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

Jim Higson, Kellogg College

November 2013

Supervised by Jeremy Gibbons

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering.

Abstract

A new design is presented for HTTP client libraries which incorporates 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 library 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 of a resource while 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 considered with regards to developer ergonomics and user perception of performance.

Contents

8 Appendix iii: Benchmarking 123

9 Bibliography 130

List of Figures

1 Introduction

HTTP was originally designed for the transfer of hypertext documents. REST (Fielding 2000) introduces no fundamentally new methods but extends the scope of HTTP to include the transfer of arbitrary data. Whereas the rival technology SOAP (Gudgin et al. 2007) largely disregards HTTP's principled design by adopting the protocol as a transport on which to bootstrap its own semantics, REST adopts all of HTTP's core phrasing. This includes the HTTP methods for fetching, creating and modifying resources: GET, POST, PUT, PATCH, DELETE, and the locating of resources using URLs. Under HTTP's original design hierarchical URLs are used to locate documents without reference to the services which produce them. REST advances this same naming strategy by likewise using URLs to locate data resources, not services. As with HTTP, REST is stateless and therefore cacheable, allowing large-scale content distribution networks to be built. Because HTTP's inbuilt headers for content type, language negotiation, and resource expiry are used according to the originally intended meanings (Fielding et al. 1999), existing intermediaries such as load balancing proxies, gateways, and caches need make no special accommodation for REST resources.

Despite REST adopting the mechanisms and semantics of HTTP, whereas documents received over the web are often interpreted in a streaming fashion, to date REST resources are not commonly examined in this way. For most practical cases where we wish to increase the speed of a system there is no reasonable distinction between acting *earlier* and being *quicker*. In the interest of creating efficient software we should use data at the first possible opportunity: examining content *while it streams* rather than holding it unexamined until it is wholly available. The purpose of this dissertation is to explore tangible benefits which may be realised if we fold HTTP streaming into the REST paradigm.

Natural languages encourage our thinking to follow patterns that they comfortably support (Whorf 1956). This idea has been applied to programming, for example the language Ruby is intentionally designed to discourage global variables by using a less attractive notation (Yukihiro 2003). It may be useful when exploring new techniques to question which established constructs are as they are because of languages which unintentionally suggest that formulation; it is perhaps significant that REST clients tend to style the calling of remote resources similarly to the call style of the host programming language. In practice one of two schemas is generally followed: a synchronous, blocking style in which an invocation halts execution for the duration of the request before evaluating to the fetched resource; or an asynchronous, non-blocking form in which some logic is specified to be applied to the response once it is available. Languages which promote concurrency though threading generally consider blocking in a single thread to be acceptable and will prefer the synchronous mode whereas languages with first class functions are naturally conversant in callbacks and will prefer asynchronous I/O. In programming the language limits the patterns that we readily see and the schemes which map most easily onto our languages are not necessarily the best possible organisation. 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 phasing encourages statements to yield single, wholly evaluated results it is comfortable to conceptualise the REST response as a discrete event. This establishment of a 'limiting comfort' extends to graphical notations such as UML whose constructs strongly reflect the textual languages of the day. UML sequence diagrams include the syntax for instantaneously delivered return values but, despite being commonly used to draw network data transfer, provide no corresponding notation for a resource whose data is progressively revealed.

No new computing techniques need be invented before this dissertation can be implemented. 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 parts together to produce a streaming interpretation. Every current web browser already implements a similar pattern. Load any complex webpage – essentially an aggregation of hypertext and other resources – and the HTML will be parsed and displayed incrementally while it is downloading, with resources such as images requested in parallel as soon as they are referenced. In the case of progressive JPEG or SVG[1] the images themselves will also be presented incrementally. The incremental display is achieved through specialised programming which applies only to displaying web pages. The contribution of this dissertation is to provide a generic analogue, applicable to any problem domain.

1.1   REST aggregation could be faster

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 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 is further to our advantage. Appendix i lists some actual limits.

---

[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 1: **Sequence diagram showing the aggregation of low-level REST resources by an intermediary.** 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 until the response is complete.
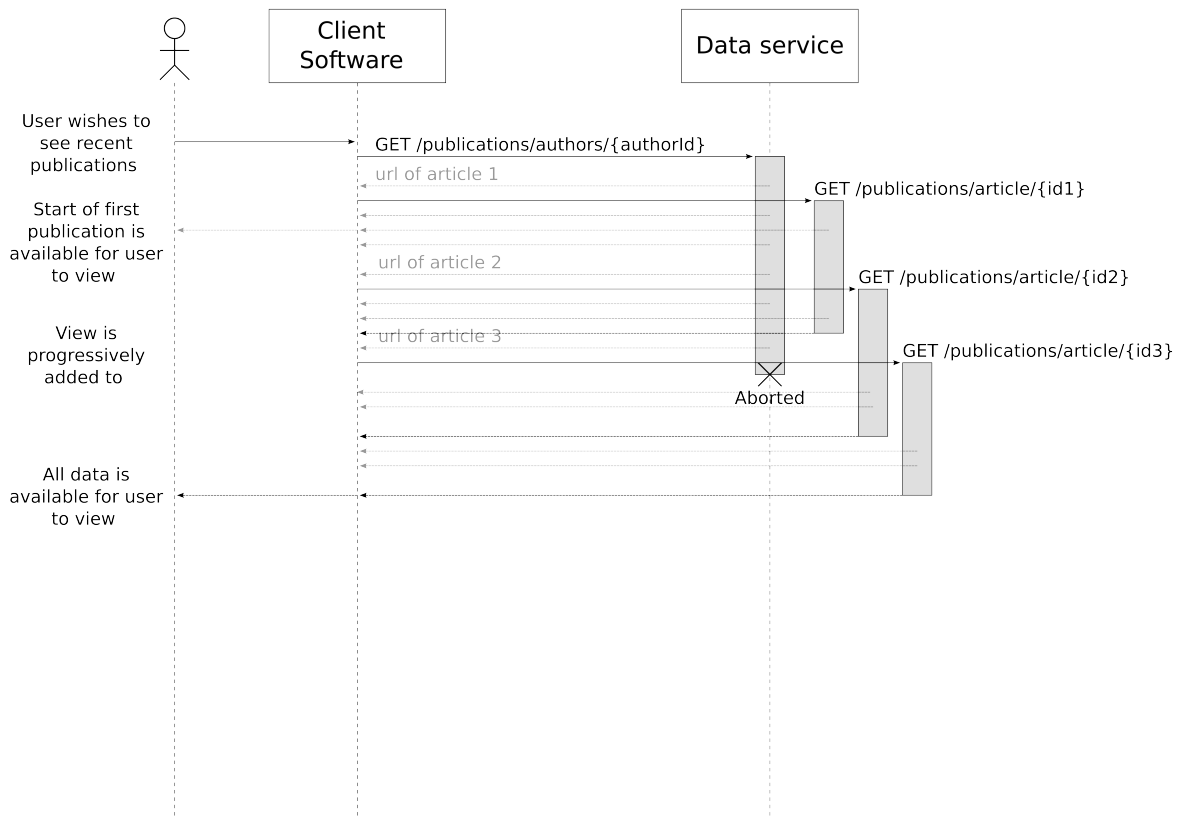
Figure 2: **Revised aggregation sequence showing a client which progressively interprets the resources.** Because UML sequence diagrams draw returned values as a one-off happening rather than a continuous process, the lighter arrow notation is added to represent fragments of an incremental response. Each request for an individual publication is made as soon as its URL can be extracted from the publications list and once all required links have been read from the original response it is aborted rather than continuing to download unnecessary data.

Nodes in an n-tier architecture defy categorisation as 'client' or 'server' in a way that 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 1 and 2 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 same 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 to the client able to interpret it progressively and, even if it is not, the overall delivery remains faster. Whilst HTTP servers capable of streaming are quite common, there seems to be no published general-purpose, streaming-receptive REST client library.

1.2 Stepping outside the big-small tradeoff

Where a domain model contains a series of data from which continuous ranges are requestable via REST there is often a tradeoff in the client design with regards to how much should be requested with 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 is maintained, albeit punctuated with pauses whenever new content is loaded. There is a tradeoff in the presentation layer between sporadically requesting many items and frequently requesting a few. At one extreme the interface would occasionally falter for a longer time, whereas at the other it would pause momentarily but with greater regularity.

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 and the interval before the interface starts updating.

1.3 Staying fast on a fallible network

REST operates over networks whose reliability varies widely. On unreliable networks connections are abruptly dropped and existing HTTP clients handle unexpected terminations wastefully. 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 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 can be extracted as raw text but handling them involves writing a special case and is not possible using standard parsers which are not amenable to incomplete markup. Because of this difficulty the canonical webapp drops partial messages without inspection. For the user checking her email, even if 90% of her inbox had been retrieved before the signal was lost, the web application will behave as if it received none and show her nothing. Later, when the network is available the inbox will be downloaded from scratch including the 90% which has already been successfully delivered. The system would be more user friendly and efficient if it used the 90% from the first request when it arrives and fetched only the lost 10% when network connectivity returns.

The delivered part of a partially successful message may be used if we turn away from this polarised view of wholly successful and unsuccessful requests. A message may be conceptualised as having many parts which are useful in themselves, in which the successful delivery of each part is handled independently before knowing if the next part will also arrive. As well as allowing partially successful messages to be used, seeing the resource as a stream of small parts allows those parts to be used earlier if they are made available to the application while the streaming is ongoing. 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 the transfer later fail.

1.4 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 and that

the components are loosely coupled and autonomous, frequent releases of any particular sub-system should pose no 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 often is hampered by tools which encourage programming against rigidly specified formats. If a data consumer is tightly coupled to the format it consumes it will resist changes to the programs which produce data in that format. As an anecdote, working in enterprise I have 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 REST clients which are able to use a resource whilst being only loosely coupled to the overall shape of the message.

1.5   Deliverables

To avoid feature creep the scope of the software deliverables is pared down to the smallest work which can be said to realise the goals of the project. 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). Given reasonable time constraints it would not be feasible to deliver a full stack while concentrating on software quality so we are obliged to focus on solutions which interoperate with existing deployments. To a third party wishing to adopt the technology it is more inviting to add small enhancements to the existing architecture than to action a shift which requires wholesale change.

Although an explicitly streaming server would improve the situation further, because all network data transfer may be thought of as a stream, it is not required to start taking advantage of progressive REST. A streaming client is the MVP and is the programming deliverable for this project.

1.6   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 application responsiveness. Because applications in the target domain are much more I/O-bound than CPU-bound, optimisation in terms of the execution time of algorithms will be de-emphasised unless especially egregious. The delivered library should allow looser coupling to the format of consumed resources than is possible with the current best tools so that it is less

disruptive when services are upgraded.

