The AtomAPI: Publishing Web Content with XML and HTTP

Joe **Gregorio**

Abstract

The AtomAPI is an emerging interface for editing content. The interface is RESTful and uses XML and HTTP to define an editing scheme that's easy to implement and extend. History, basic operation, and applications to areas outside weblogs will be covered.

Table of Contents

1. What's the point 3
    1.1. “Just” use WebDAV 3
2. History 3
    2.1. LiveJournal API 3
    2.2. XML-RPC 4
        2.2.1. Examples 4
        2.2.2. Example 4
        2.2.3. Limitations 5
3. Basic Operation 5
    3.1. Concepts 5
    3.2. Overview 6
    3.3. Example 6
        3.3.1. Create an entry 6
        3.3.2. Update an entry 7
            3.3.2.1. GET 7
4. Optimizations 8
    4.1. Concurrent Updates 9
Bibliography 9

1. What's the point

What are trying to accomplish? Publishing content to the web, primarily weblogs. That is, the creation of chronologically ordered web content authored usually by one person.

1.1. “Just” use WebDAV

What are not trying to accomplish? We are not trying to re-invent WebDAV. WebDAV (Web Distributed Authoring and Versioning) is an HTTP based protocol that uses XML and is used for editing all kinds of web content. Painting with a broad brush the goals on WebDAV and the Atom Publishing Protocol are the same, this is from the webdav.org.

The stated goal of the WebDAV working group is (from the charter) to "define the HTTP extensions necessary to enable distributed web authoring tools to be broadly interoperable, while supporting user needs", and in this respect DAV is completing the original vision of the Web as a writeable, collaborative medium.

The differences become greater when you look at the major features of WebDAV, again quoting from webdav.org[WebDavFaq].

- Locking (concurrency control): long-duration exclusive and shared write locks prevent the overwrite problem, where two or more collaborators write to the same resource without first merging changes. To achieve robust Internet-scale collaboration, where network connections may be disconnected arbitrarily, and for scalability, since each open connection consumes server resources, the duration of DAV locks is independent of any individual network connection.

- Properties : XML properties provide storage for arbitrary metadata, such as a list of authors on Web resources. These properties can be efficiently set, deleted, and retrieved using the DAV protocol. DASL, the DAV Searching and Locating [http://webdav.org/dasl/] protocol, provides searches based on property values to locate Web resources.

- Namespace manipulation : Since resources may need to be copied or moved as a Web site evolves, DAV supports copy and move operations. Collections, similar to file system directories, may be created and listed.

The Atom Publishing Protocol [AtomPubProt] provides no locking, no namespace manipulation, and while it does support resource metadata, it does not provide searches based on that metadata. Note that in this context the WebDAV documentation use of the term namespaces does not mean XML namespaces but instead server URI namespace.

2. History

Almost all of the weblog publishing protocols preceeding the Atom Publishing Protocol are RPC (Remote Procedure Call) interfaces. The idea is similar to CORBA or DCOM in that you are performing a function call remotely, the difference is that the protocols in this case aren't binary but text.

2.1. LiveJournal API

One of first weblog publishing protocols, the LiveJournal protocol formatted all the requests as ' application/x-www-form-urlencoded' and did not use XML. Key aspects:

1. Everything went through an HTTP POST Request.

2. Format was not XML.

3. Style is RPC.

2.2. XML-RPC

As the name suggests this is XML pushed over HTTP to implement an RPC interface.

1. Everything went through an HTTP POST Request.
2. Format was XML.
3. Style is RPC.

2.2.1. Examples

There are quite a few XML-RPC based protocols for editing web content. They, for the most part, do not interoperate with each other. The one exception is the Metaweblog API that does not natively support a call to delete a weblog entry but instead imports the use of a single function from the Blogger API.

1. Manila API
2. LiveJournal XML-RPC Client/Server Protocol. (In addition to the form-urlencoded interface)
3. Blogger API
4. Metaweblog API

2.2.2. Example

Here is an example of a call to create a new entry using the Blogger API:

```
POST /api/RPC2 HTTP/1.0
User-Agent: Java.Net Wa-Wa 2.0
Host: plant.blogger.com
Content-Type: text/xml
Content-length: 515

<?xml version="1.0"?>
<methodCall>
    <methodName>blogger.newPost</methodName>
    <params>
    <param><value><string>C6C.....19BF4E294</string></value></param>
    <param><value><string>744145</string></value></param>
    <param><value><string>ewilliams</string></value></param>
    <param><value><string>secret</string></value></param>
    <param><value><string>Today I had a peanut butter and pickle
    sandwich for lunch. Do you like peanut-butter
    and pickle sandwiches? I do. They're yummy. Please
    comment!</string></value></param>
    <param><value><boolean>false</boolean></value></param>
    </params>
</methodCall>
```

