|  |
| --- |
| API Design Guidelines |
| RESTful API Design Introduction and Best Practices |
| SOA Enterprise Services |

# Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Revision | Description | Author(s) | Contributors(s) |
| 2/10/15 | 1 | Initial Draft | Manoj Parimi |  |
| 2/12/15 | 2 | Inputs | Manoj Parimi | Lance Gothard  Simon Hoare  Steve Longheinrich |
| 2/16/15 | 3 | Added Section details | Manoj Parimi | Lance Gothard |
| 5/18/15 | 4 | Updated “Standardize Names and Descriptors” section | Manoj Parimi | Greg Osborn  Alan Hong |
| 6/26/15 | 5 | Updates to Standard Query Parameters, Sorting, filtering, pagination and “Standardize Names and Descriptors” section; overall review | Manoj Parimi | Lance Gothard |
| 7/27/15 | 6 | Updates to Error Handling Payload format, HTTP headers | Manoj Parimi | Suman Kondakindi, API Team |
| 1/7/16 | 7 | Updated Pagination , Custom HTTP Headers | Manoj Parimi | Chris Jayaseelan |
| 3/31/16 | 8 | Added Custom Header for Target Source system | Manoj Parimi | Alan Hong |
| 6/6/16 | 9 | Added Custom Headers for Client Transaction ID, Client Session ID, Client Message ID | Manoj Parimi | Anthem.com Re-imagined Team |
| 8/10/2016 | 10 | Versioning strategy update | Manoj Parimi | Lance Gothard |
| 10/27/2017 | 11 | API URI Pattern | Alan Hong | Pankaj Koul |
| 2/23/2018 | 11.1 | meta-senderapp explanation | Alan Hong | Timothy Southerland  Naga Mahathi |

Table of Contents

[Revision History 1](#_Toc425885054)

[Purpose 3](#_Toc425885055)

[Scope 3](#_Toc425885056)

[Introduction to REST 3](#_Toc425885057)

[RESTful API Design 6](#_Toc425885058)

[Design Goals 6](#_Toc425885059)

[Design Methodology 6](#_Toc425885060)

[Resource Naming Guidelines 10](#_Toc425885061)

[Resource Identifier (URI) 12](#_Toc425885062)

[URI Components 13](#_Toc425885063)

[Representation 17](#_Toc425885064)

[Error Handling 19](#_Toc425885065)

[HTTP Methods 20](#_Toc425885066)

[HTTP Headers 21](#_Toc425885067)

[HTTP Status Codes 23](#_Toc425885068)

[API Feature Guidelines 25](#_Toc425885069)

[Security 26](#_Toc425885070)

[Versioning 27](#_Toc425885071)

[Pagination 29](#_Toc425885072)

[Filtering 30](#_Toc425885073)

[Sorting 31](#_Toc425885074)

[Search 32](#_Toc425885075)

[Select Payload 33](#_Toc425885076)

[Content Negotiation 34](#_Toc425885077)

[Hypermedia 35](#_Toc425885078)

[References 36](#_Toc425885079)

# Purpose

The purpose of this document is to put together a set of guidelines that will help design RESTful APIs for Anthem’s API Enablement Program.

# Scope

This document will cover the usage of REST principles and best practices to consider when designing APIs for the Enterprise. This document will be an evolving collection of guidelines and best practices and will incorporate feedback from vendors and consumers.

This document will address design concerns for Internal APIs initially and External APIs at a later phase.

# Introduction to REST

Representational State transfer (REST) is an architectural style for designing distributed hypermedia systems that are highly-cohesive and loosely coupled. REST is not a standard, but rather a set of constraints that have shown to be both flexible and adaptable to many types of services.

#### REST Constraints

|  |  |
| --- | --- |
| Constraint | Description |
| Client-Server | Separates the concerns of two primary components in a distributed system, which allows their implementations to evolve independently |
| Uniform interface | A set of four constraints that standardize the communication between web-based components   * Identification of resources * Manipulation of resources through representations * Self-describing messages * Hypermedia as the engine of application state |
| Layered System | Enables network-based intermediaries to sit between a client and server without compromising the uniform interface constraints |
| Cacheable | Enables network-based intermediaries to hold on to resource state representations, which helps web servers meet the demands of their clients |
| Statelessness | Restricts a web server from holding on to any client-specific state information, which helps it support more clients |
| Code-on-demand | Optionally allows a web server to transfer executable programs to its clients on an as-needed basis |

#### REST Data Elements

|  |  |
| --- | --- |
| Data Element | Description |
| Resource | The intended conceptual target of a hypertext reference |
| Resource identifier | URL, URI |
| Representation | HTML document, JSON, XML, etc. |
| Representation metadata | Media type |
| Resource metadata | Source links, alternatives |
| Control data | Cache-control, etc. |

#### HTTP Protocol

RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol. REST is not tied to HTTP, but most commonly associated with it. For our purposes we will be referring to REST as it relates to the HTTP protocol.

