

API Standards

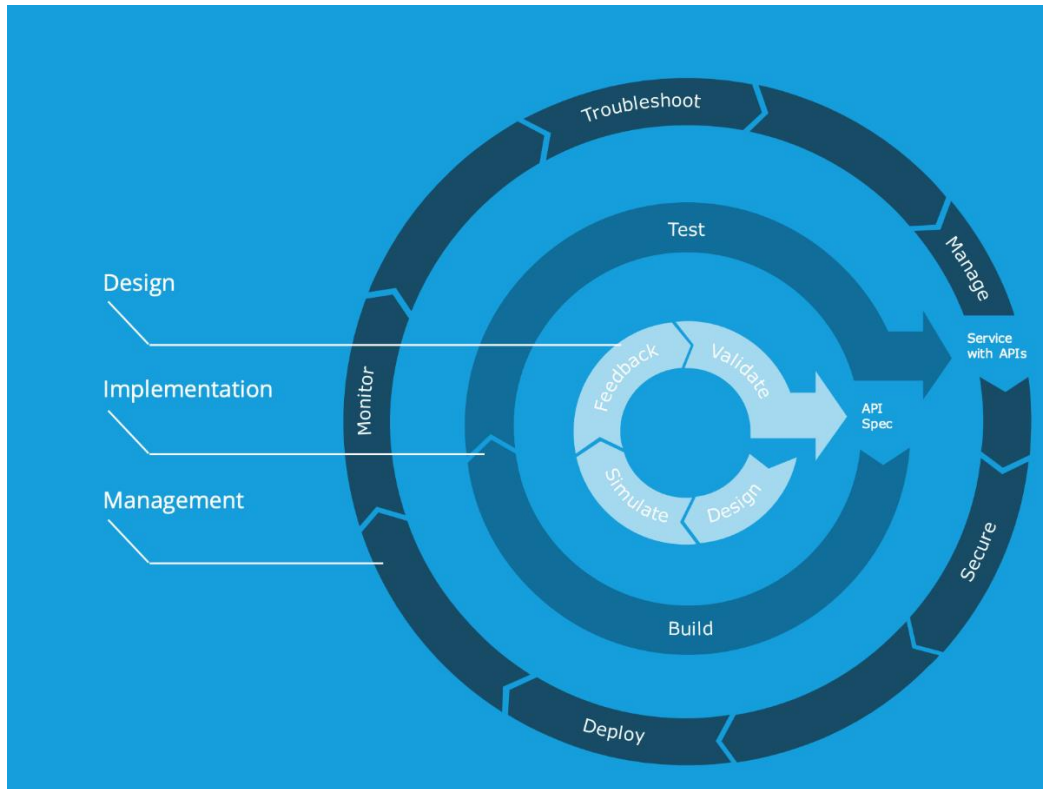
Application Programming Interfaces (APIs) provide common access to business functionality, capabilities, and data using consistent contracts. This document outlines standards for APIs in areas such as naming, taxonomy, and tooling. For questions or comments, please contact integration@gapac.com.

Contents

API Lifecycle	2
Considerations	2
Lifecycle Standards	2
Development State	2
Stable State	3
Deprecated State	3
State Promotion	3
API Discovery	3
Naming	3
API URLs	4
Resource Naming Conventions	6
Versioning	6
API Design Standards	8
RAML Naming Conventions	8
RAML Design Conventions	9
RAML Folder structure	10
Response Codes	11
API Documentation Standards	14
Outline or Description of the API	14
Endpoint URLs	15
Authorization / Authentication	15
High level Design or Diagram	15
API Asset Taxonomy	15
API Category & Tagging	16
Flow Diagram	16
Change Log	17

API Lifecycle

An API's lifecycle should follow a design-first approach, as outlined by the following image:



Source: <https://www.mulesoft.com/resources/api/what-is-full-lifecycle-api-management>

Considerations

Create APIs with the following considerations:

- Core capabilities as reusable building blocks
- APIs are a vision into business capabilities. Not IT systems
- Design-first (contract), code second
- Abstract technology stack and implementation details from consumers
- Self-service discoverability and accessibility
- Support for a variety of integration patterns

Lifecycle Standards

When creating an API, designers and developers should follow a lifecycle of development, release, and deprecation. To represent these phases, an asset moves through the lifecycle states of

development, stable, and deprecated.

