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

Jim Higson

2013

Contents

List of Figures

1 Abstract

A project which incorporates techniques including progressive parsing and http streaming to the programming of REST clients, with the aim of a significant improvement in performance, fault tolerance and encouraging a greater degree of loose coupling between systems. Particularly, loose coupling is considered in light of the application of Agile methodologies to SOA, under which resources can be expected to change as programs are constantly refactored. A study is made of a real-world situation in which SOA is used extensively to create many small systems but problems arise from a tight coupling to versions encouraged by the programmer's chosen tool set.

Performance is considered dually: in terms of an increasing the perceived responsiveness to programs presenting an interactive interface, and also the absolute completion time for a task undertaken by a system which connects to rest resources for data.

A critique is made of imperative methods under which items of interest are programmatically extracted from a resource once retrieved. Following from this critique, a ddeclarative alternative is presented under which the identification of items of interest is possible prior to a resource having been retrieved in its entirety. The declarative syntax is integrated into a javascript framework which ties this method of identification into a wider scheme of object detection within streaming resources.

2 Introduction

introduction should be 2-5 pages

Increasing the perception of speed: * Source that doing things early makes page feel faster. * Also actually faster as well as being perceived as such since useful things can often be done before whole content is loaded.

When connections fail, apps are left with non of the content. Happens a lot on mobile networks.

What a Micro-library is

2.1 Inefficiencies in performing a fairly simple task

Despite the enthusiasm for which SOA and REST in particular has been adapted, I believe this model isn't being used to its fullest potential. Consider a fairly simple task of retrieving all the images used on a web page.

Grab all the images mentioned in a web page Images may be on another subdomain * DNS lookup only after got whole page Dynamically generated pages can often load slowly, even when there is plenty of bandwidth But images could load quickly.

Diagram of timeline to get images from a webpage.

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

The above problem has many analogues and because REST uses standard web semantics applies to much more than just automated web surfing. Indeed, as the machine readability of the data increases, access early can be all the more beneficial since decisions to terminate the connection may be made. Example: academic's list of publications, then downloading all the new ones.

2.2 Agile methodologies and future versioning

SOA has been adapted widely but versioning remains a common challenge in industry.

Anecdote: test environment finds an issue. One system can't be released. Contagion.

How to cope with software that changes every week.

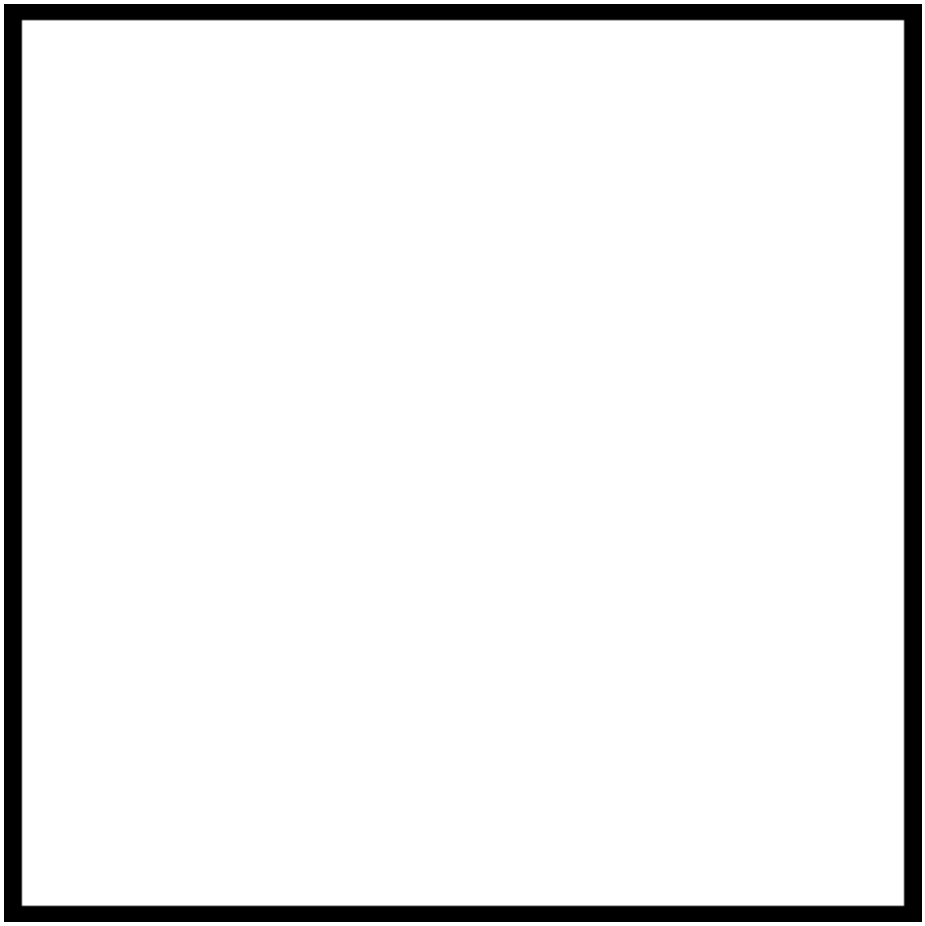

Figure 1: Potential differences in overall time taken to download a list of publications and then download any ones newer than a certain date. Assuming the publications are ordered newest first, the first connection may be terminated as soon as an older publication is found.

Because of the contagion problem, need to be able to create loosely-coupled systems.