The current Hypertext Transfer Protocol standard is HTTP/1.1.

#### HTTP Request and Response

API clients can interact with APIs by sending HTTP message using different HTTP methods, request headers and an optional representation. The server, after processing, responds with a status code, headers and response representation in the body.

The request has four parts - HTTP method, URI, request headers and entity-body.

The response has three parts - HTTP status code, response headers and entity-body.

#### REST API Maturity Model

This section helps understand the REST API maturity levels using the Richardson Maturity Model. The model breaks down the principal elements of a REST approach into three steps. Level 0 represents the existing POX (plain old XML) based SOAP and XML-RPC messaging styles.



(Retrieved from martinfowler.com: http://martinfowler.com/articles/richardsonMaturityModel.html )

Level 1 tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources.

Level 2 introduces a standard set of verbs so that we handle similar situations in the same way, removing unnecessary variation.

Level 3 introduces discoverability, providing a way of making a protocol more self-documenting.

# RESTful API Design

## Design Goals

List of primary objectives and design principles that the recommendations provided in this document are based on.

1. Simplicity of interfaces
2. Easy to use API
3. Design for consumers of the API (Humans first, machines second)
4. Reuse building blocks from widely adopted standards

## Design Methodology

Follow these 7 steps to identify requirements and design an interface using REST principles.

1. Identify Resource
2. Identify State transitions
3. Standardize Names and Descriptors
4. Define Uniform Interface
5. Documentation
6. Implementation
7. Publication

Detailed description for each step.

#### Identify Resources

1. Review business objects in domain model
   1. List all names and descriptors for data in domain
   2. Entities, their identifiers, related entities
2. Organize domain elements into resources

#### Identify State transitions

1. This step help identify the hypermedia controls for Level 3 adoption
   1. If not doing Level 3, this step can be skipped
2. Flow through domain model, relationship to other domain
   1. Define entry points to the API, root resource
   2. Use state diagrams for understanding link relationships
3. Allow applications to handle changes gracefully
   1. Provide links and link relations
   2. A client cannot be expected to know what a resource is related to and where those relations are located
   3. The server describes these relations as part of its payload

#### Standardize Names and Descriptors

1. Reconcile with Enterprise Canonical Naming Standards
   1. Logical Name
      1. Define logical name for an attribute/element based on business requirements, along with definition.
      2. Check domain to see if existing attribute/elements fulfill the definition.
         * If exists, then use existing
         * If not exists, then finalize logical name and definition. Applying primitives, such as code, amount, etc. to the end as appropriate. (note: need to define what a primitive is)
   2. Element Name
      1. The [Standard Naming Descriptors document](#_Links) describes the naming conventions for common words found in attribute/element names. Please refer to this when applying the following rules.
      2. Rules for creating attribute/element name from logical name
         * Abbreviate last word in logical name if found on the primitive tab
         * Abbreviate any word(s) that is found on the common tab
         * If over 25 characters, then revisit logical name to see if different words could be used to express business intent.
           1. If unable to reword, then refer to the [Anthem Metadata Repository (AMR)](#_Links) to abbreviate one of the words in the logical name.
           2. Add to the Pre-Defined names tab in the Standard Naming Descriptors document.
   3. Abbreviations Guidelines
      1. Abbreviations may (not *must*) be used if combined words are more than 25 chars *and* resulting abbreviated term(s) are clear and easily interpreted (i.e., use common sense)
      2. Only use Abbreviations listed in the Anthem Metadata Repository
      3. Consistently use abbreviations for common terms, prefixes, suffixes etc., (refer to the list of descriptors for common terms above), Except for these terms, do not abbreviate if the combined words or names are less than 25 characters
2. Use Web Standards for interoperability
   1. Industry standard, widely adopted formats (such as recommended by IANA, Schema.org, microformats)
   2. Both for External and Internal API

