Application Programming Interface Architecture

Version: 2.0
Date: September 12, 2018
Author: MedBiquitous Technical Steering Committee
Contact: vsmothers@jhmi.edu

Version History

Version No.	Date	Changed By	Changes Made
1.0	10 Oct 2016		Version 1.0
2.0	12 Sept 2018		Version 2.0 • Allow HTTP Patch • Refine rules for support of older versions • Handling advanced API requirements

MedBiquitous Standards Public License and Terms of Use

MedBiquitous Standards (including schemas, specifications, guidelines, sample documents, sample code, Web services description files, and related items) are 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 Standards for any use and without any fee or royalty, provided that you include the following on ALL copies of the MedBiquitous Standards 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 release] MedBiquitous Consortium. All Rights Reserved. http://www.medbiq.org"
2. Notice of any changes or modification to MedBiquitous Standards 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 Standards, 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, GUIDELINES, 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 STANDARDS 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 STANDARDS 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 STANDARDS.

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 STANDARDS 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 STANDARDS.

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

The name and trademarks of the MedBiquitous Consortium and its members may NOT be used in advertising or publicity pertaining to MedBiquitous Standards without specific, prior written permission. Title to copyright in MedBiquitous Standards and any associated documentation will at all times remain with the copyright holders.

Table of Contents

Acknowledgements

The following members of the Technical Steering Committee contributed to this architecture document.

- Joel Farrell, Chair
- Dan Rehak, Learning Technologies Architect
- Prasad Chodavarapu, American Board of Family Medicine
- James Fiore, American Board of Surgery
- Steve Kenney, American College of Surgeons
- Scott Kroyer, lumināt
- Andy Rabin, CECity
- Emmanouil Skoufos, Elsevier Clinical Solutions
- Valerie Smothers
- Luke Woodham, St. George's University of London

1 Introduction

This document defines the MedBiquitous Application Programming Interface (API) architecture, a set of definitions, structures, relationships and rules to be used by working groups when they define MedBiquitous standard APIs. The objective of the API architecture is to assure that working groups specify APIs that are light-weight, easy to implement, secure and extensible. The architecture also tries to assure that all MedBiquitous API specifications are consistent with each other. Therefore, the primary audience of this document is MedBiquitous Working Groups with API implementers being a secondary audience.

The architecture defines how programmatic interactions are to be made between member organizations via internet protocols. As such, the architecture is based on Representational State Transfer (REST) sending and receiving requests and responses whose payloads are encoded in Extensible Markup Language (XML) or JavaScript Object Notation (JSON).