2 Background
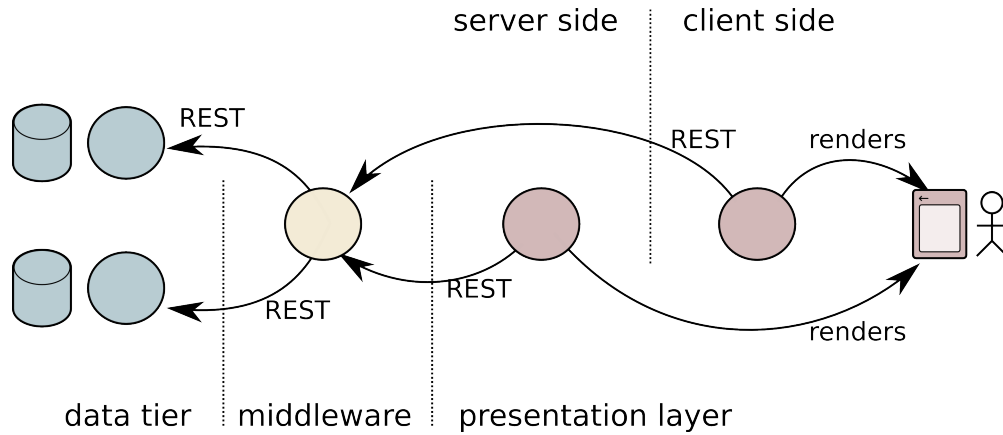
Figure 3: **Labelling nodes in an n-tier architecture**. Although network topology is often split about client and server side, for our purposes categorisation as data, middle, and presentation tier is the more meaningful distinction. According to this split the client- and server-side presentation layer serve the same purpose, generating mark-up based on aggregated data prepared by the middle tier.

2.1 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 the preferred application platform for all but the most specialised interfaces, the web perpetuates this narrative by resisting categorisation as either mode.

While the trend is generally towards more client scripting and for many sites a Javascript runtime 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 also provides progressive enhancements. Although it does not generate pages 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. In 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 REST client libraries may fit into the picture. REST is used by the presentation layer to pull data from the middle tier regardless of where the presentation resides. Likewise, rather than connect to databases directly, for portability the middle tier will often communicate with a thin REST layer which wraps the data store. This suggests three uses:

- From web browser to middle tier
- From server-side presentation layer to middle tier
- From middle tier to nodes in the data tier

Fortunately, each of these contexts requires a similar performance profile. The work done is computationally light and answering a request involves more time waiting than processing. As a part of an interactive system low latency is important whereas throughput can be increased relatively cheaply by adding more hardware, especially in a cloud hosted environment. As demand for the system increases the total work required grows but the complexity in responding to any one of the requests remains constant. Although serving any particular request might be done in series, the workload as a whole is embarrassingly parallelisable.

2.2 Node.js

Node.js (Ryan 2009) is a general purpose tool for executing Javascript outside of a browser. It has the aim of low-latency I/O and is used mostly for server applications and command line tools. It is difficult to judge to what degree Javascript is a distraction from Node's design as a tool for I/O and to what degree the language defines the platform.

For 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. Safely programming threads to share mutable objects requires great care and experience, otherwise the programmer is liable to create race conditions. Consider for example a Java HTTP aggregator; because we wish to fetch in parallel each 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 while the other requesters complete. Each thread consumes considerable resources but during its multi-second lifespan requires only a fraction of a millisecond on the CPU. It is unlikely that any two requests will return closely enough in time to be processed in parallel, shedding threading's chief advantage, that it may process simultaneously utilising multiple cores. Even if requests do return proximately, the actual CPU time required in making an HTTP request is so short that any concurrent processing is a pyrrhic victory.

Node builds on a model of event-based, asynchronous I/O which was established by browser Javascript execution. 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 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 a task 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 available 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 in which the tasks do not complete quickly. In production environments Node deployments usually take advantage of multiple cores by creating one Node instance per processor code. The separate instances act independently and do not normally use shared RAM.

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 the storage 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 any more difficult an expression than if blocking I/O were used. It is certainly harder to

make mistakes programming in this way than managing synchronised access to mutable objects that are shared between threads.

```javascript
function printResourceToConsole(url) {

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

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

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

            response.on('data', function(chunk) {
                // This function is called each time some data is
                // received from the HTTP request. The task writes
                // the response to the console and quickly exits.
                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 using a unified data streaming interface regardless of the source. The streams fit comfortably with the wider event-driven model by implementing Node's EventEmitter interface, a generic dispatcher capable of supporting any event type. Although the abstraction provided by streams is quite a thin layer on top of the host system's sockets, 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));
   });
```

Following Node's lead, traditionally thread-based environments are beginning to embrace asynchronous, single-threaded servers. The Netty project (Netty 2010) can be thought of as roughly an equivalent of Node for the Java Virtual Machine.

2.3   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 markup can be traced to document formats, JSON's lineage is in a programming language. From these roots it isn't surprising that JSON maps more directly to the metamodels that most programmers think in. XML parsers produce items such as Element, Text, Attribute, 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. The JSON nodes: *strings*, *numbers*, *objects* and *arrays* will in many cases map directly onto 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', townOrCity:'London'},
```

```
        {name: 'Jack', townOrCity:'Bristol'}
        {townOrCity:'Cambridge', name: 'Sally'}
    ]
}
```

Both JSON and XML are used to serialise orderless constructs but while expressed
as text the encoding is inevitably written according to some serialisation order. XML
specifically states that the order of attributes is not significant (Bray et al. 2008),
JSON has no such detailed specification but a similar order insignificance seems to be
implied by the JSON object's likeness to Javascript objects whose iteration order is
indeterminate (ECMA 1999, 4.3.3). In the example above the people objects would
probably have been written based on either a class with two public properties or a
hash map. On receiving this data the text would be demarshalled into similar orderless
structures and it would be quickly forgotten that the data found an ordered expression
during transport. When viewing a document as a stream and interpreting while still
incomplete it is easier to mistakenly react differently according to field order. If nodes
from the example above were used when only the first field has arrived Sally would find
a different handling than John or Jack. 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
gives different results depending on the order that fields are received.

2.4   Common patterns for connecting to REST services

For languages such as Javascript or Clojure which use 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 in any language
and some marshaling is required. In more strongly typed OO languages such as Java
or C#, JSON's classless, relatively freeform objects are less convenient. To smoothly
integrate the example JSON from the previous section, instances of a domain model
Person class with methods such as `getName()` and `getLocation()` would have to be
initialised, representing the remote 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.
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 an additional layer programmatically instantiates

the local domain model objects. DTOs are more necessary when using XML. Reading resources encoded as JSON we might say that the JSON objects are already 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, having received the response model the developer will usually start by locating the pertinent parts of the response by drilling down into its structure using the programming language itself.

```java
// 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() );
   }
}
```

```javascript
// 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 of this method for locating resource parts is that the code making the inspection is 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 changed to a fuller address structure as a street-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 should perhaps be seen as undesirable in relation to format structures as it would be elsewhere. DTOs limit the spread of refactoring inside the client because only the translation from DTO to domain object must be updated but do not avoid change altogether if a service format is refactored. In the *Red Queen's race* it took "all the running you can do, to keep in the same place". Ideally a programmer should only have to expend effort so that their code does something new, or perform

better something that it already did, not to stay still. Following an object oriented encapsulation of data such that a caller does not have to concern itself 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 difficult to later follow a narrative of updates to the logic expressed by the program. It is therefore harder to later understand the thinking behind a change or the reason for the change.

2.5 Little languages for descending into fetched data

To address the problem of drilling down to pertinent fragments of a message without tightly coupling to the format, consider if instead of programmatically descending step-by-step, a language were used which allows the right amount of specificity to be given regarding which parts to select. In VirtualStudio LINQ (Microsoft 2013), which is based on lambda-calculus and resembles SQL, is used to drill-down into data structures and may also modify the data that is found. However this style of programming requires the application developer to write significantly more code than in the simple programmatic access above so it does not meet the aims of this project.

Given model interrogation logic which is tightly coupled so that the model's structure cannot change, one suggested approach is Adaptive OOP (Lieberherr, Silva-Lepe, and Cun 1994) in which no detailed class structure is committed to when constructing the object oriented program. A REST resource could be dynamically configured into an OO model according to a formal specification in a specialisation that is capable of answering the desired queries. The model that is constructed would be sufficient to answer the queries without the programmer having to suppose any rigid form.

Certain markup languages come with 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, JSONPath's syntax resembles the equivalent Javascript:

```
// 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 notation which is not present
// in Javascript:
let betterTownSelector = "people[2]..town"
```

Consider the resource below:

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

The JSONPath `people.*..town` may be applied against the above JSON and would
continue to select correctly if the system were refactored to the version 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: 'Sally'
      }
   ]
}
```

formats tend to grow, not shrink

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 format is liable to switch between when being refactored. 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 can't be said to form a single n-way tree. XML attributes may only contain strings and have a lesser expressivity than child nodes which allow recursive structure; it is a common refactor to change from attributes to elements when a scalar value is upgraded to a compound. XPath selectors written in the most natural way do not track this change.

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

The XPath `//person@town` identifies the town in the XML above but because of the switch from attribute to sub-element fails in the revised version below.

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

Reflecting its dual purpose for marking up 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 model 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'. Inheriting from its programming language ancestry, in JSON there is no ambiguity. The space between tokens is never significant.

Programming against a changing service is always going to present a moving target but it would be easier to miss with XPATH than with JSONPath. In the JSON metamodel each node has only one, unambiguous set of children so the format author is not given a choice of logically equivalent features that must be addressed through different mechanisms. If a scalar value is updated to a compound only the node changes, the addressing of the node is unaffected.

Generally in descriptive hierarchical data there is a trend for ancestorship to signify the same relationship regardless of the number of intermediate generations. In the example above, `town` transitioned from a 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. This trend does not hold for every way that message semantics may be built 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": [
      {
         "name":     {"isProbably":"Bob"},
         "location": {"isNearTo":"Birmingham"}
      }
   ]
}
```

Considering the general case, it will not be possible to safely track every refactoring. By necessity a resource consumer should limit their ambitions to tracking ontology expansions which do not change the meanings of 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 a present and known future version would remain a valuable tool because it decouples the consumer and provider update schedules, removing the need for the client to march perfectly in sync with the service.

2.6   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. 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 progress 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 abstractions over the underlying platform differences they could be written purposefully and programmers were able to express more complex ideas.

Today JSON is generally the preferred format, especially for resources transmitted to client-side web applications. 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 integrate JSON parsers, providing a call style for working with remote resources so streamlined as to require hardly any additional effort.

```javascript
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.

   console.log('the first person is called', people[0].name);
});
```

---

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

2.7 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 available browser abilities. When the call style above was developed the most popular browser barred access to in-progress responses so the inevitable conceptualisation was drawn of the response as a one-time event with no accommodation provided for progressively delivered data.

The followup standard, XHR2 is now at Working Draft stage (van Kesteren 2012). 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.

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, and video.

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.

2.8 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. This kind of streaming intersects only narrowly with the aims of the XHR2 progress event. Whereas XHR2 enables downloads to be viewed as 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 although a streaming download might be used to make its display more responsive, the 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. On the client a page containing a table is hidden in an off-screen iframe. The frame's content is served as an HTML page containing a table 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 to the table which is then noticed by Javascript monitoring the iframe on the client. More recently, **Websockets** provides a standardised streaming transport on top of HTTP's chunked mode. Websockets requires browser implementation and cannot be retrofitted to older browsers through Javascript. It is 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 mechanisms 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 nonetheless text based and a person may take a peek inside the system's plumbing simply by observing the traffic at a particular URL. For push tables, because the transport is based on a presentation format, an actual table of the event's properties may be viewed from a browser as the messages are streamed.

