AIA

API DEVELOPMENT STANDARDS

GTO
ENTERPRISE ARCHITECTURE AND GOVERNANCE

Document Owner: Wilson Shek
Revised on:20 Aug 2021
Status: v2.3

Revision History

Version	Issue Date	Summary of Changes	Author
0.1	22nd December 2017	Initial Draft	Wilson Shek
0.2	16th January 2018	Updated Section 4.1.2: Use suffixes to indicate functional types for schema attributes	Wilson Shek
0.3	17th January 2018	Fixed typos	Wilson Shek
0.4	21st February 2018	Added Section 6.3.3 on API Key vs ClientId/Secret	Wilson Shek
0.5	28th February 2018	Updated Section 3.3.3 on Error handling	Wilson Shek
0.6	15th June 2018	Update Section 3.2.4: added API group element in url format	Wilson Shek
1.0	26th June 2018	Added "Reviewed By" section	Wilson Shek
1.1	18th December 2018	• Section 3.2.3 – defining HTTP headers in lower case recommended • Section 3.2.4 – resource id in URL not recommend • Section 4.1.2 – naming convention for HTTP headers according to scopes	Wilson Shek
2.0	25rd January 2019	• Incorporated following changes based on feedback from HK Added sections • Parameters Types 3.2.3 • Method Design 5.0 • Guideline for resource modelling 2.2 • API management 8.0 • Schema Design 4.1.4 • Appendix 9.0 Updated • Security Section 7 • Return codes 3.3.2	Charles Ip
2.1	29th March 2019	• Added section 7.3.2 – Oauth scopes	Charles Ip
2.2	30th May 2019	• Updated HK standardization section	Charles Ip

2.3	20 August 2021	• Added examples for multi-level resources in Section 2 • Removed section 2.3 Guidelines on Resource Modeling • Restructured Section 3 • Added new Section 3.2.6 on Data Protection for Parameter Values on HTTP URL	Wilson Shek

Reviewed By

20th April 2018

BU	Representative	Contact
KR	Park, Woong	Woong.Park@aia.com
CN	Yang, Chris-XB	Rocky@aia.com
ID	Rocky	Chris-XB.Yang@aia.com
SG	Munuswamy Ramu, Senthil-K	Senthil-K.MunuswamyRamu@AIA.COM
SG	Ong, Andy-KT	Andy-KT.Ong@aia.com
SG	Veerappan, Senthil-Kumar	Senthil-Kumar.Veerappan@aia.com
PH	Delgado, Roger-C	Roger-C.Delgado@aia.com
TSS	Zhou, Wes-X	Wes-X.Zhou@aia.com
TSS	Li, Sarah-J	Sarah-J.Li@aia.com

Document Information

Master location	
File Name	
Reviewer(s)	
Approver(s)	
Distribution List	

Document Change Management and Distribution Procedure

Requests to change an approved version of this document must be submitted and handled in accordance with Document Change Management Procedure (to be attached).

New versions of this document are to be distributed to appropriate staff or made accessible via the process document repository for reference. Notification of a new version must be communicated to appropriate individuals via emails.

Release Cycle

In order to reflect the ever changing nature of AIA, to keep pace with evolving technologies, and so to remain relevant to AIA, this document will be reviewed and any amendments or additions will be released for approval on a quarterly basis

Table of Contents

1.0 Overview

This document defines the standards for modelling of RESTful resources and the developing of the corresponding RESTful APIs. While the standards are technical, they are platform/product agnostic and are applicable to all projects involving API development.

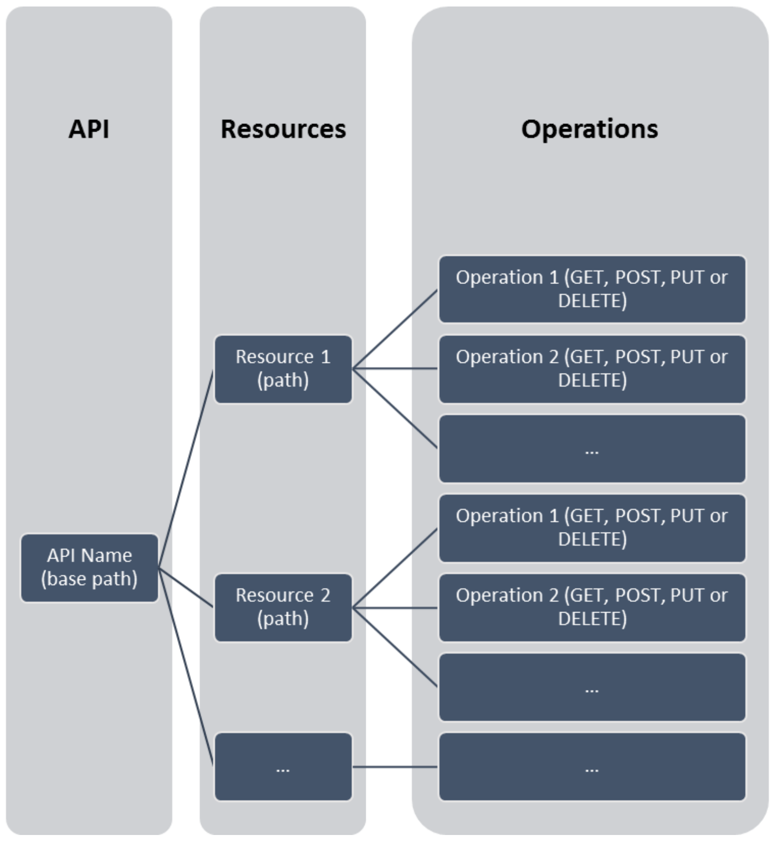
The target readers for this document are architects and API developers who should follow these standards in their API design and development work

2.0 Resource Model

- Each RESTful API consists of a number of resources which represent the fundamental concepts for the API.
- Each resource can be associated with a set of methods (or operations) which must correspond to the standard HTTP GET, POST, PUT and DELETE methods
  - A resource can be multi-level consisting of a top-level resource and a number of sub-resources
  - e.g. /policies/beneficiary
  - e.g. /policies/sum-assured/premium
- Each API should have a base path which represents the entry point for the API
  - Each resource within an API has a path relative to the API base path
  - The full path (URL) for a resource is formed by combining the API base path with the resource path
  - The URL and the HTTP method uniquely identify the operation on a resource
- A resource can be modelled as one of the following types:

Type	Description
Singleton	• When a resource is accessed as a singleton, the request must uniquely identify the single resource object to be returned in the response. • CRUD operations are applicable to a singleton resource
Collection	• When a resource is accessed as a collection, the response contains an aggregate of resources of the same type • CRUD operations are applicable to a resource collection
Controller	• A controller resource represents a functionality that cannot be mapped to one of the CRUD operations, e.g. validating a claim

The following diagram shows the high-level structure of an API based on the model described above:


API
Resources
Operations
API Name (base path)
Resource 1 (path)
Resource 2 (path)
...
Operation 1 (GET, POST, PUT or DELETE)
Operation 2 (GET, POST, PUT or DELETE)
...
Operation 1 (GET, POST, PUT or DELETE)
Operation 2 (GET, POST, PUT or DELETE)
...
...