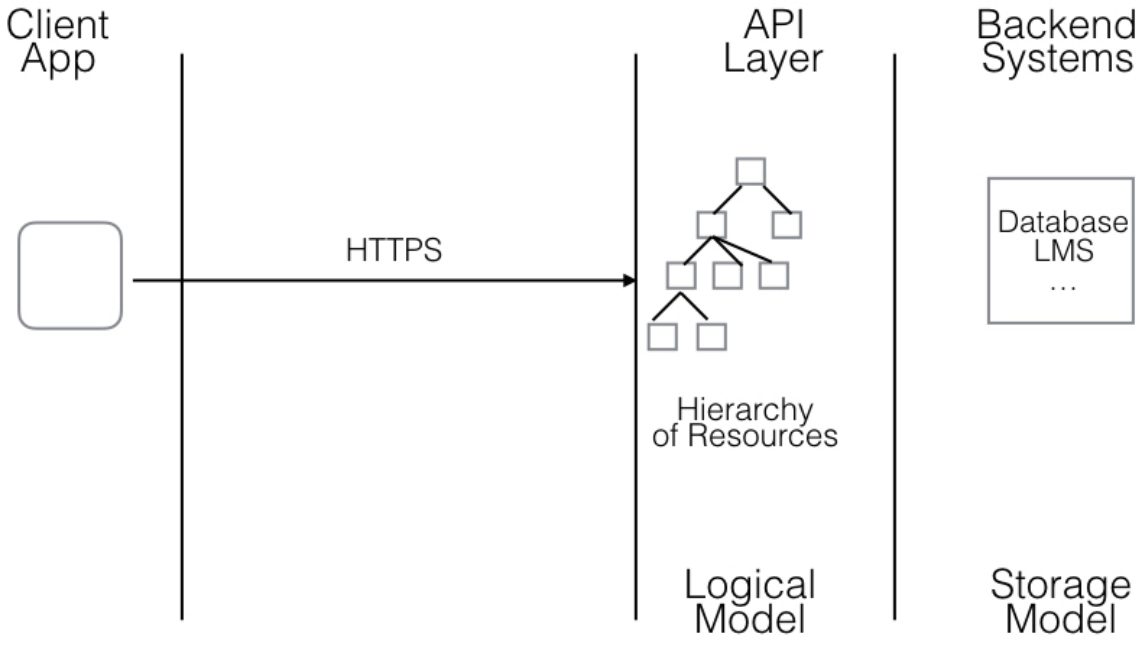

Figure 1. API Relationships

The API exposes information as a hierarchy of resources that forms a logical model of objects managed by a MedBiquitous compliant organization. This logical model is implemented at the Web server or Web application server layer. The physical storage and management of this logical model is provided by back-end systems and implemented in anyway the organization chooses. An implementation of the API uses services provided by the backend systems, but these systems should not show through the API

Layer. Interactions between a client application and the API are via standard Web protocols using standard Web security models.

2 Service Definitions

2.1 REST Concepts

All APIs will be implemented according to the REST model. In this model, the API uses HTTP methods to define operations on a set of resources. These resources are exposed via hierarchical URI's much like a directory tree. REST uses HTTP methods according to their protocol definition. These methods correspond to a simple set of standard operations as follows:

HTTP POST - Create a new resource

HTTP GET - Retrieve a copy of a resource

HTTP PUT - Update a resource

HTTP DELETE - Delete a resource

Other HTTP methods can be used if needed, such as HTTP PATCH for partial updates, but MedBiquitous API's should, if at all possible, stay with these four methods that correspond to a pure and easily used REST implementation. Resources can be any of the objects normally specified in the MedBiquitous specifications, such as Heath Care Professionals, Learners, Credentials, or Competencies.

All interactions are stateless. There is no concept of a session between the two parties in the API interaction.

In keeping with the security concepts below, all API's will require HTTPS (Secure HTTP). This generally implies an HTTP connection over the Secure Sockets Layer (SSL). All MedBiquitous REST services must follow the REST Web Service Design Guidelines [MedBiq REST] previously published

2.2 MedBiquitous REST Payload Content Types

The input and output payloads may support both XML and JSON encodings although a particular implementation does not have to support both. The content type requested must be included in the HTTP header. All services must support content negotiation to respect this HTTP content type. If the requested content type is not supported, return "type not available" (406 - Not Acceptable).

MedBiquitous uses two content types for its use of XML and JSON. They are:

For XML Payloads - application/xml or text/xml

For JSON Payloads - application/json

For example, to request an XML response include the following in the HTTP header:

```
Accept: application/xml
```

If the client includes no Accept-headers, the API will default to application/json. If the client specifies both application/json and application/xml in the header, the API will also default to application/json.

This document makes the distinction between data models, which define information, relationships, possible values, and characteristics of some real world entity; and payloads which define the specific data that is exchanged in an API call. For example, the Professional Profile is a data model of a Healthcare Professional. The data returned when querying a particular professional's certification would be a payload. MedBiquitous will not create data models in both JSON and XML. Current XML document standards define the current data models. However, when working groups define new data models (for example, when working on a new specification), the ability to support JSON and XML payloads in REST APIs must be considered.

2.3 Design Principles

MedBiquitous APIs must adhere to the following design principles:

