# FeedFeeds
## A Web Service for Feed Metadata

Igor Pesenson
Advisor: Erik Wilde

UC Berkeley School of Information
May 8th, 2008

**ABSTRACT** ..... **2**
**BACKGROUND** ..... **2**
**EXPLANATION OF SERVICE** ..... **2**
**SERVICE DESIGN** ..... **4**
DATA MODEL ..... 4
APPLICATION PROGRAMMING INTERFACE (API) ..... 5
QUERY FORMAT ..... 6
**TECHNICAL IMPLEMENTATION** ..... **7**
USER INTERFACE ..... 8
**FUTURE WORK** ..... **9**
SERVICE API ..... 9
USER INTERFACE ..... 9
SERVICE SPEED ..... 10
RESEARCH QUESTIONS ..... 10
**BIBLIOGRAPHY** ..... **11**
**APPENDIX A.** ..... **12**
**APPENDIX B.** ..... **14**

## Abstract

Feeds have become an important information channel on the Web, but the management of feed metadata has so far received little attention. It is difficult for feed consumers to locate, evaluate, and share feeds as well as manage feed subscriptions across multiple platforms. It is also difficult for feed publishers to categorize and present more than a few feeds in an effective manner.

We are proposing a web-based service to address some of these problems. The service will manage feed metadata and will allow any number of interfaces to be built on it. For example, there will be an interface for feed consumers that will enable them to manage their feed subscriptions in one location. There would also be an interface for feed producers to categorize and publish their feeds.

This report presents the design of the service and discusses the lessons learned during the implementation of the prototype.


## Background[1]

Syndication of web information using feeds is now common. There are a number of XML-based document formats used for feeds – over ten different RSS[2] standards as well as the Atom [2] standard. The Atom Publishing Protocol (AtomPub) [3] has been developed to specify how Atom XML documents are to be exchanged. It appears that a technical infrastructure for syndication using feeds is now in place.

Awareness of feeds, however, is very low – a report by Yahoo on RSS [4] use in 2005 found that 12% of users are aware of RSS and only 4% have knowingly used RSS. There are probably a number of reasons for this but one is lack of support for user adoption of the technology. Apart from feed readers there is little support for users locating, evaluating, sharing and generally managing feeds. Support for publishing and presenting feeds to users is also almost non-existent. To create a large number of feeds and let consumers select ones of interest, for example, would currently be accomplished by presenting the feeds as a long list on the publisher's website.

## Explanation of Service

The service will be a backend, accessible via an API, and will provide feed metadata storage, manipulation and user authentication. User interfaces are to be constructed on

[1] For a more complete introduction to feeds and feed use see Wilde, Pesenson [1]
[2] RSS stands for Really Simple Syndication or Rich Site Summary.

top of it and it is easiest to explain the functionality of the FeedFeeds service via one of these interfaces.

The most intuitive user interface to build is one for people who are consuming a lot of web feeds. This is analogous to how the del.icio.us service [5] works for websites. People use del.icio.us to maintain a central repository of their web bookmarks. They can organize the bookmarks by tagging them, storing descriptions of them, and combining them into bundles. Users can discover new bookmarks by viewing what other users are storing. A user interface to the FeedFeed service would be very similar except instead of storing and managing web bookmarks it would deal with web feeds. For sake of convenience we will call this user interface to the FeedFeeds service www.cafs.com (Consumers' Awesome Feed Service).

Consider the following use example. Joe the user goes to cafs.com and creates a log in. When he does so the cafs.com service contacts the FeedFeed service and asks it to create a new identity for Joe. Joe then proceeds to import feeds into his cafs.com account by uploading OPML files[3] as well as individually submitting feeds. Every time Joe adds a feed the cafs.com service sends requests to the FeedFeeds service to add the feed to his account. As the FeedFeeds service analyzes the incoming feeds it pulls out and stores some publisher-provided metadata from the feeds, especially any categories that might have been provided publishers. At the same time it stores other metadata useful for Joe such as the date he subscribed to each feed, any categories he chose to add to the feed, and any other descriptive notes he added. The FeedFeeds service then provides all of this metadata to cafs.com which can presented it to Joe. Joe can use the categories, which we will also call tags, to select subsets from all of his feeds. Clicking on category "mobile," for example, Joe will see all feeds he had marked with that tag. Joe can then click the "export" button and use the resulting OPML file to subscribe his cell phone to those feeds.