Development State

An asset version should use the development state when it is in iterative process of design and development and is not ready for consumption.

For more information on what changes reflects when asset version is republished

<https://docs.mulesoft.com/exchange/lifecycle#development-state> .

Stable State

An asset version should use the `stable` state when it is ready for consumption. Once an asset is promoted to this state, it cannot be republished or overwritten.

For more information on Deleting asset versions and its effects

<https://docs.mulesoft.com/exchange/lifecycle#stable-state>

Deprecated State

An asset version in the `deprecated` state is like an asset version in the `stable` state and it is marked as `deprecated` a when that version need be removed from the exchange. The deprecation period is one year for major version.

NOTE:

- Maximum five minor versions can be maintained after that need to update the lifecycle state to `deprecated`.
- Maximum two major versions can be maintained after that need to update the lifecycle state to `deprecated`.

State Promotion

State promotions allowed and to be followed for the asset in the exchange are:

1. `development` to `stable`
2. `stable` to `deprecated`

API Discovery

One of the biggest benefits of APIs is that they can be accessed via the inter/intranet, allowing for them to be re-used by multiple use cases. Re-used assets results in a simpler IT environment and faster realization of value for new initiatives. Consumers— software engineers or citizen developers – of APIs should have one catalog to discover any API within the organization. Georgia-Pacific uses MuleSoft's [Anypoint Exchange](#) as its API catalog.

Naming

APIs are easiest to find when their names utilize unique, intuitive, and easy-to-follow conventions. When followed, these conventions avoid duplication and enable reusability. These are the naming standards for any API in GP based on the Anypoint component. Domains and sub-domains should align to the [GP Data Taxonomy](#).

Component	Naming Pattern
-----------	----------------

Design Center	lower case, hyphen {domain}-{sub-domain}-{layer}-api
Exchange	lower case, hyphen {domain}-{sub-domain}-{layer}-api
API Manager	lower case, hyphen {domain}-{sub-domain}-{layer}-api
API URL	/ {layer} / {domain} / {version} / {sub-domain} More details below

API URLs

URL naming should reduce redundancy and minimize user confusion while making the API more accessible and user-friendly. The structure of a URL is key to distinguishing and organizing APIs across functional domains. A good URL helps govern the lifecycle of the API through versioning.

URL structure

Part	Description	Example
{base url}	Required: combination of env, company. Its load balancer server url.	dev.gpapi.gp.com, test.gpapi.gp.com, gpapi.gp.com
{version}	Required: The version of the API. The version can reflect only major versions	V1, V2
{layer}	Required: The layer of the API according to API-led connectivity approach. The possible values are sys (System), prc (Process), and exp (Experience)	sys, prc, exp
{domain}	Required: The name of the API. This typically represent a business or utility service and should be a short but descriptive name	product, procurement
{System of record}	Optional: Source/Target system name.	MP2, Asset suite

{sub-domain}	Required: The name of the resource that represents the actual object.	item, supplier
{query params}	Optional: The query string can define state transition parameters.	facility=BEL&date=2023/01/01

*URL examples***System API**

System APIs of applications should have a resource name that includes the short name of the system to which they belong. This may change for third-party (SaaS or hosted) applications where GP does not maintain the API.

<https://gpapi.gp.com/{version}/{layer}/{System of record}/{sub-domain}>

<https://gpapi.gp.com/v1/sys/mp2/facility>

Process API

Resource names of Process APIs are derived from the corresponding business object.

<https://gpapi.gp.com/{version}/{layer}/{domain}/{sub-domain}>

<https://gpapi.gp.com/v1/prc/product/item>

Experience API

Resource names of the Experience APIs dependent on the use case.

<https://gpapi.gp.com/{version}/{layer}/{domain}/{sub-domain}>

Utility API

Such as logging, incident, email notification API.

<https://gpapi.gp.com/{version}/{generalacronymy}>

<https://gpapi.gp.com/v1/logging>

Resource Naming Conventions

The key principles of REST support separating an API into logical resources and manipulating them using HTTP methods, where each method has a specific meaning. A resource represents an object type within your domain.

