

Writing a Social Content Engine with RDF

Anselm Hook
anselm@hook.org

Sept 2004

Contents

1	Rev your engines	3
1.1	Features of a Social Context Engine	3
1.1.1	Publish	3
1.1.2	Categorize	3
1.1.3	Discover	3
2	Antecedents & Influences	4
2.1	del.icio.us	4
2.2	Flickr	4
2.3	Webjay	4
3	Goals	4
3.1	Serendipity	4
3.2	Collaboration	4
4	Components	5
4.1	RDF triple-store	5
4.2	Content management system	5
4.3	Tag engine	5
4.4	XML server gateway	5
4.5	Javascript client-side User Interface	6

5	Framework	6
5.1	3rd Party Components	6
5.1.1	Sun's Java SDK	6
5.1.2	PERST	6
5.1.3	Jena's ARP	6
5.1.4	Jetty	6
5.1.5	Ant	6
6	Ideas Driving the Project	6
6.1	Signalling	6
6.2	Implicit Similar Behavior	7
6.3	Signal to Noise Ratio	7
6.4	Building the Feedback Loop	8
7	Getting Our Hands Dirty	8
7.1	Simply a Web-site	8
7.2	Application Candidates	9
7.2.1	Jetty	9
7.3	Data Store	11
7.3.1	RDF Triple Store	12
7.3.2	3rd Party Triple Stores	12
7.3.3	Roll Our Own RDF Triple Store	13
7.3.4	Mapping Object Representation to RDF	14
7.4	Getting Real with RDF	21
7.4.1	A Word to the Wise	21
7.4.2	Universal Solvent	21
7.4.3	Coming Clean	23
7.5	Navigating via Tags, Streams and Crumpled URLs	23
7.5.1	Tags	24
7.5.2	Crumpled URLs	25
7.6	The Query Engine	27
7.7	Javascript User Interface	29
7.7.1	Drawbacks to using Javascript	30
7.7.2	Uses of our Javascript Client	30
8	Conclusion	31
8.1	What could you do with this?	32
8.1.1	Make your own Craigs List	32

8.1.2	Personal knowledge tracking system	32
8.1.3	Big Bucket	32
8.2	The Next 10 Years	32

1 Rev your engines

1.1 Features of a Social Context Engine

Today we're going to build a social content engine for

This service for organizing and sharing content with our friends will allow you to:

1.1.1 Publish

observations or *stuff* onto a website.

1.1.2 Categorize

it variety of ways.

1.1.3 Discover

other related topics or persons.

2 Antecedents & Influences

2.1 [del.icio.us](#)

2.2 [Flickr](#)

2.3 [Webjay](#)

The code itself will be a rewrite based on what I've learned from developing Thingster and BooksWeLike over the last year.

3 Goals

3.1 Serendipity

Social content services have a strong emphasis on implicit social discovery. Users use these services to organize their own content for later recollection.

3.2 Collaboration

Since the services are public, other users can peek into the collective space, and discover similar items, topics or persons. We're going to look for opportunities in this project to stress the 'synthesis' aspect of social discovery; to escape from the pattern of curated collections managed and presented by one person.

4 Components

4.1 RDF triple-store

We are going to push RDF quite hard. We are going to write our own lightweight persistent and embeddable RDF triple store in Java - possibly being the first people to do so. This will be the cornerstone of our application and represents significant value even beyond this particular project. We'll also seek to use official RDF vocabularies as much as possible. We want to have something that is not only functional for our own use but that can interact with the rich ecology of the web - publishing data via RSS or RDF/XML to a wide variety of other services.

4.2 Content management system

4.3 Tag engine

One of the specific things we're going to build into our service is a 'tags' mechanism, as popularized by deli.cio.us, used to categorize our observations. Users will be able to publish tags to categorize items of interest and other users will be able to pivot on those tags to discover items of like interest.

4.4 XML server gateway

Overall the pattern of the finished project is to build an XML driven web-service built on top of industrial strength concepts that can be re-used for almost any conceivable knowledge management application.

4.5 Javascript client-side User Interface

We are also going to push Javascript quite hard to express the client side interface. Again we will seek to build fairly powerful components that will have significant reuse for other projects.

5 Framework

5.1 3rd Party Components

5.1.1 Sun's Java SDK

5.1.2 PERST

5.1.3 Jena's ARP

5.1.4 Jetty

5.1.5 Ant

The results should be quite fun to drive and fairly industrial.

