SnapLogic REST API Design - Best Practices

V 1.0

Roberto Oliva

Introduction

This document provides a guide on best practices for REST API design within the SnapLogic Platform.

The REST API design guidelines are a collection of API design patterns and principles that all API teams within an organisation should adhere to when developing APIs. The definition and implementation of API design guidelines are among the most influential drivers of an API strategy — fostering a consistent approach to the creation of an API platform across the enterprise.

The JSON:API, a specification for building APIs in JSON, is the foundation for these guidelines. Whilst the focus is on JSON APIs, guidelines for other media types have been included in this document.

1. General API Design Best Practices

This section describes the rules and recommendations that need to be followed along the entire API lifecycle of an API. The principles are the following:

- **API Design first approach**: always start designing the API contract first in close collaboration with your stakeholders. Using a design first approach allows faster development and early feedback from an API consumer. It is imperative to get the contract right to have stable and long-lasting APIs. All API contracts should be described following the OpenAPI/Swagger specification format (the Swagger editor can be used to create the API contract)
- **Message format**: there are many message formats available for APIs (XML, JSON or YAML), however JSON is the most common one for REST APIs
- **Data types**: ensure to represent correctly common data types such as date and time, strings, boolean, numbers, language codes, country codes or currencies in the OpenAPI/Swagger specification
- **Common operations**: almost all non-trivial APIs offer features like pagination, search, long running tasks, batch operations, query requests with many parameters, localisation, or rate limiting. It's important that all APIs offer these features in a consistent manner. The potential to cache responses and excel in performance optimisation is directly related to consistency in common operations

- **API Versioning**: any API modification must try to maintain backward compatibility and avoid breaking changes. URIs, naming conventions, and upgrading from a minor to a major version (which is a breaking change) need to be kept in mind during the API design and deployment phase
- **Secure by design**: it is important to design APIs that are inherently secure. OpenAPI 3.0 allows to describe APIs protected using the term **security scheme** for authentication and authorization schemes
- **Publish API specifications to the Developer Portal**: it is important to deploy an API specification to the Developer Portal as soon as it is ready. This will make the API discoverable and ready to be consumed/reused either privately within the organisation, or publicly to third-party consumers. If an API specification is not deployed to the Developer Portal, it will not be consumed (it will be invisible), leading to potential duplications if another team/business unit creates the same API all over again
- **API Tags:** when deploying an API specification to the Developer Portal, choose tags that accurately describe the purpose or category of each endpoint or resource in the API. Use clear and concise language to convey the functionality or domain of each tag. Organise tags in a hierarchical structure to reflect the logical relationships between different endpoints and resources. Use parent-child relationships to group related endpoints under broader categories or domains. Tags can be leveraged for searching and filtering endpoints in the developer portal, however limit the number of tags as this can lead to clutter and over-confusion in the documentation
- **Design for reuse**: an API-first culture focuses efforts on digitising an organisation's capabilities using APIs. Each time a new solution is built, it starts with designing and building one or more APIs to support it. Teams are encouraged to design APIs with the intent of delivering reusability. Over time, the portfolio of APIs continues to grow. Designing reusable APIs allows to save time by speeding up development, composability, optimise performance, and make tracking data a much simpler process.
- **CI/CD**: as part of the platform enablement function and full API lifecycle, it is important to enable API automation deployment using CI/CD toolchain, git repository layout, testing framework, etc.

2. Design Guidelines

The following words are used to signify the requirements in these guidelines:

- **Must**, this is required.
- **Should**, this is recommended and a deviation from this must be agreed by the Centre of Excellence (CoE) agreement during Peer Review in the API Design phase.
- **Could**, this is optional.

2.1. REST

REST is an acronym for Representational State Transfer. It is an architectural style defined by Roy T. Fielding and is derived by applying a set of constraints to the elements of a system, so the system exhibits certain properties such as loose coupling and scalability. REST is **data-orientated**, "the nature and state of an architecture's data elements is a key aspect of REST" - Fielding.

2.2. JSON:API

"JSON:API is a specification for how a client should request that resources be fetched or modified, and how a server should respond to those requests." JSON request/response APIs **must** follow the JSON:API and **should** follow the JSON:API Recommendations, unless explicitly overridden below.

The JSON Schema definition for JSON:API is at http://jsonapi.org/schema. The JSON Schema format is specified at http://json-schema.org. JSON:API FAQs can be found at https://jsonapi.org/faq/

2.3. Resources

Resources are an abstraction of information: a thing, an entity, an event or any information that can be named and are identified by URIs (Universal Resource Identifier). Resources are often real-world entities such as customers, invoices, accounts, etc. The relationship between resources are typically nested, for example each customer might have multiple accounts, each account has multiple invoices.

