MedBiquitous REST Web Service Design Guidelines

Version 1.0
28 October 2009

Revision History

Date	Version	Description	Author
October 28, 2009	1.0	Initial release	Valerie Smothers vsmothers@medbiq.org

Table of Contents

MedBiquitous Consortium XML Public License and Terms of Use

MedBiquitous XML (including schemas, specifications, sample documents, Web services description files, and related items) is provided by the copyright holders under the following license. By obtaining, using, and or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

The Consortium hereby grants a perpetual, non-exclusive, non-transferable, license to copy, use, display, perform, modify, make derivative works of, and develop the MedBiquitous XML for any use and without any fee or royalty, provided that you include the following on ALL copies of the MedBiquitous XML or portions thereof, including modifications, that you make.

1. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the following notice should be used: "Copyright © [date of XML release] MedBiquitous Consortium. All Rights Reserved. http://www.medbiq.org"
2. Notice of any changes or modification to the MedBiquitous XML files.
3. Notice that any user is bound by the terms of this license and reference to the full text of this license in a location viewable to users of the redistributed or derivative work.

In the event that the licensee modifies any part of the MedBiquitous XML, it will not then represent to the public, through any act or omission, that the resulting modification is an official specification of the MedBiquitous Consortium unless and until such modification is officially adopted.

THE CONSORTIUM MAKES NO WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, WITH RESPECT TO ANY COMPUTER CODE, INCLUDING SCHEMAS, SPECIFICATIONS, SAMPLE DOCUMENTS, WEB SERVICES DESCRIPTION FILES, AND RELATED ITEMS. WITHOUT LIMITING THE FOREGOING, THE CONSORTIUM DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY, EXPRESS OR IMPLIED, AGAINST INFRINGEMENT BY THE MEDBIQUITOUS XML OF ANY THIRD PARTY PATENTS, TRADEMARKS, COPYRIGHTS OR OTHER RIGHTS. THE LICENSEE AGREES THAT ALL COMPUTER CODES OR RELATED ITEMS PROVIDED SHALL BE ACCEPTED BY LICENSEE "AS IS". THUS, THE ENTIRE RISK OF NON-PERFORMANCE OF THE MEDBIQUITOUS XML RESTS WITH THE LICENSEE WHO SHALL BEAR ALL COSTS OF ANY SERVICE, REPAIR OR CORRECTION.

IN NO EVENT SHALL THE CONSORTIUM OR ITS MEMBERS BE LIABLE TO THE LICENSEE OR ANY OTHER USER FOR DAMAGES OF ANY NATURE, INCLUDING, WITHOUT LIMITATION, ANY GENERAL, DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, OR SPECIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF ANY USE OF MEDBIQUITOUS XML.

LICENSEE SHALL INDEMNIFY THE CONSORTIUM AND EACH OF ITS MEMBERS FROM ANY LOSS, CLAIM, DAMAGE OR LIABILITY (INCLUDING, WITHOUT LIMITATION, PAYMENT OF ATTORNEYS' FEES AND COURT COSTS) ARISING OUT OF MODIFICATION OR USE OF THE MEDBIQUITOUS XML OR ANY RELATED CONTENT OR MATERIAL BY LICENSEE.

LICENSEE SHALL NOT OBTAIN OR ATTEMPT TO OBTAIN ANY PATENTS, COPYRIGHTS OR OTHER PROPRIETARY RIGHTS WITH RESPECT TO THE MEDBIQUITOUS XML.

THIS LICENSE SHALL TERMINATE AUTOMATICALLY IF LICENSEE VIOLATES ANY OF ITS TERMS AND CONDITIONS.

