McCord Document Conversion

This page summarises the results of a number of techniques for converting the McCord XML data into a JSON form suitable to be stored in CouchDb. A sample of the current XML schema can be seen at https://source.fluidproject.org/svn/fluid/engage/trunk/src/testdata/artifacts. Note that this schema is not stable, and is expected to change repeatedly over the course of the project. Indeed, each museum will likely have their own variations on a schema, and so our approach fundamentally embraces this variability.

We have evaluated multiple approaches, in several languages. In particular, "pure" benchmarks written in Python and Java, using the respective languages' XML parsers, and Javascript benchmarks written in three styles -

1. using the "fastXmlPull" parser which Fluid uses to parse HTML for its Renderer
2. using John Resig's HTML parser (now delivered in env.js). Resig's parser was first announced on his blog at http://ejohn.org/blog/pure-javascript-html-parser/ and the freshest instance of the source seems to be held in his github for env.js at http://github.com/jeresig/env-js/tree/master
3. using the browser's native DOM, available as the return from an XmlHttpRequest
4. Thomas Frank's "xml2json" parser, held at http://www.thomasfrank.se/xml_to_json.html. Frank references a better implementation held at www.terracoder.com, however this site as of the time of writing has been down for several weeks.

The Python code is currently attached to JIRA FLUID-2816, whereas the Java and Javascript tests are held in incubator SVN at https://source.fluidproject.org/svn/incubator/engage-sketches/trunk.

Javascript measurements in-browser

Firstly the in-browser measurements, performed in various browsers on different machines:

Machine /Browser	fastXmlPull	Resig	DOM	Frank
Antranig/FF3	40 documents in 1057ms: 26.425ms per call	4 documents in 1604ms: 401ms per call	40 documents in 1132ms: 28.3ms per call	4 documents in 3247ms: 811.75ms per call
Justin/FF3	16.8ms per call	467.75ms per call	17.175ms per call	
Yura/FF3	40 documents in 770ms: 19.25ms per call	4 documents in 2399ms: 599.75ms per call	40 documents in 669ms: 16.725ms per call	
Antranig/FF2	40 documents in 2187ms: 54.675ms per call	4 documents in 2422ms: 605.5ms per call	40 documents in 3406ms: 85.15ms per call	parsed 4 documents in 2328ms: 582ms per call
Antranig/FF3.5b4	40 documents in 951ms: 23.775ms per call	4 documents in 1666ms: 416.5ms per call	40 documents in 756ms: 18.9ms per call	4 documents in 2071ms: 517.75ms per call
Justin/Chrome	8.2ms per call	41.75 ms per call	10.5 ms per call	

The first three lines establish basic normalisation between the three machines. Basically Antranig:Yura:Justin is around in a ratio 1.5:1.2:1

Observations on Javascript results

The Frank and Resig parsers are extremely slow. Whilst they make good use of RegExps to potentially accelerate performance in browsers, their overall approach to parsing is very suboptimal, frequently duplicating the document text in memory several times over during the course of parsing. The Frank parser in particular makes very frequent use of "eval" which will result in poor performance on any runtime. These two parsers take roughly an order of magnitude longer than the other approaches.

NB - the Resig parser is somewhat specialised for HTML, at the expense of behaviour on XML documents. The base code has been tweaked in a few instances to allow parsing to continue on general XML but the produced JSON is not correct, as a result of an assumption in the code that any node type which is capable of containing CDATA is not also capable of containing child nodes. However, the overall run time is probably still a good estimate of the speed of this parser.

The "fastXmlPull" parser (the Fluid homegrown approach) is in general the fastest. Despite the name it is actually capable of parsing realistic browser HTML as well as XML. Whilst on some platforms, the DOM approach appears marginally faster (on the very latest Firefox beta at 19ms against 25ms) these tests give the DOM method an unfair advantage since they only measure the cost of *iteration* over the DOM. The process of building the DOM structure itself occurs inside the browser's XHR engine and is not possible to profile separately from the overall fetch process for the files. Once this is taken into account, fastXmlPull would probably have equal or greater performance on all platforms.