The reason the user interface is decoupled from the FeedFeeds service is because this allows for a number of different interfaces to be built on the same base. Figure 1[4] shows what information flow might look like if two different interfaces were built on the FeedFeed service – one for feed consumers and one for feed publishers.

[3] Outline Processor Markup Language (OPML) is a common method of exporting and importing lists of feeds into and out of feed readers, see [6]
[4] Taken from [1]

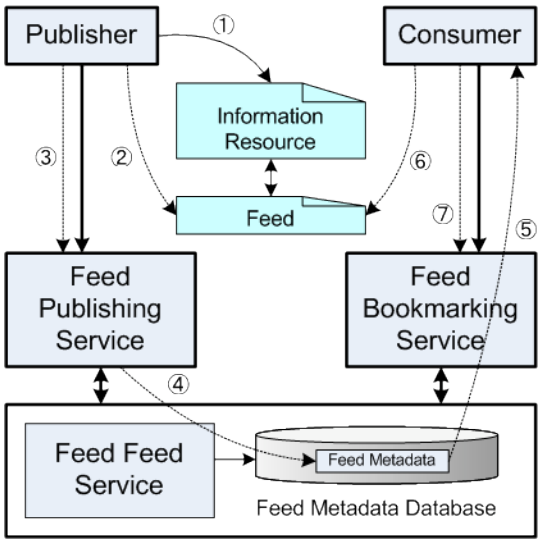


**Figure 1. How FeedFeed Service could work with a user interface for publishers and consumers.**

Consider the following information flow. A large publisher, such as a newspaper, is creating lots of content on the web (information resources). At the same time its content management system is producing dozens of feeds that summarize this content. The content management system is set up to submit the feeds to a feed publishing service – we will call it www.pafs.com (Publishers' Awesome Feed Service). The pafs.com service passes the feeds to the FeedFeeds service which validates their classification against a scheme the publisher had provided. The FeedFeeds service stores the feeds and provides the metadata for management at pafs.com and consumers to discover at cafs.com. Joe is browsing for feeds tagged "news" and "africa" using cafs.com and comes across several feeds produced by the newspaper publisher. Joe then adds these feeds to his account.

# Service Design

## *Data Model*

The data model for FeedFeeds is demonstrated in Figure 2. There are four major entities – Users, Feeds, Tags, and Resources. Resources are the web resources where the feeds originate. So the resource could be "BBC – World News website" and one of the feeds could be "Daily Africa news." Every feed subscription by a given user is distinct from subscription to the same feed by a different user.

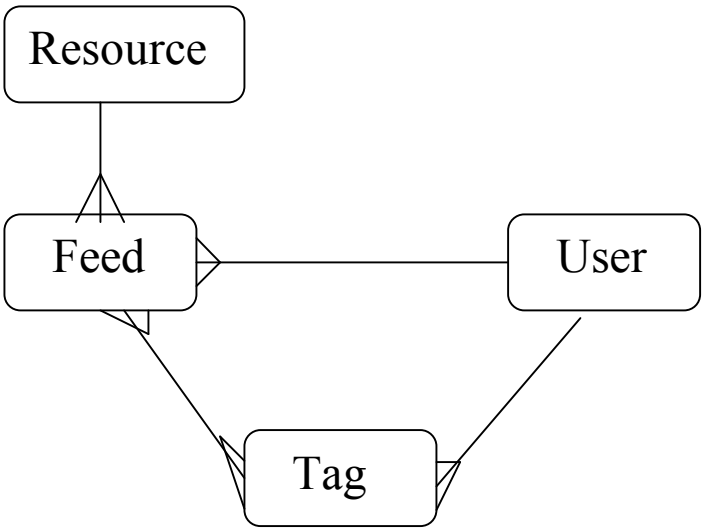


**Figure 2. FeedFeeds data model**

## *Application Programming Interface (API)*

Since the major resources managed by the service are feeds the service will use the Atom feed format to store and exchange information. The service will use the Atom Publishing Protocol so that its application programming interface (API) is REST-ful and standardized. The API is detailed in Figure 3.