MedBiquitous REST Web Services Design Guidelines

1. Acknowledgement

These guidelines are based on work authored by Michael Gilfix of IBM and submitted by Joel Farrell of IBM, Chair of the MedBiquitous Technical Steering Committee. The following members of the MedBiquitous Technical Steering Committee also contributed to this document:

- James Fiore, American Board of Surgery
- Dan Rehak, Learning Technologies Architect
- Andy Rabin, CECity
- Carl Singer, CECity
- Valerie Smothers, MedBiquitous

The committee gratefully acknowledges Jeroen Donkers, Maastricht University, and Rachel Ellaway, Northern Ontario School of Medicine, for their feedback.

2. Scope

This document provides general guidelines for those developing Representational State Transfer (REST) Web services. The document does not strictly adhere to the definition of REST presented by Roy Fielding in his dissertation [Fielding]. This document interprets REST as a general approach broadly following the architectural style for distributed resources.

3. Status

This document is a technical guideline developed for the MedBiquitous community It is designed for newcomers to REST. MedBiquitous technical guidelines are intended to address questions regarding common technical needs related to the implementation of MedBiquitous standards. We welcome your comments on the document. These guidelines are designed to complement the MedBiquitous Web Services Design Guidelines [MedBiq1] and the Knowing When to REST: Simple Object Access Protocol vs. Representational State Transfer [MedBiq2].

4. Introduction

REST is an abstraction of the architecture commonly used by websites. A Web resource has a URL; accessing this URL returns a representation of the resource, such as a web page [Ray]. A link within that resource then leads the client to another resource, or a change in state [Ray]. Resources may be dynamic or static.

A key concept in REST architecture is that of the resource. A resource is any object that can be named. A resource may be a document, blog, or search result, to name a few examples [Ray]. Every resource must have an identifier in REST architecture. Another key concept of REST is that the resource is separate from its representation [Fielding]. So “Ray’s Blog” may be a resource and have its own unique identifier, but the html page that you get back when you read Ray’s blog is the representation of Ray’s Blog in a particular format at a particular point in time. The content may change over time, and it may be represented in more than one format, XML or HTML, for example.

The ATOM Publishing Protocol is an example of a “RESTful” Web service, i.e. a Web service designed using REST principles. The protocol provides a standard way to create, edit, and delete resources, such as newsfeeds. It also provides protocols for retrieving sets of resources and discovering resource collections [RFC5023]. Retrieving a collection of e-learning objects would be an example of RESTful Web service using ATOM. Another RESTful Wes service could allow for the retrieval of metadata for one learning object in the collection.

This document assumes a general conceptual understanding of Web services.

5. Getting Started

If you're new to REST, the references below can help to get you oriented:

- Wikipedia provides a good starting point: http://en.wikipedia.org/wiki/REST
- The day trader REST API design example shows mapping existing APIs to RESTful design: http://bitworking.org/news/201/RESTify-DayTrader
- Finally, the ultimate resource is the HTTP RFC2616. REST fully exploits the HTTP protocol, so familiarity is a necessity.

6. General Approach to Structuring REST Services

REST APIs begin by modeling the service as a set of resources and relationships. These relationships become part of the structuring of the API through navigation of URI paths. This is in contrast to RPC design where APIs can have logical groupings, with implied dependencies between different methods on a given interface. REST URIs are typically dynamic and described in terms of patterns; portions of the URI will be substituted dynamically by the client before invoking a service based on the resource it wishes to interact with. An example is as follows:

```
http://foo.com/accounts/{accountID}/subscriptions/{subID}
```

This is an example of a RESTful pattern. Encoded in this pattern is the relationship that a given account may have multiple subscriptions. An individual account can be referenced by an account ID, and to access a subscription, you have to know both the subscription ID and the owning account for the subscription. Following along with this example, you could have a request like

```
GET http://foo.com/accounts/bob/subscriptions/sub1234
```

to get a description of Bob's subscription with ID sub1234.

