Feed Subscription Management

Erik Wilde and Yiming Liu
School of Information, UC Berkeley

UC Berkeley School of Information Report 2011-042
May 2011

Available at http://escholarship.org/uc/item/2r6031th

Abstract

An increasing number of data sources and services are made available on the Web, and in many cases these information sources are or easily could be made available as feeds. However, the more data sources and services are exposed through feed-based services, the more it becomes necessary to manage and be able to share those services, so that users and uses of those services can build on the foundation of an open and decentralized architecture. In this paper we present the *Feed Subscription Management (FSM)* architecture, which is a model for managing feed subscriptions and supports structured feed subscriptions. Based on FSM, it is easy to build services that manage feed-based services so that those feed-based services can easily create, change and delete feed subscriptions, and that it is easily possible to share feed subscriptions across users and/or devices. Our main reason for focusing on feeds is that we see feeds as a good foundation for an ecosystem of RESTful services, and thus our architectural approach revolves around the idea of modeling services as interactions with feeds.

Contents

1 Introduction

Nowadays, a growing number of web applications offer APIs and web services, as means to share data and interoperate with other services and devices. Moreover, many services now expose RESTful APIs, following a number of general design patterns. One popular set of implementation technologies for RESTful API design are feeds, with the most robust standard for them being *Atom* [1]. Atom has been extended by additional specifications for the read side of the service (for example introducing capabilities to expose *paged feeds* [2]), and it also has been extended to cover services beyond read-only access with the *Atom Publishing Protocol (AtomPub)* [3].

iTunes podcasts, for example, are enclosed audio files delivered as a service via feeds, albeit wrapped within the Apple iTunes client. Large web content management systems, like Wordpress or Blogger, offer full-featured read-write Web services for publishing, consuming, and managing content. In application areas such as the *Web of Things* [4], a large number of sensors, appliances, and other everyday objects may emit data streams, made accessible on the Web as real-time data feeds. Generally speaking, any federated and decentralized ecosystem of producers and consumers of information services can be modeled as a feed-based architecture, and such an architecture rooted in plain Web technologies [5] can be especially advantageous if the participants may use very different in-house implementation platforms, and still need a shared service platform for services such as reporting and report aggregation [6, 7].

In this paper, we present and demonstrate an architecture for how consumers of feed-based services can organize the services that they are using, can share them or subsets of them, and can build tools which implement a decentralized system for publishing, consuming, and managing feed subscriptions.

This *Feed Subscription Management (FSM)* architecture is based on the observation that an increasing number of services are exposed via feed-based interfaces.[1] In such an ecosystem, it becomes apparent that management of all these services becomes essential. FSM allows those services to be aggregated into *subscriptions feeds* (which can be nested using *bundles*), which can then be made available and access controlled (if required) using standard Web technologies. Since FSM subscriptions are managed in feeds themselves, managing subscriptions is based on Atom and AtomPub and thus can be done using the same technologies and toolsets that are used for accessing the managed services themselves. Our goal is to provide a decentralized architecture where the requirements for supported technologies are as minimal as possible, so that it is easy to use from a wide variety of programming and runtime environments.

Whether the feeds are consumed by browsers or by applications running on mobile platforms is irrelevant for our architecture, but access to these feeds can be easily shared between clients on those different platforms, so that users can for example easily use the same set of services, regardless of whether they use a Web application, a smartphone application, or an embedded application in a car navigation system.

2 Managing Feed Access

Feeds started as a way of representing machine-readable updates on Web sites, often for news sites or blogs. The first feed formats were various RSS versions, and after RSS ran into problems because of disagreements in the developer community, the XML-based Atom format was developed as an alternative resolving some of RSS's technical problems as well as being developed in a context that has a well-defined process and path for future developments. For the news and blogs feeds that marked the early days of feeds on the Web, *feed readers* emerged, which give users the ability to subscribe to feeds, and then allow users to see updates to all those feeds in one unified view.

Technically speaking, feed readers exist in various configurations. Some of them are standalone programs working with any kind of feed, so that users can subscribe to a wide variety of data sources. Others are

[1]For example, Google's *GData* API builds on Atom feeds and AtomPub and adds additional features to this foundation.

similarly generic in the feed sources/types they accept, but they work Web-based (*Google Reader* is probably the most popular example for this kind of feed reader). Others only work with special types of feeds such as Apple's *iTunes* which for its *podcast* support also manages feed subscriptions (as part of an iTunes user's preferences), but only works with feeds that use the specific extensions defined for the podcast format. Another variety are many of the "single-serving" iOS applications for "company news," which in many cases simply consume and render a set of feeds published by that company, because this is the most cost-effective way to implement such a publishing scenario. This approach restricts users to the small set of feeds pre-configured in the application.

Some of these single-serving applications consume feeds and implement their own UI and specific functionality, whereas a substantial share of them are essentially nothing more than chrome-less browsers, simply displaying the HTML contents of a predefined set of HTML-carrying feeds. In this latter case, the application often is little more than a thin pre-configured layer of feed access layered over the OS's native support for Web content (e.g., WebKit in the case of iOS), packaged and deployed as an application mostly for non-technical issues, such as visibility in an "app store" and the ability to monetize the application. In this case of "pseudo-apps" (i.e., applications being essentially feature-reduced browsers being deployed as applications for non-functional reasons), it becomes apparent that managing feed subscriptions (in this case subscriptions may be hard-coded into the application, or at least they can not be freely controlled by the application's user) becomes an essential part of managing access to Web content, or more generally speaking, Web services.