Inside systems also, even with automatic refactoring tools, only automate and therefoer lessen but do not remove the problem that coupling causes changes in one place of a codebase to cause knock-on changes in remote other parts of the code. A method of programming which was truly compatible with extreme programming would involve designing for constant change without disparate parts having to be modified as structural refactoring occurs.

I propose that in a changing system, readability of code's changelog is as important as readability of the code itself. Extraneous changes dilute the changelog, making it less easily defined by code changes which are intrinsically linked to the actual change in the logic being expressed by the program.

It is often stated that understandability is the number once most important concern in a codebase (CITE) - if the code is suitably dynamic it is important that changes are axiomic and clarity of the changelog is equally important.

3 Background

background should be 2-10 pages

3.1 Some high-level stuff about webapps and where processing is done

Ie, front-end client-side, front-end server-side.

!A webapp running with a front end generated partially on server and partially on client side

Separated from services by http calls regardless.

Contrast: mainframes, thin clients, X11, Wayland, PCs. NextCubes (CITE: get from old dis)

Next is closest pre-runner to current web architecture.

3.2 SOA

REST/WebServices (WSDL etc)

What is a rest client in this context (a client library)

Marshalling/ de-marshalling. Benefits and the problems that it causes. Allows one model to be written out to XML or JSON

Big/small message problem and granularity. With small: http overhead. With big: not all may be needed.

Javascript as mis-understood language (CITE: Crockford) - list features available.

(correctly, ECMAScript) Misleadingly named after Java as a marketing ploy when Java was a new technology (CITE) - in true more similar to Scheme or Lisp but with Java or C inspired syntax.

3.3 Anatomy of a SOA client

First stage after getting a resource is usually to programmatically extract the interesting part from it. This is usually done via calls in the programming language itself, for example by de-marshaling the stream to domain objects and then calling a series of getters to narrow down to the interesting parts.

This part has become such a natural component of a workflow that it is barely noticed that it is happening. In an OO language, the extraction of small parts of a model which, in the scope of the current concern are of interest is so universal that it could be considered the sole reason that getters exist.

However subtly incorporated it has become in the thinking of the programmer, we should note that this is a construct and only one possible way of thinking regarding identifying the areas of current interest in a wider model.

```
// an example programmatic approach to a domain model interrogation under Java

List<Person> people = myModel.getPeople();
String firstPersonsSurname = people.get(0).getSurname();
```

One weakness of this imperative, programatic inspection model is that, once much code is written to interogate models in this way, the interface of the model becomes increasingly expensive to change as the code making the inspections becomes more tightly coupled with the thing that it is inspecting. Taking the above example, if the model were later refactored such that the concepts of firstName and surName were pulled from the Person class into an extracted Name class, because the inspection relies on a sequence of calls made directly into domain objects, the code making the query would also have to change.

I believe that this coupling defies Agile methods of programming. Many Java IDEs provide tools that would offer to automate the above extraction into a Name class, creating the new class and altering the existing calls. While reducing the pain, if we accept the concept as I stated in the Introduction that the code should not be seen as a static thing in which understanding is

More declarative syntaxes exist which are flexible enough that the declarative expressions may still apply as the underlying model is refactored. Whilst not applicable to use in general purpose programming, XPATH is an example of this. As an analogue of the Java situation above, Given the following XML:

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

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

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

A few models exist which do not follow this pattern such as XPATH. However, these are useful in only a small domain.

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

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

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

CSS. Meant for presentation of HTML, but where HTML markup is semantic it is a selector of the *meaning of elements* for the sake of applying a meaningful presentation more so than a selector of arbitrary colours and positions on a screen.

Unlike XML, in the model created by most general programming languages, there is no requirement for the data to be tree shaped. Graph is ok. This make this slighlty harder but nontheless attempts have been made.

Linq. (CITEME)

3.4 Parsing: SAX and Dom

Why sax is difficult

DOM parser can be built on a SAX parser

3.5 State of http as a streaming technology

Dichotamy between streaming and downloading in the browser for downloading data. But not for html (progressive rendering) or images (progressive PNGs and progressive JPEGs)

Lack of support in browser Long poll - for infrequent push messages. Must be read Writing script tags

All require server to have a special mode. Encoding is specific to get arround restrictions.

JsonPath in general tries to resemble the javascript use of the json language nodes it is detecting.

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


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

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

What 'this' (context) is in javascript. Why not calling it scope.

3.6 The web browser as REST client

Browser incompatability mostly in presentation layer rather than in scripting languages.

Language grammars rarely disagree, incompatability due to scripting is almost always due to the APIs presented to the scripting language rather than the language itself.

3.7 Progressive UI

Infinitely scrolling webpages. Need a way to 'pull' information, not just push if reacting to scrolling. Use oboe with websockets? Eg, ebay home page, Facebook. Adv of infinate scroll is page loads quickly and most people won't scroll very far so most of the time have everything needed right away.

3.8 State of rest: Json and XML

Json is very simple, only a few CFGs required to describe the language (json.org) - this project is listed there!

3.9 Javascript

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

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

Explain var/function difference, ie construct pluck and explain why var keyOf = partial(pluck) is declared with a var and not a function, why some prefer to do always via . operator can't be made into a function with (.) or similar and so has to be wrapped in a function is a less direct manner. Unfortunately, can make it difficult for a reader to know the types involved. For example, on seeing: `var matchesJsonPath = jsonPathCompiler( pattern )` there is no way (other than examining the source or doucmentation of the function being called) to know that this is a higher order function and will return another function to be assigned as matchesJsonPath.

