Getting to know the Atom Publishing Protocol, Part 1: Create and edit Web resources with the Atom Publishing Protocol

Take an introductory walk through the basic operation of the protocol

James Snell October 17, 2006

The Atom Publishing Protocol is an important new standard for content publishing and management. In this article, explore a high-level overview of the protocol and its basic operation and capabilities.

View more content in this series

During the past few of years, Web content syndication technology has grown in importance both on the Internet and behind the firewall. In July of 2005, the Internet Engineering Task Force's (IETF) Atom Publishing Format and Protocol Working Group (known simply as "atompub") published the first of two standards specifications intended to provide "a feed format for representing and a protocol for editing Web resources such as Weblogs, online journals, Wikis, and similar content." The Atom Syndication Format, or Atom 1.0 as it is known commonly, has since been deployed to millions of Web sites and is supported by every major syndication platform on the market. Today, just over a year later, work nears completion on the second of the two specifications: The Atom Publishing Protocol.

The Atom Publishing Protocol is an HTTP-based approach for creating and editing Web resources. It is designed fundamentally around the idea of using the basic operations provided by the HTTP protocol (such as GET, PUT, and DELETE) to pass around instances of Atom 1.0 Feed and Entry documents that represent things like blog entries, podcasts, wiki pages, calendar entries and so on.

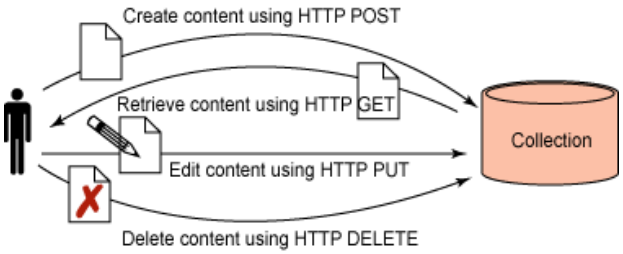
The discussion that follows will present an introductory walk-through of the basic operation of the protocol. This discussion will assume that you have an good understanding of content syndication using the Atom 1.0 Syndication Format and a rudimentary understanding of HTTP. As you read through this overview, I recommend that you keep a copy of the Atom 1.0 (RFC 4287) and HTTP 1.1 (RFC 2616) specifications handy as a cross reference for the various elements and methods

discussed. If you are not familiar with the Atom format, I recommend you look at an article I wrote for developerWorks last year, "An overview of the Atom Syndication Format" (See Resources).

The high level overview

Central to the Atom Publishing Protocol is the concept of *collections* of editable resources that are represented by Atom 1.0 Feed and Entry documents. A *collection* has a unique URI. Issuing an HTTP GET request to that URI returns an Atom Feed Document. To create new entries in that feed, clients send HTTP POST requests to the collection's URI. Those newly created entries will be assigned their own unique edit URI. To modify those entries, the client simply retrieves the resource from the collection, makes its modifications, then puts it back. Removing the entry from the feed is a simple matter of issuing an HTTP DELETE request to the appropriate edit URI. All operations are performed using simple HTTP requests and can usually be performed with nothing more than a simple text editor and a command prompt.

Figure 1. The Atom Publishing Protocol uses simple HTTP methods for publishing and managing content.



Listing 1. Interacting with an Atom Publishing endpoint using the open source curl HTTP client

```
curl -s -X POST --data-binary @entry.xml http://example.org/atom/entries
curl -s -X GET http://example.org/atom/entries/1
curl -s -X PUT --data-binary @entry.xml http://example.org/atom/entries/1
curl -s -X DELETE http://example.org/atom/entries/1
```

Discovering what collections are available

The first step to using any APP-enabled service is to determine what collections are available and what types of resources those collections can contain. The Atom protocol specification defines an XML format known as a service document that a client can use to introspect an endpoint.

To retrieve the service document, send a HTTP GET request to the service document URI.

Listing 2. Retrieving an APP service document from a server

```
GET /servicedocument HTTP/1.1
Host: example.org
```

The server should respond with a service document that enumerates the collections available to the client as illustrated in Listing 3.