Exchange of subscription information between feed readers so far cannot be based on any existing standard for how to represent this data. The *Outline Processor Markup Language (OPML)* is the closest thing to a convention that has emerged, but it is a language that is not very well-suited for the task (since it was originally intended for a different application scenario) and has no associated protocol for managing subscriptions in a Web-based way (i.e., it is supported as export/import format in some feed readers, but does not support network-based synchronization of subscription information). In earlier work about *feed feeds* [8], we explored the possibility of serializing feed subscriptions as feeds themselves, and the FSM architecture presented here builds on this earlier work and extends it in several dimensions:

- *Subscription Feeds:* The fundamental goal is to use feeds for managing feed subscription information. Section 3 explains the details, and in summary our architecture represents feed subscriptions in a subscription feed, and allows AtomPub-based interactions for managing this information (adding/changing/removing subscriptions in subscription feeds and adding/removing subscription feeds).
- *Bundling:* In order to provide support for structuring feed subscriptions, FSM supports *bundles* which is a way of referring to a subscription feed from within a subscription feed. Bundles always are used "by reference" (i.e., referred to by URI) and never "by value" (i.e., included literally where they are used), because feeds are a flat data structure and thus do not support hierarchical structures.
- *Decentralization:* Since bundles are always used "by reference," FSM naturally supports decentralization where one subscription feed can reference a bundle (another subscription feed) anywhere on the Web, as long as it is made available at an accessible URI. This structure can be nested, allowing arbitrarily complex subscription structures.
- *Query Support:* Many feed-based services support queries of some kind, sometimes just based on categories or tags, sometimes based on more sophisticated filter or query capabilities. Section 5 describes the support for query capabilities built into FSM, which makes sure that both query information and the resulting URI-based encoding for a feed query can be represented in feed subscription information, making it easier for clients to "understand" the feeds listed in subscription feeds.
- *View Support:* In addition to "data feeds," FSM also supports spatial feeds as explained in more detail in Section 6. In this case, subscriptions not only refer to feed-based services, they also contain *views*

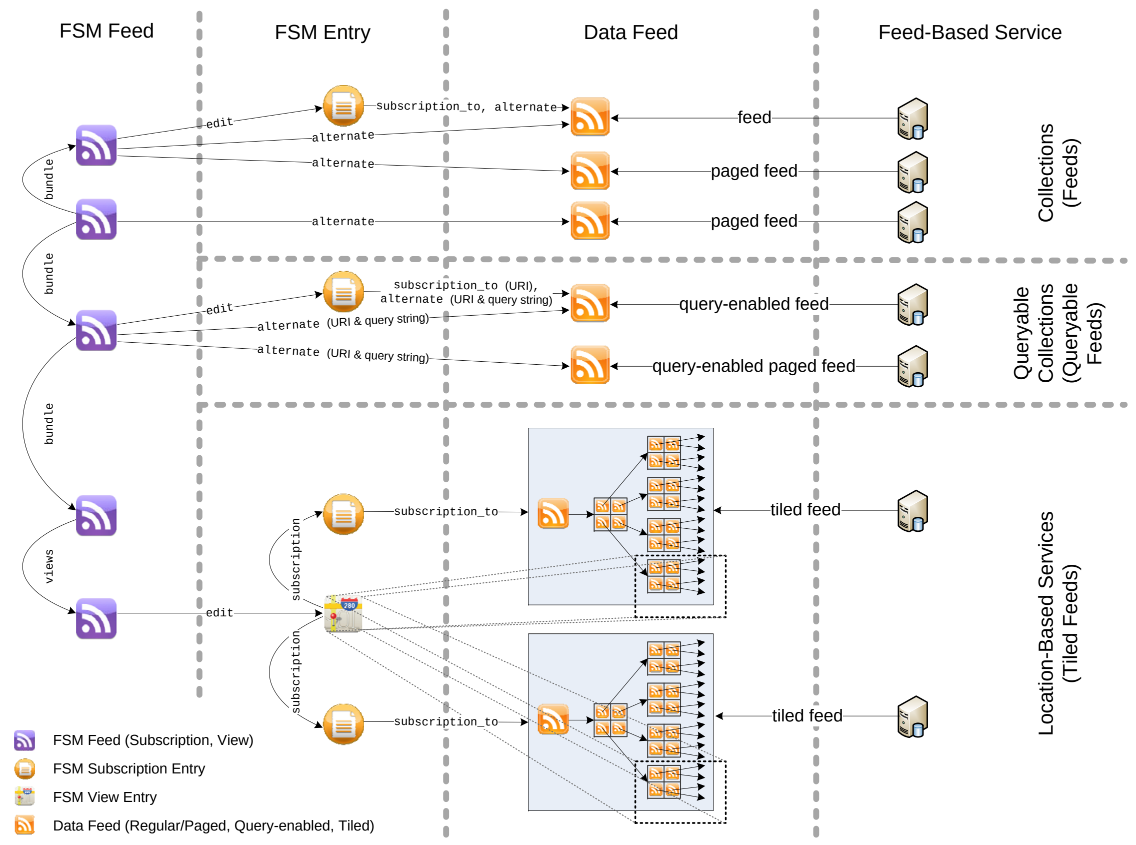


Figure 1: FSM Architecture

which are encoding a spatial region for which subscribed services should provide information, as well as a number of subscriptions which specify which services should be used to populate that view.