C-style brackets around all function arguments hampers a natural expression of functional style code. For example, this requires a lot of arguments and without checking of function airity, it is easy to misplace a comma or closing bracket.

```
function map(fn, list){
   if( !list ) {
      return emptyList;
   } else {
      return cons(fn(head(list)), map(fn,tail(list)));
```

```
    }
}
```

3.10 Node

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

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

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

Streams in node are the observer pattern. Readable streams emit 'readable' events when they have some data to be read and 'end' events when they are finished. Apart from error handling, so far as reading is concerned, that is the extent of the API.

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

Using Node's http module provides a stream but handles setting headers, putting the method otu etc.

What Node is V8. Fast. Near-native. JIT. Why Node perhaps is mis-placed in its current usage as a purely web platform "the aim is absolutely fast io". This happened because web specialist programmers took it up first

Why Node is significant * Recognises that most tasks are io-bound rather than CPU bound. Threaded models good for CPU-bound in the main.

How Node is different

Criticisms of Node. Esp from Erlang etc devs.

Node's standard stream mechanisms

3.11 Browser

XmlHttpRequest (XHR)

Xhr2 and the .onprogress callback. polling responseText while in progress * why doesn't work in IE (built on an activeX object that provides buffering)

Older style of javascript callback. Assign a listener to onprogress, not call an add listener method means can only have one listener.

> While the request entity body is being transmitted and the upload complete flag is unset, queue a task to fire a progress event named progress on the XMLHttpRequestUpload object about every 50ms or for every byte transmitted, whichever is least frequent. w3c, XHR Working Draft

Websockets More like node Can connect to any protocol (as always, easier to program if text based but can do binary) Can use to do http but not sufficient advantage over using

4 Applicaiton and Reflection

40 to 60 pages

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

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

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

4.1 breaking out of big/small tradeoff

Best of both modes

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

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

4.2 choice of technologies

can justify why js as:

Most widely deployable.

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

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

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

In comparison to typical Tomcat-style threading model. Threaded model is powerful for genuine parallel computation but Wasteful of resources where the

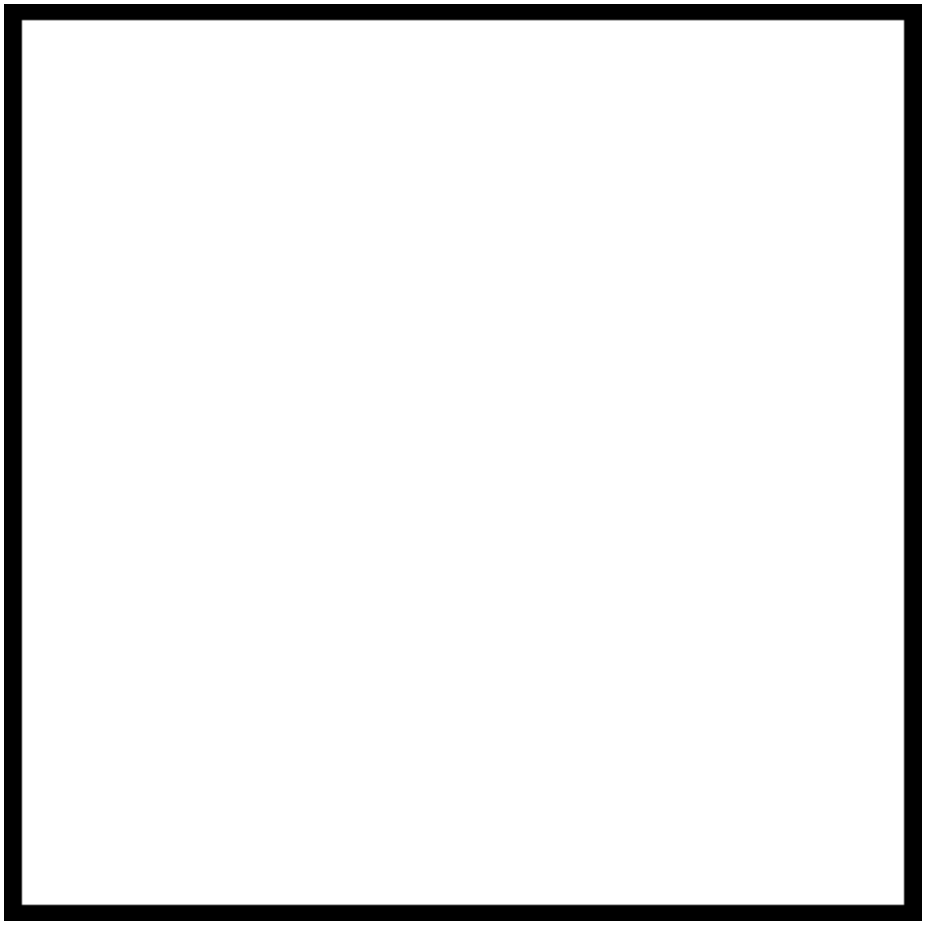

Figure 2: Over several hops of aggregation, the benefits of finding the interesting parts early

tasks are more io-bound than cpu-bound. Resources consumed by threads while doing nothing but waiting.

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

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

easy to distribute softare (npm etc)

4.3 summary of json

Why json?

A bridge between languages that isn't too different from the common types in the languages themselves a common bridge between languages

Also very simple. Easy to parse.

