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

100-200 words

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.

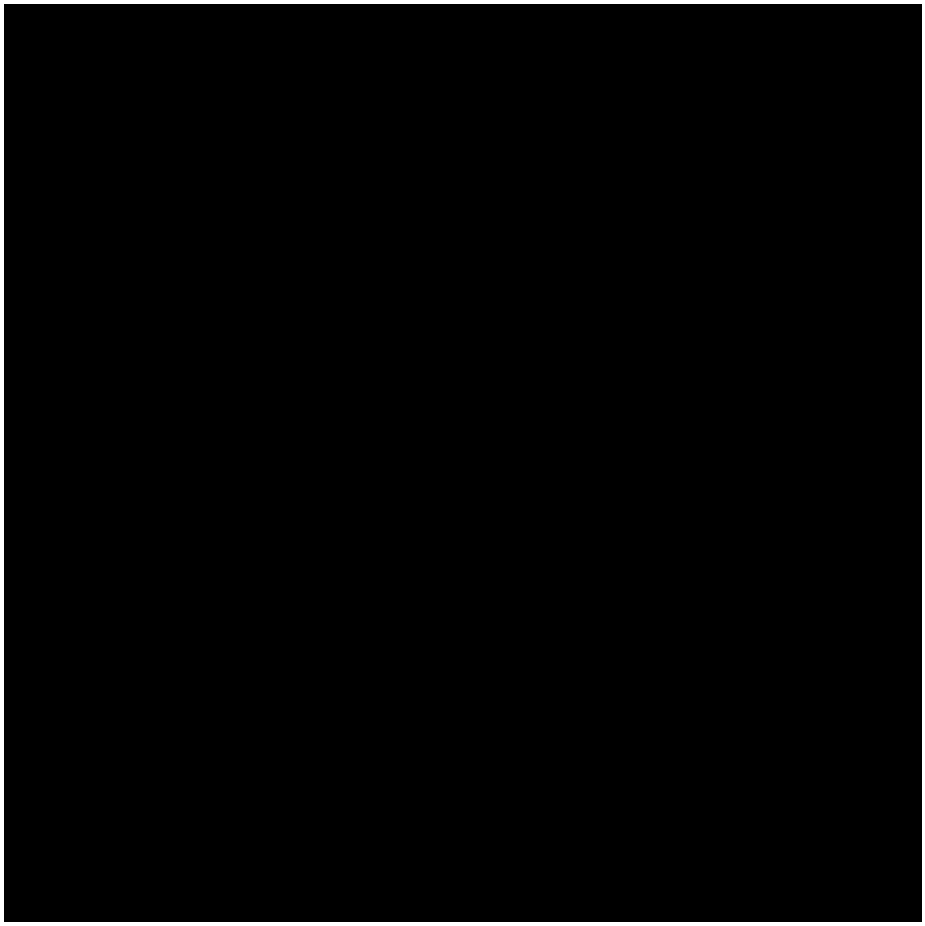

Figure 1: Potential differences in overall time taken to download a list of publications and then download any ones newer than a certain date. Assuming the publications are ordered newest first, the first connection may be terminated as soon as an older publication is found.

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

3 Background

background should be 2-10 pages

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.

3.1 Parsing: SAX and Dom

Why sax is difficult

DOM parser can be built on a SAX parser

3.2 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 addressss = 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.

3.3 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.

3.4 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!