6 Ideas Driving the Project

6.1 Signalling

There's an old saw that goes *actions speak louder than words*. A car can have its left signal flashing but be travelling blindly down the road not turn-

ing at all ... Or oncoming traffic may suddenly and mysteriously slow down suggesting the presence of a fine officer of the law doing his part to help keep a community orderly – or even just a kid crossing the street without illuminating the crosswalk signal.

In vehicular traffic drivers wheel and race making moment to moment decisions on the basis of each others inputs; signalling to each other in a variety of both intentional and unintentional ways. As a participant you end up creating a mental model of the things around you, the situational landscape, and the best navigation choices.

6.2 Implicit Similar Behavior

Personal activity on the net, when tracked, could show more of what is valued highly than what a person 'says' is valued highly. If we could watch what people were tracking, keeping and organizing for their own personal recollection then we could perhaps have better insight into what was valuable overall. In fact this could be better than if those same people told us explicitly what they thought was valuable.

People on the net do of course signal to each other with a variety of intentional and explicit mechanisms. There is craigslist, vanilla websites, listservs, email, wikis, moblogging sites and on and on.

6.3 Signal to Noise Ratio

But that space has started falling over. There is incessant spam, and almost everything has become saturated with 'adwords by google'. The language and phrasing of traditional content has steered sharply towards maximizing ad revenue. The intentional signals are polluted and noise-ridden.

On the other hand implicit behavior has not been tracked - at least not democratically. Watching flocks of humans pinwheel about has up until now been the domain of web portals. Now we're seeing this become more democratic

as new peer-visible psychographic behaviour tracking services such as A9 and Ask Jeeves are rolled out.

6.4 Building the Feedback Loop

The newer services that are emerging seem to have few parallels to existing services. Wikipedia of course does offer social benefit but it has content organized and massaged by hand. Orkut, Friendster, Multiply, LinkedIn? are social but don't have any particular organizational utility; there is no personal activity that others observe - most behavior is explicit. Craigslist? and Meetup and Upcoming do provide community but the signalling is all explicit again.

The newer services seem to be simply more pleasing in some way; more human. They do automate the synthesis of many peoples observations and that immediacy is more satisfying than dealing with slow curated collections. There is simply something pleasing about feeling as if there is zero-latency between oneself and ones peers; an instinctive sense of connectedness perhaps.

7 Getting Our Hands Dirty

We can take the set of casual observations above and keep them in mind as we now begin to dig into the actual code. Likely as not you'll think of thousands of other things that I've scarcely considered.

7.1 Simply a Web-site

One thing that we do know is our service is simply a web-site.

We don't have to think much about "what kind" of web-site yet. And in fact we'd prefer not to. We'd like to pluck away all the orthogonal pieces and

erase them from consideration as early as possible.

Since this "serving web content" is a well defined goal we can at least take it off our list. This will reduce the total number of things that we have to think about.

7.2 Application Candidates

In broad strokes our candidate applications for serving web-pages are going to be either Apache, Mason, Tomcat, Jetty or even possibly just `mod_perl` or `cgi` support. My personal experience is that `perl`, Mason and `mod_perl` have too many dependencies to ever be run in embeddable environments. Admittedly these are extremely pleasing and rapid development tools but one of the constraints of this project is portability across devices - where those devices are not necessarily running full blown LAMP or UNIX capable operating environments. Java is the only language that currently has widespread portability (well `C#` as well) and this leaves us with Jetty and Tomcat as choices.

7.2.1 Jetty

Jetty has become my personal favorite web-server of choice because it is quite visible and transparent all the way to the bottom. You can run it in an embedded mode and step-trace the logic all the way through to see what is going on. I have found and fixed bugs in my code because of this transparency. In particular being able to debug the application without having to 'attach' to a running warfile does tremendously expedite development. A lot of people talk about XP development processes; and my experience is that being able to step trace with a debugger can be just as effective.

Of course we are going to want to present a dynamic interface to the user. Traditionally people use Velocity templates or JSP templates to wrangle the user interface. In our case we are going to use Javascript and have the server serve static web-pages and dynamically generated XML content. What that

means is that for now we do not have to think about how we are going to develop a server that serves dynamic content. We just have to think about the basic server core.

Obviously we need a `main()` entry point of some kind.

Presumably it starts up Jetty and does Jetty like things. Such as starting up a Jetty Resource Handler that will handle the incoming user web requests.