#### Define Uniform Interface

For each resource

1. Identify Resource Representation
   1. Resources can be represented in different ways
      1. Media Type - JSON, XML, HTML, plain text, etc.
   2. Content types are negotiated using headers
      1. A client doesn't know what a server is going to send it
      2. A server doesn't know what a client can handle
      3. Client describe what it wants with Accept header
      4. Server (and client during POST and PUT) describes what it is sending with Content-Type header
   3. Define request and response payload
2. Identify Uniform Interface Methods
   1. Identify the subset of HTTP method the resource will support
3. Identify Headers
   1. Identify a subset of headers that need to be supported by both request and response
   2. Define Metadata that need to supported
   3. Define Control data that need to be supported
4. Identify Return Codes
   1. Identify the HTTP status codes that need to be support in the response
5. Identify API Features
   1. Security need to be supported
   2. Version need to be supported
   3. Identify other optional features to be supported

#### Documentation

Make documentation available on the API Management Platform

Document generation tools to consider

* Apigee SmartDocs
* Swagger
* Ignite

#### Implementation

Steps to consider for implementation of backend API and API Management Platform configuration

* Implement server side components of the API
* Identify and define NFR , SLA
* Enable API on the Management Platform
  + Create API Proxy
  + Apply Policy sets
  + Configure Security, Authorization
  + Create API Product
  + Register Consumer App
  + Register Developer/Team

#### Publication

Publish API with documentation to the developer portal and make it available to use

## Resource Naming Guidelines

A resource is the main abstraction of an API Design. A resource can be any “thing” on the server that can be accessed by a web interface. Below are the best practices and guidelines for naming resources.

#### Nouns for Resource names

A resource name should be a noun, not a verb.

#### Collection as Resource

The primary entities in a domain should be modelled as collections. Use plurals for maintaining consistency across the APIs.

##### Example

providers instead of provider for resource name

#### Compound Noun phrases

Compound noun phrases spanning more than two separate words may be represented by path hierarchy. If it cannot be represented in a hierarchy, it can be separated by using a hyphen (-).

##### Examples

/provider-claims

/providers/claims

#### Business Process as Resource names

Can be used for task services which are not basic CRUD operations. This resource might in turn utilize other resource to implement the business process.

This will help avoid business logic spill into client code.

This will also reduce API chattiness, by reducing the number of calls the client need to make to complete a task.

#### Domain Events as Resource names

Can be used to represent a controller that can process domain events which encompasses other resources.

#### Concrete names are better than abstract

Don't use models that can be interpreted differently

#### Naming Consistency

Maintain consistency in naming across the API portfolio. The below order can be used to determine priority in a conflicting situation.

1. Names need to be consistent with formatting guidelines
2. Names need to be consistent across the APIs
3. Names need to be consistent with Enterprise Canonical Model
4. Names need to be consistent with Web Standards

## Resource Identifier (URI)

A resource is identified by a Uniform Resource Identifier (URI). A URI uniquely identifies a web resource and makes it addressable.

#### Guidelines

* URI are used to designate resources
  + Do not use to designate operations on resources
* AURI identifies exactly one resource, but a resource may have more than one URI
* A resource must have at least one URI
* Use a strict lowercase for all names
  + Use a hyphen (-) to connect name phrases

#### Examples

GET /resource HTTP/1.1

Host: www.example.com

GET /resource/path HTTP/1.1

Host: www.example.com

GET /resource/path?param1=value1&param2=value2 HTTP/1.1

Host: www.example.com

### URI Components

Each URI consist of three parts, the hostname, path and query parameters.

#### Hostname

Resource owner, an Authority identified by hostname.