In this paper, we focus on the entire set of features listed above, and we describe a use case and an implementation (Section 7.2) that use all these features. However, it is important to notice that the FSM architecture has been designed in a way so that it easily usable even in the simplest scenarios (no query or spatial features supported by the subscribed feeds, i.e. only using feeds in the way how they are used on the Web today). As such, we envision that FSM can be used as a foundation in any scenario that uses feed-based data sources and that is in need of representing and/or managing feed subscription information. Section 3 describes the FSM architecture and the design decisions that have been made when creating it; Section 4 describes how to convert existing feed subscription data to and from FSM-based implementations; and Sections 5 and 6 then discuss the specifics of how FSM supports the use cases of query-enabled feeds and of spatial feeds.

3 FSM Architecture

The main goal of the *Feed Subscription Management (FSM)* architecture is to provide a foundation on which providers and/or consumers can build to represent and exchange information about feed-based services.

From the provider perspective, a typical scenario might be that of an office building where each room has an associated feed that publishes the room's current temperature; the manager of the building could create a representation of all those feeds in an FSM *subscription feed*. From the consumer perspective, in the same scenario a tenant of the office building might choose to create a subset of the original dataset to only represent those rooms which are occupied by him, and might use this dataset as a starting point for implementing value-added services about the current office environment, combining the room temperature feed with a bundle that contains the status of all printers located in the office rooms.

Feeds in FSM are always represented as Atom, because FSM is based on both Atom and AtomPub and requires Atom's extensibility for its design. This does not mean, however, that the feeds managed with FSM have to be Atom; those can use any feed format such as the various RSS formats, and as long as clients understand these RSS versions when accessing the actual feeds, FSM can be easily used to manage subscriptions to RSS feeds.

It is important to notice that FSM does not attempt to create or represent an aggregate data feed that is composed out of all the individual feeds which are listed in a subscription feed. Such an aggregation might be an interesting service to provide, in particular when considering value-added services for this aggregate feed such as filter/query capabilities or push services, but the FSM architecture is strictly limited to organizing information about feed subscriptions. As such, the main components of the FSM architecture are two types of resources (both of them shown in Figure 1, and for the simple case of data feeds shown in the upper part of the figure labeled "Collections (Feeds)"):

- *Subscription Feed:* A subscription feed represents a collection of feeds which are exposed by the publisher of the subscription feed for a specific purpose, either because they are subscribed to by some feed user, or because they are provided as a possible set of feeds to subscribe to for potential feed users.
- *Subscription Entry:* Each managed feed in the subscription feed is represented by a subscription entry which carries all of the subscribed feed's metadata as exposed in the feed itself (and none of the entries); subscription feed managers are free to change feed metadata such as the feed's title or summary information, though. Subscription entries must have URIs which are linked by `edit` URIs. The entries link to the feed they are representing by using `subscription_to` and `alternate` links.

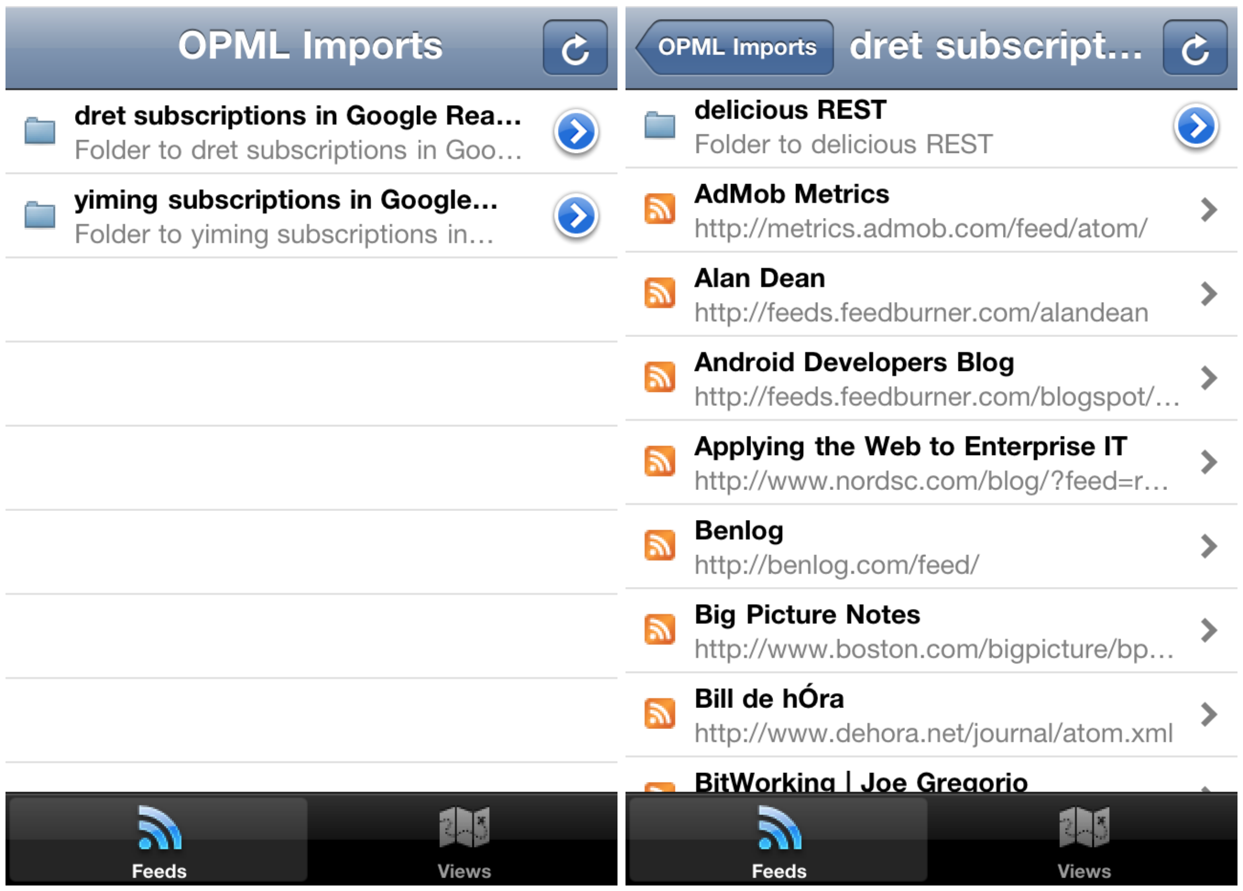
FSM covers both read-only scenarios as well as scenarios in which subscription feeds can be managed using AtomPub. Because AtomPub has no support for creating or removing collections, FSM augments AtomPub with link relations for linking from FSM feeds to the AtomPub service document, and for linking from the AtomPub workspace to `edit` URIs where clients can add new collections to a workspace (which will be represented as FSM feeds). Those links are optional (because they are only required if the FSM provider is supporting AtomPub) and are not shown in Figure 1.