Basically something like this:

```
static public void main(String[] args) {  
  
    server = new jetty server  
  
    context = new jetty context  
  
    session = new subclassed instace of a jetty handler  
  
}
```

Here we're not bothering to package up the system as a servlet. We want this to be easily accessible to the debugger and we're basically in a hurry overall. We want to build the whole project in less time than our boredom threshold. Considering how to package something as a servlet will multiply the total number of considerations in this project and create spurious complexity.

Jetty tells us that we need to subclass a Jetty Resource Handler to do actual work. In this case we invent a 'Session' concept that will be responsible for replying to user requests as per our application. In broad strokes this will look like this:

```
public class session extends org.mortbay.jetty.ResourceHandler? {
```

```
handle_event() {  
  
    if the request is for a vanilla web page then just return it  
  
    if the request is a database query then pass it off to some kind of query handler  
  
    return query results as an xml graph  
  
}  
}
```

Our Session handler will be shallow - we're going to push most of the work off to an XML query handler layer.

At this point therefore we are done at least one piece - at least conceptually. Please refer to the attached tarball to see the actual grunt work itself.

One complexity that we have to keep in mind is that multiple response handlers can be active at the same time so we'll have to remember to put semaphores or synchronized blocks around any code that isn't thread-safe. This will require a careful audit of the project when it is done.

7.3 Data Store

Now that we can "start up our app" we need to pick another piece to do. Our choices are the query layer or the user interface. But it really does seem like we are going to have to do a bit of real work now and deal with our actual persistent datastore. Since we have a `main()` entry point we should be able to do quick tests anything we now write. Writing the Triple Store

7.3.1 RDF Triple Store

The first piece of real work is to write a lightweight RDF Triple Store. This section will get the most discussion in fact; there are many details here.

Again here we don't have to think much about "what kind" of application is going to use the triple store. In a sense we're making a decision that will enforce design a priori - because of previous experience I happen to have with RDF and influences I've gotten from other people who have used RDF quite successfully.

RDF is a perpetually emerging and increasingly recognized notation for expressing the relationships between objects. It will be the cornerstone of this project and just about every other project that we walk through. Parsing

One of the things we need to do is to load up RDF content off disk. Although we're interested in writing a datastore we're actually not that terribly interested in writing an RDF parser. And excellent ones already exist. To load content into our RDF database we'll use Jena's RDF parser called ARP:

<http://www.hpl.hp.com/semweb/jena2.htm>

7.3.2 3rd Party Triple Stores

Another thing we need to do is to store stuff. We are not really keen on writing a BTree on Disk or some other storage system. Java 1.4 does support NIO - memory mapped IO and it is somewhat appealing to write our own system based on that. Also there are some rather bizarre systems such as Prevalyer which offer transparent persistence but I'm just not sure about the idea of inhaling hundreds of thousands of RDF triples every time we start up - regardless of performance. In this case we're going to go with PERST - which is a very nice datastore written by some crazy russian guy:

<http://www.garret.ru/~knizhnik/perst.html>

Nota Bene We could in fact avoid writing our own triple store if we used Jena or Kowari:

<http://www.hpl.hp.com/semweb/jena2.htm> <http://www.kowari.org>

And in fact we could just grab an open source blogging tool off the shelf:

<http://www.opensourcecms.org> <http://wordpress.org>

7.3.3 Roll Our Own RDF Triple Store

We're not going to go with the completely off-the-shelf solutions in this project because:

1. Part of what we want to do is to build a system that we can understand all the way to the bottom. Getting a comfort level with RDF and a few of its peculiarities will let us make better decisions when we want to pick that industrial strength RDF store for subsequent projects.
2. As well we may want to run this project as a mini-server on a local home PC or even on a cellphone type device. Our approach should be light enough to at least run on a circa 2004 HP IPaq and possibly even on newer smartphones. The fact is that building a triple-store is not hard given the power of tools such as PERST and ARP which we are going to leverage heavily.

The main thing we're going to miss is a real query language. In fact even in avoiding a full blown query language one effectively ends up writing ones own. Query languages do introduce complexity and unpredictability. But mostly what we're trying to do is to understand the landscape of RDF; how and why exactly one works with RDF *all the way down to the bottom* in a sense.

As far as I know nobody else has written an embeddable persistent Java based RDF Triple store yet. As soon as one comes out we can chuck all

of this code out the window - but to achieve our learning and portability constraints we are (for now) forced to use a solution that we write ourselves.