2.1 Resource Naming & Grouping

- Resource names should be meaningful for consumers
- Design resources for your clients, not for your data
- Be consistent across resources
- Reference the AIA API Capability Model first, it should cover most of the common resources required to support the business

2.2 Fine Grained Resources vs. Coarse Grained Resources

When modelling resources, it is very important to model resources at the right granularity. This generally translates to how many levels or sub resources a resource will be allowed to use and ensuring that each sub-resource will have their own CRUD operations.

Having too many levels of sub-resources will start to create a need for the API consumer to understand the business logic around the resource, which creates a very chatty and tight coupling between API consumer and API provider. When CRUD operations are required on a sub-resource, it should also be modelled as a top-level resource in the API.

Having no sub-resources will introduce control information or directives into the interface to help the API provider to determine what to do with the request. This added logic will make it harder to maintain the API but there is a less chance of data inconsistencies. Changes will generally affect more consumers of coarse grain resourced than fine-grain resourced.

To strike a balance between these two extremes, an API can make use of resources that are based on the business process or actions. These are the resources that are described as controller type resources.

3.0 Resource Accessing

- The HTTP requests for accessing resources should conform to specifications defined in RFC 2616
    - Note that HTTP headers are case-insensitive according to RFC 2616

3.1 Sync vs Async

- An API operation should be synchronous, i.e. one call to an API operation corresponds to a request/response operation implemented in a single HTTP call
- Asynchronous API operations should be avoided except for the following scenarios:
    - Backend system is not able to provide a timely synchronous response for an API request. The response is sent back asynchronously in a second HTTP call from provider to consumer
    - API consumers are expecting notifications which are delivered through asynchronous callbacks.

3.2 HTTP Request

3.2.1 Components

HTTP Request Component	Description
HTTP Verb	• Each request should have only one HTTP verb which should be on of PUT, GET, POST or DELETE • See section below on HTTP Verbs for details
HTTP URL	• Each request is represented by one HTTP URL which is the entry point to a resource within an API • See section below on URL format for details
HTTP Request Headers	• Each request typically has at least one HTTP header • HTTP headers can include standard HTTP headers or custom headers specifically defined for AIA or a particular API resource • HTTP headers can be used for carrying the following information: ○ Authorization information that are common across endpoints ○ Meta-data that will be used to assist the routing of a request. Other business data parameters should be defined elsewhere. ○ Standard Tracking identifiers • See section below on Headers below for details
HTTP Body	• HTTP request body is applicable only to POST and PUT • Input parameters for an API can be included in the request body • GET and DELETE do not have HTTP body • See section below on HTTP body for more details

3.2.2 Verbs

HTTP Method	Description
GET	• For reading resources (singleton or collection) • GET requests never change the state of the resources
POST	• For creating new resources (singleton or collection) or • For a controller resource type (an operation on a resource that cannot be mapped to one of the CRUD operations)
PUT	• For updating existing resources (singleton or collection) • Implies state changes in the resources
DELETE	• For removing resources (singleton or collection) • Implies state changes the resources

- Use of other HTTP methods not listed in the table above is not recommended

3.2.3 URL Format

URL Component	Description
API Domain	• The API domain e.g.api.aia.com • A DNS domain that can be mapped to the server and port representing the actual address of the API • The server name and port number should not be present in the URL
API Group	• Currently defined groups: ○ il (Individual Life) ○ cs (Corporate Service) ○ vitality (Vitality)
API Name	• Name of the API
Version	• Version number with the format v[n] where [n] is a positive integer indicating the major version number for the APIs of this resource type • Note that is the version number on the resource type level and is not the version for an individual API method • Refer to Section below on API versioning for more details
Resource Path	• Path to the resource (can have multiple levels) • Should be plural for singular and collection types, e.g. customers • In the case of a controller type, this path represents the controller action, e.g. claims/validate
[Path parameter/Resource Id] (before the ? in URL)	• Optional Resource Id (also referred to as path parameter) • Used to reference a resource by identifier (Resource Id). When specified, it is referring to a singleton resource • Path parameters will always be mandatory if defined • Not to be used for carrying other information besides resource identifiers • While specifying resource id in URL is a valid syntax, as a best practice, it is not recommended for the following reasons ○ Resource Id in the URL clutters the URL and can be difficult to handle in coding. Path parameters are not well handled in some API frameworks or products ○ Resource id can include specials characters like “/” (slash) or “:” (colon) which can have unexpected behaviour across different API runtime platforms ○ If null is a valid value for a resource id, it is difficult to be represented as a parameter embedded in the URL Avoid using path parameters where null values are valid.
[Query Parameters] (after the ? in URL)	• Optional query parameters which can be used for filtering, sorting, paging and specifying partial response (See sections below on Query parameters for recommended usage) • If Resource Id is specified, there should be no query parameters

c

Other Rules for URL Format

- For resource names, use nouns not verbs

- All resources should be plural
- Avoid empty spaces in the URL.
- URL must not contain sensitive unencrypted data.
- Since URLs are case insensitive, do not rely on camel case in URLs, use hyphen instead

Examples

- Retrieving a collection of customers:

HTTP Verb and URL	GET https://api.aia.com/il/pos/v1/customers
API Domain	api.aia.com
API Group	/il
API Name	/pos
Version	/v1
Resource Path	/customers

- Retrieving a singleton customer (resource id in URL is not recommended, use query string or header to specify id instead)

HTTP Verb and URL	GET https://api.aia.com/il/pos/v1/customers/1234
API Domain	api.aia.com
API Group	/il
API Name	/pos
Version	/v1
Resource Path	/customers
[Resource Id]	/1234

HTTP Verb and URL	GET https://api.aia.com/il/pos/v1/customers?cust-id=1234
API Domain	api.aia.com
API Group	/il
API Name	/pos
Version	/v1
Resource Path	/customers
[Query Parameters]	?customer-id=1234

- Controller type

HTTP Verb and URL	POST https://api.aia.com/il/pos/v1/claims/validate
API Domain	api.aia.com
API Group	/il
API Name	/pos

Version	/v1
Resource Path	/claims/validate

3.2.4 HTTP Headers

- Although HTTP headers are case-insensitive according to RFC 2616, it is recommended that all headers should be defined in lower case in API specifications.
  - A HTTP request may be processed by network operators, such as Akamai, before reaching the API gateway. This pre-processing of the request, e.g. compression, may convert all headers to lower case
  - Defining HTTP headers in lower case in the API specifications can ensure consistency in the end-to-end processing of the request and avoid confusion.