- APIs must be as specific and granular as possible. Input or output payloads should correspond to types and vocabularies defined in MedBiquitous document standards. Payloads do not have to be proper document-fragments of full MedBiquitous document. However, if a document-fragment (or its JSON equivalent) contains the needed information, it should be used as is, instead of inventing a new representation.
- Related to the above, new MedBiquitous Document Standards should be easily decomposable into document-fragments that could be appropriate to an API payload. A decomposable XML document is one in which each document element that includes sub-elements could be used as a stand-along document to describe some resource.
- Free text should be avoided in payload elements if they must be programmatically interpreted. Free text is difficult for an application to interpret. Unless the free text is intended for subsequent display or human inspection, a controlled vocabulary should be used. Such a vocabulary specifies all the possible values of a data element. If the vocabulary changes so quickly that maintenance of the controlled vocabulary is not practical, use of free text might be a fall back solution. Consult the TSC in this case.
- HTTP POST and PUT (Create and Update) operations must return the data that will actually be stored by the service. This might differ from the input payload due to the assignment of Identifiers, URI's or other representational considerations.
- All HTTP GET (Read) and Query operations must be idempotent, that is, they cannot cause any changes to the resources either directly or through side effects.
- MedBiquitous APIs are not required to support transactional updates, However, they must define what happens when the same resource is updated by two REST requests that are received concurrently.
- Batch services, those that update a set of resources rather than an individual one, are not included in this architecture. REST services defined by MedBiquitous APIs should create or update a single resource, although they may return a list of resources via a read operation. For situations where the scale of the data is too large for such a single-update-at-a-time process,

alternatives outside of the REST API can be employed. For example, a curriculum inventory for a medical school contains millions of records. In such a case, a separate "batch file" can be defined (based on MedBiquitous standards) that can be uploaded via some alternative mechanism, including an FTP upload. This is analogous to the situation in relation databases in which the API (the SQL language) is used for programmatic interactions, but the process of loading large sets of data into the database is done via a separate procedure, namely Extract, Transform and Load (ETL).

- This architecture does not define how a MedBiquitous API should address advanced and highly sophisticated REST API requirements. Such requirements are addressed by new and emerging standards, including OData from OASIS, and JSONAPI. At this point, though both specifications cover a broad range of requirements and each has a set of tooling available to support it, such tooling is not mature across the languages and platforms needed by the MedBiquitous community. MedBiquitous APIs should try to avoid complex constructs as much as possible. Two advanced requirements will likely need to be addressed, so for these the architecture gives the following guidance.
    - Output paging (handling large returned lists of objects by accepting only a subset and then requesting the next "page" of results) can be supported if the expected size of the output would be difficult to process or cause unacceptable response time. Keep in mind that paging can cause "chatty" interactions that may consume network capacity. If needed, paging should be implemented using a client-side approach in which the API request includes a limit on the number of objects returned. Subsequent calls can obtain another page of output by specifying the limit and the logical index to the place in the output that will be the start of the page (or the number of objects at the beginning of output to skip). Note that if the resource being queried is dynamic, the pages of output may not be consistent.
    - Output filtering can be included, but should avoid advanced string matching and filter hierarchies.

3 API Signatures and Resource URIs

3.1 URLs

All REST API service URLs should be of the form:

```
{host}/medbiq/api/{api-name}/…
```

where {host} is the top-level URL of the provider, plus the path to the API service, and {api-name} is the MedBiquitous standardized service. For example, for a Professional Profile service, the URL would look like:

```
{host}/medbiq/api/pprofile
```

This example URL references an aggregate resource, the list of all professional profiles managed by this API provider. If a path to such an aggregate is accessed, the API should return a list of resources, preferably with each element in the list containing the URL that can be used to access the corresponding resource. If security rules do not allow the full list to be return to a particular client, the API can return an authorization error and return and empty list or a list of only those for which the client has authorization. To operate on a particular profile, say for Dr. John Doe, reference the URL

```
{host}/medbiq/api/pprofile/johndoe
```

Note that although we use profiles as an easy to understand example, an actual professional profile API might use much different path components.

3.2 Persistent URI's

This architecture does not define an approach for Persistent URIs (URI's that are guaranteed to be valid indefinitely or at least for a long, predefined period of time). Such URI's are beginning to be proposed to represent reusable objects and for representing resources that describe long lasting entities. The MedBiquitous API architecture will simply use its API versioning approach to address URI stability, although it will not define a full Persistent URI approach. MedBiquitous working groups that are defining APIs based on an existing outside standard that employs a Persistent URI mechanism, should follow the approach defined by that standard. For example, the MedBiquitous profile for the Experience API (xAPI) should follow the approach defined by the xAPI specification.

4 Versioning

MedBiquitous API's must support a versioning strategy to maintain upward compatibility in the face of changes. An API implementer will support different versions of an API via a different URL for each version. The following rules apply.

1. A new version must be used with either the data model or the method definitions change. This implies that if the resources, the logical model represented by the resource hierarchy, or the semantics of the operations change, a new version is required. Some changes to the API are possible without forcing a new version, but these are not common. Working groups should consult the TSC to confirm these cases.