2.9   Parsing: SAX and DOM

From the XML world two standard parser types exist, SAX and DOM, with DOM by far the more popular. Both styles of parsers are also available for JSON. DOM performs a parse as a single evaluation and returns an object model representing the whole of the document. Conversely, SAX parsers are probably better considered as enhanced tokenisers, providing a very low-level event driven interface that notifies the programmer of each token separately as it is found. Working with DOM's level of abstraction the markup syntax is a distant concern whereas for SAX each element's opening and closing must be noted so the developer may not put the data's serialisation aside. SAX comes with the advantages that it may read a document progressively and has lower memory requirements because it does not store the parsed tree. It is a popular choice for embedded systems running on constrained hardware and may be used 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 the Javascript Clarinet SAX parser, 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 required to maintain this state.
        inPeopleArray = false,
        inPersonObject = false,
        inNameAttribute = false,
        found = false;

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

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

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

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

---

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

```
    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 onto the programmer the responsibility for managing state regarding the current position in the document and storing data extracted from previously seen nodes. This maintenance of state tends to be programmed once per usage rather than assembled as the composition of reusable parts. The ordering of the code under SAX is also 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, its unfriendly developer ergonomics have presented too high a barrier for adoption for all but fringe use cases.

3   Design and Reflection

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. The main tool to achieve this is the SAX parser whose model presents poor developer ergonomics because it is not usually convenient to think on the markup's level of abstraction. 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. It is possible 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, 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. This approach combines most of the desirable properties from SAX and DOM parsers into a new, hybrid method.

The interesting parts of a document may be identified before it is complete if we turn 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, the locating logic currently found inside the callback can be extracted, expressed as a selector language, and used to declare the cases in which the callback should be notified. The callback will receive complete fragments from the response once they have been selected according to this declaration.

Javascript will be used to implement the software deliverables because it has good support for non-blocking I/O and covers both environments where this project will be most useful: web browser and web server. Focusing on the MVP, parsing will only be implemented for one mark-up language. Although this technique could be applied to any text-based, tree-shaped markup, JSON best meets the project goals because it is widely supported, easy to parse, and defines a single n-way tree, making it more amenable to selectors which span multiple format versions.

JSONPath is well suited for selecting nodes while the document is being 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 we 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 a streaming context.

Given that we are selecting specifically inside a REST resource it is unlikely that we will be examining a full model. Rather, the selectors will be applied to a subset that we requested and was assembled on our behalf according to parameters that we supplied. We can expect to be interested in all of the content belonging to a particular category 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 the existing language is followed somewhat loosely, specialising the matching by adding features which are likely to be useful when detecting entities in REST resources while avoid unnecessary code by dropping others. Later adding new features to a language is easier than removing them once a userbase has built up so where the utility of a feature is not clear the default position should be to not include it. It is difficult to anticipate all real-world matching requirements but it should be possible to identify a core 20% of features that are likely to be useful in 80% of cases. For the timebeing any functionality which is not included may be implemented by registering a more permissive selection and then further filtering programmatically from inside the callback. Patterns of programmatic filtering which arise from use in the wild can later be mined and added to the selection language.

3.1 Detecting types in JSON

As seen in the 'all books' example above, it is intuitive to support identifying sub-trees according to a categorisation by higher-level types. 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
  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' on two nodes prior to examining their contents because of the field name which maps to them from 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 as 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 markup, `addresses.*` would correctly identify three address nodes. The pluralisation of field names such as 'address' becoming 'addresses' is common when marshaling from OO languages because the JSON keys are extracted from getter names which reflect the method's cardinality: `public Address getAddress()` or `public List<Address> getAddresses()`. To identify members of a type held singularly or plurally it might help if a system that understands natural language pluralisation such as Ruby on Rails were investigated. Unions were also considered as a simpler solution, resembling `address|addresses.*`. It was decided that until the usefulness is better demonstrated, with no obvious best solution, it is simplest to handle plurals outside of the JSONPath language by expecting the programmer to register 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. Instead the idea of *duck typing* was imported from Python, 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)

An address 'duck-definition' for the above JSON would say that any object which has number, street, and town properties is an address. Applied to JSON, duck typing takes an individualistic approach by deriving type from the node itself rather than the situation in which it is found. As discussed in section 2.5, 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 Javascript notation is that for object literals so a derivative duck-type syntax was created 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 it is often useful if subtypes, 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 resources marshaled from an OO representation. To conform to a duck-type a node must have all of the required fields but could also have any others.

3.2 Importing CSS4's explicit capturing to 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 CSS has previously 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.

CSS4 selector capturing will be incorporated into this project's JSONPath implementation. By duplicating a syntax which the majority of web developers should become familiar with over the next few years the learning curve should appear more gradual. Taking on this feature, the selector `person.$address.town` would identify an address node with a town child, or `$people.{name, dob}` can be used to locate the same 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 the integration is more convenient with explicit capturing because we can hand over the collection as it expands.

3.3 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 (Job 2011) 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, itself named after the **SAX**ophone.

3.4 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 widely adopted 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 be self-contained and fit within a fairly short timeframe.

jQuery is by far the most popular library for AJAX today. The basic call style for making a 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" );
   });
```

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

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

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

This pattern of passing arguments as object literals is common in Javascript for functions which take a large number of parameters, 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 given at the call site, avoids an anti-pattern where a call may only be understood after counting the position of the arguments.

Taking on this style while extending it to incorporate progressive parsing, we arrive at the following API:

```
oboe("resources/people.json")
   .node( "person.name", function(name) {
      console.log("There is somebody called", name);
   })
   .done( function( wholeJson ) {
```

```
        console.log("That is everyone!");
    })
    .fail( function() {
        console.log("There might may be more people but",
                    "we do not know who they are yet.");
    });
```

In jQuery the whole content is given back at once so usually only one `done` handler is added to a request. Under Oboe each separately addressed area of interest inside the JSON resource requires its own handler so it is helpful to provide a shortcut style for adding several selector-handler pairs at a time.

```
oboe("resources/people.json")
    .node({
        "person.name": function(personName, path, ancestors) {
            console.log("You will hear about", name, "...");
        },
        "person.address.town": function(townName, path, ancestors) {
            console.log("...they live in", townName);
        }
    });
```

Note the `path` and `ancestors` parameters in the example 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 lists the ancestors starting with the JSON root node and ending at the immediate parent of the found node. For all but the root node, which in any case has no ancestors, the nodes given by the ancestor list will have been only partially parsed.

```
oboe("resources/someJson.json")
   .node( "medalWinners.*", function(person, path) {
      let metal = lastOf(path);
      console.log( person.name, "won the", metal,
         "medal with a time of ", person.time );
   });
```

Being loosely typed, Javascript does not enforce that ternary callbacks are used as selection handlers. Before a callback is made the application programmers must have provided a JSONPath selector specifying the locations in the document that they are interested in. The programmer will already be aware enough of the node location so for most JSON formats the content alone will be sufficient, the API purposefully orders the callback parameters so that in most cases a unary function can be given.

Under Node.js the code style is more obviously event-based. Listeners are normally added using an .on method where the event name is a string given as the first argument. Adopting this style, Oboe's API design 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, the dual interface is designed to encourage adoption on the client and server side. The two styles are similar enough that a person familiar with one should be able to work with the other without difficulty. Implementing the duplicative parts of the API should require only a minimal degree of extra coding because they may be expressed in common and specialised 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'.

When making PUT, POST or PATCH requests the API allows the body to be given as an object and serialises it as JSON because it is expected that REST services which emit JSON will also accept it.

```
oboe.doPost("http://example.com/people", {
   "body": {
      "name":"Arnold", "location":"Sealands"
   }
});
```

3.5 Earlier callbacks when paths are found prior to nodes

Following 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. Under Oboe each node found in the JSON document can potentially trigger notifications at two stages: when it is first addressed and when it is complete. The API facilitates this by providing a `path` event following much the same style as `node`.

```
oboe("events.json")
   .path( "medalWinners", function() {
      // We do not know the winners yet but we know we have some
      // so we can start drawing the table:
      gui.showMedalTable();
   })
   .node( "medalWinners.*", function(person, path) {
      let metal = lastOf(path);
      gui.addPersonToMedalTable(person, metal);
   })
   .fail( function(){
      // Revert!
      gui.hideMedalTable();
   });
```

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

3.6 Choice of streaming data transport

As discussed in section 2.8, current techniques to provide streaming over HTTP encourage a dichotomous split of traffic as either stream or download. This split is not a necessary consequence of the technologies used and streaming may instead be viewed as the most efficient means of downloading. Streaming services implemented using push pages or websockets are not REST. Under these frameworks a stream has a URL but the 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. Like SOAP, these frameworks use HTTP as the underlying transport but do not follow HTTP's principled design.

Although Oboe is not designed for live events, it is interesting to speculate whether it could be used as a REST-compatible bridge to unify live-ongoing feeds with ordinary REST resources. Consider a REST service which gives per-constituency results for UK general elections. If historic results are requested 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 dispatched 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 forms a complete HTTP response, the data that was streamed is not incompatible with HTTP caching and a cache which saw the data while it was live could later serve it from cache as historic. More sophisticated caches located between client and service would 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 and a service would only have to provide one stream to serve the same live data to multiple users fronted by the same cache. 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 well current caches handle requests which do not finish. A REST service which provides infinite length 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 (section 2.7) a progressive conceptualisation of the data transfer is not possible. Streaming workarounds such as push tables will not be used because they would result in 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, an application author will not have to write special cases and the performance should be no worse than with traditional AJAX libraries such as jQuery. On legacy browsers Oboe could not be used to receive live data – in the election night example no constituencies would be shown

until they had all been called.

One benefit of a unified model for streamed and finite-size content is that it allows a simpler security model. Because the demands of the transport are different, streaming security is usually implemented separately from other HTTP requests. Schneier often argues that "complexity is the worst enemy of security" (Schneier 2000 Software Complexity and Security) and in one online debate paints a buildings analogy (Schneier and Harris 2013):

> More specifically, simplicity tends to completely remove potential avenues of attack. An easy example might be to think of a building. Adding a new door is an additional complexity, and requires additional security to secure that door. This leads to an analysis of door materials, lock strength, and so on. The same building without that door is inherently more secure, and requires no analysis or assumptions about how it will be secured. Of course, this isn't to say that buildings with doors are insecure, only that it takes more work to secure them. And it takes more work to secure a building with ten doors than with one door.

Unifying two means of data transfer into a single model is analogous to a building having only one entrance. A better level of security should be possible given the same effort taken to secure it.

Node's standard HTTP library provides a view of the response as a standard ReadableStream so there will be no problems programming to a streaming interpretation of HTTP. In Node because all streams provide a common API regardless of their origin allowing arbitrary sources to be read is no extra work. 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.

3.7   Handling transport failures

Oboe cannot know the correct behaviour when a connection is lost so this decision is left to the containing application. On failure one of two behaviours is expected: if the actions performed in response to data so far remain 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 rollback.

3.8 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. When measuring a library's download footprint it usually makes more sense to compare post-compression. Smaller is better to encourage adoption 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, but micro-libraries also tend to follow the ethos that it is better for an application developer to gather together several tiny, simple libraries than find a complex one which aims to solve many problems. As well as being small, a micro-library should impose as few restrictions as possible on its use and be agnostic as to which other libraries or programming styles it will be combined with, echoing the UNIX philosophy for small, easily combined programs (McIlroy 1978).