- Definitions and usages of commonly used HTTP headers

HTTP Header	Description	GET	POST	PUT	DELETE
Accept Header	Representation format (e.g. application/json)	Mandatory	Mandatory	Mandatory	Mandatory
Authorization Header	Mandatory when bearer authentication (token authentication) is used	Mandatory	Mandatory	Mandatory	Mandatory
Accept-Language	The set of natural languages that are preferred as a response to the request.	Optional	N/A	N/A	N/A
Content-Type Header	Representation format of request body (e.g. application/json)	N/A	Mandatory	Mandatory	N/A
Accept-Encoding	List of acceptable encodings. (e.g. Accept-Encoding: gzip, deflate)	Mandatory	Mandatory	Mandatory	N/A

- Definitions of standard AIA HTTP headers to be used

HTTP Header	Description	GET	POST	PUT	DELETE
x-aia-request-id	Unique id for the request for tracking purposes	Recommended	Recommended	Recommended	Recommended
x-aia-correlation-id	Unique id used for tracking across API endpoints, if required (optional)	Optional	Optional	Optional	Optional
x-aia-lbu	Signifies the Local Business Unit	Optional	Optional	Optional	Optional
x-aia-env	Target environment of the request. Note that this header is required only when accessing non-production environments	Recommended	Recommended	Recommended	Recommended

3.2.5 HTTP Body

- Input parameters for an API can be included in the request body. Typically submitted as content-type: application/json with the body as a JSON. This style of request body parameter should be used by default for all new APIs.
- An API invoked directly by a traditional HTTP form post will use content-type: x-www-form-urlencoded with the body content encoded like query string parameters. This style of request parameters should not be used unless it is used to support an existing front-end application that cannot work with JSON request body parameters.
- For controller type resources, a common request wrapper can be used to enclose the payload to provide a consistent interface across APIs, but this is not mandatory by the standard. Refer to Appendix (Section 9.1.1) for an example.
- Not applicable for GET or DELETE method

3.2.6 Data Protection for Parameter Values on HTTP URL

- As described in Section 3.2.3 above, a HTTP URL can embed two types of parameters:
    - Path Parameter/Resource Id (before the ? in the URL)
    - Query Parameter (after the ? in the URL)
- The information carried by these parameters, e.g. email address, national id number, can be considered confidential and required security protection as specified in the Group Data Protection Standard (ITSR_036) or local regulations
    - ITSR_036 includes definitions for data classification as well as the required protection for the defined classifications
- It is generally not acceptable for a HTTP URL to include confidential information.
    - This is because an API request generally has to pass through different network components such as routers and servers and the HTTP URL for the request can be logged without proper data protection measures
- Passing these confidential parameters as HTTP headers is not recommended:
    - It is not a best practice to introduce custom API headers that are interface-specific as they are difficult to maintain when CORS (Cross-Origin Resource Sharing) is required
- Recommended approaches for passing confidential parameters to an API
    1. Put the parameters in the HTTP Body
        - Note that this approach will not work if the HTTP verb used is GET or DELETE as HTTP body is not allowed for these verbs
    2. Encrypt the parameter values and put them in query parameters
        - This implies that the API client has to be issued with a public key for encrypting the parameter values before calling the API and the API provider holds the private key for the decryption

3.2.7 Recommended Usage for Query String Parameters

Filtering

- Use a unique query parameter for each field to implement filtering.
    - e.g. GET /policies?type=auto,home&product-id=1590,1900
- Avoid empty spaces in the URL, with the exception of <Parameter Value> strings

Sorting

- Accommodate complex sorting requirements by letting the sort parameter take in list of comma-separated fields, each with a possible unary negative to imply descending sort order.

- Use special string query parameter: $sort,
    - e.g. GET /policies/<id>/operations?$sort=-date,+type

Paging

- API developer must provide a mechanism for paging through large amounts of data.
- Use special string query parameters: $limit and $offset
    - The $limit parameter indicates the maximum number of items to return.
    - The $offset parameter indicates the starting position of the query in relation to the complete set of unpaginated items.
    - If “offset” is not included in the query parameter and only “limit” is included, by default first page will be returned, otherwise page will be returned based on the given “offset”.
    - If “limit” is not included in the query parameter, default limit (configurable value) will be used to return the items
    - e.g. GET /policies?$offset=0&$limit=5
- Use custom header X-Total-Count to return the total number of records to be expected from the query

Partial Response

- Provide the API consumer with the ability to choose returned fields.
    - Mobile clients usually require a small list of attributes for display
    - This can reduce the network traffic and speed up the usage of the API.
- User special string query parameter $select
    - e.g. GET /policies?$select=id,commercial-type,code,subscriber

3.3 HTTP Response

3.3.1 Components

HTTP Request Component	Description
HTTP Status Code	• Each response should have one Status code • See following section on “Return Codes” for details
HTTP Response Headers	• Each response should have one or more response headers • Standard headers: e.g. Content-Type • Conditional header for caching • Meta-data and Standard tracking identifiers from the request should be returned if provided in request
HTTP Body	• In the case of a successful GET request, the body should contain the resource

	• In the case of an error, the body should contain the error message (see following section on "Error Handling" for details • For controller type resources, a common response wrapper can be used to enclose the payload, return messages and errors. This is to provide a consistent interface across APIs, but it is not mandated in the standard. Refer to Appendix (Section 9.1.1) for an example

3.3.2 Return Codes

- An API request must be responded with a standard HTTP response code appropriate for the result of the request.
- Return code of 2xx indicates the client's request was successfully received, and processed.
- Return code of 3xx indicates that further action by the user-agent (API consumer) is required to fulfil the request. Usually this is used for redirections, usage of 3xx return codes in API implementation is not expected for request/response APIs. For example, OAuth 2.0 authorization code grant type flow makes use of return code 301 to redirect the user-agent between the api client and the authentication server.
- Return code of 4xx are client related errors due to bad or incorrect data being requested or presented. API should map any client request related errors to a 4xx return code, and provide a detailed error response in the body using the AIA standard error JSON structure.
- Return code of 5xx are server related errors. API should map any backend errors to a 500 return code and provide a detailed error response in the body using the AIA standard error JSON structure.