4.4 creating a losely coupled reader

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

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

Adding of semantics should could include adding new fields to objects (which could themselves contain large sub-trees) or a "push-down" refactor in which what was a root node is pushed down a level by being suspended from a new parent. See 3

(CITE: re-read citations from SOA)

4.5 Design of the jsonpath parser

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

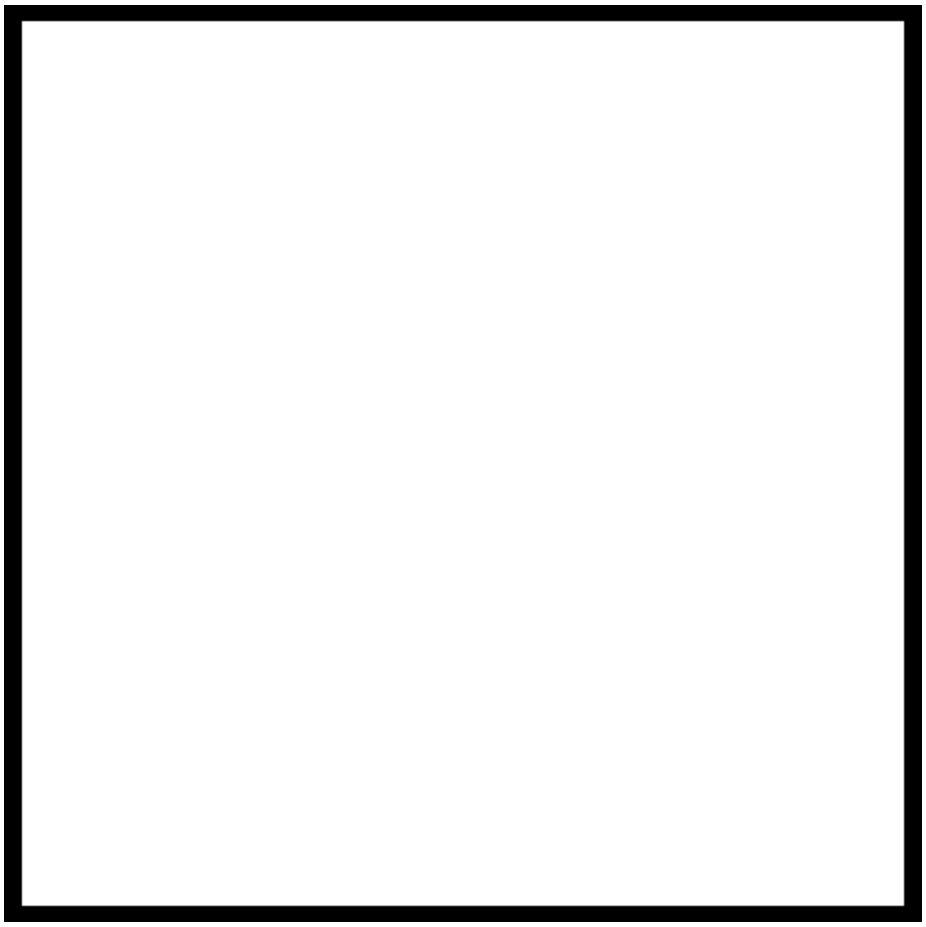

Figure 3: extended json rest service that still works - maybe do a table instead

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

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

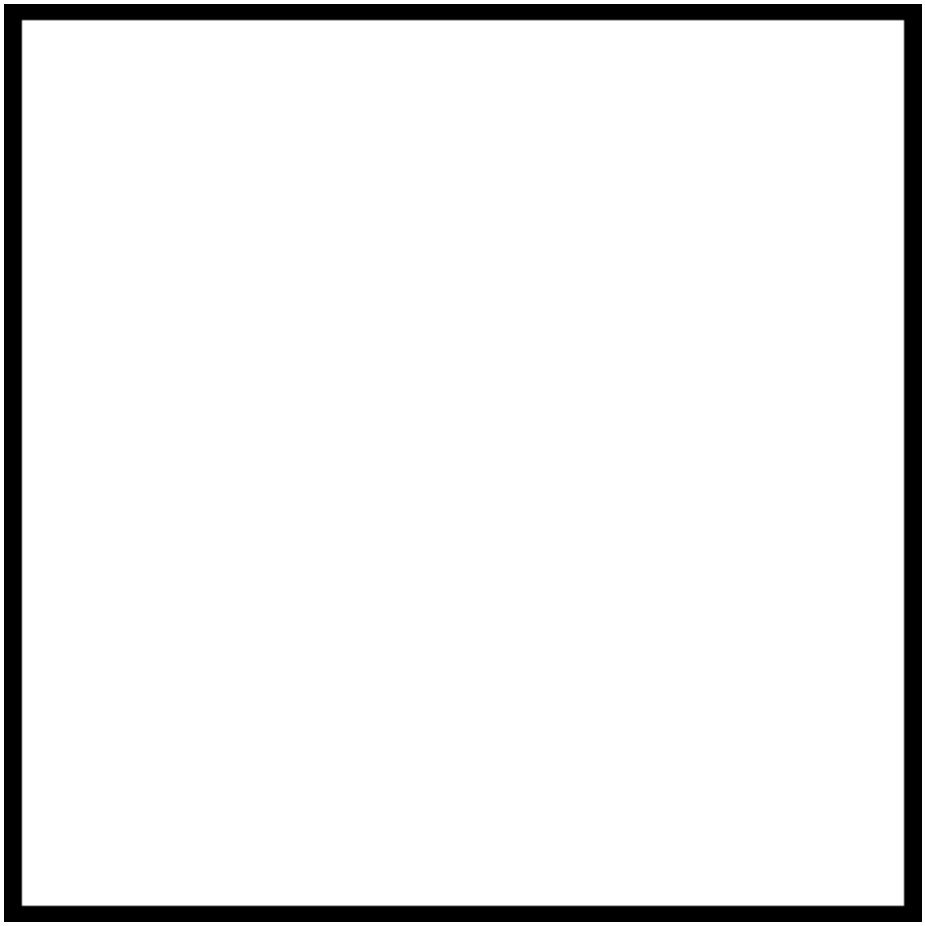

Figure 4: Diagram showing why list is more memory efficient - multiple handles into same structure with different starts, contrast with same as an array

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

4.6 Incrementally building up the content

Like SAX, calls from clarinet are entirely 'context free'. Ie, am told that there is a new object but without the preceding calls the root object is indistinguishable from a deeply nested object. Luckily, it should be easy to see that building up this context is a simple matter of maintaining a stack describing the descent from the root node to the current node.