7.3.4 Mapping Object Representation to RDF

Another big question - possibly the biggest question of this entire project - is what is the best mapping between RDF (say in an XML file) and RDF in memory.

There are a number of excellent W3C sponsored articles on RDF mappings to RDBMS. (In this case we're looking for a mapping from RDF to an OODB - but the ideas are the same). This article:

http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/

and

http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/

talks about some of the data-type requirements and implementation issues than an RDF Store might have for example. Some of the completely reasonable considerations they cite are:

Text Searching

Storing URI's efficiently

Supporting Datatypes (int, float, string)

Supporting RDF Containers

Supporting RDF Schemas

Inferencing rules and reasoning hooks.

Triple Provenance (tracking what website a triple came from)

We're actually going to respectfully ignore quite a bit of this good advice - but it is worth reading.

Our RDF database is going to have only a single kind of persisted object - an RDF triple. Where an RDF Triple consists of a:

```
{ Subject, Predicate, Value }
```

Each of these parts can be represented in Java:

Subject represents a canonical URI string describing the topic at hand. It can be represented by a Java String.

Predicate describes a relationship such as "knows" or "owns" and can also be represented by a Java String. There is some argument that for conservation of memory one could store the XMLNS encoding of the predicate such as 'geo:long'. We'll store the whole unrolled predicate for now and revisit the idea later on possibly.

Value is either a literal such as "12" or "Mary" or alternatively it is a reference to another Subject. This can be either a literal of type integer, float, String or another Subject reference. Another consideration might be different language encodings for values. And yet another consideration might be providing full text search on the Values as well (probably best done using Lucene). We are going to just treat this as a string and not actually differentiate except by context of usage.

In Java our simple triple container would look like this:

```
public class Triple extends Persistent {  
  
    public String sub;  
  
    public String pred;
```

```
public String val;  
  
}
```

Nota Bene Even if we're not going to be formal we should be at least aware of the weaknesses of both the data model and the representation of that data model being used here:

It duplicates the same 'Subject' and 'Predicate' and 'Value' Strings over and over in the cache and even on disk. This is quite wasteful. Often in fact (say when implementing a PostgreSQL? based store) one would index all the strings once only in a shared global index. The triple-store can then just be integer keys that refer to the String Index. The problem with one common bucket of strings is that one doesn't know if wildcard matches are returning Subjects, Predicates or Values without also checking the triples - and this can be slow. An alternative would be to have three string pools. Another alternative could be using a key that is say an md5 or sha1 hash of the string in question; thus allowing exact searches without having to goto disk to discover the strings key value first. In any case these approaches are easy enough to retrofit under a working RDF store later on.

This approach doesn't specify the 'type' of a Value. One could argue that it is the role of an OWL based description to formalize those facts. The system will be able to store OWL terms just as it stores ordinary RDF content but at the same time we're not going to be writing any code to validate a collection of RDF triplets against an OWL definition.

Some people would also add 'provenance' here - turning the triple into a quad and tracking the originating site of each triple. For our purposes we will simply treat it as a 'String' for now and revisit the issue later on possibly. Yet more bulky RDF triple stores might specify 'owner' concepts on each triple for fine-grained privacy. I prefer to have concepts of ownership be 'in' the grammer itself rather going from triples to quads. Another consideration might be to date-stamp triples as well - again something we're not doing.

Another way to store RDF triples would be to bind all triples associated with

a given subject as a single Subject node. Doing this in Java would look like so:

```
public class Reference extends Persistent {  
  
    public String subject;  
  
    public Hashtable values = new Hashtable();  
  
}
```

Although we're not doing it this way - this second way does have a subtle advantage. It would allow a query engine to operate across disjoint database back ends. For example you might have a spatial database and a vanilla subject-sorted keyword index and you might want to return some features from each. Since each reference is fully self contained you could easily emit a stream of blended features - without having to duplicate those features into each database. This is a significant benefit - but again something we're not doing.

Yet another way to do this would be to use an IDL to generate your java objects from an OWL definition. This is completely insane but I can see cases where people might do it:

```
public class MyRDFPerson? {  
  
    public String uri;  
  
    public int age;  
  
    public float height;
```

```
}
```

We are going to use the first approach however we will wrap the triples inside of a Reference Class as exemplified above so that from the outside you won't really care about the implementation that much - and in fact it will be very easy to swap implementations even as far as switching to Jena or directly backing your persistence requirements with PostgreSQL?.