**Figure 3. AtomPub based API for FeedFeed**

| Action | Http Method | URI | HTTP Content | Return Value |
|---|---|---|---|---|
| Get service document for feedfeed | GET | http://feedfeed.org | none | Service document for feedfeed listing all collections; 200 |
| Get user's subscriptions | GET | http://feedfeed.org/userid | none | atom:feed document where each entry is a feed. It includes the complete list of categories; 200 |
| Input a feed | POST | http://feedfeed.org/userid | Feed document as created by the publisher | feedID or URI identifying the new resource in location attribute of header; 201 |
| Get user's tags | GET | http://feedfeed.org/userid/categories | none | Categories document for the user's collection; 200 |

| | | | | |
|---|---|---|---|---|
| Modify tags or other user metadata for the feed | PUT | http://feedfeed.org/userid/feedid | atom:entry document with with no content element. Contains metadata elements | 200 |
| Get popular feeds | GET | http://feedfeed.org/popular | none | Feed document where each entry is a feed. It includes the complete list of categories; 200 |
| Export feeds in OPML format | GET | http://feedfeed.org/userid | none; accept header set to 'text/x-opml' | OPML XML document |
| Import OPML document of feeds | POST | http://feedfeed.org/userid | none; content set to 'text/x-opml' | Feed document specifying URI for each input feed; 201 |
| Get feeds filtered by tag | GET | http://feedfeed.org/userid?category='tag1' | none | Feed document with feeds as entries; 200 |
| Get a user's feed | GET | http://feedfeed.org/userid/feedid | none | Feed document with a single entry representing the feed; 200 |
| Delete a user's feed | DELETE | http://feedfeed.org/userid/feedid | none | none; 200 |

## *Query Format*

FeedFeed will need to support a wide range of queries. A client service, for example, might want to get a list of the most recent feeds or feeds with specific characteristics. AtomPub is silent on queries and needs to be extended to handle them. One common way to do is to use GData [7], Google's extension to AtomPub API used for its own services. Instead we chose FIQL, the Feed Item Query Language [8], for this purpose. FIQL is currently a draft submitted for comments in the IETF approval process. Although choosing a standard before it has been approve is risky we decided FIQL offered greater flexibility than GData. Example queries using FIQL are shown in Figure 4.

**Figure 4. Example FIQL queries**

| Action | Http Method | URI | HTTP Content | Return Value |
|---|---|---|---|---|
| Get user's feeds with title starting with 'foo' | GET | http://feedfeed.org/userid?title==foo* | none | Feed document with feeds as entries; 200 |
| Get all feeds in itunes news category | GET | http://feedfeed.org?itunes:category=news | none | Feed document with feeds as entries; 200 |
| Get all feeds since march 2008 | GET | http://feedfeed.org?updated=gt=2008-01-01T00:00:00Z | none | Feed document with feeds as entries; 200 |
| Get user's feed with "ag tech" in summary | GET | http://feedfeed.org/userid?summary==*ag%20tech* | none | Feed document with feeds as entries; 200 |
| Get all feeds from the last day | GET | http://feedfeed.org?updated=lt=-P1D | none | Feed document with feeds as entries; 200 |

# Technical Implementation

The choice for the database was straightforward – we settled on MySQL because it is freely available and scales well. We next spent some time looking for an implementation of AtomPub. Appendix D shows the results of the evaluation efforts. We decided to use Amplee – it seemed the most complete implementation and had support for a number of storage options. This also dictated the choices for language of implementation and the web framework – Amplee is Python based and was designed to run on top of the CherryPy, a Python based object-oriented HTTP framework [9]. Because Python became the language of implementation we decided to use Dejavu [10] as the interface to the MySQL database.

After a lot of following time and effort we determined that Amplee was not working well for our purpose. Although the exact method of storage was abstracted away in Amplee the data model was not. It did not appear possible, or at least possible in our time frame, to rewrite Amplee to confirm to our data model. After we jettisoned Amplee we did not have time to implement the full AtomPub API for the service. Instead we put together a makeshift API for the service prototype and it is detailed in Figure 5.

**Figure 5. API implemented in prototype.**

| Action | Http Method | URI | HTTP Content | Return Value |
|---|---|---|---|---|
| Input a feed | POST | http://feedfeed.org/userid | feed=[feed/url/here] | none |
| Tag a feed | POST | http://feedfeed.org/userid/feedid | tag=[tagtext] | none |
| Get user's subscriptions | GET | http://feedfeed.org/userid | none | A feed XML with feeds as entries |
| Get user's tags | GET | http://feedfeed.org/userid | info=servdoc | AtomPub service document with categories for the collection |
| Get feeds filtered by tag | GET | http://feedfeed.org/userid | | A feed XML with feeds as entries |
| Get all tags | GET | http://feedfeed.org | info=servdoc | AtomPub service document with categories for the collection |

## *User Interface*

Since the FeedFeeds service is designed as a back end system it requires a user interfaces to demonstrate its functionality. A potential user interface mock-up is shown in Figure 6. Proof of concept PHP code based on the FeedFeeds API has been developed by Jim Miller. The interface allows the user to view his subscriptions alongside the tag cloud. The user can import new feeds into his account and tag them.