2.3.1. URI format

URIs **should** be easily understood with plain English words. Resources **must** be named as nouns:

- For URI format, the forward slash (/) used in the path portion represents the hierarchical relationship between resources. For example: ***https://api.example.com/v1/employees/{employeeId}***
- Hyphens (-) **should** be used to improve the readability of a URL, i.e.: ***https://api.example.com/v1/employees/{employeeId}/recent-pay-slips***
- URI **should** be in lowercase letters.
- File extensions **should not** be included in the URLs.
- Avoid Nested Resource Paths - minimise the depth of URI paths by avoiding excessive nesting of resource paths. Instead, use query parameters to filter or refine resource collections.
- Maintain consistency in URI formats across different endpoints within an API to make it easier for developers to understand and use the API

2.3.2. Archetypes (Collection, Item, Processor, Stores)

API Resource archetypes serve as a design pattern model that can be followed to design the structure and behaviour of each API endpoint. The four basic archetypes in the API resources are: collection, document, store and controller. Each endpoint **should** only be aligned to one type to ensure separation of responsibilities and uniformity.

2.3.2.1. Collection

A collection resource is a list of resources of zero or more of the same resource type. The collection resource **must** be plural. For example:

https://api.example.com/v1/users

2.3.2.2. Document (Item)

A document resource is a singular object instance or a database record that typically includes both fields with values and links to other related resources. For example:

https://api.example.com/v1/users/{userId}

2.3.2.3. Controller (Processor)

A controller resource model is a procedural concept that cannot be mapped to create, retrieve, update and delete. The controller is a thing which is named by a noun with a task or a process that is performed by a controller. A POST to a controller will perform a task or operation, such as performing computations or complex operations atomically. Even though the controller performs some action, the controller is a thing and the resource is still named as a noun. For example:

POST https://api example.com/v1/users/{userId}/password-reset

2.3.2.4. Store

A store resource archetype is used for a client managed resource repository. It allows the client to perform CRUD action where client chooses the resource ID/URI. This archetype is not commonly used, since collection type entities are usually more appropriate and sufficient in most cases.

GET https://api example.com/v1/users/{userId}/favourite-movie
PUT https://api example.com/v1/users/{userId}/favourite-movie/beetlejuice

2.3.2.5. Query Design

Snake case **must** be used for query parameters (not camelCase). For example:

***GET https://api.example.com/v1/users?id_type=Student*&university_degree=Engineering**

Querying for information can include reducing or ordering of collections, or reducing the number of data fields returned in the results by using query parameters in the URI. Those are **filtering**, **sorting** and **sparse fieldsets**. Filtering is the process of selecting a subset of records based on some criteria. Sorting is a process to arrange the results in the response and sparse fieldset is the process of selecting certain fields in each entity to be included in the results.

2.4. Representations

Representations are the information retrieved from or sent to a resource. A representation can be more than one media type, i.e. a resource could be represented in HTML, XML, JSON, etc. A representation is the current resource state containing data and possibly URIs.

JSON representations **must** be documents that follow the JSON:API specification and have the structure specified in Document Structure which is identified by the media type "application/json". Although the same media type is used for both request and response documents, certain aspects are only applicable to one or the other.

2.4.1. Message Body Format

A JSON object **must** be at the root of every JSON:API request and response containing data, this object defines a document's "top level". A meta object **should** be included in a JSON request and response and follow the JSON:API Meta Information, as well a "top level" response document can contain a "**hateoas-links**" element as defined in the HATEOAS section of this document.

2.4.1.1. Service Requests

2.4.1.1.1. GET

A GET request fetching a JSON representation must follow the JSON:API Fetching Data. GET request URI query parameters should adhere to:

- Sorting **must** follow JSON:API Sorting and the sort query parameter value **should** use the attribute names
- Filtering **must** follow JSON:API Filtering. One or more filter[resource-attribute] parameters can be applied
- Sparse fieldsets **must** follow JSON:API Sparse fieldsets
- Pagination **must** use the query parameters specified in the JSON:API Pageable.

If the number of query parameters goes beyond 5, a POST Controller **should** be used instead of GET.

For example, a URI that returns a list of users:
GET https://api.example.com/v1/users

To rearrange the list by name in the alphabetical order:
GET https://api.example.com/v1/users?sort=name

To list all users that have the name call "James":
GET https://api.example.com/v1/users?filter[name]=James

2.4.1.1.2. POST

A resource can be created by sending a POST request to a URL that represents a collection of resources. A POST request of a JSON representation **must** follow the JSON:API Creating Resources. Note that the POST method is not idempotent, that is if the POST is repeatedly called with the same data, then new resources will be repeatedly created and create duplicate resources. If this is an issue, then the API needs to put in the necessary mechanisms to prevent duplication.