Subscription feeds and subscription entries are the main components of FSM and can be used with any feed-based service. Whether the managed feeds provide additional capabilities such as paging is of no concern to FSM, as long as the feeds are published at URIs. The topmost row of Figure 1 shows this basic structure of FSM feeds and entries. It also shows an additional entry type (in the leftmost column) that can be used in a subscription feed, which is a *bundle entry*. A bundle entry represents a pointer to a subscription feed, and can be regarded as the equivalent of a "folder" in feed readers that do support this feature for managing feed subscriptions. A bundle in a subscription feed always points to a subscription feed, and the bundle entry's title is the "folder name" for the linked subscription feed. Since bundles in subscription feeds simply link to subscription feeds, this structure allows the decentralized storage and management of subscription feeds.

Since subscriptions and bundles are two types of entries that can occur in FSM subscription feeds and that need to be distinguishable for FSM clients, FSM uses an extension element that signifies the `subtype` of an entry. This design has been chosen because on the media type level, both subscribed feeds and subscription feeds are of type `application/atom+xml`, but since they are representing different concepts on the FSM level, there has to be a way how to differentiate between those two cases.

The FSM architecture has no built-in support for the identification, authentication, or authorization of users. These issues are left to mechanisms that can be determined by specific FSM implementations and deployments, and since FSM is based on the principles of Web architecture and all information is managed through URI-identified resources, and since all interactions are modeled according to the principles of *Representational State Transfer (REST)*, layering access control mechanisms on top of FSM can be accomplished fairly easily.

4 OPML Conversion



dret subscriptions in Google Rea…
Folder to dret subscriptions in Goo…

yiming subscriptions in Google…
Folder to yiming subscriptions in…

delicious REST
Folder to delicious REST

AdMob Metrics
http://metrics.admob.com/feed/atom/

Alan Dean
http://feeds.feedburner.com/alandean

Android Developers Blog
http://feeds.feedburner.com/blogspot/…

Applying the Web to Enterprise IT
http://www.nordsc.com/blog/?feed=r…

Benlog
http://benlog.com/feed/

Big Picture Notes
http://www.boston.com/bigpicture/bp…

Bill de hÓra
http://www.dehora.net/journal/atom.xml

BitWorking | Joe Gregorio

Figure 2: View of OPML Imports and one Import

The *Outline Processor Markup Language (OPML)* is not as expressive or backed by a protocol as FSM is, but it currently is the most widely supported import/export format supported by feed readers. We have thus implemented transformations from OPML to FSM and vice versa, so that feed subscriptions in existing readers can be easily migrated, and so that FSM subscriptions can be exposed in a way that makes them accessible to feed readers supporting OPML. In this latter case, feed subscriptions will only be used in the simple way supported by plain feed readers, and the more sophisticated features for query-enabled and spatial feeds (as described in Sections 5 and 6) will not be supported.

```
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:fsm="http://dret.net/fsm/1.0">
  <title>Uploads from dret</title>
  <summary>Uploads from dret, tagged campanile</summary>
  <id>...</id>
  <link rel="edit" href="..."/>
  <link rel="alternate" href="http://api.flickr.com/services/feeds/photos_public.gne?
                              id=20266194@N00&amp;format=atom&amp;tags=campanile"/>
  <link rel="http://dret.net/fsm/1.0/subscription_to"
   href="http://api.flickr.com/services/feeds/photos_public.gne"/>
  <fsm:subtype>subscription</fsm:subtype>
  <fsm:query scheme="http://api.flickr.com/services/feeds/" uri="true">
    format=atom&amp;id=20266194@N00&amp;tags=campanile
  </fsm:query>
</entry>
```