Since these relationships are captured in the URL, there is usually a form of navigation that's supported within the API to allow applications to inspect capabilities and available data as they work with the resource. So, in the example above, the corresponding API may have numerous resources building up to that full pattern:

```
http://foo.com/accounts
```

Lists all accounts. This may be a list of IDs or even of URLs to the individual accounts.

```
http://foo.com/accounts/{accountID}
```

Provides a description of an account. This might include account information and a list of references to subscription URLs .

```
http://foo.com/accounts/{accountID}/subscriptions
```

This could provide a list of all subscriptions for that account, again with URI references embedded.

```
http://foo.com/accounts/{accountID}/subscriptions/{subID}
```

Gets information on a specific subscription.

REST services implement Create, Read, Update, and Delete (CRUD) operations on resources as a standard access mechanism for calling those resources. CRUD operations map as follows to HTTP methods:

- POST - create
- PUT - update

- GET - read
- DELETE - delete

For some services, it may be necessary to introduce additional HTTP methods and constructs, but that should be considered a last resort.

Regarding the use of query arguments, query arguments are typically used to filter the contents of a resource or to provide specific criteria on how to assemble the representation.

6.1 Digging in a Bit Deeper on REST Service Design

The following are some guidelines to take into account when thinking about your services:

- Make individual things of interest addressable with a single URL: This allows a REST URI to be used like a reference for a specific piece of data.
- However, also think about 'bulk' operations: API design should also allow the client to operate on multiple pieces of data within a single call for efficiency. These APIs should take a list of datums on which to apply the REST CRUD operations.
- Try to make IDs or variable navigation elements human readable. That may or may not be possible, depending on the uniqueness requirements for the navigation.
- Try to minimize the number of IDs or entity relationships that get exposed through a single REST URL. Exposing too many nested IDs in a single REST URL may make it harder to evolve the service going forward. This is a tradeoff between REST navigability, simplicity, and efficiency.
- Try to design RESTful links to be additive: since links are considered as references, they should be considered permanent or there should be something to support backwards compatibility. API design should attempt to foresee how future extension to the REST resource model might occur. See more on versioning later.
- When providing links between different resources (e.g., a resource returns a response that references another resource), relative paths should be used rather than absolute URLs. Absolute URLs are ones that contain the virtual host and port that are seen by the client, where as relative URLs contain only the path, query, and fragment components of the URI. This is important for several reasons:
    - It tells the client that it can use the same security credentials to access that resource.
    - The real virtual host may be difficult to determine accurately.
    - Using absolute URLs may inadvertently create cross-domain scripting issues.

7. REST Resource Representations

Navigating to the resource is separate from the representation of the resource, or the HTTP entity to be returned from a GET or accepted as part of a PUT/POST exchange. Within the REST style, each resource can have multiple representations or formats for representing the resource. These are signified using the MIME types in standard HTTP headers. The `Content-Type` header indicates the mime type of the body of the HTTP message. Some common MIME types are `text/xml` for XML content and `application/json` for JSON content.
Since a given resource can have multiple representations, it is a good idea for business logic to be structured to work with a common data model. The representation formats should then be generated off this common model. The common model should be rich enough to support an appropriate representation in JSON and XML structures.

A later section provides guidelines on how services should determine which representation the client wants.

7.1 What Should a Service Support?

MedBiquitous defines XML documents for healthcare education and competence assessment. For supporting REST usage enabled by MedBiquitous standards and specification, XML is preferred and must be supported at a minimum. JSON can be used as an alternative format. For more information about JSON, see Introducing JSON [JSON]. If both XML and JSON are available, see Determining Which Representation to Use.

7.2 Determining Which Representation to Use

When a service supports more than one output format, the service may support the accept parameter or Accept http header in order to allow the client to select the format to use.

To determine what format the client wants for the representation, the client should do the following:

- Look for an `accept` URI parameter. This parameter's value should follow the same format as the Accept header.
- Else, look for the Accept http header.

Clients should use the Accept header as the preferred indicator of format. The URI parameter is provided as a convenience for clients that have limited header support.

If the format is unsupported, then the REST service should return an error. If no content header is present but there is body content along with the request, then the service should use the same content type for the response. If the request contains no body and no content header, then the service should select a default. The default for MedBiquitous is XML.