2.4.1.1.3. PATCH

A resource can be updated by sending a PATCH request to the URL that represents the resource. A PATCH request of a JSON representation **must** follow the JSON:API Updating Resources.

2.4.1.1.4. PUT

A PUT, as defined by the REST specification, can be used to *update* a representation by **completely** replacing the existing state.

To update a partial JSON representation, a PATCH **must** be used rather than PUT. This will maintain backward compatibility for clients.

2.4.1.1.5. DELETE

An individual resource can be deleted by making a DELETE request to the resource's URL. DELETE requests **must** follow the JSON:API Deleting Resources.

2.4.1.2. Service Responses

2.4.1.2.1. GET

GET response of a JSON GET request **must** follow JSON:API Get Responses

2.4.1.2.2. POST

POST response of a JSON POST request **must** follow JSON:API POST Responses

2.4.1.2.3. PATCH

PATCH response of a JSON PATCH request **must** follow JSON:API PATCH Responses

2.4.1.2.4. DELETE

DELETE response **must** follow JSON:API Delete Response

2.4.1.3. Error Representation

An error response **must** follow the JSON:API Errors along with the error responses for service requests above.

2.5. Hypermedia

2.5.1. Linked Data

Hypermedia in JSON responses **should** be implemented using JSON:API Links. Links are used in JSON:API Relationships, Resource Linkage and Related Resource Links. Relationships should be fetched using JSON:API Fetching Relationships.

2.5.2. HATEOAS

Hypermedia as The Engine of Application State (HATEOAS) is a constraint of the REST architecture. To implement this constraint, each API must contain resources and verb actions that have direct relationship with the API. The JSON:API does not cover HATEOAS. To use HATEOAS, the new *top-level* member "**hateoas-links**" **should** be used containing an array of objects that have "href", "rel" and "method" members, for example:

```
{
  …
   "hateoas-links":[
      {
         "href":"api.example.com/v1/accounts/1/transactions",
         "rel":"transactions",
         "method":"GET"
      },
      {
         "href":" api.example.com/v1/accounts/1/billingInfo",
         "rel":"billing",
         "method":"GET"
      }
   ]
}
```

2.5.3. Link Headers

Link headers **should** be used only when the representations are in binary format such as images, rich-text documents, PDF, spreadsheet etc. Link headers **should not** be used for JSON resources.

Link Header format:
Link: <{URI}>;rel="{relation}";type="{media type}";title="{title}"

Example Response:
HTTP/1.1 200 OK Content-Type: image/jpeg
Link: <https://api.example.com/v1/users/112/imagesInfo/img324>;rel="alternate;type="application/json"

2.6. Versioning

In general, there isn't a golden rule for versioning an API, but the version number depends on a case by case scenario. An API version can contain both a major and minor version, i.e. v1-0, v1-1, v1-2, etc.

The following key points can be taken into consideration when versioning an API:

1) **Minor versions are backwards-compatible**: this means that if a new resource is added to the API or optional fields are added as part of the input/output data model, they won't break the previous consumer implementation as the consumer will see those changes transparently.

2) **Major version changes indicate breaking changes**: an API major version change means that either the data model has changed (because some fields have been made mandatory or their type has changed) or some resources have been deprecated. In this case, the consumer will need to use the new version of the API.

For simplicity, a new SnapLogic API version (and corresponding Pipelines) **should** only be created if the API can no longer maintain backward compatibility.

2.7. HTTP

2.7.1. HTTP Status Response Codes

HTTP status response codes for JSON:API were defined previously in the Service Responses section of this document. For other media types, the common HTTP response codes below **should** be used.

Code	Status	Description
200	OK	Success code with response body content.
201	CREATED	Successful resource creation status. Set the Location header to contain a link to the newly created resource. Response body content may or may not be present.
202	ACCEPTED	The request has been accepted for processing, but the processing has not been completed.
204	NO CONTENT	Success code without expecting a response body content.