Figure 3: FSM Subscription for Flickr's Query-Enabled Feed API

A single OPML document may produce several FSM feeds as output. The reason for this is that OPML supports hierarchical structures within one file (in hierarchically structured XML), whereas FSM feeds are flat structures, and an OPML hierarchy is thus mapped to bundles that refer to other FSM subscription feeds (as shown in Figure 1). If the FSM feeds are to be added to a FSM server that supports AtomPub, the software running the upload process must take each feed and create a new FSM subscription feed for it, and then `POST` individual entries to this feed either representing FSM subscriptions (OPML subscriptions), or FSM bundles (OPML folders).

Figure 2 shows the iOS client views (more about the client in Section 7.2) of two OPML imports being managed in one subscription feed as two bundles (two individual OPML imports of Google Reader subscriptions have been grouped in this scenario), and the list view of one of these bundles, which corresponds to the OPML subscriptions as they have been transformed into FSM. Because OPML folders are translated into FSM bundles and individually referencable FSM feeds, OPML imports can also be used in a way where individual folders of imported OPML subscriptions can be shared and thus be reused as FSM bundles in multiple subscription feeds. In the simple OPML-based scenario shown in Figure 2, the iOS feed client essentially becomes a feed reader that is configured with Google Reader feed subscriptions.

Converting from FSM to OPML requires an approach which takes FSM's decentralized approach and turns it into a single OPML document. In this case, FSM subscription feeds are "harvested" (following the links found in bundle entries) and then converted into one OPML document, where each FSM bundle in a subscription feed is converted into an OPML folder. The conversion from FSM to OPML is also implemented in XSLT and uses the built-in support for document access to `GET` individual FSM subscription feeds. Queryable feeds are converted to OPML by hardcoding the query string into the URI of the subscribed feed, so that the exported OPML information reflects the subscribed-to query that is represented in the query information that is managed by FSM for queryable feeds (as described in the following section).

5 Queryable Feeds

For simple data feeds and subscriptions to them, the basic architecture shown in the upper part of Figure 1 is sufficient. However, in many cases, feeds are more than one-dimensional datastreams and can be regarded as query result serializations [9].[2] There have been proposals for standardizing query languages for feeds,

[2]Since collections exposed as feeds have no inherent structure, feeds as they are often found on the Web actually can be regarded as having a built-in "query" of always returning the most recent entries first. Explicit query features simply add the

examples for this are the *Feed Item Query Language (FIQL)* [10] and the *Resource Query Language (RQL)*. No such language has been standardized or has gained widespread adoption so far. However, despite the lack of an established standard or de-facto standards, many feeds backed by structured data sources or services do provide service-specific query languages or filtering capabilities.

As one example, the feeds provided by the popular *flickr* photo sharing service provide a limited set of query capabilities. They allow to query for user ID(s) and tag(s), and they allow to specify how to query for specified tags (supported operators are `and` and `or`). The flickr API[3] defines a simple set of parameters that can be used in the URI query string. This constitutes a query-enabled feed service, but it requires specific knowledge of the API to understand the syntax and semantics of the query parameters. The parameters are then composed into a query string according to the same rules that are used for HTML form parameters. While this query parameter encoding is by no means required for feed query parameters, it is a widely used practice because it is easy to understand for users of the feed, and because it is well-supported by standard URI parsers (which are part of many Web-oriented development frameworks) on the provider side of the feed.

FSM supports extensions to the Atom format that allow to capture query parameters. The idea behind this design is that, as shown in Figure 1, it thus becomes possible to specifically support query-enabled feed services. Since it currently is impractical to assume the existence of a single feed query language, query information for feed subscriptions requires a *query scheme*, which is identified by a URI. Based on the query scheme, the extension element then contains the *query string*, which is a serialization of the query for a subscription. An example for this is the flickr API subscription shown in Figure 3.

In this example, the query string not only encodes the query parameters (a user ID and a tag), it also can be used directly as a URI query string, which is indicated by `uri="true"` attribute. The `alternate` link points to the feed URI containing the query string, whereas the `subscription_to` link only uses the feed's base URI without the query string. This means that even without knowing anything about queryable feeds, a generic feed client can follow the `alternate` link and will see the contents of the subscribed feed using the query string. An FSM-aware feed client, on the other hand, can use the feed's base URI as specified in the `subscription_to` link, and can, if it does understand the `http://api.flickr.com/services/feeds/` query scheme, give the subscriber the ability to change the query, by parsing the current subscription query string, and giving the user the possibility to change it. If the FSM client wants to change the subscription's query parameters, it has to be able to `PUT` an updated version of the FSM subscription entry to the member URI that is specified in the `edit` link.

6 Spatial Feeds

Spatial feeds represent a more complex use case for FSM. By spatial feeds, we mean not only the relatively simple idea of putting geospatial features into a feed-based format for reader consumption (one example for this approach is *GeoRSS* [11]), but also the delivery of full-featured location-based services via open and standard feed-based mechanisms and protocols. In earlier work [12], we explored the problem of providing open, scalable, and mashable location-based services on the Web.