And the response if the request succeeded:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 125
Content-Type: text/xml
Date: Mon, 6 Aug 20001 19:55:08 GMT
Server: Java.Net Wa-Wa/Linux

<?xml version="1.0"?>
<methodResponse>
    <params>
    <param>
    <value><string>4515151</string></value>
    </param>
    </params>
</methodResponse>
```

2.2.3. Limitations

The interfaces above all suffer from one or more of the following limitations:

1. Native HTTP authentication is not used.
2. Passwords sent in the clear.
3. Poor internationalization. (The XML-RPC protocol restricts element content to ASCII).
4. No built-in extensibility mechanisms. (No namespaces)
5. No ability to specify the title or date of an entry being edited.
6. Every request is a POST to the same URI.

   This means that if you were trying to build a large system to handle many such requests and wanted to dispatch different requests to different servers you would have to parse the XML body before knowing where to dispatch it to. Obviously a solution that provided information in the headers, such as the URI, the method and the content-type would be much faster to pull out and dispatch on than parsing an XML document.

So these protocols take XML and try to use it in a protocol and remove two of the most important qualities that XML brings to the table; I18N and extensibility. In addition they use HTTP but ignore the built in authentication mechanism and the ability of servers to quickly dispatch requests based on the headers of a message and not just the contents.

3. Basic Operation

The Atom Publishing Protocol seeks to overcome the limitations of XML-RPC based protocols and to better leverage XML and HTTP.

3.1. Concepts

1. Take full advantage of HTTP (including authentication).
2. No RPC (Use a document centric approach)
3. Take full advantage of XML, in particular all the effort put into the Atom Syndication Format.

3.2. Overview

What are the basic operations when editing a weblog? Creating an 'entry' is one operation. After an 'entry' has been created then the user may wish to edit the entry or possibly delete the entry.

There are two types of resources used in the Atom Publishing Protocol. The first kind of resource is a Collection, it represents a set of entries. The second kind of resource is an entry. Each resource has it's own URI and has a representation. For entries, that representation is a document that is an Atom Entry.

3.3. Example

3.3.1. Create an entry

To create an entry for a site you must POST an Atom Entry to the Collection Resource. For example, if the Collection is located at http://example.org/reilly then the following would be sent to port 80 of the server example.org:

```
POST /reilly HTTP/1.1
Host: example.org
Content-Type: application/atom+xml

<?xml version="1.0" encoding='utf-8'?>
<entry  xmlns="http://www.w3.org/2005/Atom" >
    <title>My First Entry</title>
    <summary>A very boring entry...</summary>
    <author>
        <name>Bob B. Bobbington</name>
    </author>
    <updated>2003-02-05T12:29:29</updated>
    <content type="text" xml:lang="en-us">
        Hello, world!
    </content>
</entry>
```

and a successful POST might receive the following response:

```
HTTP/1.1 201 Created
Location: http://example.org/reilly/1
<?xml version="1.0" encoding='utf-8'?>
<entry   xmlns="http://www.w3.org/2005/Atom" >
    <title>My First Entry</title>
    <summary>A very boring entry...</summary>
    <author>
        <name>Bob B. Bobbington</name>
    </author>
    <updated>2003-02-05T12:29:29</updated>
    <link rel="alternate" type="text/html"
    href="http://example.org/reilly/2003/02/05#My_First_Entry" />
    <id>http://example.org/reilly/1</id>
    <content type="text" xml:lang="en-us">
        Hello, world!
    </content>
</entry>
```

There are several important aspects to note about this transaction. First note that we are just POSTing an Atom entry. This is important since we are re-using the Atom Syndication Format and not creating a different XML format for the Protocol versus the Syndication format[AtomPubFormat]. This also gets use away from the silly ASCII only restrictions of XML-RPC. Secondly the response include a Location: header which specifies the URI at which the new entry was created. In this case the URI is of an Atom entry and not the URI of the HTML resource created. Also, the filled in Atom entry was returned in the response. Additional things to notice that come from the use of the Atom Syndication Format are the use of xml:base and xml:lang.

3.3.2. Update an entry

Once an entry is created we may want to edit it. The first step is to retrieve the latest version of the entry. You may notice that the updated Atom entry was returned in the response from the POST that created the entry. We'll ignore that for now to introduce another part of the protocol. In the headers of the returned response was a Location: header which contains the URI for the newly created Entry resource. If we wish to update the entry just created then we will need to do a GET on the URI returned above.

3.3.2.1. GET

The URI returned above for our first entry was http://example.org/reilly/1 . We send the following request to port 80 of the server at example.org:

```
GET /reilly/1 HTTP/1.1
Host: example.org
Content-Type: application/atom+xml
```

The response is again an Atom entry, the same one we POSTed but with some more infomation filled in by the server:

```
HTTP/1.1 200 Ok
Content-Type: application/atom+xml

<?xml version="1.0" encoding='utf-8'?>
<entry   xmlns="http://www.w3.org/2005/Atom" >
    <title>My First Entry</title>
    <summary>A very boring entry...</summary>
    <author>
        <name>Bob B. Bobbington</name>
    </author>
    <updated>2003-02-05T12:29:29</updated>
    <link rel="alternate" type="text/html"
    href="http://example.org/reilly/2003/02/05#My_First_Entry" />
    <id>http://example.org/reilly/1</id>
    <content type="text" xml:lang="en-us">
        Hello, world!
    </content>
</entry>
```

Now that we have the entry we can modify it then PUT it back to the same URI to update the entry and the corresponding HTML resource.

4. Optimizations

Because we are using HTTP and in particular GET there are optimizations that we can apply to increase the performance of the protocol. Note that these optimizations are not available to protocols that try to push everything through POST. First, we can speed up the GET by using compression. We must tell the server that we will accept compressed documents when we send the GET request:

```
GET /reilly/1 HTTP/1.1
Host: example.org
Accept-Encoding: compress, gzip
Content-Type: application/atom+xml
```

The response indicates what if any compression was used:

```
HTTP/1.1 200 Ok
Content-Type: application/atom+xml
Content-Encoding: gzip

...gzipped stuff goes here...
```

XML is very amenable to compression and will gzip down to 1/2 to 1/3 of it's original size.

But wait, there's more optimizating we can do. The server could return an ETag: header in the response:

```
HTTP/1.1 200 Ok
Content-Type: application/atom+xml
ETag: 3948018403940943

...gzipped stuff goes here...
```

The ETag value is a key that we can use the next time we make a request to that URI. When making a request we can include an If-Match header in the request:

```
GET /reilly/1 HTTP/1.1
Host: example.org
If-Match: 3948018403940943

```

And the response if the entry is unchanged since the last time we requested it is a status code of 304 with no message-body. Now that's an optimization!

```
HTTP/1.1 304 Not Modified
Content-Type: application/atom+xml
```

Note that we can mix these two techniques:

```
GET /reilly/1 HTTP/1.1
Host: example.org
Accept-Encoding: compress, gzip
If-Match: 3948018403940943

```

Now if the entry is unchanged we will get a response with no message-body, and if it is changed the message-body may still be compressed.

These same optimizations can also be applied to the initial POST used to create an entry. We could specify Accept-Encoding: on the POST. In addition the response, if it includes the full Atom entry, can also contain an ETag: header, thus speeding up subsequent GETs.

4.1. Concurrent Updates

The ETag: header also provides for protection against consurrent updates. When we wanted to speed up GETs we provided an If-Match: header in the GET request. That same If-Match: header can be provided on a PUT request and the request will only succeed if the entry is unchanged from when the ETag was generated.

Bibliography

[TypePad] *TypePad Atom API [http://sixapart.com/developers/atom/typepad/]*

[AtomWiki] *An Atom-Powered Wiki [http://www.xml.com/pub/a/2004/04/14/atomwiki.html?page=1]*

[AtomPubProt] *The Atom Publishing Protocol [http://bitworking.org/projects/atom/]*

[AtomGateway] *The Atom-Wiki Gateway [http://interwiki.sourceforge.net/cgi-bin/wiki.pl?AtomGateway]*

[AtomPubFormat] *The Atom Publishing Protocol [http://atompub.org/2005/07/11/draft-ietf-atompub-format-10.html]*

[WebDavFaq] *The WebDAV FAQ: Question 2 [http://webdav.org/other/faq.html#Q2]*

Biography

Joe **Gregorio**
President
BitWorking, Inc. [http://bitworking.org]
Apex
North Carolina
United States of America

Joe Gregorio is President of BitWorking, Inc. He has over 14 years of software design and project management experience working on a range of applications from embedded and web-based systems to Windows desktop applications. He is active in the syndication community, the author of the Atom Publishing Protocol, writes "The RESTful Web" column for the O'Reilly publication "XML.com", and spends time exploring the limits of XML and HTTP.

He holds a Masters of Arts in Mathematics from Dartmouth College and Bachelors Degrees in Mathematics and Computer Science from Eastern Connecticut State University.

Joe Gregorio maintains a personal weblog at http://bitworking.org.