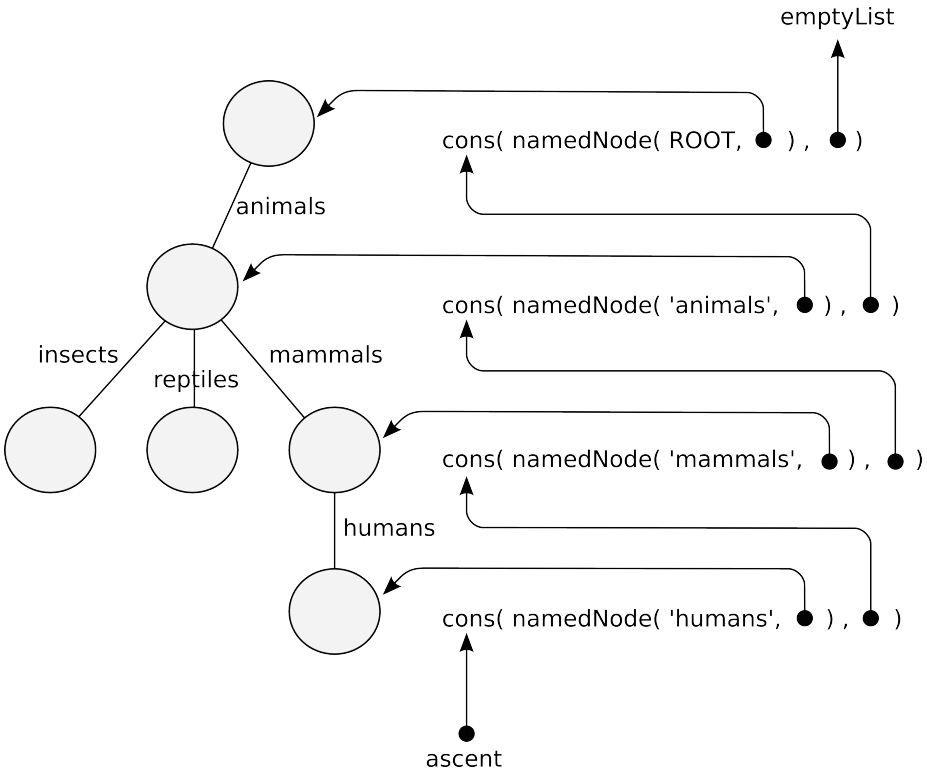

Figure 8: List representation of an ascent rising from leaf to root through a JSON tree. Note the special ROOT value which represents the location of the pathless root node. The ROOT value is an object, taking advantage of object uniqueness to ensure that its location is unequal to all others.

frozen objects.[5] The list is arranged as an ascent with the current node at the near end and the root at the far end. Although paths are typically written as a *descent*, ordering as an *ascent* is more efficient because every SAX event can be processed in constant time by adding to or removing from the head of the list. For familiarity, where paths are passed to application callbacks they are first reversed and converted to arrays.

For each Clarinet event the builder provides a corresponding handler which, working from the current ascent, returns the next ascent after the event has been applied. For example, the `objectopen` and `arrayopen` event types are handled by adding a new item at the head of the ascent but for `closeobject` and `closearray` one is removed. Over the course of parsing a JSON resource the ascent will in this way be manipulated to visit every node, allowing each to be tested against the registered JSONPath expressions. Internally, the builder's handlers for SAX events are declared as the combination of a smaller number of basic reusable parts. Several of Clarinet's event types differ only by the type of the node that they announce but the builder is largely unconcerned regarding a JSON node's type. On picking up `openobject` and `openarray` events, both pass through to the same `nodeFound` function, differing only in the type of the node which is first created. Similarly, Clarinet emits a `value` event when a string or number is found in the markup. Because primitive nodes are always leaves the builder treats them as a node which instantaneously starts and ends, handled programmatically as the composition of the `nodeFound` and `nodeFinished` functions.

Although the builder functions are stateless and side-effect free, while visiting each JSON node the current ascent needs to be stored. This is handled by the ascent tracker which serves as a holder for this data. Starting with the ascent initialised as an empty list, on receiving a SAX event it passes the ascent to the handler and stores the result so that when the next SAX event is received the updated ascent can be given to the next handler.