> This is the Unix philosophy:
> Do one thing and do it well.
> Write programs to work together.
> Write programs to handle text streams, because that is a universal interface.

4 Implementation

4.1 Componentisation of the project
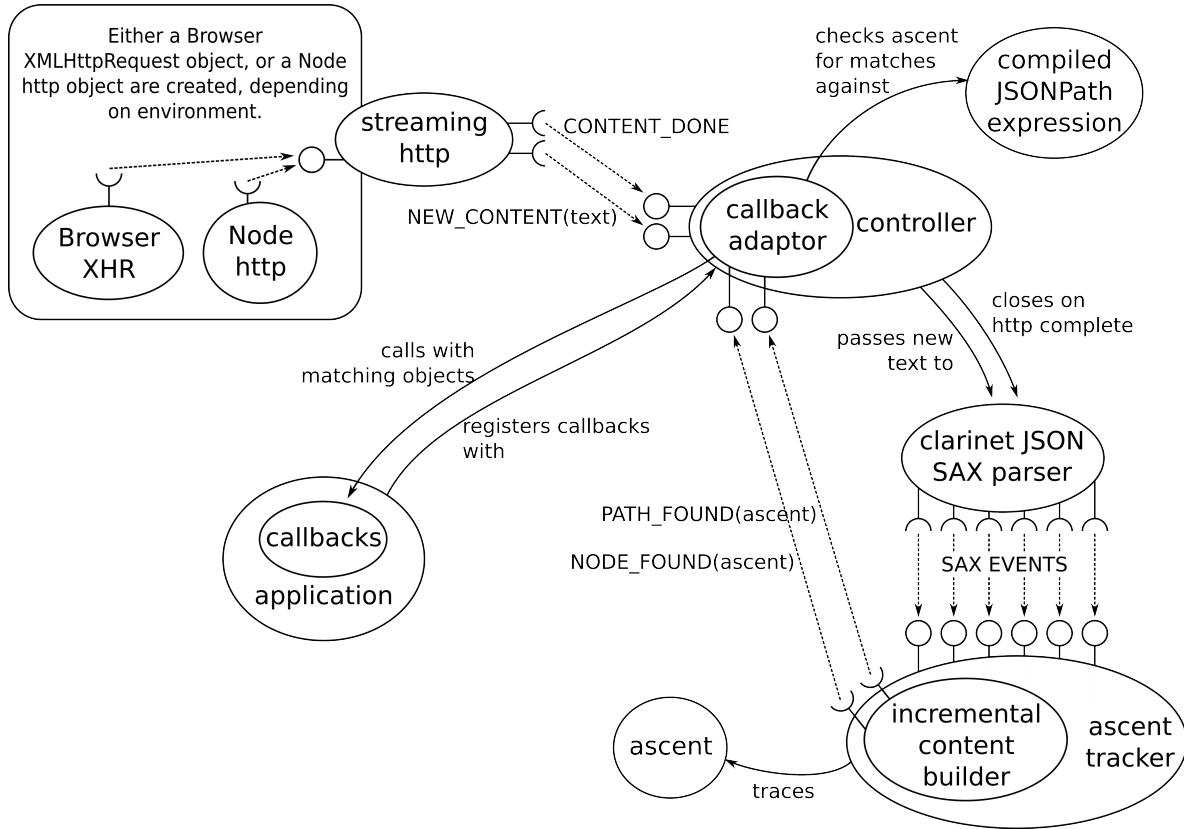
Figure 4: **Major components of Oboe.js illustrating program flow from HTTP transport to application callbacks.** UML facet/receptacle notation is used to show the flow of events and event names are given in capitals. For clarity events are depicted as transferring directly between publisher and subscriber 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 an 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 removes the need for each unit to locate other units before it may listen to their output, 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

43

emerge as a consequence of the interactions between finer ones.

4.2   Design for automated testing
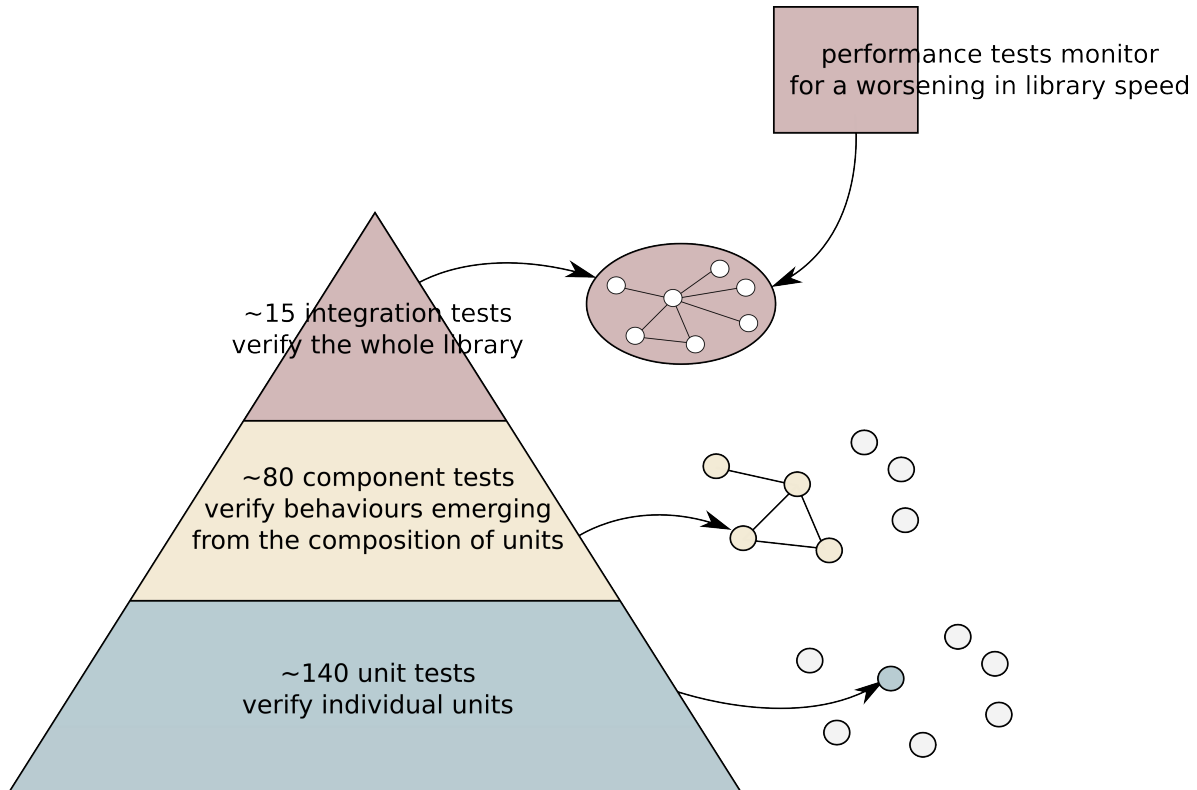
Figure 5: **The test pyramid**. Many tests specify the low-level components, fewer on their composed behaviours, and fewer still on a whole-system level.

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 certain input events the expected output events are consequently published.

The *Component tests* step back from examining individual components to a position where their behaviour in 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. HTTP traffic cannot be faked without access to the internals so before these tests are 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 four 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 where all ten will have already been downloaded.

Confidently black-box testing a stateful unit is difficult. Because of side-effects and hidden state we can only rely on inductive reasoning to say that similar future calls won't later result in different behaviours. Building up the parse result from SAX events is a fairly complex process which cannot be implemented efficiently using wholly side-effect free 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. For whichever results the functions give while under test, uninfluenced by state one may be confident that they will later yield the same result if given the same input. The separate unit to maintain the state has exactly one responsibility, to hold the incremental parse output 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 the departure will be justified if a more testable codebase promotes greater reliability.

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: its data source is no longer a hidden implementation detail but exposed as a part of its API as the responsibility of the caller. This disadvantage is mitigated by the interface being purely internal. Dependency injection 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 also been 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.

4.3   Running the tests

The Grunt task runner (Alman 2012) is used to automate routine tasks such as executing the tests and building, configured so that the unit and component tests run 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 the programmer is thinking about 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 nearly 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 proxies. He attempts to close this gap by merging programming with the results of programming (Conway 2004 pp.8-9). If we bring together the medium and the message by viewing the result of code while we write it, we can build in 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 is not merely time-saving, this build process noticeably improved the quality of the project's programming.

Integration tests are not run on save. They intentionally simulate a slow network so by the time they complete a programmer will have already context-switched to the next micro-task. Oboe's source 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.

4.4   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 Oboe in their own project a single file is much more convenient than the multi-file raw source. If they are not using a similar

46

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 request.

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 Require.js (Burke 2011) was chosen for this task. Javascript does not have an import statement; Require adds one from inside the language as 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 provides 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 when a safe concatenation order cannot be found Require falls back to lazy loading. In practice this is no problem because imports are not generally subject to branching. For larger webapps lazy loading is actually 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.

Require's `optimise` was originally chosen to automate the creation of a combined Javascript file for Oboe. Oboe would not benefit from AMD because everybody who uses it will need all of the library but using Require to find a working source order would avoid 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. With about 15 source files and a fairly sparse dependency graph, finding a working order on paper proved a simpler task than integrating with tools offering to automate the process. After finding a Grunt plugin analogous to the unix `cat` command it was trivial to create a build process which produces a distributable library while requiring no dependency management code to be loaded at run-time.

For future consideration there is Browserify (Browserling Inc. 2012). 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 browser APIs in the call style of their Node equivalents, Browserify leaves no trace of itself in the final Javascript. When run on browsers supporting progress events Browserify's HTTP adaptor[4] presents XHRs using Node's streaming interface so it should be capable of adapting the Node version of Oboe to run under modern browsers.

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 (Bazon 2010) was chosen. Uglify performs surface optimisations, rearranging the syntax tree locally to create a compact expression. Consider the code below:

```javascript
// If the 'cached' flag is set, add a query parameter '_' with the value
// of the current timestamp to the url. This guarantees the the request
// will not be served from browser cache

if( cached === false ) {

   if( baseUrl.indexOf('?') == -1 ) {
      baseUrl += '?';
   } else {
      baseUrl += '&';
   }

   baseUrl += '_=' + new Date().getTime();
}
return baseUrl;
```

Uglify compresses this example into a single statement, reducing the size by about 55%.

```javascript
return b === !1 && (a += -1 == a.indexOf("?") ?
"?" : "&", a += "_=" + (new Date).getTime()), a
```

---

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

An alternative minifier would be Google Closure Compiler (Google 2009), a more sophisticated project which leverages a deeper understanding of the programming it reduces. 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 it was decided that even constrained by micro-library size limits, a slightly larger file is a worthwhile tradeoff for a safer build process.

4.5   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 OO 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 attached to the central bus which, as inner-functions inside the same outer function, share access to values caught in a common closure. From outside the closure the values are not only private as would be seen in a Java-style OO model, they are inherently unaddressable.