3.5 Node

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

3.6 XmlHttpRequest

XmlHttpRequest (XHR)

Xhr2 and the .onprogress callback

4 Applicaiton and Reflection

40 to 60 pages

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

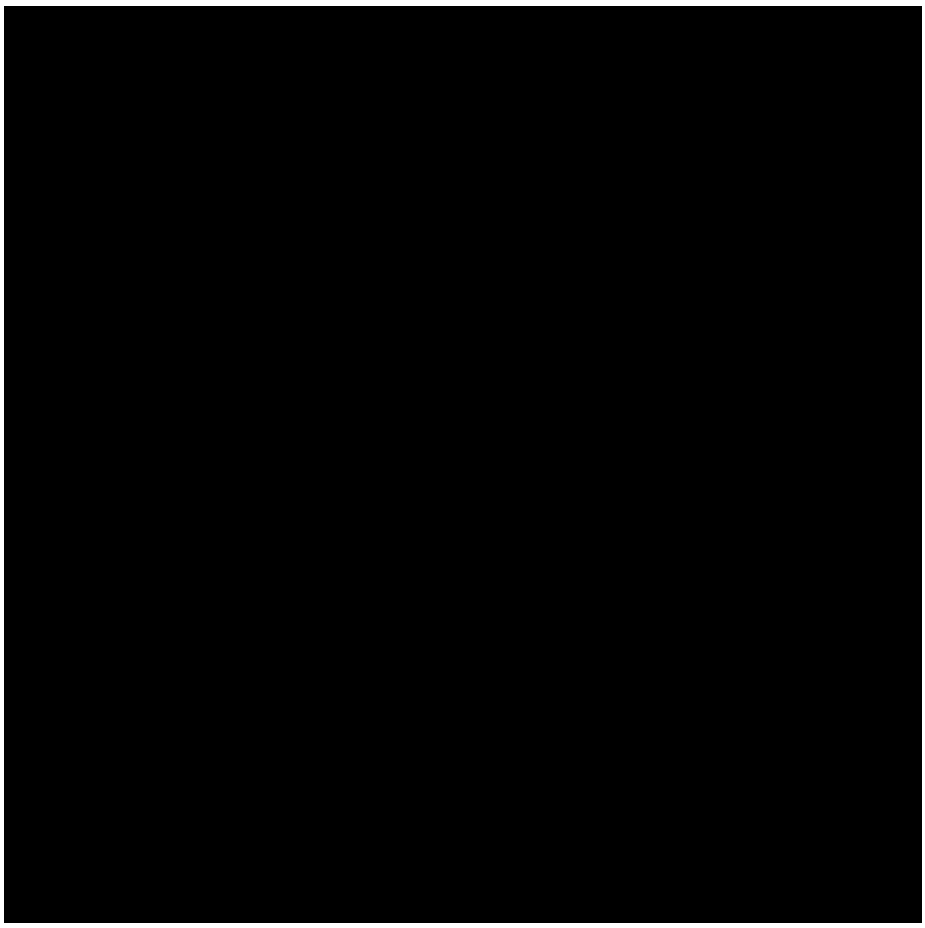

Figure 2: Over several hops of aggregation, the benefits of finding the interesting parts early

4.1 choice of technologies

can justify why js as:

Most widely deployable.

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

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

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

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

Compare to Erlang. Waiter model. Node 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)

4.2 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.

4.3 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)