In terms of general strategy, fastXmlPull is optimised for "slow" browser environments. In "last-generation" Javascript engines, there is a considerable performance premium on applying native operations such as RegExps and indexOf (see blog commentary at How Long, Oh Lord). However, the more modern VMs such as those in Firefox 3.5 and Chrome in Google will increasingly make this approach suboptimal as the speed of Javascript language primitives increases. A taste of this can be seen in the form of the narrowing headroom between FastXmlPull (strongly regexp based) and the other approaches on the more modern VMs. For truly excellent performance in these modern JS environments, we will require a ground-up rewrite of the parser, more along the lines of the "FSM character creeping" model seen in highly successful XML parsers in environments like Java, such as Aleksander Slominski's XPP3 parser that we benchmark later on.

Measurements in Python and Java

Now, for measurements in other languages. Yura on his machine has run benchmarks on CPython and JPython, and on mine I ran an equivalent conversion in Java:

Machine/Language	Reps	Total time	Per doc

Yura/Jython	20000	jython:real 2m27.557s, user 2m26.105s, sys 0m1.316s	7ms/doc
Yura/CPython 2.6	20000	real 1m6.248s, user 1m4.452s, sys 0m0.480s	3.3ms/doc
Yura/CPython 3	20000	real 1m4.939s, user 1m3.556s, sys 0m0.360s	3.3ms/doc
Yura/CPython	400	real 0m1.408s, user 0m1.344s, sys 0m0.040s	3.5ms/doc
Antranig/Java	4000	3600ms	0.9ms/doc

So, the above figures show that the best Javascript performance we have, in Chrome, broadly equivalent to the worst Python performance, in Jython. The pure Java implementation is at least 3x faster than the best Python performance. In Python we use the framework standard "xml.sax.xmlreader" package, whereas in Java we use the XPP3 pull parser implementation from Aleksander Slominski.

This test set only contains 4 documents, read repeatedly from the filesystem. Therefore it neglects caching and memory image effects. These documents are around 10k each - so the Java performance equates to a conversion speed of around 10MB/sec, which would be roughly the expected sustained read speed from a fairly good disk. If the files were kept compressed in a ZIP volume, the CPU cost would begin to dominate.

Persisting the JSON Data in CouchDB

In terms of writing the converted JSON documents to store, we can look at the following resource: http://aartemenko.com/texts/couchdb-bulk-inserts-performance/. McCord indicates that they have around 117,000 records of this form - corresponding to around 1.2Gb of raw data. The number of records puts us towards the left end of the CouchDB insertion graphs - insertion rate is still high. The second graph indicates that the very largest bulk sizes will lead to highest insertion rate - perhaps in excess of 1500 documents/second. This again would place conversion speed as the bottleneck - on this hardware, CouchDb can perform an insert in around 0.6ms. This would imply buffering data in units of around 100Mb. If the fileset and DB were on the same machine, alternating reading and writing in these large units would also lead to better use of the machine's I/O capacity.

Summary

Runtime	Time (ms /doc)	Throughput (Mb /sec)	Total McCord XML-JSON Conversion (minutes)
Python C	3.5	2.86	6.99
Python JVM	7	1.43	13.99
JavaScript JVM	41	0.244	81.97
Java 6	0.7	14.23	1.40

Some ballpark estimates for whole workflow—including conversion and persistence—on the various platforms:

- If Java were used for the conversion, we would expect to be able to commit the set of 117,000 documents into CouchDb in around 3 minutes.
- In Python, this time would probably extend to around 8-9 minutes.
- Perhaps a Javascript V8 (Chrome-like) solution might take more than 15 minutes.
- A Rhino Javascript solution, performing somewhat Firefox 2-like, might take 1-2 hours.