In particular, spatial feed support in FSM introduces the additional concept of *views*. Each view consists of a particular spatial region and a set of feed subscriptions providing data of interest for that region. For example, a researcher may be interested in exploring earthquake occurrences relative to population density figures in the San Francisco Bay Area. In this case, a view may be created consisting of the Bay Area as the region, and two data feeds of earthquakes and population density figures as data subscriptions for that view. The bottom section of Figure 1 provides a depiction of FSM's relationships with spatial feeds.

capability to augment or overwrite those default query semantics.

[3]http://www.flickr.com/services/feeds/docs/photos_public/

The FSM architecture manages not only the set of interesting geospatial data sources as feeds, but also the set of views that the user is interested in, and any query parameters that may be used for feeds within that view.

7 Implementation

To prove the concept and demonstrate all of the features of FSM, we have implemented a read/write enabled FSM server and an iOS-based spatial FSM client prototype that not only processes normal feed subscriptions and queryable subscriptions, but also supports spatial service subscriptions and views.

7.1 FSM Server

The prototype FSM Server is a general-purpose AtomPub server, implemented in Ruby on Rails, that is modified with our previously documented FSM extensions. As a prototype, it assumes that it will only handle subscriptions and views for one user, but is otherwise fully functional.

The server exposes two workspaces, which are `feeds` and `views`. Following the FSM extension to AtomPub workspaces, it also links to `edit` URI for each workspace within the service document. Sending an HTTP `POST` with an appropriately formatted feed document will create a new collection, which will then be published as an FSM subscription feed. Re-reading the service document and the new collection will be reflected, with its own `edit` link so that its attributes can be updated via `PUT` or deleted via `DELETE`.

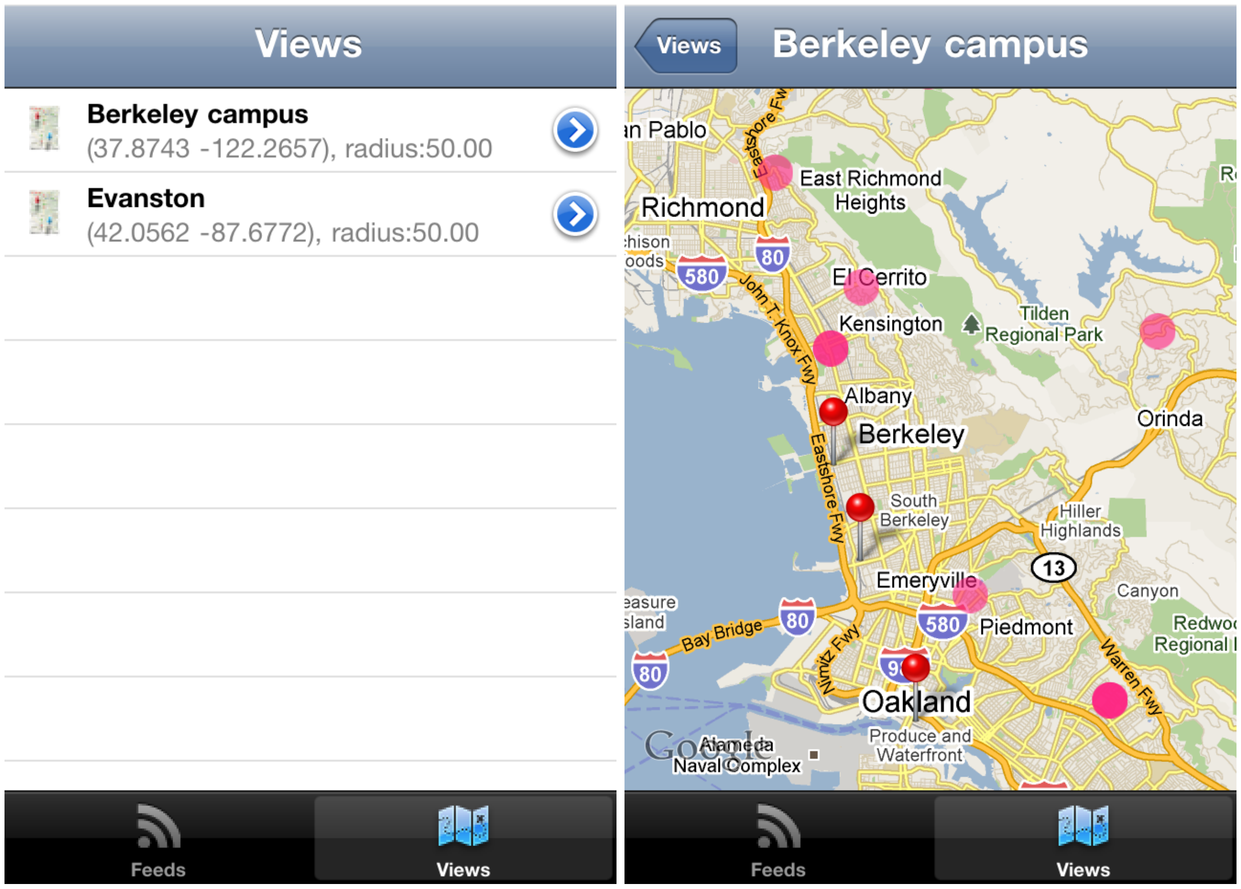
For each collection thus created, subscriptions and views can be added to them. Again, via the AtomPub protocol, sending an HTTP `POST` with an appropriately formatted subscription entry document to a subscription feed will create a subscription. To create a bundle, a two-step process is needed. The client first creates the collection via the workspace, and then creates the bundle reference with a link to to the new subscription feed by `POST`ing a bundle entry document to an existing subscription feed. Because creating the "bundled subscriptions" and the bundle entry are two separate steps, they can be executed on different servers, thus creating a bundle that references an FSM subscription feed managed elsewhere.