Listing 3. A simple APP service document

```
HTTP/1.1 200 OK
Date: ...
Content-Type: application/atomserv+xml; charset=utf-8
Content-Length: nnn

<service xmlns="..." xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title>My Weblog</atom:title>
    <collection href="http://www.example.org/blog/entries">
      <atom:title>Entries</atom:title>
      <accept>entry</accept>
    </collection>
    <collection href="http://www.example.org/blog/photos">
      <atom:title>Photos</atom:title>
      <accept>image/*</accept>
    </collection>
  </workspace>
</service>
```

Each collection element listed in the service document represents a container within which some piece of content can be stored. Workspace elements in the document serve only to group related collections into logical sets. For instance, a single user might have multiple accounts for a given blogging service that provides different containers for blog entries, uploaded files, bookmarks, and so on. Each service can be represented as a separate workspace in the service document.

The collection element provides the address of the collection (the href attribute) and a listing of the types of content that can be added to a collection (identified by mime type in the accept elements). The example in Listing 3 has two collections, one that accepts only Atom Entry Documents and one that only accepts image files (such as PNG, GIF, JPEG, and others).

Adding an entry to a collection

Once you have the address of a collection, we use the HTTP POST method to add new resources as illustrated in Listing 4.

Listing 4. Posting an entry to an APP collection

```
POST /blog/entries HTTP/1.1
Host: www.example.org
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Atom-Powered Robots Run Amok</title>
  <link href="http://example.org/2003/12/13/atom03"/>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
  <updated>2003-12-13T18:30:02Z</updated>
  <author><name>James</name></author>
  <summary>Some text.</summary>
</entry>
```

In Listing 4, the example adds an Atom Entry Document to the collection located at http://example.org/blog/entries. The URI of the collection is retrieved from the service document in Listing 3. Note that the entry posted must be valid -- that is, it must have the id, author and

updated elements even though many APP servers will choose to ignore and overwrite the client-provided values.

A successful response to the POST request, illustrated in Listing 5 below, provides the client with two critical pieces of information: the status of the request (the HTTP response code) and the address of the resource that was just created, contained in the Location header.

Listing 5. A response to a successful POST operation

```
HTTP/1.1 201 Created
Date: nnnn
Content-Type: application/atom+xml; charset=utf-8
Content-Location: /blog/entries/1
Location: /blog/entries/1
ETag: "/blog/entries/1?1"
Last-Modified: Sat, 12 Aug 2006 13:40:03 GMT

<?xml version="1.0" ?<
<entry xmlns="http://www.w3.org/2005/Atom" >
  <id>tag:example.org,2006:/blog/entries/1</id>
  <title>Atom-Powered Robots Run Amok</title>
  <link href="http://example.org/2003/12/13/atom03"/>
  <link rel="edit" href="http://example.org/blog/entries/1" />
  <updated>2006-08-12T13:40:03Z</updated>
  <author><name>James M Snell</name></author>
  <summary>Some text.</summary>
</entry>
```

Because some APP servers can modify various key aspects of the entry (such as the id, author and updated elements), the response returned by the server might include a copy of the entry that was actually added to the collection. This gives clients a way to reconcile the entry they sent to the sever with the entry was actually created.

Listing the entries in a collection

Once an entry is added to a collection, clients can retrieve listings of its member resources by issuing a GET request to the collection's URI as shown in Listing 6.

Listing 6. Retrieving a collection feed

```
GET /blog/entries HTTP/1.1
Host: example.org
```

The response to this request will be an Atom Feed Document whose set of entries each represent exactly one member resource in the collection as illustrated in Listing 7.

Listing 7. An Atom Feed Document for an APP Collection

```
HTTP/1.1 200 OK
Date: ...
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnn
ETag: "/blog/entries?132"
Last-Modified: Sat, 12 Aug 2006 13:40:03 GMT

<feed xmlns="http://www.w3.org/2005/Atom"
      xml:base="http://example.org/blog/entries">
  <id>http://example.org/blog/entries</id>
  <title>My Blog Entries</title>
```

```
  <updated>2006-08-12T13:40:03Z</updated>
  <link rel="self" href="/blog/entries" />
  <link href="http://blog.example.org" />
  <entry>
    <id>tag:example.org,2006:/blog/entries/1</id>
    <title>Atom-Powered Robots Run Amok</title>
    <link href="http://example.org/2003/12/13/atom03"/>
    <link rel="edit" href="http://example.org/blog/entries/1" />
    <updated>2006-08-12T13:40:03Z</updated>
    <author><name>James</name></author>
    <summary>Some text.</summary>
  </entry>
  <entry>
     ...
  </entry>
  ...
</feed>
```

Treat the feed returned by the collection like a type of index over that collection, much like performing a "dir" or "ls" command on a file system.