For example, `/projects` would represent projects within your organization.

Use Nouns, Not Verbs

A URI should refer to a resource that is a "thing" instead of referring to an action. Therefore, avoid using "actions" within your resource name.

For example, `getLogs` or `deleteLog` should be avoided instead you could just name your resource `/logs` and send a request against that resource.

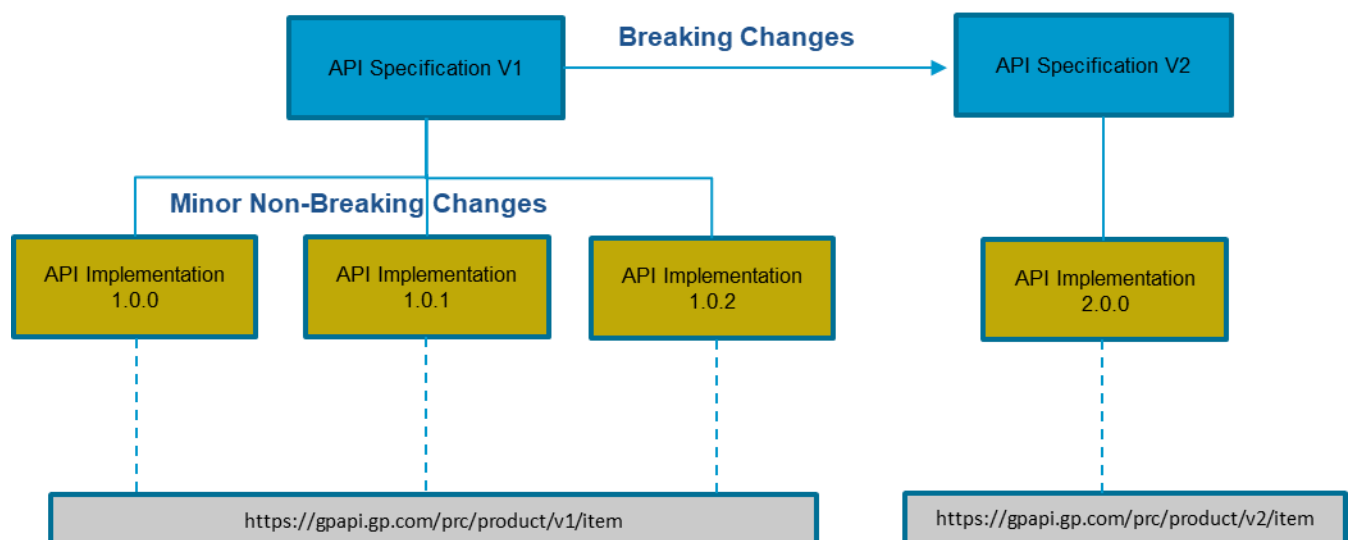
The action on the resource should be served by the HTTP methods, i.e., GET or DELETE so that the same resource can be used to get a list all the logs, create new logs and delete all the logs.

Use Plural Nouns

Additionally, it is helpful to keep endpoints in the plural form. This helps to understand and avoid confusion when using the API. For example, the resource `/logs` are self-explanatory because you know you are acting upon the collection of logs, as opposed to just one log.

Versioning

Versioning helps handle feature changes even if the nature of the change is unknown. An API version is tied to an API specification version and included as part of the URL. Major versions should change when breaking changes are introduced to the latest version.



When introducing changes to the API that do not require the API user to changes or contain enhancement/bug fixes, these are non-breaking changes. These changes should be backwards-compatible and existing consumers should not be impacted upon release of the latest version.

If the change can impact the consumers, then it should be deemed as a major version change and should be moved to a new API specification version.

The table below outlines a decision framework for when to introduce a major version. Typically, when one clicks **Add new API operation** a minor version increase occurs.