7.3 Guidelines for XML

For XML services, it is typically a good idea to support schemas. Schemas are useful for the construction of clients and validation of input. XSDs for the web service should be bundled with the REST service and made available when a request to the resource is made with the following URI parameter:

- `xsd -` This parameter indicates that the client wishes to receive an XSD for the request. If no XSD exists, then a 404 should be returned.

If there are multiple schemas for a particular request, decide which schema to use by default.

7.4 Guidelines for using ATOM

The ATOM feed format is used for managing a collection of data items. While initially used to syndicate data such as articles on the Web, it has become increasingly popular as a general envelope / metadata wrapper for working with RESTful resources. The ATOM feed format contains a list of entries, where each entry may directly contain content (with its own content type) or contain links to further access content. When building REST services, the ATOM format is often associated with the ATOM Publishing Protocol (APP). APP defines additional CRUD semantics around using ATOM feeds to work with REST services. When using the ATOM format, adherence to APP semantics for operating on feeds should be adhered to. The APP also defines additional concepts around working with workspaces or views of collections. This may or may not be applicable to the REST service -use if appropriate.

The following are some general guidelines for using ATOM:

- All content should go within the content tag within the ATOM entry structure with an appropriate MIME type set. XML extensions to the ATOM entry that are not well known extensions (i.e., appear in an RFC somewhere) should be avoided; this technique essentially builds a proprietary data format on top of ATOM and breaks compatibility (like Google's GData).
- It's a good idea to make all entry ID's globally unique
- For collections that may return large collections, where results are limited in size, the ATOM paging capabilities should be used

8. Date and Time

ISO 8601[ISO 8601] defines a robust set of date and time features that fulfill many use cases. A subset of ISO 8601 is defined in RFC 3339 [RFC 3339] which greatly reduces the variability and complexity that the full ISO 8601 supports. Thus, it is recommended to follow the RFC 3339 recommendations and constrain to just the UTC implicit

"Z" and UTC offset variations. This is useful for server-side development because the Atom syndication format also constrains to these two formats.

These recommendations only address timestamps, dates, and times and do not address formats for duration and intervals.

8.1 Timestamp

If using timestamps or dates as URI parameters, use ISO 8601. Consider using UTC time to avoid time zone confusion.

This MUST be a timestamp in either ISO 8601 formats of **YYYY-MM-DDThh:mm:ssZ** in UTC time or **YYYY-MM-DDThh:mm:ss-0:00**. When the value is presented in a serialized JSON structure, the value must be a string in either formats.

Timestamp examples of UTC implicit "Z"

```
2008-03-13T19:49:00Z 1978-01-22T00:00:00Z
```

Timestamp examples of UTC offset

```
2008-03-13T19:49:00-05:00 1978-01-22T00:00:00+03:00
```

8.2 Dates

In some contexts, such as when not labeling a specific 24 hour range, like a birthday, only the date part is necessary. In these cases, it is permissible to serialize just the date (the format preceding the "T" in the ISO 8601 examples above).

This MUST be a date in the format of YYYY-MM-DD. However, it is recommended that you use the above timestamp format with time values of zero instead of this format unless you need to transfer only the date part. When serialized over JSON, it must be a string.

Date Examples

```
2008-03-13 1978-01-22
```

9. Localization

MedBiquitous standards define how to handle localization issues when relevant to the content of the standard. For example, Healthcare Learning Object Metadata provides a mechanism for encoding metadata in multiple languages. Follow the approach recommended by the applicable MedBiquitous standard or specification.

10. Resource Navigability

Where practical, REST payloads should provide navigation aids to clients to navigate between resources, without having to construct the URI from scratch. These navigation aids should be relative URIs, made available as part of the payload.

In the following example, a REST service provides a list of available learning objects in an html page using the following URI.

```
http://foo.com/learningobjects/
```

Each learning object listed provides a link to metadata for that learning object for those that want more information.

