API Design

Recommendations

<customer_name>

v1.0, 20xx-xx-xx

Table of Contents

Overview

The API is one of the most common means through which many developers interact with data. A well-designed API should be well-structured, consistent, and intuitive. The developer's experience is the most important metric in measuring the quality of an API. If the API is confusing and difficult to work with, it will not be consumed.

Mule APIs are founded on RESTful principles first introduced by Roy Fielding[1]. In a nutshell, APIs manage multiple logical resources using pre-defined HTTP actions where each method has a specific function in the management of the resource.

Resource

- A resource is an object or representation of something, which has some associated data with it and there can be set of actions to operate on it.
    - For example, employee, order and report are resources.
    - Delete, add, update are operations to be performed on these resources.
- A collection is a set of resources.
    - For example, employees is the collection of employee resource.
- A URL (Uniform Resource Locator) is a path through which a resource can be located and some actions can be performed on it.

Use Nouns, Not Verbs

Unlike traditional RPC-based web services, the URL should refer to the resource instead of referring to an action. Therefore, avoid using "actions" within your resource name.

- For example, getOrders or deleteOrder should be avoided.
- The action prefix should be implied by the HTTP method (i.e. GET or DELETE).
    - GET /orders
    - DELETE /orders/12345

Resource Granularity

Resources should be coarse grained. When accessing data for a resource, return all information as required to satisfy the request. Retrieving all data in a single call is much more efficient then making separate API calls.

---

[1] Fielding, Roy Thomas. "Architectural Styles and the Design of Network-Based Software Architectures." Architectural Styles and the Design of Network-Based Software Architectures, UNIVERSITY OF CALIFORNIA, IRVINE, 2000, www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

- Retrieves *all* addresses for customer 1234:
    - GET /customers/1234/addresses

Above single call to get all addresses for customer 1234 versus as below, individual call to get shipping address and billing address

- Retrieves each address for customer 1234:
    - GET /customers/1234/shipping-addresses
    - GET /customers/1234/billing-addresses

Use Plural Nouns

A common practice is to standardize on plural nouns over a mixture of both singular and plural nouns in URLs. This practice makes it consistent and predictable for developers.

- For example:
    - Retrieve all employees: /employees
    - Retrieve single employee: /employees/12345

Use Concrete Nouns

Avoid making a resource name too abstract. Tunneling a number of objects through an abstract resource name makes it difficult to understand what the resource actually represents or how it should be used.

- For example, consider the different types of resources: order, payment and shipment. Representing all these resources as a single /services resource is too abstract.

Naming Standards

Single-word resources should always be in lowercase:

- /customers
- /orders

For resources with 2 or more words, use kebab-case (i.e. hyphen-separated):

- /line-items
- /security-groups

For resource fields, use camelCase:

- /customers/1234/lastName
- /employees/1234/contactInfo

Resource States

Most resources have states. In certain cases, the state of the resource can only change if certain preconditions are met.

- For example, it should be possible to change the shipping address or cancel an order only if the order has not been shipped yet.
    - This implies that the API resources could have different states (e.g. an order could be created, shipped or cancelled).
- A good practice is to capture the various states of a given resource as well as the events that change the state of a resource in a State Transition Diagram.

Identify Actions

Identifying associated actions for each resource is the next step in the design process. When designing a RESTful API, the common starting point are the resource's CRUD actions (i.e. Create, Retrieve, Update and Delete). Similar to the "Nouns, not Verbs" rule for resources, candidate actions should be "Verb, not Nouns".

- For example, a customer should be able to create an order, to add products to that order and to retrieve the status of that order.
- Further, the customer wants to be able to update the shipping address of an order.
- Also, the customer may want to completely delete the order if it has not already been processed and shipped.

Map Actions to HTTP Verbs

On first glance, CRUD operations map seamlessly to standard HTTP methods:

CRUD Operation	**Description**	**HTTP Method**
Create	Creation of an instance of the resource.	POST
Read	Retrieve an instance of the resource.	GET
Update	Alters the state of an instance of the resource.	PUT (full update)
		PATCH (partial update)
Delete	Removes an instance of the resource.	DELETE

Upon closer inspection, there are subtle differences in which certain HTTP methods support POST, PUT and PATCH methods:

- Use PUT when a resource is updated completely through a specific URL.
    - For instance, to completely replace an existing order (order 1234567): PUT http://example.org/orders/1234567
    - The complete and updated version of resource must be placed in the body of the request.
- Use PATCH when a resource is updated partially through a specific URL.
    - For instance, to partially replace an existing order (order 1234567): PATCH http://example.org/orders/1234567
    - Only the desired changes to the resource's fields must be placed in the body of the request.
- Use POST if a new order is placed that doesn't already exist in the system. This method allows the target system to determine the location of the new resource.
    - POST http://example.org/orders
    - The response to the POST should include the URL location of the resource upon creation (e.g. http://example.org/order/1234568)

Some additional considerations:

- PUT and PATCH are idempotent, so if either PUT/PATCH are submitted twice, it has no effect.
- PUT *can* be used to create a resource with the same resource URL by simply overwriting it.
    - If the URL location refers to an already existing resource, the completely updated resource SHOULD be considered as a modified version of the one residing on the target system.
    - If the URL location does not refer to an already existing resource, the target system (or calling API) creates the resource with that location.