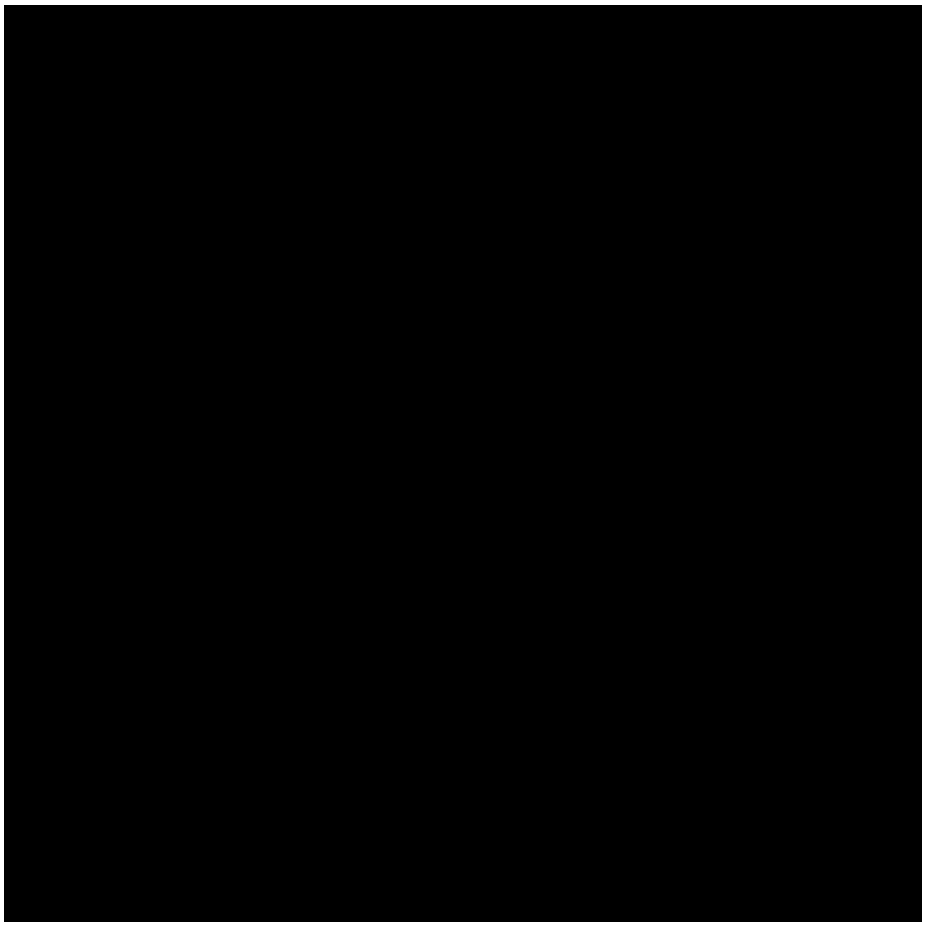

Figure 3: extended json rest service that still works - maybe do a table instead

4.4 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. Not very good, won't persue further.

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 4 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 = {
   "name": "...",
   "address": "...",
   "email": "...:
}