For the ascents linked lists were chosen in preference to the more conventional approach of using native Javascript Arrays for several reasons. Firstly, I find the program more easy to test and debug given immutable data structures. Employing native Arrays without mutating would be very expensive because on each new path the whole array would have to be copied. Secondly, while debugging, unpicking a stack trace is easier if I know that every value revealed is the value that has always occupied that space and I don't have to project along the time axis by imagining which values were in the same space earlier or will be later. Thirdly, the lack of side effects means that I can try 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

[5] 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, we can be fairly certain that the code does not mutate these objects or the tests would fail with attempts to modify in environments which are able to enforce it.

and shrinking so for collections whose size changes often, arrays are relatively inperformant. Finally, lists are a very convenient format for the JSONPath engine to match against as will be discussed in the next section. The Javascript file lists.js (Appendix p.93) implements various list functions: `cons`, `head`, `tail`, `map`, `foldR`, `all`, `without` as well as providing conversions to and from arrays.

5.7 Oboe JSONPath Implementation

On the first commit the JSONPath implementation was little more than a series of regular expressions[6] but has slowly evolved into a featureful and efficient implementation. The extent of the rewriting was possible because the correct behaviour is well defined by test specifications[7]. The JSONPath compiler exposes a single higher-order function. This function takes the JSONPath as a string and, proving it is a valid expression, returns a function which tests for matches to the pattern. The type of is difficult to express in Javascript but expressed as Haskell would be:

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

The match result is either a hit or a miss. If a hit, the return value is the node captured by the match. Should the pattern have an explicitly capturing clause the node corresponding to that clause is captured, otherwise it is the node at the head of the ascent. Implementation as a higher-order function was chosen even though it might have been simpler to create a first-order version as seen in the original JSONPath implementation:

```
(String, Ascent) -> JsonPathMatchResult
```

This version was rejected because the pattern string would have to be freshly reinterpreted on each evaluation, repeating computation unnecessarily. Because a pattern is registered once but then evaluated perhaps hundreds of times per JSON file the most pressing performance consideration is for matching to execute quickly. The extra time needed to compile a pattern when new application callbacks are registered is relatively insignificant because it is performed much less often.

The compilation is performed by recursively by examining the left-most side of the string for a JSONPath clause. For each clause type there is a function

[6] 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 for contrast, the current source can be found in the appendix on page 81 or at https://github.com/jimhigson/oboe.js/blob/master/src/jsonPath.js.

[7] 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.

which tests ascents for that clause, for example by checking the field name; by partial completion the field name function would be specialised to match against one particular name. Having generated a function to match against the left-most clause, compilation continues recursively by passing itself the remaining unparsed right-side of the string. The recursion continues until the terminal case where there is nothing left to parse. On each recursive call the clause function generated wraps the result from the last recursive call, resulting ultimately in a concentric series of clause functions. The order of these functions mirrors the ordering of paths as an ascent, so that the outermost function matches against the node at the near end of the ascent, and the innermost against the far end. When evaluated against an ascent, each clause function examines the head of the list and, if it matches, passes the list onto the next function. A special clause function, `skip1` is used for the `.` (parent) syntax and places no condition on the head of the list, unconditionally passing the tail on to the next clause, thus moving matching on to the parent node. Similarly, there is a function `skipMany` which maps onto the `..` (ancestor) syntax and recursively consumes the minimum number of ascent items necessary for the next clause to match or fails if this cannot be done. In this way, we peel off layers from the ascent as we move through the function list until we either exhaust the functions, indicating a match, or cannot continue, indicating a fail.

This JSONPath implementation allows the compilation of complex expressions into an executable form by combining many very simple functions. As an example, the pattern `!.$person..{height tShirtSize}` once compiled would resemble the Javascript functional representation below:

```
statementExpr(              // outermost wrapper, added when JSONPath
                            //    is zero-length
   duckTypeClause(          // token 5, {height tShirtSize}
      skipMany(             // token 4, '..', ancestor relationship
         capture(           // token 3, '$' from '$person'
            nameClause(     // token 3, 'person' from '$person'
               skip1(       // token 2, '.', parent relationship
                  rootExpr // token 1, '!', matches only the root
               )
            'person' )
         )
   ), ['height', 'tShirtSize'])
)
```

Since I am using a side-effect free subset of Javascript for pattern matching 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 left side they could share the inner part of their functional expression. Given the patterns `!.animals.mammals.human` and `!.animals.mammals.cats`, the JSONPath engine will currently create two

identical evaluators for `!.animals.mammals`. Although Javascript doesn't come with functional caching, it can be added using the language itself, probably the best known example being `memoize` from Underscore.js. I suspect, however, that hashing the cache 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.