#### Scheme

###### Examples

https

http

#### Domain Name

###### Examples

api.anthem.com

domain.api.anthem.com

partners.api.anthem.com

#### Formatting

The below standard convention will be used for domain names

* All lowercase for domain and sub domain names
* Words and word phrases separated by hyphen (-)

#### Resource path

#### Guidelines

* Resource path and URIs should be descriptive, meaningful and well structured
  + A client should be able to construct the URI with less effort
* Avoid using variable part of the URI as the first path segment
* Using a trailing slash is not needed, the presence or absence of a trailing slash should mean the same
* The query names used in the path should be consistent across all API

#### Formatting

The resource path is case sensitive, the below standard will be used for all URL resource naming

* All lowercase
  + example /members
* Words and word phrases separated by hyphen (-)
  + example /member-data

#### URI Template Notation

Use {} to denote path variable in documentation

#### Best Practices

* Have a base URI for the resource and the resource collection

###### Example

/members

/members/{id}

* Limit to two base nodes

###### Example

/providers/{id}/claims

/providers/{id}/claims/{id}

* Use path variables to separate elements of a hierarchy

###### Example

/parent/child

* Use punctuation characters in path variables to separate elements of same hierarchy
  + Use semicolons when order is not important

###### Example

/parent/child1;child2

* + Use commas when order is important

###### Example

/coordinates/x,y

* Use path variables for specifying mandatory elements though GET

###### Example

/claims/{startdate}/{enddate}

#### Query Parameters

#### Guidelines

* Use query variables to imply inputs into an algorithm
* Refer to API features guidelines for standardized query parameters

#### Formatting

The query parameter names are case sensitive, the below standard formatting will be used

* All lowercase
  + example ?name=abc
* Words and word phrases separated by hyphen (-)
  + example ?name=abc&alternate-name=xyz

#### Standard Query Parameters

List of standard query parameters to use across all APIs. These are reserved parameter names and cannot be used for other puprposes.

|  |  |
| --- | --- |
| Parameter Name | Used for |
| page | Pagination |
| size | Pagination |
| filter | Filtering |
| criteria | Filtering |
| fields | Partial Payload |
| view | Partial Payload |
| sort | Sorting |
| q | Search |
| apikey | Access Token |
| env | Environment |

#### Standard Delimiters

List of standard delimiter to use in query parameters across all APIs

* For separating **values** use a comma (,)
  + Example /entity?listofvalues=123,4356.00,63.20
  + An Exception to this is values with comma in them (ex: dollar amount) use [“value”,”value”]
  + Example /entity?listofvalues=[“123”,”40,356.00”,”63,000.20”]
* For separating **fields** use a pipe(|)
* For **key/value pairs** use a double-colon (::)

#### URI Design Pattern

URI should follow a standardized pattern described in the following pattern:

[**https://[server**](https://[server) **name]/[owner ID]/version/[consumer ID/SenderApp]/domain/[sub domain]/resource**

* For true enterprise API which is owned by SOA, **[owner ID]** should be omitted
* For true enterprise API which should be used by all consumers without further customization, the **[consumer ID]** should be omitted

## Representation

A resource can have many representations. A representation of the resource is rendered for the client consumption at the time the request is made.

#### Media Type

The media type of the data in a representation; it is indicated by the Content-Type header field. Most commonly used media type for APIs is application/json.

##### Guidelines

* Application-specific media types should be used
* Media type negotiation should be supported when multiple representations are available

##### Examples

Content-Type: text/html

Content-Type: application/json

Content-Type: application/xml; charset=UTF-8

##### Registered Media Types

List of registered media types to use

|  |  |
| --- | --- |
| Media Type | Description |
| application/json |  |
| application/xml |  |
| application/pdf |  |
| text/plain |  |
| text/html |  |
| image/png |  |
| application/atom+xml |  |
| application/javascript |  |
|  |  |

##### Vendor Specific Media Types

List of custom media types to use

|  |  |
| --- | --- |
| Media Type | Description |
|  |  |
|  |  |

#### Message Body Format

* JSON as default
  + JSON should be supported for resource representation
  + JSON must be well-formed
* XML and other formats may optionally be used for resource representation
* Avoid additional wrappers

#### Hypermedia Format

* A consistent form should be used to represent links
* A consistent form should be used to represent link relations

#### Descriptor Name Formatting

Standard Format for property(element or attribute) names

* Use JavaScript notation
  + camelCase
* lowercase is preferred in general

### Error Handling

Provide Error handling using HTTP status codes and response payload.