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

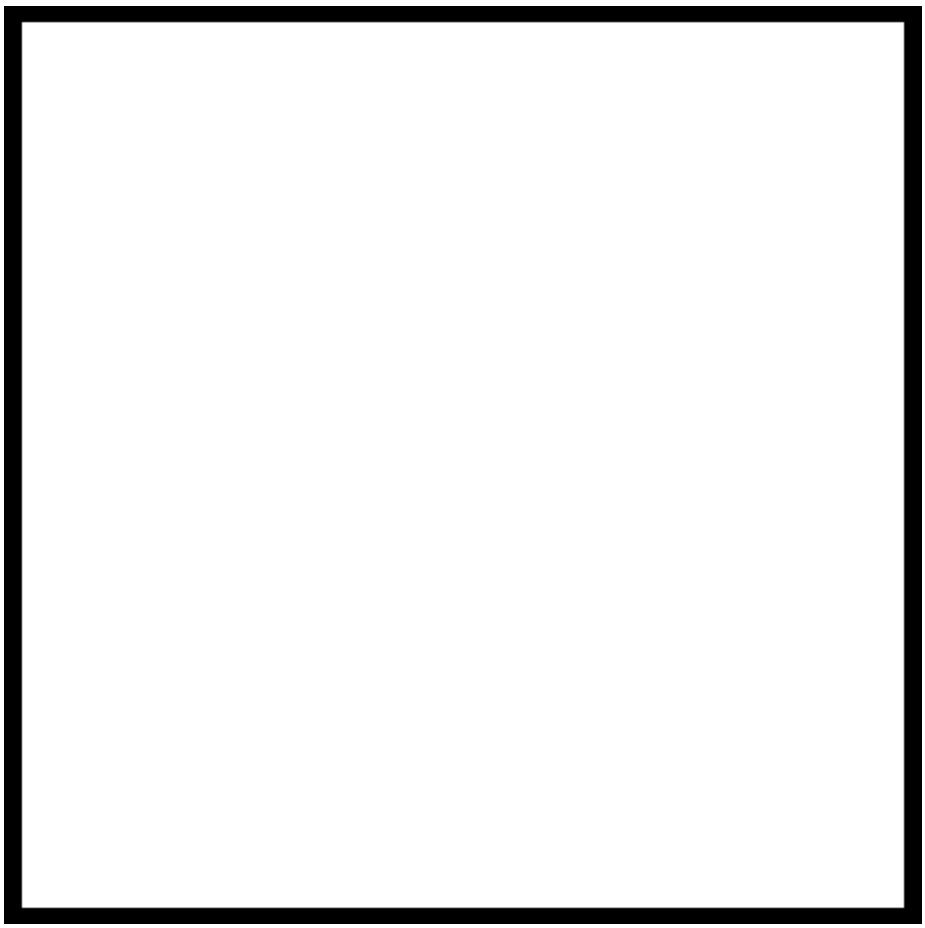

Figure 5: Show a call into a compiled jsonPath to explain coming from incrementalParsedContent with two lists, ie the paths and the objects and how they relate to each other. Can use links to show that object list contains objects that contain others on the list. Aubergine etc example might be a good one