Functions describing the tokenisation of the JSONPath language are split out to their own source file and tested independently of the compilation. Regular expressions are used because they are the simplest form able to express the clause patterns. Each regular expressions starts with ^ so that they only match at the head of the string, the 'y' flag would be a more elegant alternative but as of now this lacks wider browser support[8]. By verifying the tokenisation functions through their own tests it is simpler to create thorough specification because the tests may focus on the tokenisation more clearly without having to observe its results though another layer. For JSONPath matching we might consider the unit test layer of the test pyramid (figure 6 p.44) is split further into two sub-layers. Arguably, the upper of these sub-layers is not a unit test because it is verifying more than one unit, the tokeniser and the compiler, and there is some redundancy since the tokenisation is tested both independently and through a proxy. However, a more purist approach would not be any more useful because stubbing out the tokeniser functions before testing the compiler would be a considerable effort and I do not believe it would improve the rigor of the JSONPath specification.

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

6 Conclusion

6.1 Differences in the programs written using Oboe.js

I find it quite interesting that a program written using Oboe.js in the most natural way will be subtly different from one written using JSON.parse, even if the programmer is trying to express the same thing. Consider the two examples below which use Node.js to read from a local file and write to the console.

```
oboe( fs.createReadStream( '/home/me/secretPlans.json' ) )
   .on('node', {
      'schemes.*': function(scheme){
         console.log('Aha! ' + scheme);
      },
      'plottings.*': function(deviousPlot){
         console.log('Hmmm! ' + deviousPlot);
      }
   })
   .on('done', function(){
      console.log("*twiddles mustache*");
   })
   .on('fail', function(){
      console.log("Drat! Foiled again!");
   });


fs.readFile('/home/me/secretPlans.json', function( err, plansJson ){
   if( err ) {
      console.log("Drat! Foiled again!");
      return;
   }
   var plans = JSON.parse(err, plansJson);

   plans.schemes.forEach(function( scheme ){
      console.log('Aha! ' + scheme);
   });
   plans.plottings.forEach(function(deviousPlot){
      console.log('Hmmm! ' + deviousPlot);
   });

   console.log("*twiddles mustache*");
});
```

The first observation is that the explicit looping in the second example requires roughly the same amount of code as the pattern registration that serves as an

analogue in the first. The first example has a more declarative style whereas the second is more imperative. If two levels of selection were required, such as `schemes.*.premise`, other than a longer JSONPath pattern the first example would not grow in complexity whereas the second would require an additional loop inside the existing loop. We can say that the complexity of programming using Oboe stays roughly constant whereas in a more traditional style it grows linearly with the number of levels which must be traversed.

While the intended behaviours are very similar, the unintended accidental side-behaviours differ between the two examples. In the first the order of the output for schemes and plans will match the order in the JSON, whereas for the second scheming is always done before plotting. In the second example the order could be easily changed by reversing the statements whereas in the first to change the order would require a change in the order of the JSON. Whether the programmer has been liberated to ignore order or restricted to be unable to easily change it probably depends on the situation. The error behaviours are also different. The first example will print until it has an error. The second will print if there are no errors.

In the second example it is most natural to check for errors before output whereas in the first it feels most natural to register an error listener as the last of the chained calls. I prefer the source order in the first because the the normal case is listed before the abnormal one emphasises their roles as main and secondary behaviours.

Finally, the timings of the printing will be different. The first code prints the first output in constant time regardless of the size of the file and then outputs many lines individually as the JSON is read. The second will print all output at once after a time proportional to the size of the input.

6.2 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.[9] 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	Good	984	1,064	9,0
JSON.parse	Poor	2550	2,609	8.9
Clarinet	Good	34	781	5.5
Clarinet	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[10] 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

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

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

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.

6.3 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	30	lots!