Description for Specific Return Codes

HTTP Return Code	Name	Description
200	OK	Indicates general success and must not be used to communicate errors in the response body
201	Created	Indicates that a new resource has been created
202	Accepted	Sent by controllers to indicate the start of an asynchronous action
204	No Content	Indicates that the body has been intentionally left blank
301	Moved permanently	Indicates that a new permanent URL has been assigned to the client's requested resource
303	See other	Sent by controllers to return results that it considers optional
304	Not modified	Sent to preserve bandwidth (with conditional GET)
307	Temporary redirect	Indicates that a temporary URL has been assigned to the client's requested resource
400	Bad request	Indicates a general client error
401	Unauthorized	Sent when the client either provided invalid credentials or forgot to send them

403	Forbidden	Sent to deny access to a protected resource
404	Not found	Sent when the client tried to interact with a URL that the REST API could not map to a resource
405	Method not allowed	Sent when the client tried to interact using an unsupported http method
406	Not acceptable	Sent when the client tried to request data in an unsupported media type format
409	Conflict	Indicates that the client attempted to violate resource
412	Precondition failed	Tells the client that one of its preconditions was not met
415	Unsupported media type	Sent when the client submitted data in an unsupported media type format
500	Internal server error	Tells the client that the API is having problems of its own.
502	Bad Gateway	Service is down or being upgraded
503	Service Unavailable	Service is up, but overloaded with requests
504	Gateway Timeout	Servers are up, but the request couldn't be serviced due to some failure within our stack

3.3.3 Error Handling by API implementation

API implementation must cover the following 4 return codes

- 200 – OK for Success
- 400 – Bad Request with details returned in AIA standard error JSON structure.
- 401 – Unauthorized (for fine grain authorization errors)
- 500 – Internal Server error with details returned in AIA standard error JSON structure

The following 2xx and 4xx return codes can be implemented by API if required

- 201, 202, 204, 404, 403, 409, 412, 415

3.3.4 Error Handling provided by API Gateway

- Return Codes 404, 502, 503, 504 will be covered by the API gateway
- Gateway will cover Return Code 401 for coarse grain authorization (identity of application, permission to call API, validity of token)
- If APIs are exposed externally, return codes 400 and 500 will have its detailed error response mapped to generic ones to reduce exposure of sensitive details of the server implementation

3.3.5 Standard Error JSON Structure

- In an error condition, in addition to returning an appropriate HTTP return code to reflect the error category, a useful error message in a known consumable format should also be provided in the HTTP response body
- The error body should be in JSON format and include at the minimum the following:
    - a unique error code(reasonCode)
    - a useful error message (reason)
    - a detailed description of the error (message)
- Use a fixed top-level error message to indicate the error category, e.g. validation Error, and include a more detailed or precise description if available
- Where error codes are not used in API design, populate reasonCode with HTTP return code.
- Standardized error codes and format are intentionally excluded from the development standards to allow some flexibility depending on the context and purpose of the API. Eg. Front-end APIs versus system integration focused APIs would have different requirements.
- For reference, Hong Kong's standard for reason code can be found in the Appendix (section 9.1.2)
- API implementation can extend the JSON structure from the minimum standard structure to provide a more detailed error response structure if required

<u>Minimum Error JSON Structure</u>

```
{
 "reasonCode": "ERR-0001"
 "reason": "Validation Error"
 "message": "Name validation failed",
}
```

<u>Example of Extended Error JSON Structure</u>

```
  {
   "reasonCode": "ERR-0001"
   "reason": "Validation Error"
   "message": "Name validation failed",
   "developerMessage": "Validation failed",
   "api": "/rest/customer-portal/v1",
   "errors": [ {
        "location" : "firstName",
        "message" : "First name cannot have fancy
characters"
        "messageCode": "INVALID_FORMAT"
   }]
  }
```

3.3.6 Headers for Caching

- An API may optionally use the following response headers for supporting web caching

HTTP Response Header	Description
ETag	• In response to a request the API returns the current resource representation and the corresponding ETag value (hash or checksum) in the ETag header • Example : Etag:"686897696a7c876b7e“ • The client may then decide to cache the representation, along with its ETag. • Later, if the client wants to retrieve the same URL again, it will send its previously saved copy of the ETag along with the request in a "If-None-Match" field. • On this subsequent request, the server may now compare the client's ETag with the ETag for the current version of the resource. • If the ETag values match, meaning that the resource has not changed, then the server may send back a very short response with a HTTP 304 Not Modified status. • The 304 status tells the client that its cached version is still good and that it should use that.
Last-Modified	• This basically works like to ETag, except that it uses timestamps. • The response header Last-Modified contains a timestamp in RFC 1123 format which is validated against If-Modified-Since. • If the cached entry on the client side has a last modification date, subsequent request for the same URL should have the If-Modified-Since header in the GET request message • Sever responds with HTTP 304 if resource has not changed
Cache-Control: max-age	• The Cache-Control header with max-age value (in seconds) enables the API to inform the freshness lifetime. • Example: Cache-Control: max-age=3600

4.0 Interface Signatures

4.1.1 Input Parameters

- The input parameters for an API (including HTTP headers, request body, query strings) must have relevant business context on the consumer side.
- System-specific parameters should be defined with certain degree of business abstraction such that the parameters can provide business context relevant to the API consumers
- An API specification should only require the consumer to provide the minimum inputs needed to complete the business operation

- Input parameter names should not contain any system names

4.1.2 Naming Convention

Element	Naming Convention
URL	• A URL, including query parameters and resource id definitions, should be all in lower case • Use hyphen "-" as word separator • Resource element in an URL should be plural • e.g. https://api.aia.com/mypage/v1/customers?cust-id=1234 • e.g. https://api.aia.com/mypage/v1/customers/{cust-id} (definition in swagger)
HTTP Header	• All headers should be defined in lower case • Use hyphen "-" as word separator • Header should be defined according their scopes in the following order: ○ HTTP standard header ▪ e.g. authorization ○ AIA-specific header ▪ Prefixed with x-aia ▪ e.g. x-aia-request-id ○ API-specific header ▪ Common across different swagger files. ▪ Prefixed with x-<name> where <name> is a reference name for grouping the headers ▪ e.g. x-vitality-entity-id ○ Operation-specific header ▪ header that is specific to the operation being defined ▪ does not have to start with x- ▪ lower case with "-" as separator • AIA-specific headers that are common across different operations or APIs should start with x-aia (lower case) • Start each word with upper case • e.g. x-aia-message-id