Type of Change	Changes Requiring a New Major Version	Changes Not Requiring a New Major Version
Service contract	<ul style="list-style-type: none"> Remove API endpoint Change in the effect of an API operation Change in the response type of an API operation Add new required operation argument 	<ul style="list-style-type: none"> Add new API operation Changes in error codes
Data contract	<ul style="list-style-type: none"> Remove an existing element (or attribute) Add new required element (or required attribute) Change an existing element (or attribute) 	<ul style="list-style-type: none"> Add an optional element (or attribute) Add a derived element type
Representation format	<ul style="list-style-type: none"> Remove existing representation 	<ul style="list-style-type: none"> Add new representation/content type (adding application/xml).
Accessibility	<ul style="list-style-type: none"> Restrict permissions 	<ul style="list-style-type: none"> Relax permissions

Best Practices for versioning

- Always release an API with a Major version number.
- Minor versions should be backwards-compatible.
- Support one previous major version along with the current major version. Upon introduction of a new major version, deprecate the current version - 1, and decommission any older versions.
- Maintain at least one major version prior to the current one to ensure backwards compatibility and allow time to bring code in compliance with the latest version.
- Online documentation of versioned services must state the support status of each previous API version and provide a path to the current version.




API Design Standards

This document describes the various best practices in designing APIs for Anypoint Platform.

RAML Naming Conventions

Match RAML file name with API functionality, following [Naming standards](#).