**Figure 6. Example feed consumer service screenshot.**

# Future Work

## *Service API*

Although the developed prototype demonstrates the functionality of the service well it does not closely adhere to the original design specifications. The API of the service will have to be redesigned to implement the AtomPub standard. It would be worthwhile to analyze Amplee again and evaluate how much effort it would take to redesign the underlying data model. If such redesign is deemed unfeasible then AtomPub should be implemented using the current setup. This setup, the combination of Python as the language, CherryPy as the web framework, and Dejavu as the database interface works well – implementing AtomPub on it should be relatively easy if somewhat tedious.

## *User interface*

The consumer user interface should be built out to take full advantage of the FeedFeed service functionality. This will ensure the FeedFeed service is useful and usable to at least one community. To make sure that the service is abstract enough for different purposes a feed publisher interface will need to be created. Development of these interfaces will also help point out deficiencies to address in the FeedFeeds service.

## *Service speed*

The service is currently running on CherryPy routed through Apache. The CherryPy website claims that this set up can support a modest to large service. Some load testing should be done to test this. At this time the response time is slower than expected. Some of this could probably be addressed by careful review of database access code.

## *Research Questions*

What kind of publishers use feeds and how often do they use them? What fraction of the feeds produced get read at all? For a feed that's read, what proportion of the contents are read? How do most people read feed content? What is the ranking between the different motivations behind reading feeds? Are feeds used differently on intranets as opposed to the internet?

# Bibliography


1. Erik Wilde and Igor Pesenson, "Feed Feeds: Managing Feeds Using Feeds" (May 5, 2008). School of Information. Paper 2008-025. http://repositories.cdlib.org/ischool/2008-025
2. M. Nottingham and R. Sayre, “The Atom Syndication Format”. Internet RFC 4287, December 2005. http://www.ietf.org/rfc/rfc4287
3. Joe Gregorio and Bill de Hora. “The Atom Publishing Protocol”. Internet RFC 5023, October 2007. http://www.ietf.org/rfc/rfc5023.txt
4. Joshua Grossnickle, Todd Board, Brian Pickens, Mike Bellmont. “RSS—Crossing into the Mainstream”. Yahoo! White Paper, October 2005. http://publisher.yahoo.com/rss/RSS_whitePaper1004.pdf
5. Del.icio.us, http://www.del.icio.us
6. Dave Winer, Outline Processor Markup Language (OPML), November 2000, http://www.opml.org/spec
7. GData, http://code.google.com/apis/gdata/overview.html
8. N. Nottingham, “FIQL: The Feed Item Query Language”. Internet draft, December 2007. http://tools.ietf.org/html/draft-nottingham-atompub-fiql-00
9. CherryPy, http://www.cherrypy.org/
10. Dejavu, http://www.aminus.net/dejavu

# Appendix A.

XML Documents

Below is an example of a feedfeed document. The document is a valid Atom feed document with each entry corresponding to a feed in the collection. The content element of the entry contains data provided by the feed publisher – this could be the title, summary, links to the feed and similar. The metadata provided by the user and stored by the FeedFeed service is provided inside the entry but outside the content element.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>FeedFeed subscriptions of igor</title>
    <id>ff:feedfeed,igor,2008:05:05</id>
    <updated>2008-05-05T23:20:50.52Z</updated>
    <entry>
        <link
href="http://groups.ischool.berkeley.edu/feedfeed/igor/41" rel="self"/>
        <category term="toread"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <category term="boring"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <category term="crazy"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <content>
            <feed>
                <title>This American Life Updates</title>
                <id>NULL</id>
                <updated>Fri, 02 May 2008 16:00:00 -0600</updated>
                <link
href="http://feeds.thisamericanlife.org/talupdates"
type="application/rss+xml"
                    rel="self"/>
                <link href="http://www.thisamericanlife.org"
type="text/html" rel="alternate"/>
                <summary>Presented by Chicago Public Radio, "This
American Life" is an award-winning radio program as well as a
television program on the Showtime network. Hosted by Ira Glass, we
feature first-person stories and short fiction pieces that are
touching, funny, and </summary>
            </feed>
        </content>
    </entry>
    <entry>
        <link
href="http://groups.ischool.berkeley.edu/feedfeed/igor/42" rel="self"/>
        <category term="fun"
      scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <category term="bioproject"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <category term="toread"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor"/>
        <content>
            <feed>
                <title>Atom</title>
```