MedBiquitous must publish which versions of MedBiquitous API's are currently supported. All APIs must support at least the current version and the previous version. In the case where future revisions to the API specification are not expected for a lengthy period of time, if at all, the working group can decide on the appropriate time to cease requiring support for the previous version.

2.
3. Each API root path should include a version component that starts with v1 and is incremented for each version up to v[k] for 'k' versions. For example: https://webapi.test.org//medbiq/api/pprofile/v1 is Version 1 of the API
4. https://webapi.test.org//medbiq/api/pprofile/v2 is Version 2 of the API and is hosted on a different end point.
5. The working group responsible for the API defines when a new version number should be used.

5 Concurrency

This architecture defines a simple approach to handling concurrent requests from multiple client applications to a single resource. When a client updates a resource, it must first check to see if it is still operating on the most current version of that resource. This is done in a optimistic fashion using HTTP ETag headers when using POST/PUT/DELETE methods on the resource from the client to the server. The sequence flow of this interaction is described below for an update operation:

1. Client sends a request to the Server for a resource with a GET request.

2. Server responds with the requested resource along with a ETag header associated with the resource as below:

   ETag: "1AB2C3D4E5D6"

3. Resource is updated on the Client side and it wants to update the resource on the Server side and issues conditional PUT request to the resource URI with a HTTP header

   If-Match: "1AB2C3D4E5D6"

4. The server compares the value of the if-match request header to the value that represents the latest version of the resource. If the ETag value does not match, the server responds with a HTTP Error code 412 (Precondition Failed) response. If ETag matches, then the update operation is allowed to proceed.

Another way a client might be using a stale version of a resource is a result of caching. Most enterprise architectures include some sort of cache point such as a caching proxy. The ETag header approach can deal with this case as well as illustrated in the flow below.

1. Client sends a request to the Server for a resource with a GET request.

2. Server responds with the requested resource along with a ETag header associated with the resource to the client.

   ETag: "1AB2C3D4E5D6"

3. On subsequent requests, client issues a conditional GET request with the ETag of the resource it received from the Server

   If-None-Match: "1AB2C3D4E5D6"

4. Server verifies whether the ETag value supplied in the conditional GET request matches the latest ETag value for the requested resource. If there's a match, Server returns a 304-not modified response to the client.

5. Client on receiving not modified response sends the cached copy of the resource to the user.

For more information see the discussion of ETags in [HTTP1.1].

6 Binary Attachments

Some resources have associated with them a binary object such as an image or a PDF file. Logically this can be seen as a binary attachment concept in an API. However, MedBiquitous APIs will not use binary attachments, but rather will stay with pure rest concepts. The binary objects are simply objects that have their own positions in the API resource path hierarchy. The standard REST Create, Read, Update, and Delete operations can be performed on them. The path to a photo for a professional profile could be:

```
{host}/medbiq/api/pprofile/v1/johndoe/photo
```

The binary resource can also be referenced from within a resource, potentially from multiple places, via URL references. For example, within …pprofile/johndoe a reference to the photo could be via a fully resolved URL such as:

```
"Photo": "https://myhost/medbiq/api/pprofilev/1/johndoe/johndoephoto.jpg"
```

An API should allow the user to create or update a resource containing such a reference as a path reference such as pprofile/v1/johndoe/photo. Then the API would update the reference to be a fully resolved URL that can be directly used by the client.

Note that unlike attachments in SOAP Web services, the resource and its binary object cannot be added or updated as a composite unit. They must be added or updated in separate REST operations. This should not be a significant problem given the types of use cases currently projected. If a working group identifies a use case in which the lack of atomicity is problematic, it should consult the TSC.

7 HTTP Responses

REST Services must report error conditions by means of HTTP status codes. These response codes, particularly those in the 4XX range are used by the API provider to return error information. For particular error codes, additional information can be returned in the HTTP response body using a common MedBiquitous error information resource. Services can also return any of the normal response codes in the 2XX and 3XX range.

7.1 HTTP Redirection Codes

Two status codes in the 3XX range are particularly applicable to MedBiquitous APIs.

301 - Moved Permanently

Used for redirection when the location of a resource changes. This must be minimized, as discussed in section 3.2.