Here is what that Reference class is going to look like:

```
public class Reference {  
  
    String uri;  
  
    public String get(String predicate);  
  
    public String set(String predicate, String value, boolean allowDuplicates );  
  
}
```

The rules we'd think of normally associating with `set()` would say that duplicate predicates are not allowed per subject. In a Java class for example you can't say "int myvalue; int myvalue;". But in RDF this method can explicitly allow a given predicate to be declared more than once if `allowDuplicates` is true. You'd typically however want an `rdf:Bag`. Let's say that for example you wanted to associate several tags with a given subject - you'd want to declare a child bag that belongs to that subject and have that child cite all of the tags in question.

At this stage we have a concept of a 'Reference'. This acts as a bag for predicates and values associated with a given Subject. Sticking things together

What we need now is actual persistence and a way to manufacture and store handles on our Reference objects. Basically now we're going to just glue all of the pieces into one huge blob called 'Database'.

So this is where we call upon PERST to do the heavy lifting for us:

```
import org.mortbay.perst.*;

class DatabaseRoot? extends Persistent {

FieldIndex? subs;

FieldIndex? preds;

FieldIndex? vals;

}
```

This incantation declares 3 persistent field indexes using PERST. Now when we commit triples into the database we commit them to all 3 indexes. And to query for any triple we can query any of the indexes.

PERST supports range queries, exact queries, and "subject starts with" string queries. Queries can be done in forward or reverse index order.

For our needs this will suffice. For example:

If we want to discover all triples whose subject begins with `http://playground` then we ask PERST to efficiently search the `subs FieldIndex?` for subjects with that term. If we want to discover say all predicates that are `http://www.w3.org/2003/01/geo/long` then we do something similar. If we want to discover all values that are say greater than "118.35" and less than "120.35" we can do that as well using PERST. However to do more complex queries such as say find all things that are within a certain value

of predicate "geo:long" and predicate "geo:lat" we have to issue multiple queries and do explicit joins by hand. Technically speaking however one can actually avoid fully explicit joins (where one has a full copy of each set) by using java code to iterate through the second set with the first set in hand. (In the particular case where we are doing something that looks like a spatial query - we could use the spatial indexing that PERST provides).

There's one more piece on top of all this that we need to add. We need some concept of an overall "database" that can yield instances of References that the application logic can then manipulate. That database layer will wrap PERST completely; making it invisible to the outside world and will look something like this:

```
interface Database {  
  
public Reference get(String key);  
  
}
```

With a little bit of glue this layer is basically done. Please refer to the associated tar-ball for the exact details.

Now we're done most of the hard stuff. We just have to think about the user experience and build out some UI. Actually that will also be quite a bit of work - but hard in a different way - as we wander a thicket of possible UI choices.

7.4 Getting Real with RDF

7.4.1 A Word to the Wise

A lot of people wonder if RDF is really any kind of improvement over other ways of expressing objects. People often complain that RDF/XML is overly verbose and not human editable for example. And people do wonder if the same content couldn't be packaged under some other schema altogether. Here are some of my thoughts as a first-time-user from playing with RDF over the last few months:

These days I find it easiest to think of RDF documents as simply big buckets full of triplets consisting of { `subject`, `predicate`, `object` }. This corresponds fairly well to the definition of a simple english sentence being { `subject`, `verb`, `noun` }.

In RDF one can talk about 'decorating' any arbitrary subject with any arbitrary fact. It can be a reasonable way to think about and verbally discuss RDF system architecture in general - having something of a 'tools not rules' or 'just do it' flavor that can expedite thought. Subjects can be extended later on in the development process - meaning less time spent anticipating and pre-planning the system. Pre-planning and discussion time can be exponential with the number of elements that need to be considered and RDF can help de-stress that part of the work.

7.4.2 Universal Solvent

RDF implies an underlying database model (for better or worse). If you're using RDF triplets consisting of `subject`, `predicate`, `object` then you're likely to find yourself somewhat coerced into having a database implementation that consists of a huge bucket of RDF triplets (rather than say one with a lot of specialized schemas that are being translated to RDF dynamically). RDF forces debate up a level of abstraction. Using RDF/XML specifies agreement not only on the transport notation (XML), but now also on the database structure. Since RDF specifies a grammar - not simply a syntax

- it seems to coercively imply how that grammar is stored at least to some degree. Agents written to crawl one RDF database can potentially crawl another one that they were not originally meant to consider. Where people used to argue about how to transport data and how to structure meaning in the data now they are arguing about what the words mean.