The server is seeded with subscriptions and bundles imported from OPML documents. As described in Section 4, converting from OPML to FSM is implemented using XSLT which takes a single OPML as input and produces potentially multiple FSM subscription feed documents on output. We currently use OPML files exported from a user's Google Reader account — but one can conceivably use any form of OPML output produced by a feed-managing piece of software. A second tool then parses the base FSM feed generated, retrieves each entry in turn (dereferencing links to other files if the entry is a bundle), and `POST`s them as individual entry documents to an appropriate collection (or workspace) on the FSM server. This process highlights that the FSM server does not support "batch uploads", but this is a restriction of AtomPub itself.

7.2 iOS FSM Client

The FSM client prototype is intended as a proof of concept FSM feed reader. It focuses on read-based interactions with the FSM server. It has two main views. In the first tab (shown in Figure 2), the user interacts with his subscriptions as retrieved via FSM subscription feeds. In the example shown, there are two sets of subscription feeds in the user's workspace, created from Google Reader OPML files. As per the decentralization benefit of FSM, each subscription feed can be hosted by a different organization on a different FSM server, or even in a hypothetical FSM cloud service. They may not even "belong" to the same user. As long as the user has read access, a subscription feed can be imported/shared with another user's own subscription feed, and can thus show up in the client interface. Any changes to the subscriptions are reflected on the next refresh, in a typical pull-based feed interaction.

Tapping a bundle feed will dereference the bundle into its component subscriptions and any bundle references of its own. Tapping a normal data feed will show a typical feed reader interface, with an option of opening the HTML alternate in the iOS standard browser, Mobile Safari. Tapping a spatial feed or a view will present a map-based visualization.



Berkeley campus
(37.8743 -122.2657), radius:50.00

Evanston
(42.0562 -87.6772), radius:50.00

Figure 4: iOS FSM client — in (a) showing views, and in (b) showing map visualization of the Berkeley view

8 Related Work

There are few direct analogues to FSM-style feed-based subscription management, beyond aforementioned OPML – which itself is not a feed-based subscription language, but a generic representation language for URIs. Research in mobile push services often discuss service subscription management (for example, in [13]), especially for push-based content delivery to mobile clients, but more specifically intended for notification and SMS-like services than Web-based services. On the WS* front, web service management involve fairly tightly coupled mechanisms, via WSMF implementations [14].

Research in geospatial and enterprise mashups are perhaps the most closely related to the spatial feeds use case. Damia [15] is an enterprise system that generates feeds from multiple sources — such as IT intranets, user desktop machines, and the Web, and allows users to create on-the-fly data integrations and mashups.

Similarly, mashups are becoming more important in various scientific and government contexts, such as water management [16], which currently still require toolkits to construct and are cumbersome to share. An FSM back-end would relieve much of the infrastructural responsibility from these mashup systems, allowing them to focus on visualization or other techniques that enhance user experience or efficiency.

The combined architecture of FSM and spatial feeds is targeted at similar use cases as the standards produced by the *Open Geopspatial Consortium (OGC)*, in particular the *Web Feature Service* [17]. Our architecture avoids the heavier WS-* stack of protocols employed by the OGC standards, and instead builds on simple and widely supported coupled Web standards. The goal of this design is to avoid the focus of the OGC standards, which is mostly back-end integration on fully-featured platforms, and design an architecture which can be easily used across all kinds of platforms, from fully-featured back-end integration scenarios to resource-constrained environments of embedded devices bandwidth-constrained networks.

9 Future Work

The current FSM architecture is based on plain feeds and thus works pull-based. The assumption is that clients of FSM subscription feeds learn about all feeds in a subscription, and then aggregate those feeds themselves (often using simple `GET`s to retrieve the contents of all subscribed feeds). One attractive service that might be provided based on FSM would be to provide *PubSubHubbub (PuSH)* support for all feeds listed in an FSM subscription feed. In such a scenario, a value-added FSM-based service would not only manage FSM subscription feeds, it would also implement PuSH services for those feeds, allowing clients to subscribe to the subscription feed's PuSH service via the PuSH protocol. We believe that such a notification service aggregating multiple data sources and pushing them to one consumer might be a very interesting scenario for how to implement near-realtime notifications in an efficient way; it could be using PuSH as a delivery mechanism or could even be tunneled through platform-specific push channels such as the iOS *Apple Push Notification (APN)* service, Android's *Cloud to Device Messaging (C2DM)*, or HTML5's server-sent events or Web sockets. One part of our future work will be to explore this facet of adding push capabilities to FSM as an additional service layer.