Schema	• Schema names should be in camel case starting with upper case
	• Attribute names should be in camel case starting with lower case
	• Resource Identifiers must be well documented in property description
	• Mandatory fields must be defined
	• Use of suffixes to specify the "functional type" of an attribute:
	  ○ If the attribute is a number of items: suffix his name with "Count"
	  ○ If the attribute is an amount : suffix his name with "Amount"
	  ○ If the attribute is a percentage : prefix her name with "Percentage"
	  ○ If the attribute is an identifier : suffix his name with "Id"
	  ○ If the attribute is a date : suffix his name with "Date"
	
	• e.g. json definition for Customer object
	
	`Customer:`
	`  type: object`

	```
	          properties:
	            id:
	              type: integer
	              format: int64
	              description: customer id
	            name:
	              type: string
	            firstName:
	              type: string
	            lastName:
	              type: string
	            email:
	              type: string
	            phone:
	              type: string
	              description: phone number
	          required:
	            - id
	            - name
	```

4.1.3 Weak Typing

- Weak typing means using the value of one type to represent the value of another type, e.g. using string to represent a date or a number value
- Weak typing is commonly found in interface specification especially for HTTP headers and query strings. Fields in a data model may also be weakly typed.
- When weak typing is used, the documentation for the corresponding attribute must clearly state the expected format that represents the target value.
- e.g. when a string is used to represent a date value, the documentation should specify the date format in string, such as YYYY-MM-DD
- e.g. when a string is used to represent an integer within a range such as 1 to 100, the documentation should state that the string represents an integer value within the range 1 to 100

4.1.4 Schema Design

- Follow JSON Schema Draft-07 Specification (from http://json-schema.org) when defining JSON structures
- The specification is defined in 3 parts
  - JSON Schema Core - defines the basic foundation of JSON Schema
  - JSON Schema Validation - defines the validation keywords of JSON Schema
  - JSON Hyper-Schema - defines the hyper-media keywords of JSON Schema
- Section 7.3 of the JSON Schema Validation defines formats for many common non-primitive data types like date, datetime, email, hostname, etc, which is represented as a string value.
  - Eg. for dates and time, it is derived from RFC3339, section 5.6, which follows ISO8601 standard.

Data Type	**RFC3339 format**	**ISO8601 format** **(Expressed using Java SimpleDateFormat pattern)**
Timestamp	date-time	yyyy-MM-dd'T'HH:mm:ss[.SSS]XXX
Date	full-date	yyyy-MM-dd
Time	full-time	HH:mm:ss[.SSS]XXX

5.0 Method Design

RESTful APIs are designed around the concept of a resource. A resource is an object with data attributes and relationships to other resources. A resource will also have a set of methods that can be executed to perform some work against it.

For RESTful APIs, these methods are based on a subset of HTTP methods, which roughly maps to the four basic programming functions for data records, create (POST), read (GET), update (PUT) and delete (DELETE).

Depending on the method executed, the underlying resource will be accessed or changed.

When a resource is only accessed, there is very little implications, but when a change is made to a resource, the API implementation will need to ensure the changes made to the resource is performed in an atomic way.

For simple resources, atomic transaction behaviour can easily be implemented. For complex resources, where there may be many relationships with other resources, ensuring proper transaction behaviour of the API is much more challenging.

This section of the document will step through the key design considerations for implementing each type of method.

5.1 Method Behaviours

When a method produces the same results when executed once or multiple times, it is deemed idempotent. For RESTful APIs, this is applicable to the state of the resource the method is executed on, the response sent back to the client can be different.

For example, a request is sent the first time to set a resource attribute value, a 200 standard response is returned after updating the attribute. If the same request is sent again, 2 things can happen,

1. The same attribute is updated again with the same value and a 200 response is returned, repeating the process of the first request.

2. The API can have logic to detect this as a duplicate (eg. same request id was presented), and a standard 200 response is sent with a warning message in the response body to state that the duplicate request was ignored.

At the end of either scenario, the state or the resource is the same.

Now if we look at the same example again, the API is behaving in an idempotent manner, but the resource is changed at least once. This kind of behaviour is deemed un-safe.

When an API is idempotent and does not modify the state of the resource at all, the methods is deemed be Safe.

5.1.1 Implementation Requirement by Method

HTTP Method	Idempotent?	Safe?
GET	Yes	Yes
PUT	Yes	No
DELETE	Yes	No
POST	No	No

- API implementations must adhere to this design
- It serves as an implicit design contract to API clients as defined in RFC 2616
- Un-safe method implementations must be accompanied by sufficient test cases to proof of its correctness and atomicity behaviour including rollback on failures.

5.1.2 Resource State vs Resource Representation

Resource state is the current state of a resource on a server at any point of time. Physically it may be persisted as one or more records in a datastore.

Additional control or tracking information may also be stored along with the record somewhere on the server side. This additional information would not be considered part of the resource state if has no meaning in the context of the resource.

For example, if a resource is modelled from a database table, and the table contains control or audit fields, these fields are not considered part of the resource state when we judge it in the context of method idempotency or safety.

This also applies to any system or application log records that may be generated in the path of processing. These are classified as application state.

When we talk about the resource representation, it is referring to the API response, represented in a specific format. The standard is JSON preferred over other formats like XML or URL encoded string. This can be different to the resource definition of the underlying record in the datastore.

5.2 GET Method

- Never changes the resource state on server

- Purely for retrieving the resource representation or meta data at that point of time

5.3 PUT Method

- Used to update the resource state
- Used for the replacement of the whole resource. This is the well-known practice that traces back to HTTP RFC 2616
- Use POST (as controller) for partial resource update
- Protocols like OData builds on top of REST will use PATCH for partial updates, but for building microservices and sharable APIs, it is not recommended
- Upserts using PUT is not recommended, implement as a separate POST operation
- Underlying implementation needs to ensure the update is atomic over the whole resource. If the underlying resource is composed of multiple resources (eg. resides on multiple tables or systems), proper transaction control is required in the API design.

5.4 DELETE Method

- Deletes the resource as whole by resource identifier
- Do not implement delete by reference or by relative position.

  Eg. DELETE address/last
  DELETE policy/expired
- Ensure implementation of deletion is atomic over the whole resource. Transaction control may be required to co-ordinate this in the API Design

5.5 POST Method

- Creating a new resource
  - By default, creates a new resource. If invoked multiple times with the same request, it will create another new instance of the resource.
  - No duplication check unless it is part of interface design, which needs to be clearly documented in API specification
  - Resource creation needs to be atomic

- Controller resource type
  - Used to perform a unit of work on a resource
  - For operations on a resource that cannot be mapped to one of the CRUD operations.