RDF is something of a 'universal solvent'; things tend to be dissolveable in RDF whereas they are not dissolveable in other grammars. Some of the tension with other grammars such as say VRML, GML and the like come out of this fact: people want to represent extremely diverse collections of facts. Even if you can't succinctly represent a concept as a single RDF triplet you can pretty much always find some transformation of your original idea into two or more RDF triplets.

Often (in other grammars) facts that were not core considerations are attached as kind of barnacles. Late arriving concepts are not considered to be first-class citizens. Because of this classical grammars often attempt to re-invent the wheel 'better' - rolling in all the new thinking. Grammars such as say SVG, Flash, VRML, GML, Avalon 'steal' ideas from other grammars - re-implementing and repackaging them - whereas with RDF you just 'use' the snippets of the other RDF vocabularies that you like. One example of this is the 'Locative Packet' that in and of itself specifies no new vocabulary but simply denotes a convenient intersection of already existing vocabularies:

http://locative.net/workshop/index.cgi?Locative_Packets

RDF has an appealing simplicity and formality. It is quite pleasing for example that OWL (a grammar for specifying the legal attributes of any RDF subject) is itself in RDF. In other language such as say Java or any IDL - there are separate notations for specifying 'abstract' versus 'instance'. Even XML has the infamous DTD notation which is not itself XML. This lucky happenstance of RDF seems to be more than just accidental thinking - it looks like there were many predecessor ideas that ended up emerging here such as this paper on Associative Databases: http://www.lazysoft.com/docs/other_docs/AMD.pdf and these general comments on database normalization: http://en.wikipedia.org/wiki/Database_normalization

7.4.3 Coming Clean

A weaknesses of RDF is that work is pushed over to logic. Instead of having a declarative schema that fully constrained an object one tends to ignore constraints and simply use application logic to traverse the complex relationships that describe an objects state. The fact that a person may belong to an organization for example can be expressed in OWL but is more likely - practically speaking - expressed implicitly in the logic that walks persons and finds organizations they belong to. This might be as simple as an RDQL query or could be as complicated as explicit hard-coded logic in the application. Ultimately what is needed is a programmatical model of RDF where the OWL schema is itself directly exposed to the procedural logic. FABL for example moves in that direction.

7.5 Navigating via Tags, Streams and Crumpled URLs

At this point we have a way to serve content, and we have a way to store content. Now we have to consider exactly how the user is going to interact with and define what is being stored. Here is where we move into the thinking that specializes the design away from being any generic web driven database application.

Effectively we're building a CMS - it understands what a user is, what posts are, how to perform various useful queries and enforces a permissions policy such that users cannot overwrite each others space. The kinds of concepts we're needing to manage include:

Users

User posts

User tags

Perhaps some statistics as well

Users and Posts

Users, preferences and posts are fairly clear. Basically we define some RDF predicates in our vocabulary to capture basic post data. In fact we don't even have to do any work - we can just use RSS as is with `<title>` `<link>` and `<description>` being perfectly adequate.

7.5.1 Tags

Tags are a new concept here and get a little bit more discussion.

Tags are introduced as a mnemonic to help users recall their own posts later on. A user can categorize a new post under any arbitrary string that they wish such as say 'politics, satire, humor' or say 'politics, art, prague'. The user can then see posts under a specific topic or intersection of topics and this helps with overall recall.

If enough users have similar ideas about similar tagging systems then presumably even in groups you'll begin to see certain tags evolve and become representative of certain ideas. In the dating advertisements in the back of magazines for example you often see 'm4w' or 'w4m' as examples of tags that have evolved to represent certain ideas.

To keep people from running amok with tags we have to enforce some tag naming constraints. In this system all user tags are lower case, may start with a number, must not have any symbol in them except '/' and may not have spaces in them (even with quotes). Heirarchical tags are allowed although their value is low and they are treated as single atomic tags in most cases. As well 'sys:' and 'system:' are reserved.