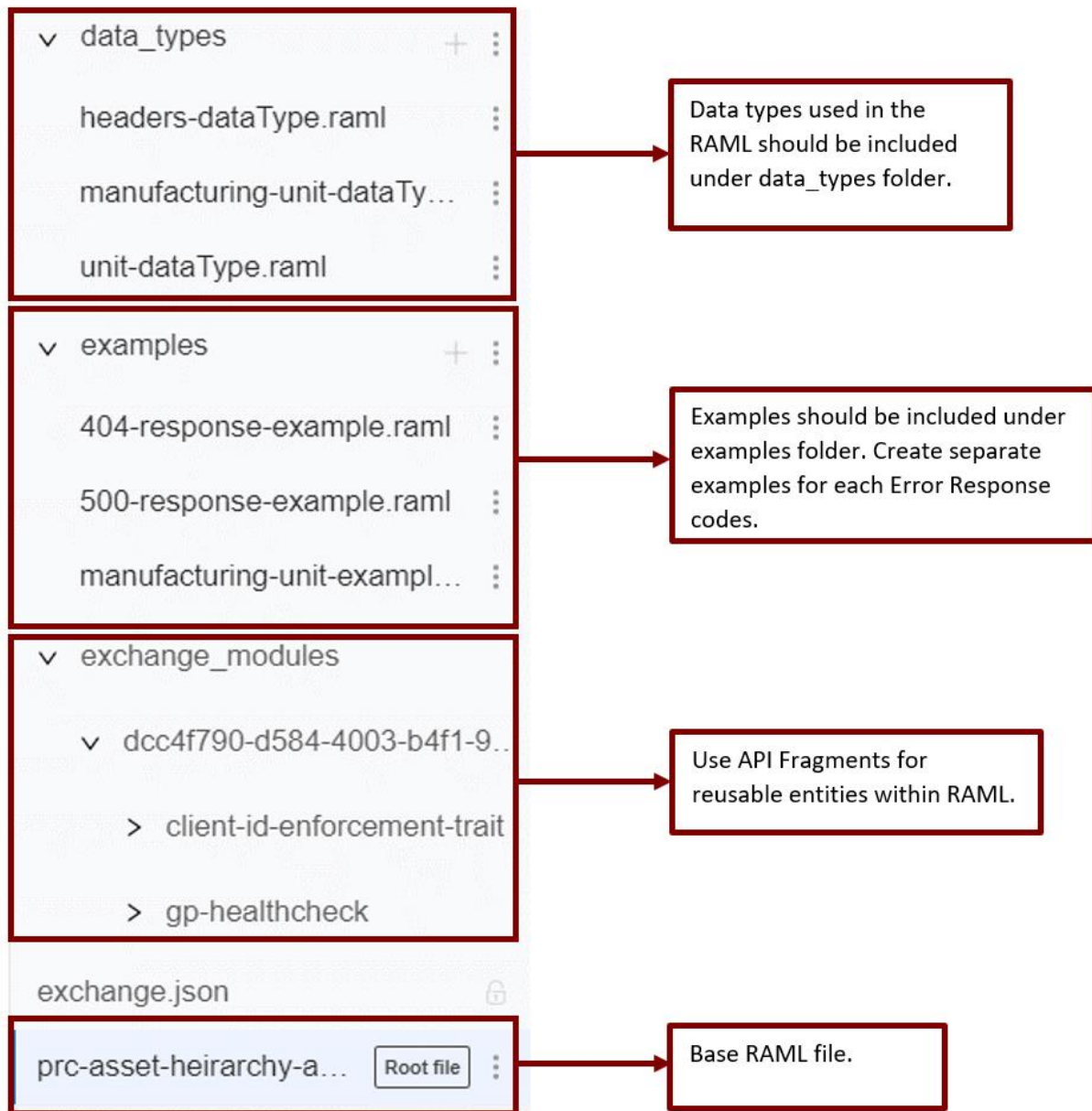
RAML Design Conventions

Section	Best Practices	Example
RAML header	Include the following elements: <ol style="list-style-type: none"> 1. title: API title 2. description: Description of the API 3. version 4. baseUri 	<pre> #%RAML 1.0 title: Location API description: The Location API is intended to provide basic operations to access the Locations master data version: v1 baseUri: https://dev.gpapi.gp.com/{version}/prc </pre>
Security Schemes	Secure Experience APIs with client-id enforcement and OAuth 2.0. Exchange  : oauth2-security-scheme (mulesoft.com)	<pre> securitySchemes: oauth_2_0: !include exchange_modules/dcc4f790-d584-4003-b4f1-9a9332027202/oauth2-security-scheme/1.0.0/oauth2-security-scheme.raml </pre>
Traits	Secure System and Process APIs with client-id enforcement. Exchange  : client-id-enforcement-trait (mulesoft.com)	<pre> traits: client-id-required: !include exchange_modules/dcc4f790-d584-4003-b4f1-9a9332027202/client-id-enforcement-trait/1.0.0/client-id-enforcement-trait.raml </pre>
Resource types	Include health check under resource types. Exchange  : gp-healthcheck (mulesoft.com)	<pre> resourceTypes: health-check: !include exchange_modules/dcc4f790-d584-4003-b4f1-9a9332027202/gp-healthcheck/1.0.6/healthcheck.raml </pre>
Resources / Methods	Each resource-method pair should include the following elements: <ol style="list-style-type: none"> 1. description 2. displayName 	<pre> /asset-heirarchy: get: description: Used to fetch the asset heirarchy based on Facility ID Follow resource naming convention from Resource Naming Conventions </pre>
Headers /Query Parameters	Each header and query parameter should include the following elements: <ol style="list-style-type: none"> 1. description 2. type 3. required 	<pre> queryParameters: author: description: Author type: string required: false </pre>
Data Types	Each Data Type should include the following elements:	<pre> description: The address data type type: object </pre>

	<ol style="list-style-type: none">1. description2. type <p>Each property should include the following elements:</p> <ol style="list-style-type: none">1. description2. type3. required4. displayName	<p>properties:</p> <p>address1:</p> <ul style="list-style-type: none">required: truedisplayName: address1description: Address Line 1type: string <p>address2:</p> <ul style="list-style-type: none">required: truedisplayName: address2description: Address Line 2type: string
Responses	<p>Each response code should include the following elements:</p> <p>description</p>	<p>responses: 200:</p> <ul style="list-style-type: none">description: List of projects was returned successfully <p>body:</p> <p>application/json:</p>

RAML Folder structure

Externalize examples, types, RAML fragments, Traits from the base RAML file as shown below.



Response Codes

These response codes should be used as standard, the use of undefined return codes is discouraged and should only be done in exceptional circumstances.