Standard precautions and infection control
Hand washing basics
When to practice hand hygiene
Antiseptic agents
Antimicrobial organisms
Personal protective equipment
Respiratory Hygiene
Patient placement

Selecting Respiratory Hygiene executes the following service, :

```
http://foo.com/learningobjects/respiratory/md00123
```

And returns the following metadata:

Title: Respiratory Hygiene
Keywords: standard precautions, cough, cough etiquette
Audience category: professional
Educational objective: By the end of this lesson, students should be able to identify standard precautions that protect healthcare providers and patients against the spread of respiratory illness.
Location: http://www.foo.com/respiratoryhygiene.html

A note for ATOM: the ATOM format provides a means for the server to specify navigation links as part of the metadata. For services whose XML payloads only support the ATOM format, navigation links may be embedded within the ATOM payload.

11. HTTP Caching

One benefit of RESTful design is leveraging HTTP caching. Caching can be used to reduce server-side computation, as well as reduce the amount of traffic to be sent over the network. Caching is achieved through communication to both HTTP proxies (which typically do the caching) and end clients using HTTP headers. Each REST service design should consider potential caching of REST responses to improve performance.

Caching is a complicated topic and reading the RFC 2616 [RFC 2616] is the best way to truly understand caching. However, the following is a high level overview:

- **Last-Modified and If-Modified-Since**: When a response is returned, the server indicates the modification time of the document and can include cache directives which indicate how long the document is to be cached. Granularity in times are on the order of seconds. When the client submits a subsequent request, it includes the last known modification time in a conditional If-Modified-Since header. That determines whether or not the data can be served out of the cache. If no cache is present, it might even be used at the server side to determine whether any further computation is necessary.

- **ETags and If-None-Match**: ETags can be used for both caching and reducing network overhead. ETags allow for exchange of content only once it has been changed at the server side. The basic scheme is that the client first requests a resource and discovers the ETag. In subsequent GET requests, it includes the last known ETag in an 'If-None-Match' header. If a caching proxy or the server has determined that the cached value is still valid or that the content hasn't changed (this last part is the network optimization bit), then a 304 response is returned to the client indicating that it still has valid content. This also cuts down on the

amount of data that needs to be exchanged. Proxies can keep track of the validity of these ETag cache keys; this scheme only works if the ETag is a strong validator, meaning that it changes when the content changes. The HTTP 'Vary' header can be used to direct the cache to use the ETag as part of its cache key. This can be useful for globally unique ETags, where the returned response might be unique per client for the same dynamic HTTP resource.

12. REST APIs and Versioning

A good practice is to version your REST API by including a version number as part of the base URI for your REST services. Using the example above, it would be better if the base URI was:

```
http://foo.com/myservices/v1
```

So, the accounts listing would be:

```
http://foo.com/myservices/v1/accounts
```

This prefix allows for handling radical changes to the REST API set that would break compatibility with existing REST clients; the base URI version number can be incremented. Clients can also inspect the URI to determine whether they understand the URI through the version number.

Version numbers do not need to be incremented provided that additions to the REST API do not break backwards compatibility. Additions to the API should always strive to avoiding having to increment the version number. A change is considered backwards compatible if:

- It adds new REST resource relationship to the resource model, without affecting existing navigation
- It adds additive information to REST representations that won't affect existing clients
- It adds a new MIME type / representation support. Clients can request explicit representations by use of the Accept header (and it is good practice to do so for client compatibility reasons).

.

13. URI Length Concerns

GET URIs with complex queries may result in long query strings. The best practice is to keep URIs to a length that browsers can handle. In case URI length is a concern, the client should detect this condition and change the GET request to a POST request. In the body of the POST request will be the query parameters (without the '?'), with the content type set to:

```
application/x-www-form-urlencoded
```

URL form encoding is discussed on Wikipedia in the article Query String [Query String]. REST services should still treat this as a logical GET. If query string size is considered an issue, then REST resource design must be able to clearly differentiate between a query that might result in GET/POST behavior vs. a POST create request. REST clients should also consider this a logical GET action, with the decision to switch to a POST style request being a transport-level decision.