Beyond CRUD

In some cases, the desired action does not map cleanly to CRUD operations. There are several approaches to deal with this:

- Restructure the action to operate on a field of the resource. This works if the action does not use parameters.
    - For example, ship order could be mapped to PATCH that updates only the status "shipped".
    - PATCH http://example.org/orders/1234567
        - { "status": "shipped" }

- Use a sub-resource to capture the state that results from the execution of the action.
    - For example, the status of an order could be changed by creating a sub-resource shipped-order. Then, the state of the order state could be changed:
    - POST http://example.org/orders/1234567/shipped-order

- In some cases, it is impossible to map an action to a reasonable resource structure.
    - For example, a search action over a collection of multiple, different resources.
        - In such cases, it makes sense to define a "fake" resource search but this should be documented clearly to avoid confusion.
    - GET http://example.org/search?rsrc=order,invoice&term=fulfilled

HTTP Headers & Media Types

Where necessary, include the Content-Type and Accept headers with every request and the Content-Type with every response.

- The Accept request-header field specifies the media types that are acceptable for the response.
    - Accept: application/json
- The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient.
    - Content-Type: application/json

There are also other headers [2] that can be used with http request/response based on the requirement for the API.

HTTP Status Codes

Get to know your HTTP Status codes to provide the consumer with standard response messages on the status of their request. Ensure there is consistency for all HTTP methods (GET, POST, PATCH, PUT, DELETE) based on the desired operation, Below is a list of the most common HTTP Status Codes:

- 2xx - Successful

  This class of status codes indicates that the client's request was successfully received, understood, and processed.

[2] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

    - 200 Ok
        - Indicates the standard response representing success for GET, PUT or POST.
    - 201 Created
        - Indicates that a new instance is created for POST or PUT.
    - 204 No Content
        - Indicates that the request is successfully processed but has not returned any content (e.g. DELETE of a resource).
- 3xx - Redirection
  This class of status code indicates the client must take additional action to complete the request.
    - 304 Not Modified
        - Indicates that the client has the response already cached and thus the request does not need to be re-processed.
- 4xx - Client Error

  This class of status codes is intended for cases in which the client has erred.

    - 400 Bad Request
        - Indicates that the request by the client was not processed since the request was not valid (i.e., malformed request, missing/incorrect payload, etc.).
    - 401 Unauthorized
        - Indicates (*ironically*) that the client is not allowed to access resources due to missing/bad authentication credentials.
    - 403 Forbidden
        - Indicates that the request is valid and the client is authenticated, but the client is not authorized to access the resource.
    - 404 Not Found
        - Indicates that the requested resource is temporarily unavailable.
    - 410 Gone
        - Indicates that the requested resource is permanently unavailable and has been intentionally removed.
- 5xx - Server Error

  This class of status codes is intended for cases in which the server has erred or is incapable of processing the request.

    - 500 Internal Server Error
        - Indicates that the request is valid but the server is unable to process the request due to an unexpected condition.
    - 503 Service Unavailable

    - Indicates that the server is down or unavailable to receive and process the request (i.e. undergoing maintenance).

Below is a list of excellent resources for using HTTP Status Codes in a RESTful environment:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/errorcodes.htm
- https://httpstatusdogs.com

Querying the Data

All of the below actions provide a consistent means to query the resource data.

Filtering

The API consumer may only need a subset of a collection of resources. The ability to filter the data can be accomplished by using query parameters. For example, to get the list of all shipped orders, the API consumer could use:

http://api.mulesoft.com/orders?status=shipped

In the above example, status is query parameter used to implement order filtering. Or to get the list of all shipped orders to a given state, the API consumer would specify:

http://api.mulesoft.com/orders?status=shipped&state=CA

Sorting

Similar to filtering, a generic query parameter (sort) could be used to describe sorting rules. The below example returns all of the orders sorted by ID:

http://api.mulesoft.com/orders?sort=orderId

To allow sorting on multiple fields, the query parameter could be designed to take a list of fields instead of a single value.

http://api.mulesoft.com/orders?sort=orderId,date

In the above example, the sort query parameter provides orders that have been sorted first by ID and then by date.