Oboe was the shortest:

```
oboe(DB_URL).node('{id url}.url', function(url){
   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. The code below is shortened by using the get-json[11] package which removes the need to explicitly parse:

```
getJson(DB_URL, function(err, records) {
   records.data.forEach( function( record ){
      if( record.url ) {
         getJson(record.url, function(err, record) {
            console.log(record.name);
         });
      }
```

[11] Https://npmjs.org/package/get-json.

```
   });
});
```

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.

6.4 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/ms)
Curl (control)	42ms	*n/a*
Chrome 31.0.1650.34 (Mac OS X 10.7.5)	84ms	9.57
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 otherwise 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.

During the project a new version of Chrome more than doubled the node throughput dur to including later version of the v8 Javascript engine. Node also uses v8 and should be updated to this version soon.

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.

6.5 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.

6.6 potential future work

Although the project delivers improvements already, the most obvious expansion to fully realise the potential would be a matching server-side component that writes JSON in a streaming way. So far this has required that the JSON be written out as strings but this scales badly as messages become more complex.

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. Plug-ins for formats.

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[12]. 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.

[12] 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

```
/**
 * 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

```
/**
 * 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++;

function errorReport(statusCode, body, error) {
   try{
      var jsonBody = JSON.parse(body);
   }catch(e){}

   return {
      statusCode:statusCode,
      body:body,
      jsonBody:jsonBody,
      thrown:error
   };
}
```

8.3 functional.js

```
/**
 * Partially complete a function.
 *
 * Eg:
 *    var add3 = partialComplete( function add(a,b){return a+b}, 3 );
 *
 *    add3(4) // gives 7
 *
 *
 *    function wrap(left, right, cen){return left + " " + cen + " " + right;}
 *
 *    var pirateGreeting = partialComplete( wrap , "I'm", ", a mighty pirate!" );
 *
 *    pirateGreeting("Guybrush Threepwood");
 *                          // gives "I'm Guybrush Threepwood, a mighty pirate!"
 */
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 emits 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 emit
 * function given.
 */
function incrementalContentBuilder( emit ) {


   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,
      emit( 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
```

```
 * 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
                           );

   emit( PATH_FOUND, ascentWithNewPath);

   return ascentWithNewPath;
```

```
}


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

   emit( 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, []);
   },
```

```
        // 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( nodeFinished, nodeFound ),

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

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(  emit, on, un,
                              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) {
   emit(
      ERROR_EVENT,
      errorReport(undefined, undefined, 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 handler( 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 ) {

         if( !notifyCallback(callback, maybeMatchingMapping, ascent) ) {

            un(eventId, handler);
         }
      }
```

```
   });
}

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)),
       keep        = true;

   oboeApi.forget = function(){
      keep = false;
   };

   callback( nodeOf(matchingMapping), path, ancestors );

   delete oboeApi.forget;

   return keep;
}

function protectedCallback( callback, context ) {
   return function() {
      try{
         callback.apply(context||oboeApi, arguments);
      }catch(e)  {

         // An error occured during the callback, publish it on the event bus
         emit(ERROR_EVENT, errorReport(undefined, undefined, 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,
         protectedCallback(callback, callbackContext)
      );
   } else {
      addListenersMap(eventId, jsonPathOrListenerMap);
   }

   return this; // chaining
}

var addDoneListener = partialComplete(addNodeOrPathListenerApi, NODE_FOUND, '!'),
    addFailListner = partialComplete(on, ERROR_EVENT);

/**
 * implementation behind oboe().on()
 */
function addListener( eventId, listener ){

   if( eventId == NODE_FOUND || eventId == PATH_FOUND ) {

      apply(arguments, addNodeOrPathListenerApi);

   } else if( eventId == 'done' ) {

      addDoneListener(listener);

   } else if( eventId == 'fail' ) {
```

```
            addFailListner(listener);
        }

        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    :  addListener,
        fail  :  addFailListner,
        done  :  addDoneListener,
        abort :  partialComplete(emit, 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){
```

```
      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) {
```

```
   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,
                                            /!/
                                          )
```

```
        //   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