The entries themselves are ordered according to the value of each entry's updated element with most recently updated entries listed first. Additionally, the listing of entries can span across multiple Atom Feed Documents linked to one another using so-called paging links as illustrated in Listing 8.

Listing 8. A snippet of a feed showing the use of paging links

```
<feed xmlns="http://www.w3.org/2005/Atom"
      xml:base="http://example.org/blog/entries?page2">
  <link rel="next" href="entries?page3" />
  <link rel="previous" href="entries?page1" />
  ...
```

Paging links provide a means to break up potentially large listings of collection member resources into smaller, more manageable subsets.

Editing an entry

To edit an entry, clients first need to retrieve an editable representation. To do this, issue a GET request to the members' Edit URI as shown in Listing 9 below. This is essentially equivalent to opening a document in a local text editor prior to editing it.

Listing 9. Retrieve an editable representation of a resource

```
GET /blog/entries/1 HTTP/1.1
Host: example.org
```

The response to this request should be an Atom Entry Document as seen in Listing 10.

Listing 10. An Atom Entry Document representing an editable resource

```
HTTP/1.1 200 OK
Date: nnn
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnn
ETag: "/blog/entries/1?1"
Last-Modified: Sat, 12 Aug 2006 13:40:03 GMT

<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom" >
  <id>tag:example.org,2006:/blog/entries/1</id>
  <title>Atom-Powered Robots Run Amok</title>
  <link href="http://example.org/2003/12/13/atom03"/>
  <link rel="edit" href="http://example.org/blog/entries/1" />
  <updated>2006-08-12T13:40:03Z</updated>
  <author><name>James</name></author>
  <summary>Some text.</summary>
</entry>
```

Once the editable representation is received, the client can generally make whatever modifications to the entry it chooses (within reason) then issue a PUT request back to the entries Edit URI to update (Listing 11).

Listing 11. A modified Atom entry sent back to the server

```
PUT /blog/entries/1 HTTP/1.1
Host: example.org
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnnn
If-Match: "/blog/entries/1?1"
If-Unmodified-Since: Sat, 12 Aug 2006 13:40:03 GMT

<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom" >
  <id>tag:example.org,2006:/blog/entries/1</id>
  <title>Atom-Powered Robots Run Crazy</title>
  <link href="http://example.org/2003/12/13/atom03"/>
  <link rel="edit" href="http://example.org/blog/entries/1" />
  <updated>2006-08-12T13:40:03Z</updated>
  <author><name>John</name></author>
  <summary>Some different text.</summary>
</entry>
```

Note the use of the If-Match and If-Unmodified-Since headers in the PUT request. While optional, use of these headers allows APP implementations to protect against overwriting modifications other clients might have made on a member resource. If either of these conditions is not met, the server should reject the request and notify the client that it is likely that there was a conflict with the resource they are attempting to modify. If the conditions are met and the server considers the modifications submitted by the client to be acceptable, it will respond with an appropriate success response.

Deleting an entry

For a client to remove a resource from a collection, it sends a DELETE request to the Edit URI as illustrated in Listing 12.

Listing 12. Deleting a resource from a collection

```
DELETE /blog/entries/1 HTTP/1.1
Host: example.org
```

Following a successful delete, the entry should no longer appear within the collections Atom feed and should no longer be available for editing.

Adding media resources to a collection

You can also add arbitrary media resources such as photographs, documents, audio recordings, and so forth to APP collections. Such items are called *media-link entries* by the APP specification due to the fact that when these resources are added to the collection, the server will create an Atom entry document linked to the media resource posted by the client.

While originally designed simply to allow Weblog authors to upload media objects they may want to include in their posts, the Atom Publishing Protocol's support for arbitrary media resources makes it ideally suited for a broad range of applications including:

- Podcasting
- Video blogging
- Photo libraries
- Wikis and situational applications
- Document management
- XML repositories
- Software distribution
- Productivity applications (such as Office Suites)
- And many others...

To create a media-link entry, a client issues a POST request to the collection URI, but instead of including an Atom Entry Document, the client includes a representation of the media resource to be linked (Listing 13).