Although not following an established object-oriented metamodel, the high-level componentisation has not departed very far from how the project might be divided following that style and OO design patterns have influenced the 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 final deliverable, short functions that can be combined to form longer ones have been generally preferred. Writing a program using short functions reduces the size of the minimum testable unit and because each test specifies a very small unit of functionality, encourages the writing of very simple unit tests. When the tests are simple there is less room for unanticipated cases to hide. Due to pressures on code size a general purpose functional library was not chosen, one was created containing only the necessary functions (functional.js, Appendix p.69). Functional programming in Javascript is known to be slower than other styles, particularly in Firefox which lacks optimisations such as Lambda Lifting (Guo 2013) but the effect should be insignificant, particularly when considered alongside the performance advantages that streaming I/O offers. Because of its single-threaded execution model, in the browser any Javascript is run 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 browser script frame. In Node no limit can be implied from a display but any CPU-hogging task degrades the responsiveness of concurrent work. Switching tasks is cheap so effective CPU sharing 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 the processor more cooperatively and because of better I/O management the overall system responsiveness should be improved.

4.6   Incrementally building the parsed content

As shown in figure 4 on page 43, there is an *incremental content builder* and *ascent tracer* which handle SAX events from the Clarinet JSON parser. Taken together these components might be considered as an Adaptor (Gamma et al. 1994 p.139) that wraps Clarinet with a simpler interface, albeit a modified version of the pattern which is event-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 received 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. The list and the items it contains are immutable, enforced in newer Javascript engines by using 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.

---

[5]See   https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global/_Objects/Object/freeze. Although older engines provide no facility 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.
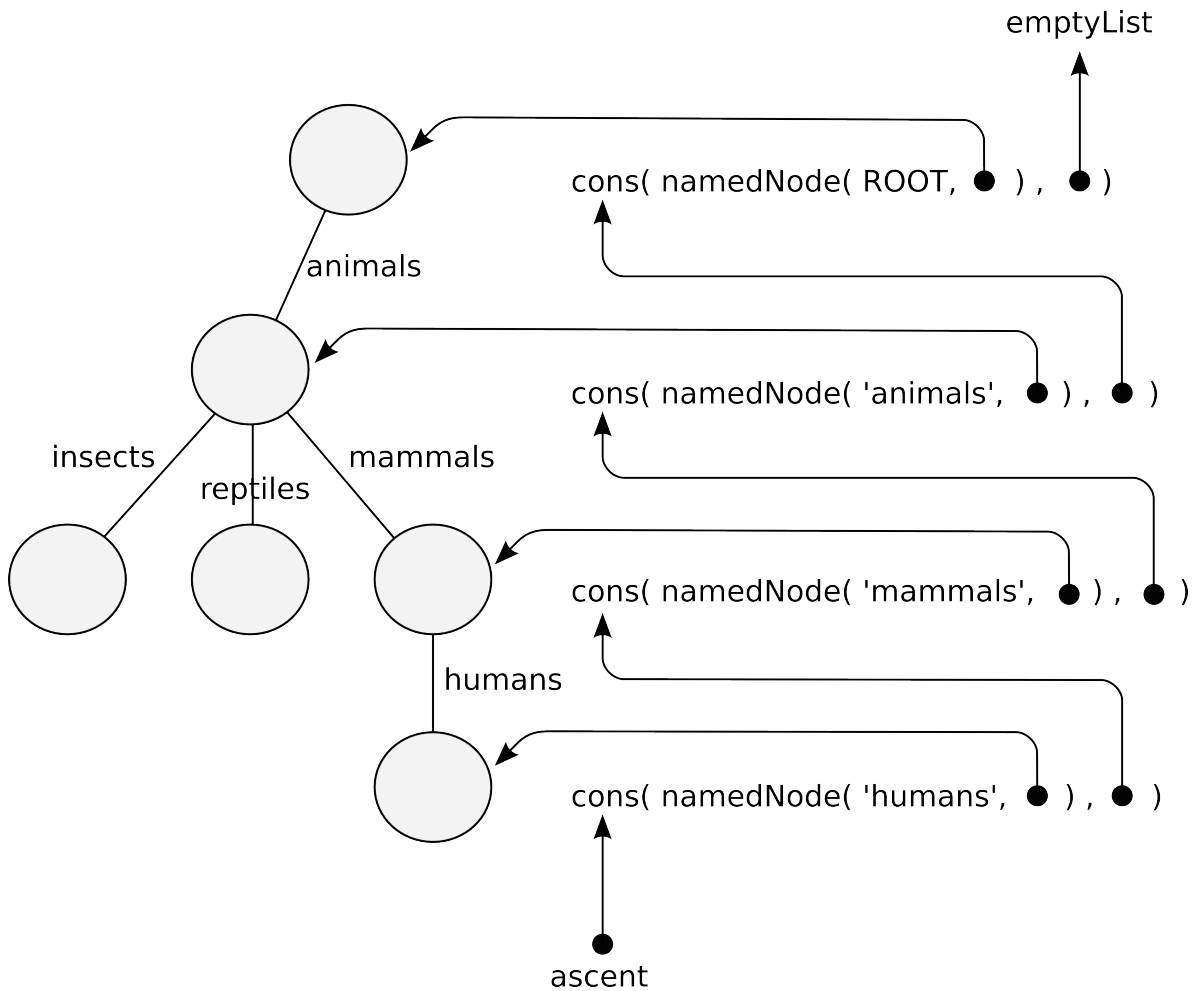
Figure 6: **List representation of an ascent rising from leaf to root through a JSON tree.** Note the special ROOT value which represents the location of the pathless root node. The ROOT value is an object, taking advantage of object uniqueness to ensure that its location is unequal to all others.

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 `openobject` and `openarray` 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 Oboe 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, between SAX events 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 the 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.

Linked lists were chosen for the ascents in preference to the more conventional approach of using native Javascript arrays for several reasons. A program is easier to test and debug given immutable data but employing the native Arrays without mutating would be very expensive because on each new path the whole array would have to be copied. During debugging, unpicking a stack trace holding immutable data requires less mental stress because every value revealed is the value that has always occupied that space and the programmer does not have to project along the time axis by imagining which values were in the same space earlier or might be there later. The lack of side effects means that new commands may be tried during a pause in execution without worrying about breaking the working of the program. In terms of speed, array-type structures are poorly suited to frequent growing and shrinking so for tracking ascents whose length changes with every event received, arrays are relatively unperformant. Taking into account the receiver of the ascent data, lists are also a convenient format for the JSONPath engine to match against as will be discussed in the next section. The Javascript file lists.js (Appendix p.98) implements various list functions: `cons`, `head`, `tail`, `map`, `foldR`, `all`, `without` as well as providing conversions to and from arrays.

4.7   Oboe JSONPath implementation

With the initial commit the JSONPath implementation was little more than a series of regular expressions[6] but has slowly evolved into a featureful and efficient implementation. A total 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 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 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.

---

[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 86 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

The compilation is performed by recursively examining the left-most side of the string for a JSONPath clause. For each clause type there is a function 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, which repeats 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'])
)
```

Because the matching is implemented using a side-effect free subset of Javascript 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`. Likewise, while evaluating several sibling elements against a pattern that requires matches at multiple depths in the JSON hierarchy, the same JSONPath term evaluator

will be tested many times against the parent element, always with the same result. Although Javascript comes without functional caching, it can be added using the language itself, probably the best known example being `memoize` from Underscore.js (Ashkenas 2008). It is likely however that hashing the function parameters would 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 function for a node parsed out of JSON would have to walk the entire subtree rooted from that node, requiring time proportional to the size of the tree. Current Javascript implementations also make it difficult to manage caches in general from inside the language because there is no way to occupy only spare memory. Weak references are proposed in the ECMAScript 6th edition draft (Harmony 2013) but currently only experimentally supported[8]. If the hashing problem were solved the WeakHashMap would be ideal for adding functional caching in future.

Functions describing the tokenisation of the JSONPath language are given 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 expression 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[9]. 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 5, p.44) to be split into two further 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. A more purist approach would stub out the tokeniser functions before testing the compiled JSONPath expressions. This would be desirable if a general purpose compiler generator were being implemented but since the aim of the code is to work with only one language, removing the peculiarities of the tokenisation from the tests would only decrease their ability to demonstrate the correct interpretation of the JSONPath language as a whole.

---

[8]At time of writing, Mozilla Firefox is the only browser supporting WeakHashMap by default. In Google Chrome 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/_Objects/WeakMap retrieved 11th October 2013

[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular/_Expressions

One limitation is that Oboe currently only supports selections which are decidable at the time that the candidate node is discovered. This forbids some seemingly simple selections such as *the last element of the array* because when an element is found, without looking ahead and possibly finding an array closing token we cannot know if our node is the last element. Removing this restriction would require a fairly substantial rewrite of the JSONPath engine. One strategy would be to take an event-driven approach to the matching. At present matching is triggered by events but the tests themselves are expressed synchronously. Under an event-driven matching implementation, instead of returning a value, each JSONPath term evaluator would be given a callback to pass the result to. Under most circumstances it should be able to decide if a match has taken place at the time that it is called, handing the result immediately to the callback. However, for cases where more of the document must be revealed before a match can be decided the term evaluators would have the option of listening to the parse until further document nodes are revealed, replying later when the necessary information is available. While a wider selection vocabulary might be useful, such an expansion would make it difficult to offer a predictable callback order and could cause confusion for application developers. A language containing only selectors which may be evaluated against nodes as they are detected is powerful enough to handle most cases. Until a strong need is demonstrated the selector language will be kept in its current form.

4.8 Differences in the working of programs that can be easily written using Oboe.js

Because of assumptions implicit in either technique, a program written using Oboe.js will perform subtly different actions from one written using more conventional libraries, even if the programmer means to express the same thing. Consider the two examples below in which Node.js is used to read a local JSON file and write to the console.