14. Security and Authentication

REST services can be secured and its users authenticated in the same manner that web pages are secured and web users authenticated. If you are implementing REST services and have security concerns, transport REST over HTTPS. For more information on security, see Knowing When to REST: Simple Object Access Protocol vs. Representational State Transfer [MedBiq2].

15. Firewall Concerns

Some firewalls do not allow HTTP PUT/DELETE traffic to traverse the firewall, as it is considered dangerous for

regular web content that may not be properly secured. A standard alternative is available: PUT and DELETE requests can be 'tunneled' through a POST using the **X-Method-Override** or **X-HTTP-Method-Override** HTTP headers. The two headers are synonymous; check for the alias if the first isn't present.

```
X-Method-Override: PUT
```

All REST services should support this variant of specifying the HTTP request method.

16. HTTP Error Handling

The HTTP RFC2616 provides guidelines on what error codes to return and the significance of those error codes. It should be considered the ultimate guide for choosing the appropriate error to return for a given service. The following table is a guideline for more common situations:

Error Code	When it Applies	Example
400 - Bad Request	The input doesn't match the expected format or failed validation.	Key data was missing in a create or update request.
401 - Unauthorized	The client is not authorized to perform the RESTful action.	The client has requested a resource that the user is not authorized to see. An alternative is attempting an update for which the user does not have permission.
404 - Not Found	The RESTful URI does not match any resource within the system.	An invalid resource ID was passed in as part of a dynamic URL.
405 - Method Not Allowed	The REST service does not support the operation implied by the HTTP method used	Performing a PUT on a resource that is read only.
500 - Internal Server Error	Signifies that an error occurred during processing that does not relate to any input provided by the client. This error should not arise as part of normal operating conditions.	Connectivity to a database was interrupted.

17. References

[Fielding]
Feilding R, 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral

dissertation, University of California, Irvine.

[ISO 8601]
International Organization for Standardization: Numeric Representation of Dates and Time. Accessed August 11, 2009:
http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/date_and_time_format.htm

[JSON]
Introducing JSON. http://www.json.org/ Accessed August 11, 2009.

[MedBiq1]
MedBiquitous, 2009. MedBiquitous Web Services Design Guidelines ver 2.0. Accessed May 27, 2009:
http://www.medbiq.org/std_specs/techguidelines/webservicesguidelines.pdf

[MedBiq2]
Knowing When to REST: Simple Object Access protocol vs. Representational State Transfer, Accessed October 28, 2009. http://www.medbiq.org/std_specs/techguidelines/knowingwhentorest.pdf

[Query String]
Query String, Wikipedia. Accessed August 11, 2009. http://en.wikipedia.org/wiki/Query_string

[Ray]
Ray R.J., Kulchenko P., 2002. Programming Web Services with Perl. O'Reilly: Sebastapol, CA.

[RFC 2616]
Fielding, R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T. Hypertext Transfer Protocol – HTTP 1.1. Access August 11, 2009.

[RFC 3339]
Klyne G., Newman C., 2002. RFC 3339: Date and Tie on the Internet: Timstamps. Accessed August 11, 2009.
http://www.ietf.org/rfc/rfc3339.txt

[RFC 5023]
Gregorio J., de hOra B., 2007. RFC 5023: The Atom Publishing Protocol. IETF Trust. Accessed May 27, 2009:
http://www.ietf.org/rfc/rfc5023.txt

[Snell1]
Snell J., Tidwell D., and Kulchenko P., 2002. Programming Web Services with SOAP. O'Reilly: Sebastapol, CA

[Snell2]
Snell J., 2004. Resource-oriented vs. Activity-oriented Web services. Accessed May 26, 2009:
http://www.ibm.com/developerworks/webservices/library/ws-restvsoap/