Code	HTTP Method	Response Body	Description
200 OK	GET, PUT, DELETE	Resource	There are no errors, the request has been successful

201 Created	POST	URI of the resource that has been created	The request has been fulfilled and resulted in a new resource being created
202 Accepted	POST, PUT, DELETE	An URI of a resource which represents the processing status	The request has been accepted for processing, but the processing has not been completed
204 No Content	GET, PUT, DELETE	N/A	There are no errors, the request has been processed successfully but it is not returning any content
304 Not Modified	conditional GET	N/A	The resource has not been modified: there is no new data to return Note - This code means that the user has requested a resource only if it has been modified since the last cache date of the document stored by the user.
400 Bad Request	GET, POST, PUT, DELETE	{ "httpCode": "400", "httpMessage": "Bad Request", "moreInformation": "Invalid JSON payload received." }	The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modification to the syntax
401 Unauthorized	GET, POST, PUT, DELETE	{ "httpCode": "401", "httpMessage": " Unauthorized ", "moreInformation": " Invalid client id or secret." }	The user makes a server request using the wrong credentials (username/password, client id/secret, etc.)
403 Forbidden	GET, POST, PUT, DELETE	Error message	The server understood the request but is refusing to fulfill it because the user does not have access to the resource.
404 Not Found	GET, POST, PUT, DELETE	{ "httpCode": "404", "httpMessage": "Not Found", "moreInformation": "No resources match requested URI." }	The resource requested cannot be found
405 Method Not Allowed	GET, POST, PUT, DELETE	{ "httpCode": "405", "httpMessage": "Method Not Allowed", "moreInformation": "The method is not allowed for the requested URL." }	The method specified in the request is not allowed for the resource identified by the URI

406 Not Acceptable	GET, POST, PUT, DELETE	Error message	The request contains parameters that are not acceptable. Ex. the file format of the resource requested is not in a format that the user is capable of understanding.
408 Request Timeout	GET, POST, PUT, DELETE		The server timed out waiting for the request from the user that failed to respond in the time allowed by the server.
409 Conflict	POST, PUT, DELETE	Error message	The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request
410 Gone	GET, POST, PUT, DELETE	Error message	Used to indicate that an API endpoint has been turned off. Could be used to deprecate API, for example, to inform the customer that the API will soon stop functioning and to migrate to latest version of the API
412 Precondition Failed	GET, PUT	Error message	The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource meta data (header field data) and thus prevent the requested method from being applied to a resource other than the one intended
415 Unsupported Media Type	GET, POST, PUT	Error Message	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method
429 Too Many Requests	GET, POST, PUT, DELETE	Error message	Indicates that the user has sent too many requests in a specific amount of time ("rate limiting")
500 Internal Server Error	GET, POST, PUT, DELETE	{ "httpCode": "500", "httpMessage": "Internal Server Error", "moreInformation": "Cannot read property 'startsWith' of undefined." }	The server encountered an unexpected condition which prevented it from fulfilling the request

502 Bad Gateway	GET, POST, PUT, DELETE	Error message	The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request
503 Service Unavailable	GET, POST, PUT, DELETE	Error message	The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which may be alleviated after some delay
504 Gateway Timeout	GET, POST, PUT, DELETE	Error message	The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (ex. HTTP, FTP, LDAP) or some other auxiliary server (ex. DNS) it needed to access in attempting to complete the request
509	GET, POST, PUT, DELETE	Error Message	Bandwidth Limit Exceeded (not included in the W3C standard, implemented as part of the Apache tooling)
510 Not Extended	GET, POST, PUT, DELETE	Error message	The policy for accessing the resource has not been met in the request. The server should send back all the information necessary for the client to issue an extended request
511	GET, POST, PUT, DELETE	Error Message	Network Authentication Required (not included in the W3C standard)

API Documentation Standards

Outline or Description of the API

Should provide a clear view of the functionality of the API. Should contain summary of the API and its purpose and may inform potential users about the benefits of using this API over others.

Regardless of whether the API version is v1 or v2, the API specification should be thoroughly documented in Anypoint Exchange. A comprehensive set of Exchange documents should include:

- Solution Functionality

Provides an overview of the purpose of the API, why it was created, and a summary of the solution's capabilities.

- Solution Compliance

Provides a summary of the security mechanisms implemented by the API, an overview of enforced performance SLAs.

- Solution Access

Provides an overview of the processes for requesting access to the API.

- Solution Support

Provides the contact information for the team supporting the API as well as release notes detailing changes made to the API over time.

Endpoint URLs

Provide the actual endpoints of the API which the user can use to invoke the API.

Always specify the endpoints for dev, test, and prod environment of the API in your documentation.