Explain match starting from end of candidate path

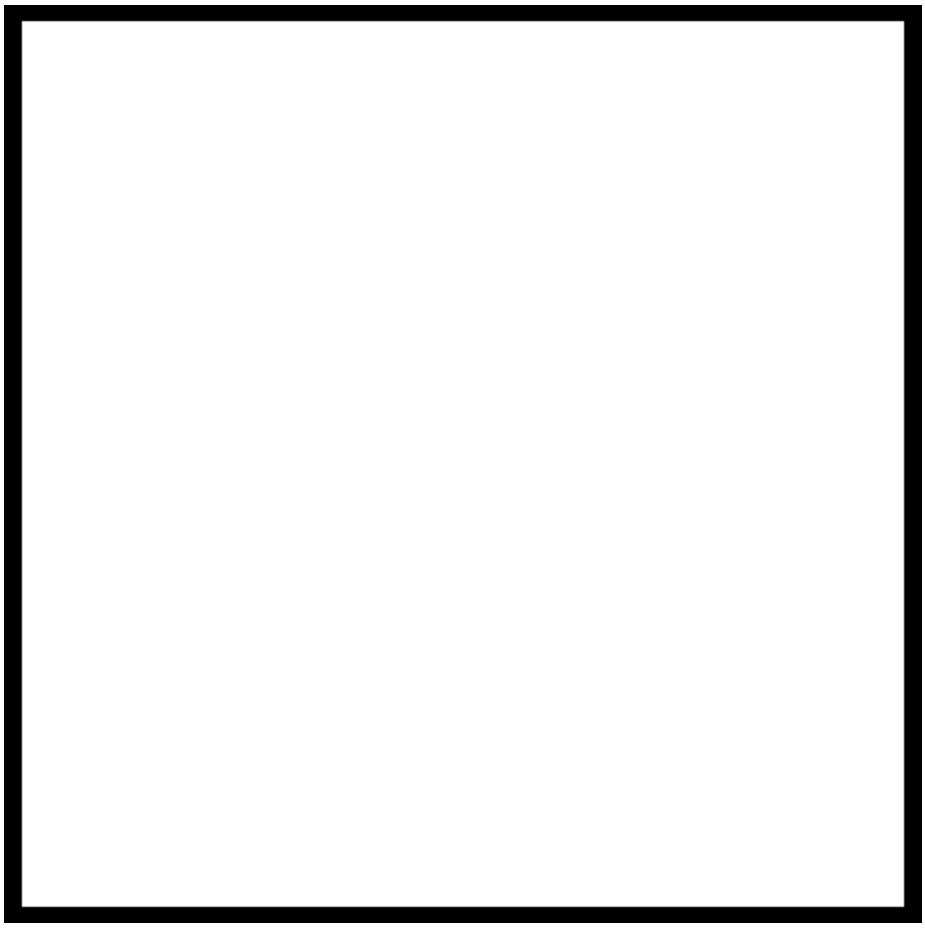

Figure 6: Some kind of diagram showing jsonPath expressions and functions partially completed to link back to the previous function. Include the statementExpr pointing to the last clause

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

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

4.7 identifying interesting objects in the stream

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

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

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

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

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

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

More important to efficiently detect or efficiently compile the patterns?

The failure of sax: requires programmer to do a lot of work to identify interesting things. Eg, to find tag address inside tag person with a given name, have to recognise three things while reieving a callback for every single element and attribute in the document. As a principle, the programmer should only have to handle the cases which are interesting to them, not wade manually through a haystack in search of a needle, which means the library should provide an expressive way of associating the nodes of interest with their targetted callbacks.

First way to identify an interesting thing is by its location in the document. In the absense of node typing beyond the categorisation as objects, arrays and various primative types, the key immediately mapping to the object is often taken as a lose concept of the type of the object. Quite fortunately, rather than because of a well considered object design, this tends to play well with automatically marshaling of domain objects expressed in a Java-style OO language because there is a stong tendency for field names – and by extension, 'get' methods – to be named after the *type* of the field, the name of the type also serving as a rough summary of the relationship between two objects. See figure 7 below.

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

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

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

Compare duck typing to the tag name in

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

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

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

let matchingObject = {
```

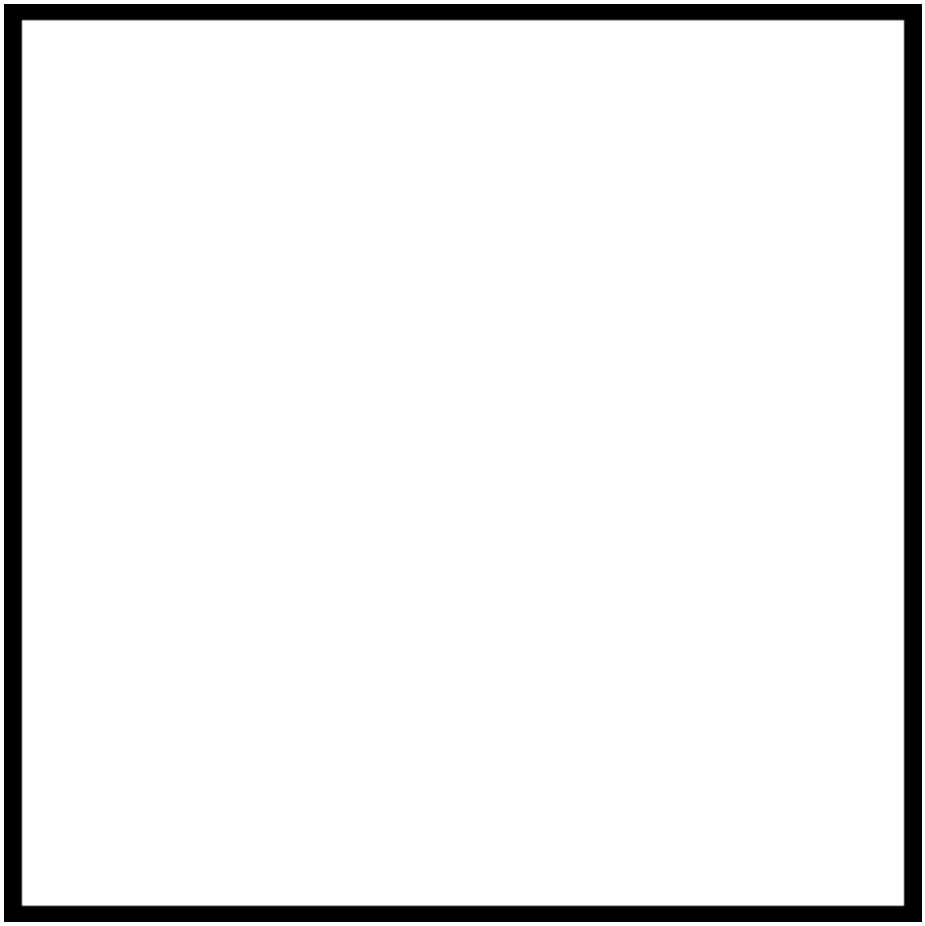

Figure 7: UML class diagram showing a person class in relationship with an address class. In implementation as Java the 'hasAddress' relationship would typically be reified as a getAddress method. This co-incidence of object type and the name of the field referring to the type lends itself well to the tendency for the immediate key before an object to be taken as the type when Java models are marshaled into json