304 - Not Modified

This status code is used in conjunction with the use of ETags described in section 4. This indicates the operation can proceed since the resource has not changed on the server since the client previously read it.

7.2 HTTP Error Codes

The following HTTP error codes should be used for API specific situations. Other codes may be returned by Web infrastructure or security implementations.

400 - Bad Request

The client sent and invalid payload to service. This is the general HTTP error code to report application errors such as invalid syntax in the REST payload or syntactic or semantic problems that are not covered by other HTTP error codes.

403 - Forbidden

The client is not authorized to perform the requested operation on the resource. This error code should be used for authorization errors rather that “401 - Unauthorized” which the Web server will used to report authentication problems and issue authentication challenges.

404 – Not Found

The resource specified in the URL is not available or not found, but may be available in the future. This is not to be used if the resource has been deleted. Use 410 Gone, instead.

406 - Not Acceptable

The service cannot return content in the format requested by the client in the HTTP Accept headers it sent in the request.

409 - Conflict

The service could not update a resource due to a conflict with another request. This can happen if the resource version number (if an API uses version numbers) of the request does not match the version number of the resource on the server, implying that the resource has been updated since the client retrieved it.

410 - Gone

The resource does not exist. This can indicate that the resource has been deleted.

412 - Precondition Failed

ETags do not match. This is used to indicate that the resource the client is using is not the current version.

7.3 Common Error Resource

More specific information describing the error should be included in the HTTP response and included in the HTTP body. For error 400 especially, the service should identify the precise error in the request. This error information must be returned in a format(XML or JSON) consistent with the Accept Header. The common error resource is a subset of the OData V4 Error Response. If an API needs to return additional details, it can use the full Error Response defined there. The common error resource is defined via XML as follows.

```
<error>
  <code>code</Code>
  <target>path to error in input</target>
  <message>description</message>
</error>
```

Where:

code (int) is an API defined error code descriptive of the specific error. Exactly one ApiErrorCode element is allowed.

target(string) is the location of the error in the input. It must at least be the name of the object in error. If practical, this string should be the path to the element that is in error. The path is specified as an XPath expression for XML payloads or a JsonPointer for JSON payloads. The API should restrict Path expressions to element names and repetition indicators. In this way, the corresponding JsonPointer would be the same (other than namespace prefixes and the array element designation). For example, to indicate that the second department within an institution the Path would be:

```
<Location>/Institution/department[2]<Location>
```

while the JsonPointer would be

```
<Location>/Institution/2/department<Location>
```

Zero or one Location element is allowed.

message (string) is descriptive text meant to be read by a human, not processed by the client application. It can be used to display an error message or to add text to an error log. Zero or one message element is allowed

Additional elements are also allowed as needed by the API, but they should not conflict with the OData Error Response.

8 Security

All interactions must that exchange personal information must be secure. At a minimum, this requires use of HTTPS and basic authorization (encrypted Userid and Password). Working Groups can specify higher levels of security for particular services and implementers can enforce higher security if needed. But HTTPS with basic authorization is the minimum requirement. No exception is made for public, non-confidential APIs due to the need for data integrity.

An authorization scheme must be implemented to assure that requestors have appropriate credentials and authorization to read or update personal information. This architecture does not require a particular authorization scheme or technology.

Single sign-on scenarios should conform to the MedBiquitous Single Sign-on Guidelines.

Other server-wide or enterprise-wide security solutions monitor and take action to address attacks. Implementers should follow common best practices to avoid exposing avenues of attack in their implementations.

8.1 Digital signatures

Digital Signatures can be included in payloads. When using digital signatures, consider the purpose of the signature: integrity of the message content, authentication of the sender, or non-repudiation of the message being sent. If integrity is the goal, determine if the integrity of the message content in isolation from the header details, including the http verb and resource URI, are sufficient (see Lascelle, RESTful Web Services and Signatures). For XML, use the W3C XML Digital Signature standard. The W3C XML Digital Signature will effectively provide assurance of the integrity of message content.

For JSON, consider JSON Web Signature JSON Serialization (JWS-JS) http://self-issued.info/docs/draft-jones-json-web-signature-json-serialization-01.html. JWS-JS also provides assurance of the integrity of the message content and header details, authentication of the sender, and non-repudiation of the message being sent.