- As POST methods are not required to be idempotent, the API design and specification needs to clearly describe the function, purpose, and expected behaviors
- Implementation must be accompanied by a comprehensive set of test cases to demonstrate the intended behavior and error handling
- Used to implement partial updates or merge on resources where the implementation will need to be atomic.

5.6 Bulk Operations

In general, bulk operations are treated no differently to operations that work on a single resource. The decision to provide bulk operations is a tradeoff between performance, simplicity and reliability.

APIs must always support operations that work on a single resource, and this is what implementations should focus on in early versions of the API.

When there is a need to introduce bulk operations for a resource, the following constraints must be catered for in the design

- Response time SLA must still be acceptable for API clients
- Action must be performed on all or none of the resources (rollback all on error)

To comply with these design constraints, the bulk operation implementation must

- Have the ability to enforce upper limits on,
  - Time that is allowed for the transaction to run. It must have a backout (rollback) mechanism on timeout
  - Number of resources that the bulk operation can process in 1 call.
- Have tunable Upper limits
  - The optimum setting for limits would be determined during testing. For example, batch size would be determined by the complexity of the transaction on backend resources and transmission size that is best suited for the network infrastructure.

5.6.1 Guideline for Implementing Bulk Operations

- PUT on a collection should only be allowed on a sub resource and not on a top level resource of an API.

- DELETE methods on collections should be avoided.

- Implement bulk operations for the above scenarios as a POST method with strict selection criteria for identifying individual resources by ID in the request parameters or body.

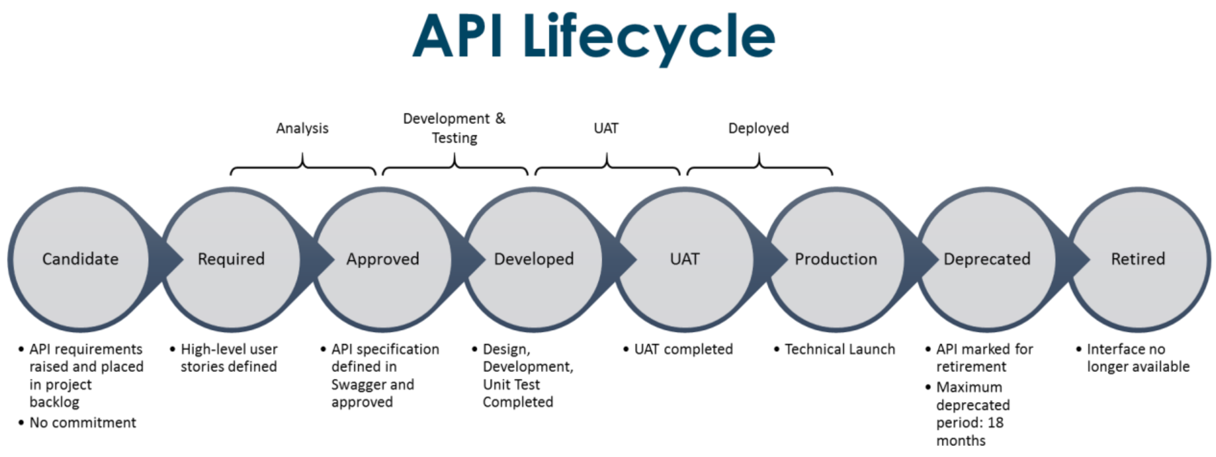
6.0 API Versioning

- Version number format: v<major version>.<minor version>, e.g. v2.1
  - Only major version number is shown in URL
    - https://api.aia.com/mypage/v1/customers/1234e.g.
  - Minor version is shown in documentation but not in URL
- Only the most recent API minor version is in production.
  - e.g. v1.1 and v1.2 will never be in production at the same time
- The version number in the URL refers to the API version as a whole.
  - Version numbers of individual operations within the API are not exposed in the URL and are managed internally
- Major version number is incremented if the new version includes incompatible changes
- Minor version number is incremented if the new version includes only compatible changes
- Compatible vs Incompatible Changes

Compatible	Incompatible
Interface Signature	
• Adding new operations in an API • Adding optional fields as input parameters	• Removing operations from an API • Changes in input model (model structure or new mandatory fields) • Changes in output mode (model structure or removing fields)
Internal Behavior	
• Changes in data sources or data mappings while the affected fields maintain the same business context to the consumers	• Changes in data sources or data mappings while the affected fields DO NOT maintain the same business context to the consumers • Changes in internal behavior of operations (can be considered on a case-by-case basis)
API Policy	
• Virtualization changes in gateway/mediator which are transparent to consumers • Policy changes in throttling or traffic control	• Change in security policy, .e.g. authentication method

6.1 API Lifecycle

Every version of an API should have a clearly defined lifecycle. The following is a reference API lifecycle definition which can be customized for a particular project.


API Lifecycle
Analysis
Development & Testing
UAT
Deployed
Candidate
Required
Approved
Developed
UAT
Production
Deprecated
Retired
• API requirements raised and placed in project backlog
• No commitment
• High-level user stories defined
• API specification defined in Swagger and approved
• Design, Development, Unit Test Completed
• UAT completed
• Technical Launch
• API marked for retirement
• Maximum deprecated period: 18 months
• Interface no longer available

7.0 Security

7.1 Transport Security

- Protecting data in transit with encryption is mandatory for all communications including
  - Communications taking place entirely within AIA internal network
  - Communications with external parties which reside outside of AIA data centers
- HTTP with Secure Socket Layer (SSL), HTTPs, is the preferred transport protocol for RESTful APIs
- For server-to-server communications or B2B integration scenarios where API level controls are not required, mutual SSL can be used for supporting client authentication between 2 servers.