```
   "name": "...",
   "address": "...",
   "email": "...:
}

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

When we aer searching

4.8 program design

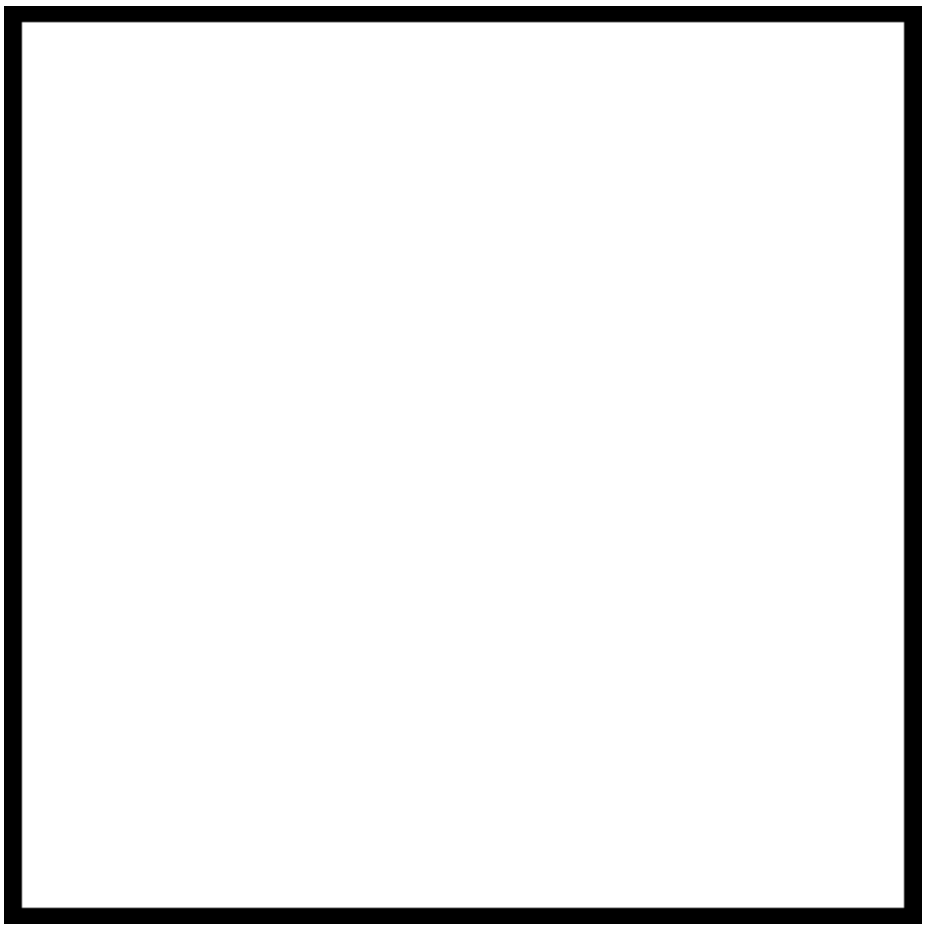

Figure 8: Overall design of Oboe.js. Nodes in the diagram represent division of control so far that it has been split into different files.

4.9 incrementally building up a model

A refactoring was used to separate logic and state:

- Take stateful code
- Refactor until there is just one stateful item
- This means that that item is reassigned rather than mutated
- Make stateless by making all functions take and return an instance of that item

- Replace all assignment of the single stateful var with a return statement
- Create a simple, separate stateful controller that just updates the state to that returned from the calls

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

How do notifications fit into this?

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

Stateful controller very easy to test - only 1 function.

4.10 styles of programming

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

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

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

The style of implementation of the generator of functions corresponding to json path expressions is reminiscent of a traditional parser generator, although rather

than generating source, functions are dynamically composed. Reflecting on this, parser gens only went to source to break out of the ability to compose the expressive power of the language itself from inside the language itself. With a functional approach, assembly from very small pieces gives a similar level of expressivity as writing the logic out as source code.

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

Different ways to do currying below:

```
// function factory pattern (CITEME)
function foo(a,b,c) {
   return function partiallyCompleted(d,e,f) {

      // may refer to partiallyCompleted in here
   }
}

function fooBar(a,b,c,d,e,f) {
}

partial(fooBar, a,b);
```

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

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

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

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

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

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

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

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

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

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

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

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

4.11 The mutability problem

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

Why this is a problem.

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

Potential solutions:

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

4.12 Performance implications of functional javascript

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

http://rfrn.org/~shu/2013/03/20/two-reasons-functional-style-is-slow-in-spidermonkey.html 9571 ms vs 504 ms

Also: copy that guy's website dude!

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

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

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

JS is much faster with "monomorphic call sites"

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

4.13 targeting node and the browser

Node+browser To use Node.js and

Need to build an abstraction layer over xhr/xhr2/node

Use best of the capabilities of each.

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

Why distributed javascript is more like a binary than a source file. Licencing implications?

Inherent hiding by wrapping in a scope.

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

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

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

Packaging for use in frameworks. * Many frameworks already come with a wrapper arround the browser's inbuilt ajax capabilities ** they don't add to the capabilities but present a nicer interface

- I'm not doing it but others are ** browser-packaged version should be use agnostic and therefore amenable to packaging in this way

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

Why not require * What it is * Why so popular ** Why a loader is necessary - js doesn't come with an import statement** How it can be done in the language itself without an import statement * Meant more for AMD than for single-load code ** Situations AMD is good for - large site, most visitors don't need all the code loaded** Depends on run-time component to be loaded even after code has been optimised ** Small compatible versions exist that just do loading (almond) * Why ultimately not suitable for a library like this

Why testing post-concatenation is good idea.