```javascript
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 moustache*");
   })
   .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 moustache*");
});
```

While the behaviours intended by the programmer are similar, some accidental side-behaviours differ between the two. It is likely that most programmers would not think of these differences as they write so it is important that they are not destructive. In the first example the order of the output for schemes and plans will match their order in the JSON, whereas for the second scheming is always done before plotting. The error behaviours are also different – the first prints until it has an error, the second prints if there are no errors. In the second example it is *almost mandatory* to check for errors before starting the output whereas in the first it feels most natural to register the error listener at the end of the chained calls. It is unusual in describing a system's desirable behaviour to state the reaction to abnormal cases first so I find that the Oboe example follows the more natural ordering.

Considering the code style that is encouraged, the first example takes a more declarative form by specifying the items of interest using patterns whereas the second is more imperative by explicitly looping through the items. If several levels of selection were required, such as schemes.*.steps.*, other than a longer JSONPath pattern the first example would not grow in complexity whereas the second would require nested looping. The cyclic complexity of programming using Oboe would stay roughly constant whereas using programmatic drill-down it increases linearly with the number of levels that must be traversed.

5 Conclusion

5.1 Benchmarking vs non-progressive REST

I feel it is important to experimentally answer the question, *is this way actually any faster?* To measure performance the Oboe repository contains a small benchmarking suite that runs under Node.js. One of the advantages of incremental parsing suggested in the introduction was a perceptual improvement in speed. The experiments do not directly gauge user perception because it would require subjective judgement and human participants, an undertaking large enough to be a project in itself. In lieu of perceptual experiments the benchmarks measure the time taken to provide the first output. This correlates with how quickly the first interface elements may be drawn and should be a good proxy indicator of perceptual speed. Node is used to host the tests because it is a minimalist platform and should give repeatable results whereas browsers may be running any number of simultaneous background tasks and are less predictable. Node also has the advantage that small changes in memory use are not overwhelmed by a memory hungry environment.

The benchmark involves two node processes, one acting as a REST client and the other as a REST server, mimicking a service backed by a relational database. Relational database client libraries pass data from a result cursor one tuple at a time to be used by the application, the service simulates this by writing out forty tuples as JSON objects, one every ten milliseconds. Half the tuples contain a URL to a further resource which will also be fetched so that an aggregation can be created. To simulate real network conditions, Apple's *Network Line Conditioner* was used with the presets *3G, Average Case* and *Cable modem* to represent poor and good internet connections respectively. Three client versions were implemented using JSON.parse DOM-style parsing, Clarinet SAX-style parsing and Oboe. Memory was measured on the client using Node's built in memory reporting tool, `process.memoryusage()` and the largest figure reported during each run is taken. The test server and client can be found in the project's `benchmark` directory, or in the appendix on pages 8.1 and 8.2.

Client Strategy	Network	First output	Total time	Max. Memory
Oboe.js	Good	40ms	804ms	6.2Mb
Oboe.js	Poor	60ms	1,526ms	6.2Mb
JSON.parse	Good	984ms	1,064ms	9,0Mb
JSON.parse	Poor	2550ms	2,609ms	8.9Mb
Clarinet	Good	34ms	781ms	5.5Mb
Clarinet	Poor	52ms	1,510ms	5.5Mb

In comparison with JSON.parse, Oboe shows a dramatic improvement of about 96% regarding the time taken for the first output to be produced and a smaller but significant improvement of about 40% in the total time required to create the aggregation. Oboe's aggregation on a good network is about 15% slower than Clarinet; since Oboe depends on Clarinet for parsing it could not be faster but I had hoped for the gap to be smaller.

Clarinet is known to be slower than JSON.parse for input which is already held in memory (Job 2012) but when reading from a stream this is more than offset by the ability to parse progressively. Compared to JSON.parse, the extra computation time needed by Oboe and Clarinet is shown to be relatively insignificant in comparison to the advantage of better I/O management. Reacting earlier using slower handlers is shown to be faster overall than reacting later with quicker ones. I feel that this vindicates a project focus on efficient management of I/O over faster algorithms.

Oboe shows an unexpected improvement in terms of memory usage compared to JSON.parse. It is not clear why this would be but it may be attributable to the large dependency tree brought in by the get-json (NPM 2013) library used in the JSON.parse client version. As expected, Clarinet has the smallest memory usage because it never stores a complete version of the parsed JSON. Clarinet's memory usage remains roughly constant as the parsed resource increases in size while the other two will rise linearly. Node is popular on RaspberryPi type devices with constrained RAM and Clarinet might be preferable to Oboe where code clarity is less important than a small memory footprint.

5.2   Comparative developer ergonomics

Writing less code is not in itself a guarantee of a better developer ergonomics but it is a good indicator so long as the program isn't forced to be overly terse. The table below reports the quantity of code required to implement the benchmark REST client under each strategy. Each version is written in the most natural expression for the library used.

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 with JSON.parse was slightly longer, requiring a loop and an if
statement, both necessary to drill down into the results. The code below is shortened
by using get-json package which combines parsing implicitly with the download:

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

This version is tightly coupled with the JSON format that it reads. We can see this
in the fragments `records.data`, `record.url`, and `record.name` which will only work
if they find the desired subtree at exactly the anticipated location. The code might
be said to contain a description of the format that it is for rather than a description
of what is required from the format. The Oboe version describes the format only so
far as is needed to identify the desired parts; the remainder of the JSON could change
and the code would continue to work. I believe this demonstrates a greater tolerance to
changing formats and that this would be useful when programming against evolving
services.

The Clarinet version of the code is too long to include here but may be seen
. By using SAX directly the code is more verbose and its purpose
is obfuscated. A person looking at this source would find it difficult to deduce what is
being done without considering it for some time. The functions receiving SAX events
must handle several different cases and so tend to have generic parameter names such as
'key' or 'value' which represent the token type. By contrast, Oboe and JSON.parse both
allow names such as 'record' or 'url' which are chosen according to the semantics of the
value. This naming aids understandability because it allows the programmer to think in
terms of the domain model rather than working on the level of serialisation artifacts.

5.3 Performance under various Javascript engines

The file `oboe.performance.spec.js`[10] contains a benchmark which concentrates on measuring the performance of Oboe's pattern matching. This test registers a complex pattern which intentionally uses all features from the JSONPath language and then fetches a JSON file containing approximately 800 nodes, 100 of which will match. Although actual HTTP is used, it is over an unthrottled connection to localhost so network delay should be negligible. The tests were executed on a relatively low-powered Macbook Air laptop running OS X 10.7.5, except for Chrome Mobile which was tested on an iPhone 5 with iOS 7.0.2. Test cases requiring Microsoft Windows were performed inside a VirtualBox virtual machine. Curl is a simple download tool that writes the resource to stdout without any parsing and is included as a baseline.

Platform	Total Time	Throughput (nodes/ms)
Curl	42ms	*unparsed, n/a*
Chrome 31.0.1650.34	84ms	9.57
Node.js v0.10.1	172ms	4.67
Chrome 30.0.1599	202ms	3.98
Safari 6.0.5	231ms	3.48
IE 10.0.0 (Windows 8)	349ms	2.30
Chrome Mobile iOS 30.0.1599	431ms	1.86
Firefox 24.0.0	547ms	1.47
IE 8.0.0 (Windows XP)	3,048ms	0.26

Table 3: time taken to match approximately 800 JSON nodes against a complex JSONPath expression under several Javascript engines

From table 3 we can see that Firefox executes the library more slowly than other modern desktop browsers. This is probably explicable by SpiderMonkey, the Mozilla just-in-time Javascript compiler being poor at optimising functional Javascript (Guo 2013). The JSON nodes are not of a common type so many of the library's internal callsites are not monomorphic which is also optimised poorly (Guo 2013). When the test was later repeated with a simpler pattern Firefox showed by far the largest improvement, indicating that the functional JSONPath matching accounts for Firefox's lower than expected performance.

---

[10]In git repository, test/specs/oboe.performance.spec.js

Version 31 of Chrome for OSX was released during the project and due to an updated version of the v8 Javascript engine performs more than twice as quickly as version 30. This reflects Javascript engine writers targeting functional optimisation now that functional Javascript is becoming a more popular style. Node and Chrome for iOS also use v8 and should catch up when they are next updated.

Of these results I find only the performance under old versions of Internet Explorer poor enough to be concerning. An improvement over the traditional model was known not to be possible since this platform forbids progressively interpreting the XHR response but I did not expect the performance to degrade by so much. Adding three seconds to a REST call will unacceptably impair the user experience of a webapp so it might be reasonable to conclude that for complex use cases Oboe is currently unsuited to legacy platforms. If we desired to improve performance on older platforms one solution might be to create a simpler, non-progressive implementation of the Oboe API for selective delivery to older browsers. However, I would argue that the time spent writing a basic legacy version would be better employed waiting for these moribund platforms to die.

For an imperative language coded in a functional style the compiler may not optimise as effectively as if a functional language were used. This is especially the case for a highly dynamic language in which everything, even the basic built-in types, are mutable. Presenting a convenient API to application developers means passing eagerly evaluated parameters to application callbacks even when the parameters are of secondary importance, such as the path and ancestor arrays that are created for every matching node, and will be predominantly ignored. Under a functional language these could be lazily evaluated without requiring any special effort by the application programmer. The choice of Javascript gives a very large number of client- and server-side applications that may potentially adopt the library. However, server-side Oboe would be very amicable to implementation using a purer functional language and it would be interesting to see how much faster it could be.

5.4   Status as a micro-library

The file `oboe-browser.min.js` is the minified, built version of Oboe ready to be sent to web browsers and can be found in the project's `dist` directory. The size fluctuates as commits are made but after gzip it comes to about 4800 bytes; close to but comfortably under the 5120 byte limit. At roughly the size of a small image the download footprint of Oboe should not discourage adoption.

5.5   Potential future work

Although all network traffic can be viewed as a stream, the most obvious future expansion would be to create a matching server-side component that provides an intuitive interface for writing JSON streams. So far, sending streaming JSON has required that the resource be written out using programmer-assembled strings but this approach is error prone and would scale badly as messages become more complex. A server-side library for stream writing would allow Oboe to be used as a REST-compatible streaming solution for situations which currently employ push tables or Websockets. This would provide a form of REST streaming that operates according to the principled design of HTTP rather than by sidestepping it.

Although JSON is particularly well suited, there is nothing about Oboe that precludes working with other tree-shaped formats. If there is demand, an XML/XPATH version seems like an obvious expansion. This could be implemented by allowing resource formats to be added using plugins which would allow programmers to create a progressive interpretation of any resource type. As a minimum, a plug-in would require a SAX-like parser and a DSL for node selection.

Oboe stores all parsed nodes for the duration of its lifetime so despite being similar to a SAX parser in terms of being progressive, it consumes as much memory as a DOM parser. The nodes remain held so that all possible JSONPath expressions may later be tested. However, in most cases memory could be freed if the parsed content were stored only so far as is required to test against the patterns which have actually been registered. For selectors which match near the root this would allow large subtrees to be pruned, particularly after the patterns have matched and the nodes have already been handed back to the application. Likewise, the current implementation takes a rather brute force approach when examining nodes for pattern matches by checking every registered JSONPath expression against every node parsed from the JSON. For many expressions we should be able to say that there will be no matches inside a particular JSON subtree, either because we have already matched or because the subtree's ancestors invariably imply failure. A more sophisticated implementation might subdue provably unsatisfiable handlers until the SAX parser leaves an unmatchable subtree.

5.6   Summing up

The community reaction to Oboe has been overwhelmingly positive with several projects already adopting it and reporting performance gains which are large enough to be obvious. While some attention may be required for optimisation under Firefox, this project meets all of its intended aims, presenting a REST client library which in the best case allows the network to be used much more efficiently and in the worse case is very close to the previous best solution, at least when used with capable platforms. At the same time the produced solution requires less code, is less tightly coupled to JSON

63

format specifics, and with a declarative style, I believe is easier to use than the previous simplest solution.

6 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

7 Appendix ii: Oboe.js source code listing

7.1 ascent.js

```
/**
 * 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};
}

/** get the key of a namedNode */
var keyOf = attr('key');

/** get the node from a namedNode */
var nodeOf = attr('node');
```

7.2   clarinetListenerAdaptor.js

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

   var state;

   clarinet.EVENTS.forEach(function(eventName){

      var handlerFunction = handlers[eventName];

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

67

7.3   events.js

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

var // the events which are never exported are kept as
    // the smallest possible representation, in numbers:
    _S = 1,

    // fired whenever a node is found in the JSON:
    NODE_FOUND    = _S++,
    // fired whenever a path is found in the JSON:
    PATH_FOUND    = _S++,

    NODE_MATCHED  = 'node',
    PATH_MATCHED  = 'path',

    FAIL_EVENT    = 'fail',
    ROOT_FOUND    = _S++,
    HTTP_START    = 'start',
    STREAM_DATA   = _S++,
    STREAM_END    = _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
   };
}
```