Tags are also used internally to categorize system concepts. There is a somewhat seductive power to a tag engine. Once you have one typing system then it becomes increasingly convenient to be able to do all of your filtering against that type system. A 'system:subscription' or a 'system:friend' or a 'system:ignore' tag could be attached to a user post to indicate that that post is about another user that that user may be subscribed to or a friend of or ignoring. If we weren't using tags then we might have defined our own RDF Vocabulary to explicitly capture concepts such as 'subscription' and

or 'friend' and would have a system that was actually less flexible (as the query layer will show). At the same time, by migrating system concepts up to the level of tags we are in a sense stepping outside of RDF a bit - it means other third party consumers of our RDF feeds have to have special logic to understand exactly what class of object an object is.

Note that there isn't any particularly deep reasoning as to why we're using tags - it's just an easy, convenient, brief and memorable concept for users. At the same time there is quite a bit of formal discussion on voluntary categorization, prototype theory and the like. You can read some of the literature in cognitive psychology for more discussion of these topics - in particular Eleanor Rosch and George Lakoff. But at the same time it's probably best to think of tags as a simple colloquial concept and not to read too much into them.

7.5.2 Crumpled URLs

The URL presents a very small text space within which a number of not completely orthogonal concepts are being 'crumpled'. We are effectively trying to represent a set of slightly irrational 'human shaped' ideas within a few dozen bytes. The URL space should be:

Memorable. To have an URL scheme for the site overall that is simple enough and clear enough that it can act as a mnemonic for the user. The user should ideally be able to type in an URL with various path and parameter qualifiers and have their browser retrieve specific content at that path. The user should not be required to visit the site and navigate solely by mouse-clicks. Unique. Each unique given page of a given type of content should uniquely map to a an URL and visa versa. Some sites that don't conform to these simple rules cannot be bookmarked; the user must manually navigate back to the site and page in order to retrieve the content. Key concepts dominate. In general the most important concepts that the service provides should be URL addressable in the URL path itself. Secondary concepts can be reached by '?' style parameters. Ego dominates. Users simply enjoy having their name be visible in the URL space. Tags dominate. Tags are an important concept and should be visible in the URL. Streams

Del.icio.us uses an especially nice pattern where the url path represents a kind of 'sum of children streams'.

We're going to do something similar where the URL is broken up like this:

Effectively the url is broken into:

```
[ domain ] / [ username ] / [ tag ] [ ?styles ]
```

Each parent folder sums up all of the content of all children folders. It's an intuitive and useful metaphor. It even works with hierarchical tags.

An alternative pattern could be to do [username].[domainname]/[user tag path]. This is problematic simply for DNS management issues and because it ruins the opportunity to use the domain name space for other kinds of more appropriate overloading and precedence order. It is (arguably) more clear to humans to say `portland.craigslist.org/anselm` than to say `anselm.craigslist.org/portland` for example. So we won't do this.

Using a streams concept helps us work in RDF. There are some nice things we can do in the database layer for indexing and discovering collections of facts under a given stream or stream with a wildcard path.

There does need to be some concept of getting a stream of all tags independent of users. To accomplish this we can create a fake user called 'tag' and copy all posts to that user. Visiting `http://domain/tag/elephant` would yield all posts with the category elephant of all users.

There is also quite a need to get at information in different 'styles' such as `/person?rss=true`. We are going to avoid as much as possible having reserved root path nodes and instead use parameter arguments where appropriate (avoiding `/rss/person` and favoring `/person?rss=true`). This isn't quite REST `http://www.xfront.com/REST-Web-Services.html` in that REST encourages using 'nouns' not 'verbs' - but the REST argument in this case isn't

quite clear to me and we are using tags so we desperately want to minimize use of the urlspace for anything else.

Another final consideration regarding streams: there are system folders and other internal things that effectively end up becoming reserved users. If for example we want to have a folder for all books such as '/isbn/' we would have to make sure that user is reserved. There is some argument to put all users under '/home/' but that is a terrific waste of root namespace and that root-namespace is highly valuable and highly sought by users for their own names. So we live with the slight irrationality and just crumple the concepts we need into the url space as best maps to human needs.

Now we're done thinking about the "way" the user sees the system.

7.6 The Query Engine

Since we have a model of user interaction - with streams and tags and all that stuff - we need to figure out how we're going to drive that interaction. We have to make a bridge between the user and the database engine.

We're going to want a query layer that can be directly queried by the client application. This is not RDQL (although it could use RDQL or another query language) but is tailored towards our specific application. It also imposes a security wall so that users cannot pollute other users content.

Basically we just want a laundry list of the kinds of capabilities we need and then we can pluck out commonalities and implement something simple that translates these high level requests into actual indexed query lookups of our RDF database.