Listing 13. Posting a binary image file to an APP collection

```
POST /blog/photos HTTP/1.1
Host: example.org
Content-Type: image/png
Content-Length: nnnn
Slug: A trip to the beach

{binary image data}
```

If the collection can store the type of media resource sent by the client, it does so and creates an Atom Entry Document linking to the media resource as illustrated in Listing 14. The Slug header contained in the request is a new HTTP Entity Header introduced by the Atom Publishing Specification used to associate a simple name with the member resource that can be used for a variety of purposes when creating and managing the resource. For instance, the server can use the value of the slug when creating the URI of the member resource or when setting the value of

the title element in the Atom Entry Document. The Slug header can be used when posting Atom entries or media resources but will most frequently be used with the latter.

Listing 14. A media-link entry created in response to a media post

```
HTTP/1.1 201 Created
Date: nnnn
Content-Location: /blog/photos/a_trip_to_the_beach
Location: /blog/photos/a_trip_to_the_beach
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnnn
Slug: A trip to the beach
ETag: "/blog/photos/a_trip_to_the_beach?1"
Last-Modified: Sat, Aug 12 2006 14:11:04 GMT

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>tag:example.org,2006:/blog/photos/a_trip_to_the_beach</id>
  <title>A trip to the beach</title>
  <link rel="edit"
    href="http://example.org/blog/photos/a_trip_to_the_beach" />
  <link rel="edit-media" type="image.png"
    href="http://example.org/blog/photos/a_trip_to_the_beach?media" />
  <updated>2006-08-12T14:11:04Z</updated>
  <author><name>James</name></author>
  <summary>A trip to the beach</summary>
  <content type="image/png"
    src="http://blog.example.org/photos/a_trip_to_the_beach" />
</entry>
```

Media-link entries will always contain a content element whose src attribute provides the URI of the media resource that was created. Consider this URI usable for publicly referencing the media resource. With the separate edit-media link, you can identify the URI that can be used to update the media resource.

Editing media resources

Editing a media resource posted to a collection is generally identical to editing an Atom entry. The first step is to retrieve an editable version of the resource by issuing a GET request on the URI specified by the edit-media link (Listing 15).

Listing 15. Retrieving an editable representation of a media resource

```
GET /blog/photos/a_trip_to_the_beach?media HTTP/1.1
Host: example.org
```

After the editable representation is returned, the client makes whatever modifications it chooses, then issues a PUT request back to the edit-media URI (Listing 16).

Listing 16. Modifying a media resource

```
PUT /blog/photos/a_trip_to_the_beach?media HTTP/1.1
Host: example.org
Content-Type: image/png
Content-Length: nnn

{new binary image data}
```

Protecting collections

While the Atom Publishing Protocol does not require that implementations use authentication, it is highly recommended that they do so in order to prevent malicious clients from creating and modifying collection members. At a minimum, implementations are required to be capable of using HTTP Basic authentication and TLS/SSL connections. In practice, however, APP clients are likely to see a variety of authentication mechanisms. Regardless of the type of authentication employed, however, servers should utilize standard HTTP-style challenges to identify the type of authentication selected.

For instance, if a server receives an unauthorized request from a client, the server should respond with an 401 Unauthorized response that includes a WWW-Authenticate header as shown in Listing 17.

Listing 17. A response to an unauthorized request

```
HTTP/1.1 401 Unauthorized
Date: nnn
WWW-Authenticate: Basic realm="my blog"
```

The client can then reissue the request with an appropriate Authorization header.

Listing 18: An authenticated request

```
POST /entries/blog HTTP/1.1
Host: example.org:443
Authorization: Basic SmFtZXM6bm90IG15IHJlYWwgcGFzc3dvcmQgOi0p
...
```

Putting APP to work

To this point, I covered the basic operation of the Atom Publishing Protocol, illustrating, through example, all of its core functions. What I have not discussed, however, are the various ways that you can put the Atom Publishing Protocol to work. In the next installment of this series, I will walk through a number of application scenarios that are considered good uses of the protocol. These include such obvious things as Weblogs, social bookmarking and photo album type applications as well as somewhat non-obvious uses in calendaring, contact management, document and media content repositories, database management, situational applications and even Service Oriented Architecture.

Beyond that, you will explore how to implement a Atom Publishing client and server in Java using the Apache Abdera open source Atom implementation currently in incubation at the Apache Software Foundation and will step through the creation of an APP-enabled application service.