JWS-JS incorporates keyed-hash message authentication codes (HMACs), which can create a digital signature based on a unique key and string consisting of a number of elements from the service call or response, including the verb and the resource URI. Some Amazon services and the Windows Azure platform use an HMAC-based approach. In order to use an HMAC-based approach, there must be a shared HMAC available only to a single consumer and the service provider. See RFC 2104 and RFC 6151 for more information on HMAC.

9 Sample API

This example illustrates how to specify an API based on a simplified use case involving the MedBiquitous Professional Profile standard. Its purpose is to show how to specify REST Services, service payloads and error information. The simple API for Certification consists of a single service to find if a professional is certified by a particular board.

9.1 Certification API (Sample)

The Certification API exposes information about the certification of a healthcare professional via two services. The first provides a professional's certification status in a particular specialty and then provides certification for all specialties for which the professional has certification information.

9.1.1 CheckCertification

This service returns the status of the certification of a particular healthcare professional relative to a particular certification board.

Verb: `GET`

Path: `Member/CertificationInfo/organization/id`

Where:

organization is the community that issues a unique id

id is the Unique ID of the professional whose certification info is requested

Output Body:

Elements	Description	Required	Multiplicity	Datatype
CertificationInfo	Input Container	Required	1	Container
CertificationBoard	Name of Certification board to be checked	Required	1	Non-null String
UniqueID	Container for Professional whose certification is to be checked	Required	1	Container
Domain	Community that is the source of the ID	Required	1	Non-null String

Elements	Description	Required	Multiplicity	Datatype
ID	Unique ID of the professional	Required	1	Non-null String
CertficationStatus	Empty on input. On output contains one of Active, Expired, Revoked, Suspended, Surrendered	Required	1	Restricted

Example Output

XML:

```
<CertificationInfo>
   <CertificationBoard>American Board of Surgery</CertificationBoard>
   <UniqueID
      <Domain>American College of Surgeons</domain>
      <ID>21556222</ID>
   </UniqueID>
   <CertificateStatus>Active</CertificateStatus>
</CertificationInfo>
```

JSON:

```
{
   "CertificationInfo": {
      "CertificationBoard": "American Board of Surgery"
      "UniqueID": {
         "Domain": "American College of Surgeons"
         "ID" : "21556222"
      }
      "CertificateStatus" : "Active"
   }
}
```

Errors

Certification Board Does is unknown
HTTP Error Code 400 - Bad Request

Certification Board Does is unknown

```
<ErrorResource>
     <ApiErrorCode>2</ApiErrorCode>
     <Location>path to error in input</Location>
     <ErrorString>The Certification Board Does in the input URI is
unknown.</ErrorString>
</ErrorResource>
```

Member id is not recognized

HTTP Error Code 400 - Bad Request

```
<ErrorResource>
     <ApiErrorCode>4</ApiErrorCode>
     <Location>path to error in input</Location>
     <ErrorString>The member identifier in the input URI is not
recognized.  The ID might be malformed or the member may not known by this
board.</ErrorString>
</ErrorResource>
```

10 References

[Archer] Archer, P. (2013, June 24). Study on Persistent URIs. Retrieved December 22, 2015, from http://philarcher.org/diary/2013/uripersistence/

[ADL Vocabulary] Advanced Distributed Learning. Companion Specification for xAPI Vocabularies. Available at https://adl.gitbooks.io/companion-specification-for-xapi-vocabularies/content/index.html

[ADL IRI] Advanced Distributed Learning (2015). Guidelines for IRI Design and Persistence. Available at https://docs.google.com/document/d/1RavkVwdzWQNszMXs8DMEth0bayAZRGBWWlSYVJtnC7M

[HTTP1.1] Hypertext Transfer Protocol -- HTTP/1.1, Section 14.24, If-Match,

https://tools.ietf.org/html/rfc2616#section-14.24

[MeBiq REST] REST Web Service Design Guidelines, http://www.medbiq.org/std_specs/techguidelines/RESTguidelines.pdf

[ODataV4] OData Version 4, 2017, http://www.odata.org/documentation/

[RFC 2104] RFC 2104, HMAC: Keyed-Hashing for Message Authentication, IETF, February 1997. https://www.rfc-editor.org/rfc/rfc2104.txt

[RFC 6151] RFC 6151, Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, IETF, March 2011. https://www.rfc-editor.org/rfc/rfc6151.txt