* Messages returned in the payload of the error should be as verbose as possible
* For unsuccessful calls return a body with error response payload
  + Should contain error codes and details that could be common errors or API specific
  + Include these elements
    - type – type of exception classified as error, warning or information. Expected values are E, W or I
    - code – existing error codes, system error codes, application error codes, business error codes
    - message – name or short description of the error
    - detail – details about the error; Include a suggested corrective action if applicable
* Map API error conditions to the subset of HTTP status codes listed in this document (refer to HTTP status codes section)
* A consistent form should be used to represent errors
* Consistent error types should be used for common error conditions

#### Payload format

{

"exceptions": [

{

"type": "E",

"code": "1001",

"message": "Name of the Error 1001",

"detail": "Detailed description of error 1001"

}, {

"type": "I",

"code": "1002",

"message": "Name of the Error 1002",

"detail": "Detailed description of error 1002"

}, {

"type": "W",

"code": "1003",

"message": "Name of the Error 1003",

"detail": "Detailed description of error 1003"

}

]

}

## HTTP Methods

Subset of HTTP Method available in the specification

|  |  |
| --- | --- |
| Method | Description |
| GET | Used to retrieve a representation of a resource’s state |
| POST | Used to create a new resource with in collection or execute a controller |
| PUT | Used to update an existing resource |
| DELETE | Used to remove a resource from its parent |

#### Idempotence

An operation should produce the same result if executed once or multiple times. GET, PUT and DELETE are considered to be idempotent. DELETE will not produce the same result on the second call as the resource will be removed on first call.

#### Safety

GET is considered a safe method. GET should never be used to manipulate a resource

#### Guidelines

* GET must be used to retrieve a representation of a resource
* POST must be used to create new resource in a collection
* GET and POST must not be used to tunnel other methods
* PUT must be used to update mutable resources
* DELETE must be used to remove a resource from its parent
* Use POST to execute controller resource, to trigger operations that cannot be mapped to one of the other HTTP method

## HTTP Headers

#### Standard HTTP Headers

There are 46 headers listed in the HTTP Standard. Link to standard HTTP headers is provided in the references. We will use a subset of headers in our API design as described in the below sections.

#### Standard Request Headers

##### Subset of request headers to use

|  |  |  |
| --- | --- | --- |
| Header Name | Purpose | Required/  Optional |
| Accept | Content-Types that are acceptable for the response | Optional |
| Authorization | Authentication credentials for HTTP authentication   |  |  | | --- | --- | | Basic | This header will be used to send client credentials.  The value will be a Base64 encoded client id and client secret.  Base64(client\_id:client\_secret) | | Bearer | OAuth 2.0 access token | | Required |
| Host | The domain name of the server, and the TCP port number on which the server is listening. | Required |
| Content-Length | The length of the request body in octets (8-bit bytes) | Required |
| Content-Type | The MIME type of the body of the request (used with POST and PUT requests) | Required |
| Date | The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)  Example:  Date: Tue, 18 Nov 2014 02:24:18 GMT | Required |
| Cache-Control | Used to specify directives that must be obeyed by all caching mechanisms along the request-response chain  Example:  Cache-Control: no-cache |  |
| If-Match | Only perform the action if the client supplied entity matches the same entity on the server. This is mainly for methods like PUT to only update a resource if it has not been modified since the user last updated it |  |
| If-None-Match | Allows a 304 Not Modified to be returned if content is unchanged, used with ETag |  |
| If-Modified-Since | Allows a 304 Not Modified to be returned if content is unchanged |  |

#### Standard Response Headers

##### Subset of response headers to use

|  |  |  |
| --- | --- | --- |
| Header Name | Purpose | Required/Optional |
| Content-Type | Identifies the entity body’s media type | Required |
| Content-Length | The size (in bytes) of the entity body | Required |
| Date | The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)  Example:  Date: Tue, 18 Nov 2014 02:24:18 GMT | Required |
| Last-Modified | The last modified date for the requested object (in "HTTP-date" format as defined by RFC 7231) Example:  Last-Modified: Tue, 18 Nov 2014 02:24:18 GMT |  |
| Cache-Control | Tells all caching mechanisms from server to client whether they may cache this object. A TTL-based caching value (in seconds)  Example:  Cache-Control: max-age=300 |  |
| Expires | Gives the date/time after which the response is considered stale (in "HTTP-date" format as defined by RFC 7231)  Example:  Expires: Tue, 18 Nov 2014 02:29:18 GMT |  |
| ETag | An identifier for a specific version of a resource |  |
| Location | Provides the URI of a resource |  |
| Pragma | Implementation-specific fields that may have various effects anywhere along the request-response chain |  |