68

7.4 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, args ) {

    // this isn't the shortest way to write this but it does
    // avoid creating a new array each time to pass to fn.apply,
    // otherwise could just call boundArgs.concat(callArgs)

    var numBoundArgs = args.length;

    return varArgs(function( callArgs ) {

        for (var i = 0; i < callArgs.length; i++) {
            args[numBoundArgs + i] = callArgs[i];
        }

        args.length = numBoundArgs + callArgs.length;

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

/**
 * 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) {
     // TODO: can this be written using foldr1 and compose2?

     var fnsList = arrayAsList(fns);

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

     return varArgs(function(startParams){

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

/**
 * A more optimised version of compose that takes exactly two functions
 * @param f1
 * @param f2
 */
function compose2(f1, f2){
   return function(){
      return f1.call(this,f2.apply(this,arguments));
   }
}


function attr(key) {
   return new Function('o', 'return o["' + key + '"]' );
}

/**
 * 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,
       slice = Array.prototype.slice;


   if( numberOfFixedArguments == 0 ) {
      // an optimised case for when there are no fixed args:

      return function(){
         return fn.call(this, slice.call(arguments));
      }

   } else if( numberOfFixedArguments == 1 ) {
      // an optimised case for when there are is one fixed args:

      return function(){
         return fn.call(this, arguments[0], slice.call(arguments, 1));
      }
   }
```

```javascript
    // general case

    // we know how many arguments fn will always take. Create a
    // fixed-size array to hold that many, to be re-used on
    // every call to the returned function
    var argsHolder = Array(fn.length);

    return function(){

        for (var i = 0; i < numberOfFixedArguments; i++) {
            argsHolder[i] = arguments[i];
        }

        argsHolder[numberOfFixedArguments] =
            slice.call(arguments, numberOfFixedArguments);

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


/**
 * 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) {
```

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

/**
 * A function which does nothing
 */
function noop(){}

function always(){return true}

function functor(val){
    return function(){
        return val;
    }
}
```

7.5   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.
 */




/**
 * 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( oboeBus ) {

   var emitNodeFound = oboeBus(NODE_FOUND).emit,
       emitRootFound = oboeBus(ROOT_FOUND).emit,
       emitPathFound = oboeBus(PATH_FOUND).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,
      emitRootFound( 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;
}


/**
 * 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
                            );

    emitPathFound( ascentWithNewPath);

    return ascentWithNewPath;
```

```
}


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

   emitNodeFound( 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: compose2( 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
    };
}
```

7.6 instanceApi.js

```javascript
function instanceApi(oboeBus){

   var oboeApi,
       fullyQualifiedNamePattern = /^(node|path):./,

       addListener = varArgs(function( eventId, parameters ){

           if( oboeApi[eventId] ) {

               // for events added as .on(event), if there is a
               // special .event equivalent, pass through to that
               apply(parameters, oboeApi[eventId]);
           } else {

               // we have a standard Node.js EventEmitter 2-argument call.
               // The first parameter is the listener.
               var listener = parameters[0];

               if( fullyQualifiedNamePattern.test(eventId) ) {

                   // allow fully-qualified node/path listeners
                   // to be added
                   addPathOrNodeListener(eventId, listener);
               } else  {

                   // the event has no special handling, pass through
                   // directly onto the event bus:
                   oboeBus(eventId).on( listener);
               }
           }

           return oboeApi; // chaining
       }),

       removeListener = function( eventId, p2, p3 ){

           if( eventId == 'node' || eventId == 'path' ) {

               // allow removal of node and path
               removePathOrNodeListener(eventId + ':' + p2, p3);
```

```javascript
      } else {

          // we have a standard Node.js EventEmitter 2-argument call.
          // The first parameter is the listener.
          var listener = p2;

          if( fullyQualifiedNamePattern.test(eventId) ) {

              // allow fully-qualified node/path listeners
              // to be added
              removePathOrNodeListener(eventId, listener);
          } else  {

              // the event has no special handling, pass through
              // directly onto the event bus:
              oboeBus(eventId).un( listener);
          }
      }

      return oboeApi; // chaining
  };


function addPathOrNodeListener( fullyQualifiedName, callback ) {

   var safeCallback = protectedCallback(callback);

   oboeBus(fullyQualifiedName).on( function(node, path, ancestors) {

      var keep       = true;

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

      safeCallback( node, path, ancestors );

      delete oboeApi.forget;

      if(! keep ) {
         oboeBus(fullyQualifiedName).un( callback);
      }
   }, callback)
```

```
}

function removePathOrNodeListener( fullyQualifiedName, callback ) {
   oboeBus(fullyQualifiedName).un(callback)
}

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

         // An error occured during the callback, publish it on the event bus
         oboeBus(FAIL_EVENT).emit( errorReport(undefined, undefined, e));
      }
   }
}


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

   for( var pattern in listenerMap ) {
      addPathOrNodeListener(
         eventId + ':' + pattern,
         listenerMap[pattern]
      );
   }
}

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

   if( isString(jsonPathOrListenerMap) ) {
      addPathOrNodeListener(
         eventId + ':' + jsonPathOrListenerMap,
         callback
      );
   } else {
      addListenersMap(eventId, jsonPathOrListenerMap);
```

```
   }

   return oboeApi; // chaining
}


// some interface methods are only filled in after we recieve
// values and are noops before that:
oboeBus(ROOT_FOUND).on( function(root) {
   oboeApi.root = functor(root);
});

oboeBus(HTTP_START).on( function(_statusCode, headers) {

   oboeApi.header =  function(name) {
                        return name ? headers[name]
                                    : headers
                                    ;
                     }
});

/**
 * Construct and return the public API of the Oboe instance to be
 * returned to the calling application
 */
return oboeApi = {
   on             : addListener,
   addListener    : addListener,
   removeListener : removeListener,
   emit           : oboeBus.emit,

   node           : partialComplete(addNodeOrPathListenerApi, 'node'),
   path           : partialComplete(addNodeOrPathListenerApi, 'path'),

   done           : partialComplete(addNodeOrPathListenerApi, 'node', '!'),
   start          : compose2( oboeBus(HTTP_START).on, protectedCallback ),

   // fail doesn't use protectedCallback because
   // could lead to non-terminating loops
   fail           : oboeBus(FAIL_EVENT).on,

   // public api calling abort fires the ABORTING event
   abort          : oboeBus(ABORTING).emit,
```

83

```
        // initially return nothing for header and root
        header          : noop,
        root            : noop
    };
}
```

7.7 instanceController.js

```javascript
/**
 * 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(   oboeBus,
                               clarinetParser, contentBuilderHandlers) {

   oboeBus(STREAM_DATA).on( clarinetParser.write.bind(clarinetParser));

   /* 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 */
   oboeBus(STREAM_END).on( 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. */
   oboeBus(ABORTING).on( function() {
      clarinetListenerAdaptor(clarinetParser, {});
   });

   clarinetListenerAdaptor(clarinetParser, contentBuilderHandlers);

   // react to errors by putting them on the event bus
   clarinetParser.onerror = function(e) {
      oboeBus(FAIL_EVENT).emit(
         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
   };
}
```

7.8 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  = compose2(keyOf, head),
       headNode = compose2(nodeOf, 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};
```

86

```javascript
    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 =  compose2(
                     hasAllrequiredFields,
                     headNode
                  );

   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 */
```

```
                    compose2(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
           );
        }
    }

});
```

7.9 jsonPathSyntax.js

```javascript
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
     );
  };

}());
```

7.10   lists.js

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

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

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

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

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

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

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


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

```
 *
 *    cons(a, cons(b, cons(c, emptyList)))
 **/
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
            ;
}

/**
 * foldR implementation. Reduce a list down to a single value.
 *
 * @pram {Function} fn     (rightEval, curVal) -> result
 */
function foldR1(fn, list) {

   return tail(list)
            ? fn(foldR1(fn, tail(list)), head(list))
            : head(list)
            ;
}


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

   return withoutInner(list, removedFn || noop);

   function withoutInner(subList, removedFn) {
```

```
      return subList
          ?  ( test(head(subList))
                    ? (removedFn(head(subList)), tail(subList))
                    : cons(head(subList), withoutInner(tail(subList), removedFn))
              )
          : 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)) );
}

/**
 * Call every function in a list of functions with the same arguments
 *
 * 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 the
 * functions being called have side-effects.
 */
function applyEach(fnList, arguments) {

    if( fnList ) {
       head(fnList).apply(null, arguments);

       applyEach(tail(fnList), arguments);
    }
}

/**
 * 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
```

101

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

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

return reverseInner(list, emptyList);
}

function first(test, list) {
   return   list &&
            (test(head(list))
               ? head(list)
               : first(test,tail(list)));
}
```

7.11 parseResponseHeaders.browser.js

```javascript
// based on gist https://gist.github.com/monsur/706839

/**
 * XmlHttpRequest's getAllResponseHeaders() method returns a string of response
 * headers according to the format described here:
 * http://www.w3.org/TR/XMLHttpRequest/#the-getallresponseheaders-method
 * This method parses that string into a user-friendly key/value pair object.
 */
function parseResponseHeaders(headerStr) {
  var headers = {};

  headerStr && headerStr.split('\u000d\u000a')
    .forEach(function(headerPair){

      // Can't use split() here because it does the wrong thing
      // if the header value has the string ": " in it.
      var index = headerPair.indexOf('\u003a\u0020');

      headers[headerPair.substring(0, index)]
                = headerPair.substring(index + 2);
    });

  return headers;
}
```

7.12   patternAdapter.js

```
function patternAdapter(oboeBus, jsonPathCompiler) {

   var predicateEventMap = {
      node:oboeBus(NODE_FOUND)
   ,  path:oboeBus(PATH_FOUND)
   };

   function emitMatchingNode(emitMatch, node, ascent) {

      /*
         We're now calling to the outside world where Lisp-style
         lists will not be familiar. 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);

      emitMatch(
         node,

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

   function addUnderlyingListener( fullEventName, predicateEvent, compiledJsonPath ){

      var emitMatch = oboeBus(fullEventName).emit;

      predicateEvent.on( function (ascent) {

         var maybeMatchingMapping = compiledJsonPath(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) {

       emitMatchingNode(
          emitMatch,
          nodeOf(maybeMatchingMapping),
          ascent
       );
    }
  }, fullEventName);

  oboeBus('removeListener').on( function(removedEventName){

     // if the match even listener is later removed, clean up by removing
     // the underlying listener if nothing else is using that pattern:

     if( removedEventName == fullEventName ) {

        if( !oboeBus(removedEventName).listeners(  )) {
           predicateEvent.un( fullEventName );
        }
     }
  });
}

oboeBus('newListener').on( function(fullEventName){

  var match = /(node|path):(.*)/.exec(fullEventName);

  if( match ) {
     var predicateEvent = predicateEventMap[match[1]];

     if( !predicateEvent.hasListener( fullEventName) ) {

        addUnderlyingListener(
           fullEventName,
```

105

```
                predicateEvent,
                jsonPathCompiler( match[2] )
            );
        }
    }
})