```
                <id>tag:www.atompub.org,2005:/blog//1</id>
                <updated>2005-07-17T01:10:32Z</updated>
                <link href="http://atompub.org/blog/index.atom"
type="application/atom+xml"
                    rel="self"/>
                <link href="http://atompub.org" type="text/html"
rel="alternate"/>
                <summary>Atom news in Atom.</summary>
            </feed>
        </content>
    </entry>
</feed>
```

Below is an example of a service document for a user's collection. Although we did not implement AtomPub for the prototype we still borrowed some of the structures. This document is currently only used to show list of categories or tags used in a collection.

```
<app:service xmlns:app="http://www.w3.org/2007/app"
xmlns:atom="http://www.w3.org/2005/Atom"
    xml:lang="en" xmlns:ff="http://www.feedfeed.org/2008/ff">
    <app:workspace>
        <atom:title>User Collections</atom:title>
        <app:collection
href="http://groups.ischool.berkeley.edu/feedfeed/igor">
            <atom:link
href="http://groups.ischool.berkeley.edu/feedfeed/igor" type="self"
                rel="alternate"/>
            <app:categories fixed="no">
                <atom:category term="bioproject"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="2"/>
                <atom:category term="boring"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="4"/>
                <atom:category term="cell"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="4"/>
                <atom:category term="cool"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="1"/>
                <atom:category term="crazy"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="3"/>
                <atom:category term="forjim"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="2"/>
                <atom:category term="fun"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="11"/>
                <atom:category term="nyc"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="3"/>
                <atom:category term="toread"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="4"/>
                <atom:category term="work"
scheme="http://groups.ischool.berkeley.edu/feedfeed/igor" num="6"/>
            </app:categories>
        </app:collection>
    </app:workspace>
</app:service>
```

# Appendix B.

Selection of AtomPub implementation results.

| | **Amplee** | **PHP AtomPub** | **Abdera** | **eXist** | **ruby-atompub @ RubyForge** |
|---|---|---|---|---|---|
| Completeness of the implementation ? | Looks pretty complete...from the tutorial<br>Does: POST, PUT, GET, DELETE, Create new indexers, paging | Appears complete but does not explicitly state so. All code dates 12/13/2007 | Appears complete, code dated 10/2007 | Complete. Unable to verify - could not find atompub interface on 1.0.2 to run APE on. | Developmental. Last updated on May 5, 2006 |
| Support of the final RFC 5023 or some draft version? | | Points to rfc5023 so probably yes | Supports AtomPub final draft | Draft (?) | Supports older draft |
| Support for additional standards? | Community working to integrate with XMPP. May be possible to extend to support tombstones / feature discovery. No support for bidirectionality | Supports Feed Paging and Archiving, does not mention others. | Supports License Extension and feed paging/archiving, also supports a GeoRSS extension | No | |
| How is the connection to the back-end data store? | Flexible. Supports S3, ZODB, etc. | PDO version available but does not implement categories. Flat files by default | Not sure, but appears to be file based. Could possibly be extended to include database support, but that would require an unknown amount of development on our end | Unstable database | |
| Is it possible to pass through query parameters? | Supports OpenSearch and its Geo extension. Supports query by collection. | Probably not | Unsure, inclined to say no (the documentation was not very helpful on this point) | No - XQuery POST to /atom/query/ <collection> | |
| Is there an abstract feed API, or do users need to construct and parse XML? | Yes | Don't know, probably the latter | Feed API seems to follow the Atom Publishing Protocol actions (postEntry, deleteEntry, getEntry, putEntry, getFeed, getCategory) | Construct/parse XML | |
| HTTP authentication support? if so, which methods (basic, digest access, cookies) | Found none. But there is a crawler which does some of this: http://www.defuze.org/oss/amplee/api-0.6.0/amplee.contrib.crawler-pysrc.html | It should | HTTP Auth support for client requests, but does support Google Login and WSSE Login | Jetty - basic; Tomcat - ? | |
| HTTPS? TLS and/or SSL? | Found none. | It should | SSL support for client requests | Jetty - SSL/TLS | |

| | | | | | |
|---|---|---|---|---|---|
| How does the GeoRSS extension index lat, lngs? Is it easily extensible? | | | | | |
| General comments | | | | Overall I get the sense that eXist really focuses on its own XML:DB API implementation. | http://rubyforge.org/projects/atompp/ |