```
/**
 * 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)))
 **/
```

```
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
            ;
}
```

```
/**
 * 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
            ;
}

/**
 * Return a list like the one given but with the first instance equal
 * to item removed
 */
function without(list, item) {

  return list
            ?  ( head(list) == item
                     ? tail(list)
                     : cons(head(list), without(tail(list), item))
               )
            : emptyList
            ;
}

/**
 * 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
   function reverseInner( list, reversedAlready ) {
      if( !list ) {
         return reversedAlready;
      }

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

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

8.9 pubSub.js

```
/**
 * Isn't this the cutest little pub-sub you've ever seen?
 *
 * Over time this should be refactored towards a Node-like
 *    EventEmitter so that under Node an actual EE acn be used.
 *    http://nodejs.org/api/events.html
 */
function pubSub(){

   var listeners = {};

   return {

      on:function( eventId, fn ) {

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

         return this; // chaining
      },

      emit:varArgs(function ( eventId, parameters ) {

         each(
            partialComplete( apply, parameters ),
            listeners[eventId]
         );
      }),

      un: function( eventId, handler ) {
         listeners[eventId] = without(listeners[eventId], handler);
      }
   };
}
```

8.10 publicApi.js

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

function apiMethod(httpMethodName, mayHaveRequestBody) {

   return function(firstArg) {

      if (firstArg.url) {

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

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

         // 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
         );
      }
   };
}
```

8.11 streamingHttp-browser.js

```
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} emit 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' 'PATCH' '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(emit, 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 ) {
      emit( 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 emitted 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();

            emit( END_OF_CONTENT );
         } else {

            emit(
               ERROR_EVENT,
               errorReport(
                  xhr.status,
                  xhr.responseText
               )
            );
         }
      }
   };

   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 emit an event here.
      // Node's streaming http adaptor receives the error as an asynchronous
      // event rather than as an exception. If we emitted 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(emit, ERROR_EVENT,
             errorReport(undefined, undefined, e)
         )
      , 0
      );
   }
}
```

8.12 streamingHttp-node.js

```
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} emit 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' 'PATCH' 'DELETE'
 * @param {String} contentSource the url to make a request to, or a stream to read from
 * @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(emit, on, http, method, contentSource, data, headers) {

   function readStreamToEventBus(readableStream) {

      // use stream in flowing mode
      readableStream.on('data', function (chunk) {

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

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

         emit( END_OF_CONTENT );
      });
   }

   function readStreamToEnd(readableStream, callback){
      var content = '';

      readableStream.on('data', function (chunk) {
```

```
        content += chunk.toString();
    });

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

        callback( content );
    });
}

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

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

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

    req.on('response', function(res){
        var statusCode = res.statusCode,
            sucessful = String(statusCode)[0] == 2;

        if( sucessful ) {

            readStreamToEventBus(res)

        } else {
            readStreamToEnd(res, function(errorBody){
                emit(
                    ERROR_EVENT,
                    errorReport( statusCode, errorBody )
                );
            });
        }
    });

    req.on('error', function(e) {
        emit(
            ERROR_EVENT,
```

```
            errorReport(undefined, undefined, e )
         );
      });

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

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

      req.end();
   }

   if( isString(contentSource) ) {
      fetchUrl(contentSource);
   } else {
      // contentsource is a stream
      readStreamToEventBus(contentSource);
   }

}
```

8.13 util.js

```
/**
 * 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, contentSource, body, headers){

   var eventBus = pubSub();

   streamingHttp( eventBus.emit, eventBus.on,
                  httpTransport(),
                  httpMethodName, contentSource, body, headers );

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

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);
```

```
      });

      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

```
/**
 */

"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 fake db 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);
```

```
            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.

Cragg, Duncan. 2006. “Duncan Cragg on Declarative Architectures: STREST (Service-Trampled REST) Will Break Web 2.0.” http://duncan-cragg.org/blog/post/strest-service-trampled-rest-will-break-web-20/.

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

Eberhart, Andreas, and Stefan Fischer. 2002. *Java Tools: Using XML, EJB, CORBA, Servlets and SOAP*.

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.”

Goessner, Stefan. 2007. “JSONPath - XPath for JSON.” http://goessner.net/articles/JsonPath/.

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{]}.

Martelli, Alex. 2000. “Discussion of typing in Python language, 2000.” https://groups.google.com/forum/?hl=en\T1\textbackslash{}#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ.

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/.