#### Common Non-standard Headers

##### A list of most common non-standard headers that are widely adopted

|  |  |  |  |
| --- | --- | --- | --- |
| Header Name | Purpose | | Required/  Optional |
|  |  |  | |

#### Custom Headers

##### List of custom headers that will be used by Anthem

|  |  |  |
| --- | --- | --- |
| Header Name | Purpose | Required/  Optional |
| apikey | Populate with Consumer Key (also known as apikey, app key or client\_id) | Required |
| meta-transid | Client Transaction ID – identifies an individual transaction | Optional |
| meta-sessionid | Client Session ID – identifies a client user session | Optional |
| meta-messageid | Client Message ID – identifies the current call/message | Optional |
| meta-senderapp | Client App Identifier; originated at gateway for internal use only. When non-QA APIKey is used, meta-senderapp will be automatically populated with the SOA SenderApp that is issued and linked to APIKey by APIGEE. If QA APIKey is used, the default SenderApp will be WBS~QA, but it can be overwritten by testers with their own specific SenderApp. | Optional |
| meta-target-source | Specify a target source system | Optional |
| meta-src-envrmt | Specify the sub-environment on the target | Optional |
| meta-page-key | Caching key for pagination implementation | Optional |
| meta-page-limit | maximum number of items to return | Optional |
| meta-page-total-items | total number of items available for the query | Optional |
| meta-page-current | current page | Optional |
| meta-page-total | total number of pages | Optional |
| meta-page-more-data | more data indicator, to use when total items is not available | Optional |

#### Guidelines

* Content-Type is mandatory
* Content-Length should be used
* Caching should be encouraged
  + Cache-Control, Expires and Date headers should be used to encourage caching
  + Cache-Control, Expires and No-Cache headers should be used to discourage caching
* Location must be used to specify the URL of a newly created resource
* Last-Modified should be used in responses
* POST response may not be cacheable

## HTTP Status Codes

Every API response includes a status code

#### HTTP status code categories

|  |  |
| --- | --- |
| Category | Description |
| 1XX: Informational | Communicates transfer protocol-level information |
| 2XX: Success | Indicates that the client’s request was accepted successfully |
| 3XX: Redirection | Indicated that the client must take some additional action in order to complete their request |
| 4XX: Client Error | Indicates client error |
| 5XX: Server Error | The server takes responsibility for these error status codes |

#### Subset of standard HTTP Status codes to use

* 200 OK
  + Everything worked
* 201 Created
  + The server has successfully created a new resource
  + Newly created resource's location returned in the "Location" header
* 202 Accepted
  + The server has accepted the request, but it is not yet complete
  + A location to determine the request's current status can be returned in the "Location" header
* 204 No content
  + An element was successfully deleted from a collection but no content is returned on the delete.
  + Contrast this with 200 on a delete, typically the deleted element is included in the response to follow Java convention of deleting an element from a collection.
* 304 Not Modified
  + Indicates that the resource has not been modified since the version specified by the request headers If-Modified-Since or If-None-Match
  + This means that there is no need to retransmit the resource, since the client still has a previously-downloaded copy
* 400 Bad Request
  + Malformed syntax
  + Should not be repeated without modification
* 401 Unauthorized
  + Authentication is required
  + Includes a WWW-Authenticate header
* 403 Forbidden
  + Server has understood but refuses to honor the request
  + Should not be repeated without modification
* 404 Not Found
  + The server cannot find a resource matching a URI
* 406 Not Acceptable
  + The server can only return response entities that do not match the client's Accept header
* 409 Conflict
  + The resource is in a state that is in conflict with the request
  + Client should attempt to rectify the conflict and then resubmit the request
* 500 Internal Server Error
  + A generic error message, given when an unexpected condition is encountered
  + This should never occur and indicates a bug
* 503 Service Unavailable
  + The server is currently unavailable
  + Should only be used for maintenance
* 504 Gateway Timeout
  + The server was acting as a gateway or proxy and did not receive a timely response from the upstream server
  + Used if doing orchestration

#### Exceptions

* Need to support suppress-response-codes optional parameter so HTTP response is always 200 for things like Adobe Flash
  + If suppress-response-codes= true, Push the response code and error message to the representation

# API Feature Guidelines

The best APIs are predictable and consistent.