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

When we aer searching

4.5 breaking out of big/small tradeoff

Best of both modes

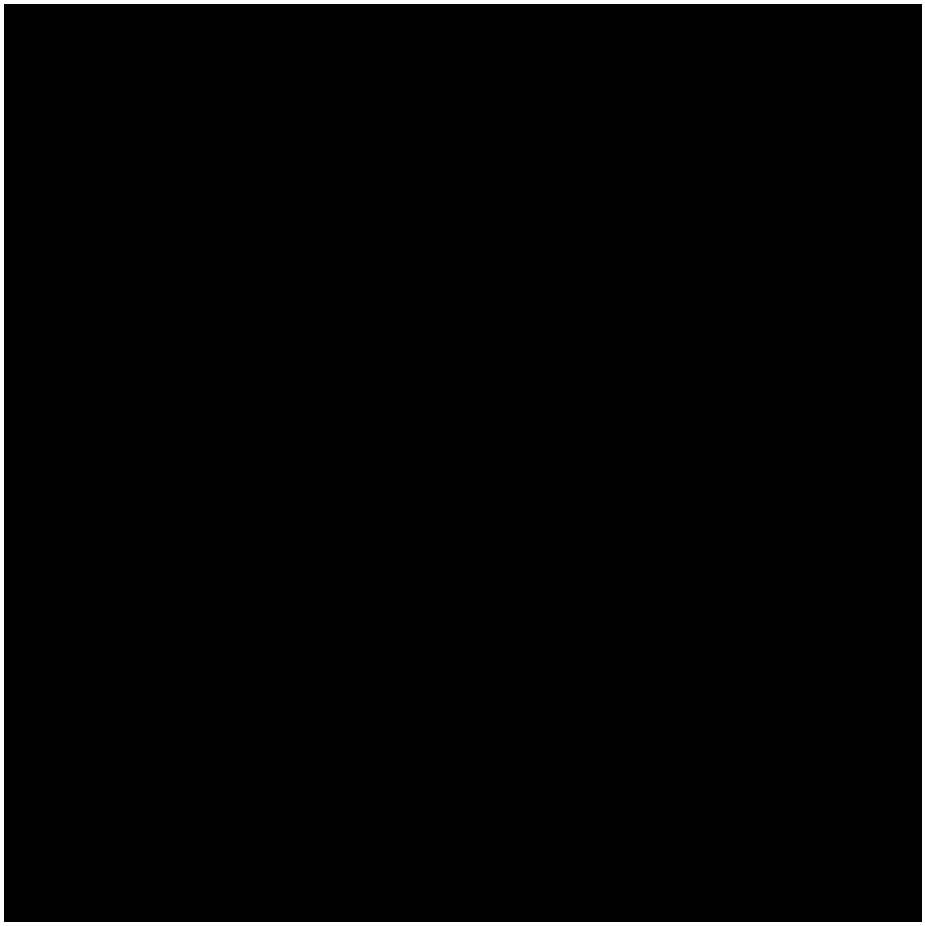

Figure 4: UML class diagram showing a person class in relationship with an address class. In implementation as Java the 'hasAddress' relationship would typically be reified as a getAddress method. This co-incidence of object type and the name of the field referring to the type lends itself well to the tendency for the immediate key before an object to be taken as the type when Java models are marshaled into json

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.

4.6 program design

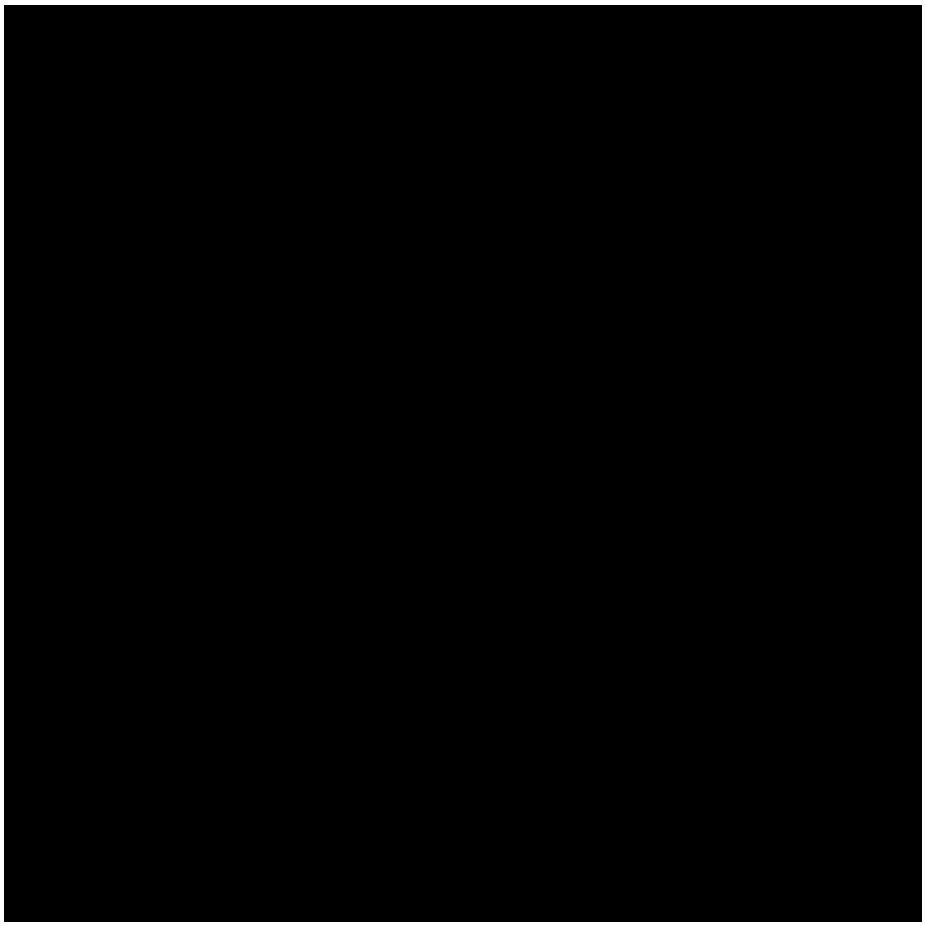

Figure 5: Overall design of Oboe.js. Nodes in the diagram represent division of control so far that it has been split into different files.