7.2 HTTP Header vs HTTP Body

- The HTTP request body should not contain any security data or message metadata
- HTTP request headers must transport the required security information

- The HTTP request header size should be less than 8 kB (the limit of the HTTP header size)
- eg. For JWT or OAuth 2.0, this will be in the form of a "Bearer Authentication" using the Authorization header parameter

7.3 OAuth 2.0

- OAuth 2.0 is the preferred access control for external facing APIs, i.e. APIs that are exposed to consumers who are in the internet, e.g. mobile apps, web apps, partners
- Supports both 'server to server' and 'delegated access' scenarios
- Applicable to both internal 'confidential' and external '3rd party or public' applications by using different grant types
- Restricts usage to certain APIs and scopes

7.3.1 Guidelines on Grant Type Usages

Grant Type	Recommended Usage
Authorization Code Grant	• This type should be used for server-side applications where source code is not publicly exposed, and client secret confidentiality can be maintained • This is the most secure grant type and is the preferred type for third-party end-user applications with access to AIA APIs
Implicit Grant	• This type should be used for mobile apps and web applications (i.e. applications that run in a web browser), where the client secret confidentiality is not guaranteed • Client authentication and refresh tokens are not supported
Resource Owner Password Credentials Grant	• One Request – Reply only for getting the Access Token • Username and password visible for the client application • Least preferred grant type. Use this type only when there is no alternative
Client Credentials Grant	• Only the client is authenticated, not the resource owner • Suitable for server-to-server communication • Can be used for AIA-internal application or B2B REST-based integration with partners

7.3.2 Scopes

- OAuth 2.0 scopes provide a way to limit the amount of access that is granted to an access token.
- Consumers can request access to one or more scopes, and the authorization server will grant one or more scope when issuing the access token.
- The list of scopes granted may be different to the list requested, this is controlled by the authorization server

- According to the OAuth specification, scopes are a case-sensitive, space-delimited, unordered list of strings.
- Standard naming convention for scopes will be
  "apigroup.apiname[.object][_subobject][_action]"

  - All lower case with "." as the separator between API group, API name and Object.
  - Beyond object, use "_" as the separator for sub object and actions if they are defined.
  - Where a scope section is composed of multiple words, define as lowercase without spaces.

Scope section	Description
API group	Following the standard for URL format for API Group Eg. cs (Corporate Service), vitality (Vitality)
API name	Name of API in lower case with no spaces between words Eg. partner, core
Object (optional)	Name of API resource or a grouping of API resources for access control purposes. Eg. sessions, memberships
Sub Object (optional)	Further separation of access level for sub objects if required. Eg. assessments_fitness
Action (optional)	Further separation of access level by action if required. Eg. read or write

Eg.	vitality.core.memberships
	vitality.core.assessments_fitness_read
	vitality.partner.vdx_accommodations
	vitality.partner.sessions_read
	vitality.partner.sessions_write

7.3.3 Authorization

- API gateway is responsible for coarse-grained, user-agnostic authorization
  - Is scope of token valid for the API being called?
  - Is the access token valid (signature, expiration)?
- API Provider is responsible for fine-grained, user-related authorization
  - Is the user allowed to access the business data?

7.3.4 ClientId/Secret vs API Key

- API keys are generally generated by API portal for identification of client apps and tracking of runtime statistics on API consumption
- When OAuth protocol is used, the OAuth authorization server generates ClientId/Secret for each client app for authentication and authorization purposes
- API key should not be used for client authentication as the key is typically not known to the OAuth authorization server
- For public APIs which do not require authentication, the API key can be used for SLA enforcement or tracking of runtime statistics by the portal

7.4 API Key

- An API Key is a piece of code assigned to a party and is used whenever that party makes an API call. It is very easy to implement
- This Key is typically a long string of generated characters which follow a set of generation rules specified by the issuer. Often, this will be a kind of UUID
- Not meant to be used as a security feature, it is only a form of identification for an API Caller, it is not enough for authorizing individual users
- Only adequate for inquiry type scenarios only where the data is not sensitive or mission critical
- Requires HTTPs to protect the API Key during transmission

7.5 JWT as access token

- JSON Web Token is an open standard (RFC 7519) for securely transmitting information between parties using a JSON object
- Information within the token is digitally signed by using a public/private key pair using RSA or other cryptosystems
- When used as an access token for authorization, the information contained within the token (as token claims) contains all the data that is required to determine the identity and scope of access. This is no need to perform an external lookup (eg. via database or an API call)
- The tradeoff for the verification performance is that the JWT, once issued, it cannot be revoked
- Access token expiry must be kept low, 15 minutes to 30 minutes depending on client type (internal, external, 3rd party) as stipulated by AIA security policy. Maximum Refresh token expiry is 6 hours
- Both OAuth 2.0 and JWT presents tokens to the server for authorization, but JWT does not define how the token is obtained. The server only needs to be able to verify the authenticity of the token

- JWT is suitable when an application has already established an authentication mechanism, and delegated access is not required. Token can be generated or exchanged from an existing authenticated identity

7.6 HTTP Basic Authentication

- Simplest method for enforcing access control on a web resource like API
- Uses standard HTTP Authorization header like Bearer Authentication but the credentials are only encoded, not encrypted.
- Must be used with SSL to ensure the credentials are not sent in plain text
- Suitable for internal network only. Additional access control to be imposed (policy enforcement) by API Gateway if API is to be used externally.
- API implementation is required to implement at least Basic Authentication for restricting access of the endpoint to the gateway only.

7.7 Using Other Authentication Methods

- While OAuth 2.0 is a widely used API security protocol and can be used in many scenarios, there are cases where the preferred OAuth 2.0 method is considered inappropriate and may add unnecessary complexity, e.g.
    - Internal API communications
    - Consumers that do not support OAuth protocol, e.g. 3rd party applications
    - Authorization flow is incompatible with OAuth 2.0 grant types
- For those cases, other security mechanisms can be considered including but not limited to the following:
    - HTTP Basic Authentication (i.e. username/pw)
    - JWT
    - Client Certificates, mutual SSL
    - Kerberos
    - SAML Tokens

7.8 Best Practices for RESTful API Security

This section highlights some of the important best practices for securing REST-based services as recommended by the Open Web Application Security Project (OWASP). More information on API security can be found from the links listed below in the References section.

7.8.1 Authentication and Session Management

- RESTful web services should use session-based authentication, either by establishing a session token via a POST or by using an API key as a POST body argument or as a cookie.
- Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

Examples:

```
OK:
https://example.com/resourceCollection/<id>/action

NOT OK:
https://example.com/controller/<id>/action?apiKey=a53f435643de32
(API Key in URL)
http://example.com/controller/<id>/action?apiKey=a53f435643de32
(transaction not protected by TLS; API Key in URL)
```

7.8.2 Whitelist allowable methods

- It is common with RESTful services to allow multiple methods for a given URL for different operations on that entity.
- For example, a GET request might read the entity while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity.
- It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).

7.8.3 Insecure direct object references

- Insecure Direct Object References occur when an application provides direct access to objects based on user-supplied input.
- As a result of this vulnerability attackers can bypass authorization and access resources in the system directly by modifying the value of a parameter used to directly point to an object.
    - Such resources can be database entries belonging to other users, files in the system, and more.
- To address this security risk, an API implementation should perform sufficient authorization check before using a user-supplied input to retrieve an object.
- The following defensive measures should be considered:

  - For direct references to restricted resources, does the application fail to verify the user is authorized to access the exact resource they have requested?
  - If the reference is an indirect reference, does the mapping to the direct reference fail to limit the values to those authorized for the current user?