* Versioning, Response Codes, etc. should be predictable across API implementation, so developers moving from one set of APIs to another have an idea of what to expect.
* The URI structure and navigation should be intuitive so a developer can easily understand how it works without having to reference the documentation.

The following list of features ensures a level of consistency and predictability across all API’s developed within Anthem, while still allowing added functionality.

The list of features is to be evaluated for every API creation to ensure consistency across API implementations. Not every feature will apply, but when it does the guidelines for that feature should be applied in the design.

#### List of API Features

API features are discussed in detail below.

|  |  |
| --- | --- |
| Mandatory | Optional |
| [Security](#_Security) | [Pagination](#_Pagination) |
| [Versioning](#_Versioning) | [Searching](#_Search) |
|  | [Filtering](#_Filtering) |
|  | [Sorting](#_Sorting) |
|  | [Select payload](#_Select_Payload) |
|  | [Content negotiation](#_Content_Negotiation) |
|  | [Hypermedia](#_Linking_to_other) |

## Security

All APIs should implement security

(TODO: Incorporate Security elements from security architecture document.)

For the current phase, the API security will be implemented using client credential authentication. The client credentials will be provided in the HTTP headers for each API request. The HTTP headers are listed in the security headers section of this document.

## Versioning

Versioning should be supported in all APIs

#### Guidelines

* Mandatory
* Maintain at least one version back, but dependent on product that is being used by consumer
* Give X amount of time for clients to react before a version obsolete
* Use major version only, do not use minor versions in interface
* Only increment the version in situations where backward compatibility is broken
* For Consumer specific API update version for all changes

#### Different types

URI Based

(\*Preferred approach)

##### Example

/v1/resource/path/entity

/v2/resource/path/entity

* Pro
  + Easier to test and communicate
* Con
  + Code and documentation is more difficult to maintain
  + Can't evolve parts of the system independently
  + Caching complexities – a post to v2 may not show up in v1

##### Custom Request Header

##### Example

api-version: 1

api-version: 2

* Pro
  + Easy to communicate
* Con
  + Not a standard. HTTP Spec has this built in if you use the accept header
  + Not recommended

##### Accept Header/Content Type

##### Example

Accept: application/vnd.anthem.api.v1+json

Accept: application/vnd.anthem.api.v2+json

* Pro
  + Rich media types allow server to uniquely identify not only types, but versions of types
  + Fine granularity allows each type to evolve at different rates
  + Types can all be based on JSON/XML (+json, +xml)
* Con
  + Defining and document a different media types for every entity in your system can be time consuming
  + Offset by the ease in which clients can consume changes

## Pagination

Support for pagination using query parameters, allowing data subsets from collections. Implement additional functionality using metadata in headers.

#### Default limit

Limit default to 20 items per page, if pagination is implemented.

#### Scenario 1

##### Query Parameters

offset - beginning item number , starts at 0

limit - maximum number of items to return

##### Examples

Retrieve only first # items

/entity?limit=50

Use default limit, and offset

/entity?offset=20

Use both limit and offset values

/entity?offset=20&limit=10

##### Request Headers

Use along with query parameters or independently.

##### Metadata

meta-page-limit - maximum number of items to return

meta-page-key - caching key ; can also be the offset for next page

##### Examples

meta-page-limit: 10

meta-page-key: xyz001

##### Response Headers

As a convenience to user-agents the following headers should be populated so payload does not have to be read to implement paging functionality. Ex. JavaScript Slider

##### Metadata

meta-page-total-items – total number of items available for the query

meta-page-current – current page

meta-page-total – total number of pages

meta-page-more-data – more data indicator, to use when total items is not available

meta-page-key - use in case of a caching key need to be sent

##### Examples

meta-page-total-items: 100

meta-page-current: 1

meta-page-total: 10

meta-page-more-data: Y

meta-page-key: xyz001

#### Scenario 2

##### Query Parameters

page - current page being requested, could apply starting with the second request

size - maximum number of items to return

##### Examples

/entity?size=20

/entity?page=1

/entity?page=2&size=20

##### Response Body

##### Metadata

number - current page

size - number of items to returned

totalItems – total number of items available for the query

totalPages – total number of pages

##### Examples

"metadata": {

"page": {

"number": 1,

"size": 20,

"totalItems": 60,

"totalPages": 3

}

}

## Filtering

Support for filtering results using query parameters.