4.7 styles of programming

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

4.8 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.

4.9 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.

4.10 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.

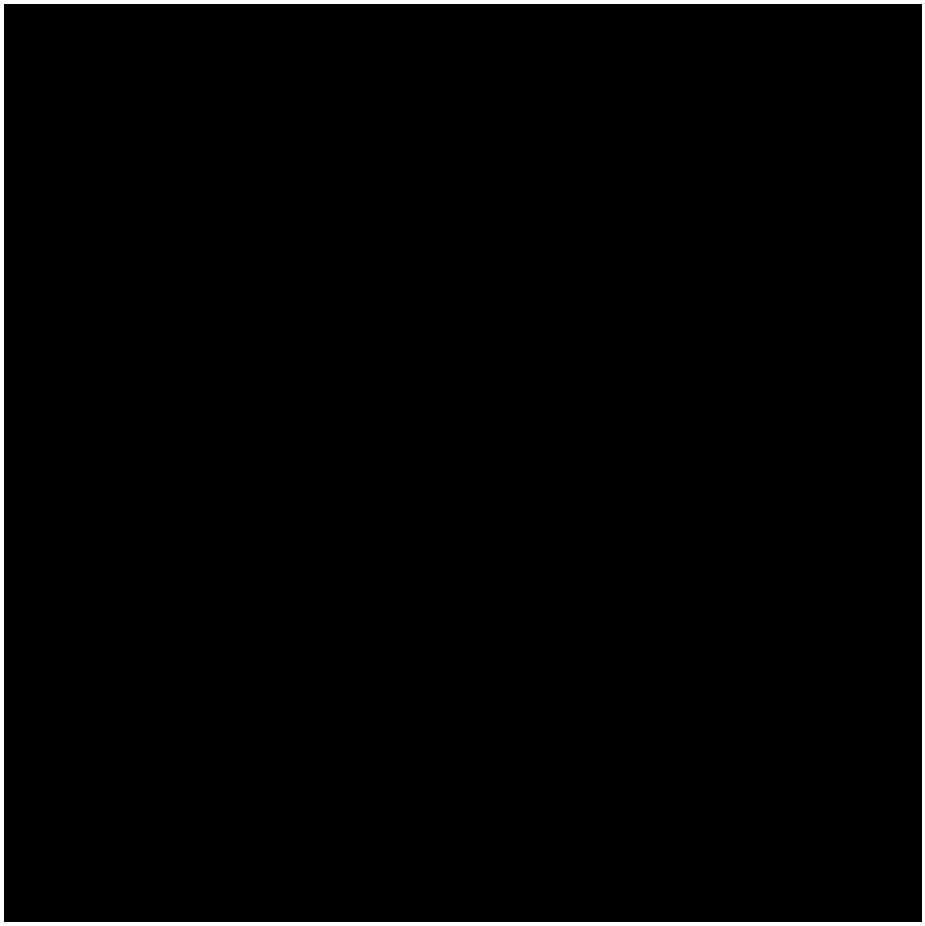

Figure 6: The testing pyramid is a common concept, relying on the assumption that verification of small parts provides a solid base from which to compose system-level behaviours. A Lot of testing is done on the low-level components of the system, whereas for the high-level tests only smoke tests are 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.

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

The test pyramid concept 6 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();
}
```

4.11 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

4.12 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)

4.13 suitability for databases

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

4.14 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

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