To support ascending/descending sort order, each field of the query parameter can be prefixed with a plus (‘+’) to denote an ascending sort order or a minus (‘-’) to denote a descending sort order. For example, the below request returns all orders sorted by date (descending) and then by product (ascending).

http://api.mulesoft.com/orders?sort=-date,+product

Another alternative is to use the ‘_asc’ or ‘_desc’ suffix to denote the sort order. Using the same parameters above, the request would be:

http://api.mulesoft.com/orders?sort=date_desc,product_asc

Searching

The API consumer may wish to search across the resource data. The ability to search can be accomplished by using a search query parameter. For example, to get all employees with ‘Mason’ in their name, the request would look like:

http://api.mulesoft.com/employees?search=Mason

Pagination

For the GET method that returns a large collection of data, pagination is necessary to ensure good performance and user experience.
Without pagination in place, a GET method may return hundreds or thousands of records causing heavy network traffic and eventually the session may timeout, rendering the API unusable.
The pagination is implemented using a technique called Offset Pagination. This technique originates from the SQL world which already have LIMIT and OFFSET as part of the SQL SELECT Syntax.

An example of the Offset Paging would look like:
GET /orders?limit=20&offset=10

In this ex, the GET method would return 20 rows starting from the 10th row.
The corresponding SQL statement would be something like:
SELECT * FROM orders_table LIMIT 20 OFFSET 10;
A typical usage is shown below:

1. Client requests for 20 most recent items:
   GET /orders?limit=20
2. To go to the next page, the client makes the second request:
   GET /orders?limit=20&offset=20
3. To go to the next page, the client makes the third request
   GET /orders?limit=20&offset=40

One could codify the pagination as a trait which can be included in your RAML, as shown in the example below:

pageable.raml

```
#%RAML 1.0 Trait
  usage: Apply this trait to a GET method that supports pagination.
  queryParameters:
    offset?:
      type: integer
      default: 10
    limit?:
      type: integer
      default: 50
```

Partial Resources

The consumer might only need specific fields within a given resource. To obtain a partial resource the API URL could be designed to take a list of fields as a query parameter and return only the fields included in that list.

For example, the following request will return only the date and the total of the purchase order: GET /orders/1?fields=date,total

http://api.mulesoft.com/orders/12345?fields=date,total

Aliases

The ultimate goal of any API is to make its invocation as effortless as possible for application developers. If there exists a common pattern of usage for the API that requires considerable configuration, the API could encapsulate this pattern of usage into an easily accessible URL.

For example, to return only the zip codes and totals associated with orders that include a specific SKU shipped in the past 24 hours, the API could become rather complex:

http://api.mulesoft.com/orders?fields=zipcode,total&include=4011200296908&to=2017-10-02T13:15:30Z&from=2017-10-01T13:15:30Z

A simple refactoring can provide a much friendlier endpoint:

http://api.mulesoft.com/orders/past-day?include=4011200296908

A resource name should remain short in order to avoid any size limitations. The base URL should also contain no more than 2-3 resources if possible. URIs can be limited in some HTTP stacks.

API URIs

The structure of a URI is central to how APIs are organised and categorised within your enterprise domain. A good URI taxonomy helps to categorise your APIs across functional domains, regions, and relationships (hierarchical) between them. A good URI also helps to govern the lifecycle of your API through versioning practices.

The recommended URI Structure is shown below:

External API:
https://${domain}/api/${context}/${version}/${resources}/${resource-id}?${queryParams}

ex:
https://mulesoft.com/api/logger/v1/entries
https://mulesoft.com/api/logger/v1/entries/123?sort=+timeCreated

Internal API:
https://${domain}/api/${layer}/${context}/${version}/${resources}/${resource-id}?${queryParams}

ex:
https://mulesoft.com/api/sys/logger/v1/entries
https://mulesoft.com/api/sys/logger/v1/entries/123?sort=+timeCreated

Part	Description	Example
${domain}	The domain of the enterprise	mulesoft.com
${layer}	The layer of the API according to API-led connectivity approach. The possible values are sys (System), prc	sys, prc, exp

	(Process), exp (Experience)	
${context}	The name of the API. This typically represent a business or utility service and should be a short but descriptive name	logger
${version}	The version of the API. Depending on requirements, the version can reflect only major versions or include a more hierarchical convention to identify minor versions	v1, v2
${resources}	The name of the resource that represents the actual object. An API may contain one or more resources. The resource can also be referred to as the API endpoint	entries
$resource-id}	The id of the resource to be fetched/updated. The resource id is optional and it is not applicable when we do a POST	12345
${queryParams}	Query parameters/strings are often used to carry identifying information in the form of "key=value" pairs	page=1&sort=+timeCreated