#### Using resource path as filtering mechanism

#### Using Query parameters for filtering

##### Query Parameters

filter – a set of field name/value pairs to apply filter the responses by

criteria – a set of field name/value pairs to search or query

##### Examples

/entity?filter=name::alan|state::CT,GA

/entity?criteria=sourcesystemid::aces|networkid::17,20,30|amounts::3000.01,2000.40

## Sorting

Functionality to sort response records in a specified order.

##### Query Parameters

To have your results sorted on a particular property, add a sort URL parameter with the name of the property you want to sort the results on. You can control the direction of the sort by adding a negative (- ) prefix to the property name.

###### Multiple sort elements

To sort the results by more than one property, keep adding as many property names as you need separated by a (|). The sort hierarchy is based on sequence the property names appear in the query string.

###### Sort order

When request sort, the default sort order on a property is ascending order, For reverse order or descending order note the (- ) prefix before distance in the below example.

* sort parameter **not present—** default sort behavior defined for the API
* sort parameter **present—** sort on property in ascending (or default)order
* Property with (- ) **prefix**— sort on this property in reverse order

###### Examples

1. Sort by indicator in default order and then by distance in default order.

/entity?sort=indicator|distance

1. Sort by indicator in ascending(or default) order and then by distance in descending (or reverse) order.

/entity?sort=indicator|-distance

## Search

Search for a specific resource using query parameters or search within a resource.

##### Query Parameters

##### Global search

##### Example

/search?q=fluffy+fur

##### Scoped search

##### Example

/owners/5678/dogs?q=fluffy+fur

##### Name/value search

Use query parameters on the collection resource to express search parameters

##### Example

/members?name=simon&state=Idaho

## Select Payload

Limit payload to requested fields supplied in the GET Query.

#### Field or view based

##### Query Parameters

view – a predefined view, or a subset of the full payload to be sent in response

fields – list of fieldnames to be sent in response, separated by a “|”.

###### Additional operators

* fields parameter **not present**— default payload as defined by the spec
* fields parameter is **present**— payload includes only the fields/objects listed
* field listed with **(-) prefix**— remove the object/field from the default payload
* field listed with **(+) prefix**— include the object/field along with the default payload

###### Examples

1. Return “summary” view of the payload as defined by the API specification

/members/3456789?view=summary

1. Return “details” object /view

/members/3456789?view=details

1. Return only specific fields— “id”, “name” and “address”

/members/3456789?fields=id|name|address

1. Return default payload but remove “address” object

/members/3456789?fields=-address

1. Return default payload and include “profile” object

/members/3456789?fields=+profile

## Content Negotiation

Support multiple formats for response media type provided in the request headers

* JSON (default), XML, HTML , plain text
* Headers describe the messages
* Use accept-headers to determine expected type in request

##### Examples

Accept: application/json

Content-Type: application/json

## Hypermedia

#### Links

Links should be included along with fields, within resource state representations. The required href value identifies the link’s target resource.

#### Link relations

A link’s rel value describes the relationship from the current resource to the resource specified by the link’s href attribute. rel values are "standardized" so client can recognize them

##### Examples

{ “rel”: “entity”, “href”: “localhost:8080/entity/0” }

#### Standard Formats

List of standard formats to consider

##### HAL (preferred)

Hypertext Application Language

HAL is a simple format that gives a consistent and easy way to hyperlink between resources in your API.

##### Collection+JSON

##### JSON-LD

##### Siren

# References

#### Links

* Anthem API Standardized Naming Dictionary
  + List of Descriptors for Common terms – [Standard Naming Descriptors](https://collaborate.wellpoint.com/sites/SOA_Strategy/API/Design%20Guidelines/Standard%20Naming%20Descriptors.xlsx)
  + List of Descriptors in Domain – TBD (for each domain or consolidated)
* Enterprise Metadata Repository
  + Anthem Metadata Repository (AMR) – http://www.wmr.wellpoint.com/amr/

#### Links (web)

* RFC2616 - Hypertext Transfer Protocol -- HTTP/1.1
  + http://www.w3.org/Protocols/rfc2616/rfc2616.html
* Hypertext Transfer Protocol (HTTP) Status Code Registry
  + http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
* IANA Media Types Registry
  + http://www.iana.org/assignments/media-types/media-types.xhtml
* Message Headers
  + http://www.iana.org/assignments/message-headers/message-headers.xhtml