Also referring to *PubSubHubbub (PuSH)*, FSM could be more immediately useful for exposing and maybe even managing PuSH subscriptions. Currently, PuSH has no way how a client can find out about its current subscriptions on a PuSH hub. FSM could be a natural choice for this, and going even further, the current PuSH-specific subscription and unsubscription interactions could be replaced by using FSM's AtomPub based methods of `POST`ing and `DELETE`ing FSM subscriptions. PuSH still needs some custom interactions (such as verifying a subscription as legitimate), but we believe that be reusing more of Atom and AtomPub standard interactions and FSM's way of using them for feed subscription scenarios, PuSH could be improved to be easier usable.

We plan to use FSM as the management platform for "Web of Things" scenarios such as aggregating and consolidating electrical appliances and their energy consumption [18], where the services often have different properties because there are many more feed sources (sensor networks often manage large numbers of data providers), and there is typically little interest in consuming unfiltered datastreams (such as measurements in 10sec intervals). In those scenarios, managing a large set of feeds and managing query parameters for them becomes essential for building useful services, and we hope that FSM will make it easier to build services and share service parameters openly.

10 Conclusions

This papers presents the *Feed Subscription Management (FSM)* architecture, which is used for managing feed subscriptions for plain data feeds as well as for more sophisticated query-enabled and spatial feeds. FSM itself is feed-based and thus models all representations according to Atom, and all interactions according to the AtomPub protocol. FSM is designed RESTfully, which means that it is only based on standardized media types (with some extensions) and standardized link relationships (also with few additions). It is meant to provide a management foundation for a decentralized ecosystem of feed-based services, and itself is built in a way which supports the decentralized management and use of FSM subscription data. While work on this management layer has finished and is the main topic of this paper, we see a lot of opportunities in further work (as described in Section 9), both implementing more sophisticated service layers on top of FSM, such as in the areas of push capabilities and better models for LBS information, as well as for using FSM in specific application scenarios that need scalability and decentralization in terms of how they manage data sources and services.

References

[1] M. Nottingham and R. Sayre, “The Atom Syndication Format,” Internet RFC 4287, December 2005.

[2] M. Nottingham, “Feed Paging and Archiving,” Internet RFC 5005, September 2007.

[3] J. Gregorio and B. de Hóra, “The Atom Publishing Protocol,” Internet RFC 5023, October 2007.

[4] T. Berners-Lee, “The Web of Things,” *ERCIM News*, no. 72, p. 3, January 2008.

[5] E. Wilde, “The Plain Web,” in *First International Workshop on Understanding Web Evolution (WebEvolve2008)*, Beijing, China, April 2008, pp. 79–83.

[6] E. Wilde, E. C. Kansa, and R. Yee, “Web Services for Recovery.gov,” School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2009-035, October 2009.

[7] R. Yee, E. C. Kansa, and E. Wilde, “Improving Federal Spending Transparency: Lessons Drawn from Recovery.gov,” School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2010-040, May 2010.

[8] E. Wilde and I. Pesenson, “Feed Feeds: Managing Feeds Using Feeds,” School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2008-025, May 2008.

[9] E. Wilde and A. Marinos, “Feed Querying as a Proxy for Querying the Web,” in *Eighth International Conference on Flexible Query Answering Systems*, ser. Lecture Notes in Artificial Intelligence, vol. 5822. Roskilde, Denmark: Springer-Verlag, October 2009, pp. 663–674.

[10] M. Nottingham, “FIQL: The Feed Item Query Language,” Internet Draft draft-nottingham-atompub-fiql-00, December 2007.

[11] Open Geospatial Consortium, “An Introduction to GeoRSS: A Standards Based Approach for Geo-enabling RSS feeds,” OGC 06-050r3, July 2006.

[12] Y. Liu and E. Wilde, “Scalable and Mashable Location-Oriented Web Services,” in *10th International Conference on Web Engineering (ICWE 2010)*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, G. Kappel, and G. Rossi, Eds., vol. 6189. Vienna, Austria: Springer-Verlag, July 2010, pp. 307–321.

[13] I. Podnar, M. Hauswirth, and M. Jazayeri, “Mobile push: delivering content to mobile users,” in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 563 – 568.

[14] M. P. Papazoglou and W.-J. v. d. Heuvel, “Web Services Management: A Survey,” *IEEE Internet Computing*, vol. 9, pp. 58–64, November 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1100860.1100912

[15] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, “Damia: Data Mashups for Intranet Applications,” in *2008 ACM SIGMOD International Conference on Management of Data*, J. T.-L. Wang, Ed. Vancouver, Canada: ACM Press, June 2008, pp. 1171–1182.

[16] C. Granella, L. Díaza, and M. Goulda, “Geospatial Web service integration and mashups for water resource applications,” in *Proceedings of the XXI Congress of the International Society for Photogrammetry and Remote Sensing (ISPRS 2008)*. Citeseer, 2008, pp. 661–666.

[17] Open Geospatial Consortium, “Web Feature Service Implementation Specification,” OGC 04-094, Version 1.1.0, May 2005.

[18] D. Guinard, V. Trifa, and E. Wilde, “A Resource Oriented Architecture for the Web of Things,” in *Second International Conference on the Internet of Things (IoT 2010)*, Tokyo, Japan, November 2010.