Example:

API Endpoint : <https://gpapi.gp.com/v1/prc/asset-heirarchy/unit/>

DEV Endpoint: <https://dev.gpapi.gp.com/v1/prc/asset-heirarchy/unit/>

TEST Enpoint: <https://test.gpapi.gp.com/v1/prc/asset-heirarchy/unit/>

NOTE: These URLs should follow the [URL Naming Convention](#).

Authorization / Authentication

Authentication is how the provider keeps the API's data safe for developers and end users, and so it might have multiple authentication schemes. The API documentation explains each authentication method, so users understand how to access the API.

Ex: If your API requires the User to provide Client Credentials (client_id and client_secret) to authenticate, you should specify how one can get the client credentials and how those credentials can be used for authentication.

NOTE: Also provide details of process for requesting credentials and key management expectations.

High level Design or Diagram

Includes a high-level design diagram of the API including the source and the target systems to provide a better understanding of the API functionality.

Example for High level Design [prc-asset-heirarchy-api \(mulesoft.com\)](#)

API Asset Taxonomy

- Reduce the time & effort of users to find the right asset.
- An Asset Taxonomy is a hierarchical structure on which Exchange uses to make assets findable through searching and filtering.
- A consistent Asset Taxonomy should be the first step in configuring Anypoint Exchange

API Category & Tagging

Categories organize the assets into different groups to improve asset's browsing and discovery. Asset categories include.

- Business Function (Canonical/Domain API)
- API Type
- Data Types
- Template

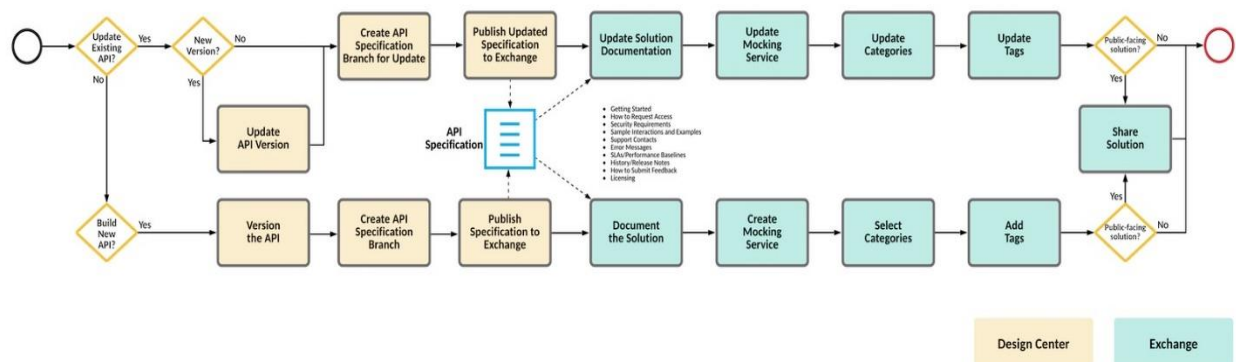
Category Name	Category Value	Asset Types	Asset Tag
Business Function	Procurement, Finance (L1 Taxonomy Name)	Rest API	Purchase pricing, Purchase Order (L2 Taxonomy Name)
API Type	Exp api, prc api, sys api, Utility API	Rest API, SOAP API, HTTP API	SAP, salesforce, Logger, incident
Data Types	Item, supplier, price	API Spec Fragments	fragment
Template	API Template, Service Template, Batch Process Template	Rest API, SOAP API, HTTP API	NA

Tag Recommendation

"tagging" is very similar to "taxonomizing" data, i.e. organizing and categorizing data for easy retrieval latter. In MuleSoft, once the API is published to Exchange, categorizing that API or the asset with a "tag" can help retrieve all APIs or assets that match the tag at a later time.

- If the API is in Business function category, tag name should be L2 taxonomy name e, g supplier, item etc.
- Tag name should be general acronym of API, system name. ex logger, incident, SAP.

Flow Diagram



Change Log

Modification	Author	Date
Initial creation	Avaya Mohanty	03/02/2023
Added design, documentation, and lifecycle standards	S Sathuryan; Tejas G C	03/15/2023
Updated Documentation page, API asset taxonomy	Avaya Mohanty	03/29/2023