7.8.4 Protect against cross-site request forgery

- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions through RESTful web services in which they're currently authenticated.
- CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. A successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth.
- Recommended standard CSRF defense that does not require user intervention:
  - Check standard headers to verify the request is same origin
  - AND Check CSRF token
- Refer to https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide for a detailed discussion on the above two checks.

7.8.5 Validate incoming content-types

- When POSTing or PUTting new data, the client will specify the Content-Type (e.g. application/xml or application/json) of the incoming data.
- The server should never assume the Content-Type; it should always check that the Content-Type header and the content are the same type.
- A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a 406 Not Acceptable response.

7.8.6 Validate response types

- It is common for REST services to allow multiple response types (e.g. application/xml or application/json, and the client specifies the preferred order of response types by the Accept header in the request.
- Do NOT simply copy the Accept header to the Content-type header of the response.
- Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types.

7.8.7 Handling of JWT

- The JWT spec that allows signature algorithm to be set to ‘none’. This should be ignored if the JWT is expected to be signed.

  - Verification should fail if the JWT is expected to be signed and the signature algorithm is set to ‘none’
- Secure the secret signing key used for calculating and verifying the signature.
  - The secret signing key should only be accessible by the issuer and the consumer; it should not be accessible outside of these two parties.
- Do not contain any sensitive data in a JWT. The tokens are usually signed to protect against manipulation (not encrypted) so the data in the claims can be easily decoded and read.
  - For example, user’s address should not be directly included in a JWT. Store a link to the user’s record or another identifier that is opaque and the application can look up the information based on the opaque reference.
  - If a JWT does contain sensitive information, consider using JSON Web Encryption (JWE) to secure the token.

7.9 References

- The Open Web Application Security Project
- OWASP Secure Coding Practices - Quick Reference Guide
- Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

8.0 API Management

API management deals with the publishing, documenting, protecting and analyzing of APIs. These activities are supported by the API Portal and API Gateway. The details of processes are outside the scope of this document. This section serves only as an overview for developers.

Once an API is ready in its runtime, it is registered on the API Gateway to expose the endpoint securely to consumers. Then from the API Gateway, it is published to the Portal to allow consumers to discover and request access to the new API. When a consumer accesses the API, the API Gateway will monitor the usage.

8.1 API Specification

- The API specification, prepared in accordance to OpenAPI Specification (2.0), serves as the main design contract between the implementation, API Gateway, and consumers (via API Portal)
- Prepare by using the AIA API Specification template

8.2 Implementation

- Must adhere to the API specification, any changes made to the interfaces should be reflected in the API specification.
- The API development standards does not dictate the technology stack used to implement APIs, but it should follow AIA Product Standard
- Fine-grain authorization is the responsibility of the API Provider. Eg. Restricting business data access by the identity of the API consumer (from header)
- API Provider needs to ensure the path from API Gateway to API implementation is secured. Implementation needs support HTTP Basic Authentication and SSL at a minimum
- API provider is responsible for the load balancing and scaling of the implementation. Only one URL endpoint will be registered on the API Gateway for each API
- Policy enforcement and overall API protection to consumers is provided by API Gateway

8.3 API Gateway

- Responsible for protecting APIs from a security and performance perspective
- It is a control point for APIs, providing the following features
  - Authentication/Authorization (coarse-grain)
    - Identifying consumer applications
    - Controlling access to APIs or API scopes based on indentity
  - Security policy enforcement
    - Checks the transport and source of request

        - Verifies the validity of the identity of the API requester
        - Expiry of the presented credentials
    - Transformation and routing
        - Routes requests to API implementation securely
        - Propagates identity information and transform (from token) if required
        - Mocking an implementation behavior to support early integration testing
        - Error handling (log to central event store), Error translation (for masking internal system errors)
    - Contract and service level agreement (SLA) management
        - Monitoring usage and performance of APIs
        - Enforce usage limits (throttling)
        - Logging
- APIs are registered on the API Gateway by importing the API specification
    - Scopes from the API specification is mapped to gateway scopes to implement runtime access control
    - Profile information is set according to API specification (propagates to API portal on publish)
    - Registered APIs are then published to the API portal from the API Gateway to allow consumers to discover and test the API (in sandbox)

8.4 API Portal

- Provides a central self-service portal for exposing APIs
- Uses the profile information to organizes APIs,
    - By communities (making API discoverable by a subset of users)
    - Grouping by categories in the catalogue
    - Allow Searching by Tags
- Allows users to download the runtime version of the API Specification as deployed on the API Gateway and Client SDKs for interfacing with the API using various languages and technology stacks
- Users can request access to APIs (sandbox) through the portal. The portal facilitates the registration of the application and credential generation on the API Gateway automatically

9.0 Appendix

9.1 Hong Kong Standardization

The following are the standardization used by Hong Kong to provide extra consistencies across APIs

9.1.1 Controller Type API Request Response Structure

Request

```
{
        client: "string" (mandatory – name of the client),
        payload : {…} (mandatory – the actual data payload defined by the API provider)
}
```

Response

```
{
        api: "string" (mandatory – URI of the API being called),
        code: int (mandatory – the application return code, see return code standard below),
        payload: {…} (mandatory – the actual data payload defined by the API provider),
        errors: [{
                reasonCode: "string" (mandatory – see return code standard below),
                reason: "string" (mandatory – reason why request failed),
                message: "string" (mandatory – a human readable message telling user what to
do),
                developerMessage: "string" (mandatory – a message for developers to debug the
error),
                api: "string" (mandatory – URI of the API being called),
                actor: "string" (mandatory – the module that raised the error),
                errors: [{…}] (mandatory – see group development standard)
        }] (optional – use in case of error)
}
```

9.1.2 Reason Codes Format

[BU][APP][TYPE][C][XXXXXXXXXXXXXXXX]

Code	Length	Description
BU	2	Business unit (e.g. HK)
APP	3	Application code (reserved, to be assigned later)
TYPE	3	ERRR – Error message WARN – Warning message INFO – Informational message
C	1	H – High M – Medium L – Low
XXX	16	Application unique message code

9.1.3 Query String Conventions

Parameter	Meaning	Default (suggestion)	Example
offset	The offset of the 1st record being returned	1	?offset=1
ps	The page size of the records being returned	10	?ps=10
order	Records returning order	<none>	?order=owner(asc), name(desc)
depth	The depth of children to return, for complex objects with multiple levels	3	?depth=3
<filter>	Filter to apply to the query	<none>	?name=Chan