4.15 polyfilling

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

Node doesn't require, built on v8.

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

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

4.16 automated testing

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

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

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

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

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

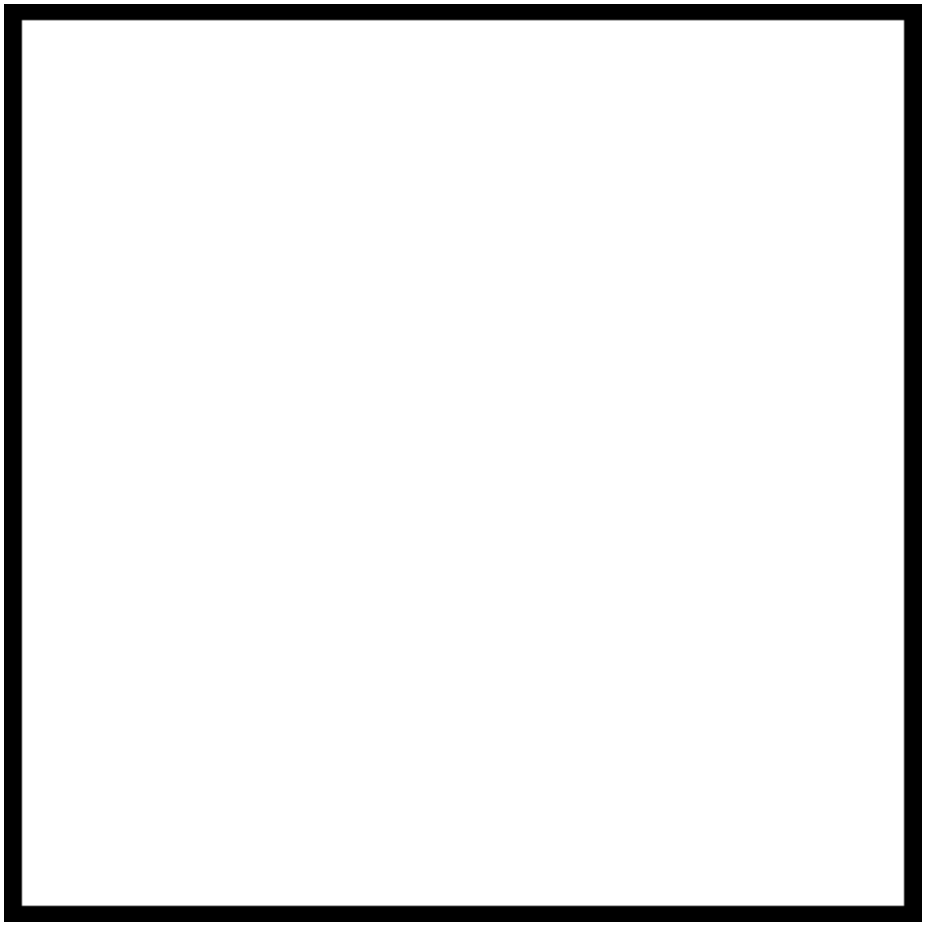

Figure 9: The testing pyramid is a common concept, relying on the assumption that verification of small parts provides a solid base from which to compose system-level behaviours. A Lot of testing is done on the low-level components of the system, whereas for the high-level tests only smoke tests are provided.

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

Tests deal with the problem of "irreducible complexity" - when a program is made out of parts whose correct behaviour cannot be observed without all of the program. Allows smaller units to be verified before verifying the whole.

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

Testing via node to give something to test against - slowserver. Proxy.

The test pyramid concept 9 fits in well with the hiding that is provided. Under the testing pyramid only very high level behaviours are tested as ??? tests. While this is a lucky co-incidence, it is also an unavoidable restriction. Once compiled into a single source file, the individual components are hidden, callable only from withing their closure. Hence, it would not be possible to test the composed parts individually post-concatenation into a single javascript file, not even via a workarround for data hiding such as found in Java's reflection. Whereas in Java the protection is a means of protecting otherwise addressable resources, once a function is trapped inside a javascript closure without external exposure it is not just protected but, appearing in no namespaces, inherently unreferenceable.

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

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

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

By testing individual tokens are correct and the use of those tokens as a wider expression, am testing the same thing twice. Arguably, redundant effort. But may simply be easier to write in that way - software is written by a human in a certain order and if we take a bottom-up approach to some of that design, each layer is easier to create if we first know the layers that it sits on are sound. Writing complex regular expressions is still programming and it is more difficult

to test them completely when wrapped in rather a lot more logic than directly. For example, a regex which matches "{a,b}" or "{a}" but not "{a,}" is not trivial.

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

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

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

Could implement a resume function for if transmission stops halfway

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

4.17 Inversion of Control

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

However, can still follow the general principles.

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

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

4.18 stability over upgrades

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

4.19 support for older browsers

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

4.20 suitability for databases

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

4.21 weaknesses

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

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

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

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

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

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

5 Conclusion

1 to 5 pages

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

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

Not particularly useful reading from local files.

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

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

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

5.1 Comparative usages

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

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

Consider: * Difficulty to program * Ease of reading the program / clarity of code * Resources consumed * Performance (time) taken – about the same. Can react equally quickly to io in progress, both largely io bound.

5.2 Community reaction

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

6 Bibliography

7 Appendix