Typical queries are probably:

- Return a list of all posts by a particular user
- Return a list of all posts by a particular user under a particular topic or 'tag'

- Return a description of a particular user
- Return a count of all posts or posts in an area
- Return posts within a certain date range
- Return posts over a certain size
- Return a list of all topics
- Return a list of all users
- Return a list of all posts
- Return all posts with certain content
- Return all posts on a particular 'kind' of topic - such as a book, music, mime-type or other disambiguatable thing.
- Return a thumbnail of an url or a file
- Return administrative gateway views of all users and all posts
- Login a user
- Logout a user
- Accept a new users description
- Accept a new post by a user
- Accept a subscription by a user to another user (ie accept other kinds of things not just posts)
- Accept a file
- Allow a sysadmin to delete or modify users and or posts
- Show statistics
- Throttle returned results; return todays or this weeks or 10 results only.
- Return not individual posts but only 'unique' posts about a given URL. An url posted twice should show up once only.

The discussion of the actual implementation of the query engine is probably too much detail for here. I'll let you look at the code to see the specifics of how these queries were implemented based on this set of use cases.

7.7 Javascript User Interface

A few web services now are starting to use Javascript. Googles gmail and Amazon's A9 service are good examples of this.

Historically most web services manufactured the user interface on the server side using Mason, ASP, JSP or other such grammars. These solutions are actually quite difficult for designers to work with and they create a security liability in that the pages can express commands that permeate the security wall between the client and server state.

The Javascript pattern has a number of appeals however:

Javascript runs on the client side, is shipped as static content from the server so less computation for server.

It can be vastly more responsive than any server driven application. Authoring tools can deal with Javascript much better than with ASP, JSP and the like.

It is simply a nice separation between server responsibilities and client responsibilities.

Using Javascript creates a practice of building XML gateways between server and client; this formalizes the server API.

A well separated server with clearly defined roles can talk to any client - a native application or other visualization tools.

Since there is a total separation between the server and the client it becomes possible to allow clients to create their own html pages and store them on our server. That means users could entirely customize the appearance of their

own pages and we wouldn't have to worry about security issues. Many web services fight over look and feel - this makes that debate totally obsolete and a little bit silly.

7.7.1 Drawbacks to using Javascript

:

Browser portability problems HTML and layout inconsistencies across browsers (that can be more easily treated on the server side).

7.7.2 Uses of our Javascript Client

- We're going to simply have a single html document on the server that we're
- going to send to the client over and over.
- The single document will change its appearance based on the current URL that the client is on.
- We will 'round-trip' form parameters back to the client document for it to do work.
- Work will always be done with the server as explicit XML requests or posts.
- In a sense we are shipping an 'application' to the client - and even though HTML is too stupid to know it - that application persists between pages and doesn't have to introduce any new pages.

To build out our Javascript client we are going to have to write a number of small Javascript functions:

- Reading XML in Javascript from the server
- Writing XML in Javascript back to the server
- Drawing XML to the screen
- Input forms
- Some layout utilities
- Determining current user page and reacting appropriately
- For now you'll have to refer to the actual code to see the examples, but I hope to complete this section soon.

8 Conclusion

Here is the tarball.

These services are fun to build from a kind of mad scientist perspective. The tools we have today to architect these large scale social systems are so powerful and so easy to use that it can be as little as a few days work to unleash an entirely new social application on an unsuspecting public.

If you're going to use this starting point professionally then there are other considerations not covered here; such as finding ways to aggregate and or federate content so that you can take advantage of laws of utility and avoid walled garden effects. As well if you are deploying a commercial service based on this code you may want to support some wiki like concepts so that users can entirely customize their own experience.

8.1 What could you do with this?

8.1.1 Make your own Craigs List

<http://frot.org/geo/craigslist.html>

8.1.2 Personal knowledge tracking system

Track your habits or even your finances.

8.1.3 Big Bucket

Effectively this becomes a big bucket that you can pour stuff into. You could attach an aggregator to this and do say brute force geo-location of news-articles and project them onto a globe; and then do peer based review of those articles or additional decoration of facts from people who are on the ground in that area...

Really the sky is the limit.

8.2 The Next 10 Years

The thing to do is to think about where all of these services are going over the next 10 years. Clearly many of them are going to go away - and clearly others will have to find ways to federate and share their knowledge.

Hope you had fun.

Please send me comments if you liked this essay.