}
```

7.13 pubSub.js

```
/**
 * pubSub is a curried interface for listening to and emitting
 * events.
 *
 * If we get a bus:
 *
 *    var bus = pubSub();
 *
 * We can listen to event 'foo' like:
 *
 *    bus('foo').on(myCallback)
 *
 * And emit event foo like:
 *
 *    bus('foo').emit()
 *
 * or, with a parameter:
 *
 *    bus('foo').emit('bar')
 *
 * All functions can be cached and don't need to be
 * bound. Ie:
 *
 *    var fooEmitter = bus('foo').emit
 *    fooEmitter('bar');  // emit an event
 *    fooEmitter('baz');  // emit another
 *
 * There's also an uncurried[1] shortcut for .emit and .on:
 *
 *    bus.on('foo', callback)
 *    bus.emit('foo', 'bar')
 *
 * [1]: http://zvon.org/other/haskell/Outputprelude/uncurry_f.html
 */
function pubSub(){

    var singles = {},
        newListener = newSingle('newListener'),
        removeListener = newSingle('removeListener');

    function newSingle(eventName) {
```

```javascript
        return singles[eventName] = singleEventPubSub(
            eventName,
            newListener,
            removeListener
        );
    }

    /** pubSub instances are functions */
    function pubSubInstance( eventName ){

        return singles[eventName] || newSingle( eventName );
    }

    // add convenience EventEmitter-style uncurried form of 'emit' and 'on'
    ['emit', 'on'].forEach(function(methodName){

        pubSubInstance[methodName] = varArgs(function(eventName, parameters){
            apply( parameters, pubSubInstance( eventName )[methodName]);
        });
    })

    return pubSubInstance;
}
```

7.14 publicApi.js

```javascript
// export public API
function oboe(arg1, arg2) {

   if (arg1.url) {

      // method signature is:
      //    oboe({method:m, url:u, body:b, headers:{...}})

      return wire(
         (arg1.method || 'GET'),
         url(arg1.url, arg1.cached),
         arg1.body,
         arg1.headers
      );
   } else {

      //  simple version for GETs. Signature is:
      //    oboe( url )
      //
      return wire(
         'GET',
         arg1, // url
         arg2  // body. Deprecated, use {url:u, body:b} instead
      );
   }

   function url(baseUrl, cached) {

      if( cached === false ) {

         if( baseUrl.indexOf('?') == -1 ) {
            baseUrl += '?';
         } else {
            baseUrl += '&';
         }

         baseUrl += '_=' + new Date().getTime();
      }
      return baseUrl;
   }
}
```

7.15   singleEventPubSub.js

```javascript
/**
 * A pub/sub which is responsible for a single event type. A
 * multi-event type event bus is created by pubSub by collecting
 * several of these.
 *
 * @param {String} eventType
 *    the name of the events managed by this singleEventPubSub
 * @param {singleEventPubSub} [newListener]
 *    place to notify of new listeners
 * @param {singleEventPubSub} [removeListener]
 *    place to notify of when listeners are removed
 */
function singleEventPubSub(eventType, newListener, removeListener){

   /** we are optimised for emitting events over firing them.
    *  As well as the tuple list which stores event ids and
    *  listeners there is a list with just the listeners which
    *  can be iterated more quickly when we are emitting
    */
   var listenerTupleList,
       listenerList;

   function hasId(id){
      return function(tuple) {
         return tuple.id == id;
      };
   }

   return {

      /**
       * @param {Function} listener
       * @param {*} listenerId
       *    an id that this listener can later by removed by.
       *    Can be of any type, to be compared to other ids using ==
       */
      on:function( listener, listenerId ) {

         var tuple = {
            listener: listener
         ,  id:        listenerId || listener // when no id is given use the
```

110

```
                                              // listener function as the id
   };

   if( newListener ) {
      newListener.emit(eventType, listener, tuple.id);
   }

   listenerTupleList = cons( tuple,    listenerTupleList );
   listenerList      = cons( listener, listenerList      );

   return this; // chaining
},

emit:function () {
   applyEach( listenerList, arguments );
},

un: function( listenerId ) {

   var removed;

   listenerTupleList = without(
      listenerTupleList,
      hasId(listenerId),
      function(tuple){
         removed = tuple;
      }
   );

   if( removed ) {
      listenerList = without( listenerList, function(listener){
         return listener == removed.listener;
      });

      if( removeListener ) {
         removeListener.emit(eventType, removed.listener, removed.id);
      }
   }
},

listeners: function(){
   // differs from Node EventEmitter: returns list, not array
   return listenerList;
```

```
        },

    hasListener: function(listenerId){
        var test = listenerId? hasId(listenerId) : always;

        return defined(first( test, listenerTupleList));
    }
   };
}
```

7.16   streamingHttp.browser.js

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


/**
 * A wrapper around the browser XmlHttpRequest object that raises an
 * event whenever a new part of the response is available.
 *
 * In older browsers progressive reading is impossible so all the
 * content is given in a single call. For newer ones several events
 * should be raised, allowing progressive interpretation of the response.
 *
 * @param {Function} oboeBus an event bus local to this Oboe instance
 * @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(oboeBus, xhr, method, url, data, headers) {

   var emitStreamData = oboeBus(STREAM_DATA).emit,
       emitFail       = oboeBus(FAIL_EVENT).emit,
       numberOfCharsAlreadyGivenToCallback = 0;

   // When an ABORTING message is put on the event bus abort
   // the ajax request
   oboeBus( ABORTING ).on( 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
```

113

```
 *   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 ) {
      emitStreamData( newText );
   }

   numberOfCharsAlreadyGivenToCallback = len(textSoFar);
}


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

xhr.onreadystatechange = function() {

   switch( xhr.readyState ) {

      case 2:
```

```javascript
         oboeBus( HTTP_START ).emit(
            xhr.status,
            parseResponseHeaders(xhr.getAllResponseHeaders()) );
         return;

      case 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();

            oboeBus(STREAM_END).emit();
         } else {

            emitFail( 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(emitFail, errorReport(undefined, undefined, e))
      ,  0
      );
   }
}
```

7.17 streamingHttp.node.js

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


/**
 * A wrapper around the browser XmlHttpRequest object that raises an
 * event whenever a new part of the response is available.
 *
 * In older browsers progressive reading is impossible so all the
 * content is given in a single call. For newer ones several events
 * should be raised, allowing progressive interpretation of the response.
 *
 * @param {Function} oboeBus an event bus local to this Oboe instance
 * @param {XMLHttpRequest} http the http implementation to use as the transport. Und
 *          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 j
 * @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(oboeBus, http, method, contentSource, data, headers) {
   "use strict";

   function readStreamToEventBus(readableStream) {

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

         oboeBus(STREAM_DATA).emit( chunk.toString() );
      });

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

         oboeBus( STREAM_END ).emit();
      });
   }

   function readStreamToEnd(readableStream, callback){
```

117

```javascript
    var content = '';

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

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

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

       callback( content );
    });
}

function fetchHttpUrl( 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;

      oboeBus(HTTP_START).emit( res.statusCode, res.headers);

      if( sucessful ) {

         readStreamToEventBus(res)

      } else {
         readStreamToEnd(res, function(errorBody){
            oboeBus(FAIL_EVENT).emit(
               errorReport( statusCode, errorBody )
            );
```

```
            });
        }
    });

    req.on('error', function(e) {
        oboeBus(FAIL_EVENT).emit(
            errorReport(undefined, undefined, e )
        );
    });

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

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

    req.end();
}

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

}
```

7.18   util.js

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


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

var 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;
}

/**
 * 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) {
```

120

```
            return (field in o);
        }, fieldList);
}
```

7.19 wire.js

```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 oboeBus = pubSub();

   streamingHttp( oboeBus,
                  httpTransport(),
                  httpMethodName, contentSource, body, headers );

   instanceController(
               oboeBus,
               clarinet.parser(),
               incrementalContentBuilder(oboeBus)
   );

   patternAdapter(oboeBus, jsonPathCompiler);

   return new instanceApi(oboeBus);
}
```

8 Appendix iii: Benchmarking

8.1 benchmarkClient.js

```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;
```

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

```javascript
    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]();
```

8.2   benchmarkServer.js

```javascript
/**
 */

"use strict";

var PORT = 4444;

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

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

function serveItemList(_req, res) {

    console.log('slow 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) {
```

```javascript
            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));
```

9   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.

Alman, Ben. 2012. "Grunt: The Javascript Task Runner." http://gruntjs.com/.

Ashkenas, Jeremy. 2008. "Underscore.js: Memoize." http://underscorejs.org/#memoize.

Bazon, Mihai. 2010. "UglifyJs." https://github.com/mishoo/UglifyJS.

Bray, Tim, Jean Paoli, C M Sperberg-McQueen, Eve Maler, and François Yergeau. 2008. "Extensible Markup Language (XML) 1.0 (Fifth Edition)." http://www.w3.org/TR/REC-xml/#sec-starttags.

Browserling Inc. 2012. "Browserify Website." http://browserify.org/.

Burke, James. 2011. "Require.js." http://requirejs.org/.

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.*

ECMA. 1999. "ECMAScript Language Specification 3rd Edition." http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf.

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

Fielding, R. 2000. "Principled Design of the Modern Web Architecture."

Fielding, R, J Gettys, J Mogul, H Frystyk, et al. 1999. "Hypertext Transfer Protocol -- HTTP/1.1: Header Field Definitions." http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.5.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.*

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/.

Google. 2009. "Google Closure Compiler." https://developers.google.com/closure/compiler/.

Gudgin, Martin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. 2007. "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." http://www.w3.org/TR/soap12-part1/.

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.

Harmony. 2013. "Draft Specification for ES.next (Ecma-262 Edition 6)." http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts.

Job, Nuno. 2011. "Clarinet - SAX Based Evented Streaming JSON Parser in JavaScript." https://github.com/dscape/clarinet.

———. 2012. "Clarinet - SAX Based Evented Streaming JSON Parser in JavaScript for the Browser and Nodejs." http://writings.nunojob.com/2011/12/clarinet-sax-based-evented-streaming-json-parser-in-javascript-for-the-browser-and-nodejs.html.

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

Lieberherr, Karl J, Ignacio Silva-Lepe, and Xiao Cun. 1994. "Adaptive Ob Ject-Oriented Programming Using Graph-Based Customization." http://www.csg.ci.i.u-tokyo.ac.jp/old-pages/muga/paper/others/gpce_gcse/ap_graf.pdf.

Martelli, Alex. 2000. "Discussion of Typing in Python Language, 2000." https://groups.google.com/forum/?hl=en/#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ.

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

McIlroy, Doug. 1978. "Basics of the Unix Philosophy." http://www.faqs.org/docs/artu/ch01s06.html.

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

Microsoft. 2013. "LINQ (Language-Integrated Query)." http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx.

Netty. 2010. "Netty Project." http://netty.io/.

NPM. 2013. "Get JSON." https://npmjs.org/package/get-json.

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.

Ryan, Dahl. 2009. "Node.js." http://nodejs.org/.

Schneier, Bruce. 2000. "Crypto-Gram Newsletter." https://www.schneier.com/crypto-gram-0003.html#8.

Schneier, Bruce, and Sam Harris. 2013. "To Profile or Not to Profile? - a Debate Between Sam Harris and Bruce Schneier." http://www.samharris.org/blog/item/to-profile-or-not-to-profile.

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

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/.

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.