304	NOT MODIFIED	When the same URI has been requested in the past and the result has been cached. When the client sends another request with the head **If-Modified-Since** or **If-None-Match** header. If the content still has the same Last-Modified or ETag, a 304 **should** be returned without a response body to tell the client the resource content stored on the client is still current.
400	BAD REQUEST	This status indicates that the data payload does not fulfil the requirements expected from the API specification. This is typically caused by invalid key value pairs or/and missing or broken structural parts in the request body.
401	UNAUTHORIZED	Error code for a missing or invalid authentication token.
403	FORBIDDEN	When the user does not have the authorisation to access a certain resource due to insufficient role or due to time constraint.
404	NOT FOUND	Resource not found
405	METHOD NOT ALLOWED	The requested method is not supported for the requested resource
406	NOT ACCEPTABLE	This error response code indicates that a response matching the list of acceptable values defined in Accept-Charset and Accept-Language cannot be served
409	CONFLICT	This status is normally used for indicating conflicts in the request. This happens when multiple clients are trying to update the same record creating conflicts in versioning.
429	TOO MANY REQUESTS	The user agent has sent too many requests in a given amount of time.
500	INTERNAL SERVER ERROR	The general catch-all error when the server-side throws an Exception.
502	BAD GATEWAY	The server was acting as a gateway or proxy and received an invalid response from the upstream server
503	SERVICE UNAVAILABLE	The server cannot handle the request (because it is overloaded or down for maintenance). Generally, this is a temporary state
504	GATEWAY TIMEOUT	The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.

2.8. Extensibility

Extensibility is the ability to manage changes in the API without breaking the existing client applications that used the same version.

2.8.1. Maintain URI Compatibility

A URI **must** be kept with the same version for a query API when adding new parameters, to continue to honor existing parameters. The new parameter **must** be optional.

When changing the format of a query parameter value, add support for both value formats, for example the parameter *lastupdated*:

https://api.example.com/v1/users?dept=sales&lastupdated=2017-01-10Z
https://api.example.com/v1/users?dept=sales&lastupdated=1492482295

2.8.2. Maintain Representation Compatibility

To maintain backward compatibility, elements in the returned representations **must** not be removed and the hierarchical structure **must** be maintained.

2.8.3. Client application design recommendations

Client applications **must** be designed to parse data by name/value pair for JSON (Attributes for XML) and not via position and order of the data returned. A client **must** not fail if it finds unrecognised data. Clients ignore properties that they don't know for backward compatibility. A client **must** use PATCH to update representations as the client may have ignored properties.

2.9. API Patterns

2.9.1. Synchronous API

A synchronous API is the simplest and most common REST API, where the entire pattern is fulfilled by a few steps/invocations from the underlying SnapLogic Pipeline(s), and the response is returned to the API consumer with the corresponding HTTP status code.

This pattern is NOT recommended for longer lived orchestrations. Longer lived orchestrations are orchestrations where the invocation of multiple activities may require a time that is much larger than the time that the original consumer is willing to wait. For this scenario, an asynchronous API is required, as described in section 2.8.4.

2.9.2. Synchronous API with Caching

This pattern is very similar to the pattern in the previous section with one main difference. In this case, the API is caching the result of the processing for storing and reusing frequently called data. Using caching reduces the processing load on the SnapLogic APIs/target systems and increases the speed of message/payload processing.

The caching mechanism is applied in SnapLogic API Manager component using the **HTTP Response Cache** policy. The policy owner can configure the cache time-to-live (TTL) per policy and the cache key used to cache each request (with every **key-value** pair).

2.9.3. Synchronous API with Orchestration

This pattern is very similar to the pattern in section 2.8.1 with one main difference. In this case, the API is orchestrating the underlying business logic rather than providing a simple abstraction layer. The orchestration steps consist of invoking multiple APIs/backend systems (either in sequence or in parallel) and finally aggregating all the invocation responses into a JSON message, before returning it to the API consumer.

2.9.4. Asynchronous API

An asynchronous API is still exposed as a REST API, and it is responsible for the reception of the message/payload from the source system/consumer to be processed in a fire and forget fashion.

The received message/payload from the API consumer is placed on a queue/topic within the Messaging Layer (i.e. JMS, Kafka, etc.). The configuration of the queue/topic will allow the message/payload to be received by multiple consumers subscribed to the queue/topic. Queue processing will also ensure **Guaranteed Delivery**.

The API then provides a progress of the processing returning the HTTP status code **202 (Accepted)**, along with a HTTP **Location** header that provides the resource to query for the processing status. Whilst the processing is in progress, the API **should** continue to return **202**. Once the processing is completed, the API should return **302** with a **Location** header containing the new resource.

2.10. API Naming Conventions

All the SnapLogic APIs should follow the below naming convention:

<<organisation/business unit-name>>-<<api-name>>-<<api-domain/layer>>**-api**

For example:

- siemensshs-salesforce-sys-api
- siemensshs-patientmanagement-proc-api
- siemensshs-servicenow-sys-api

NOTE 1: the <<api-domain/layer>> is optional. In the above examples, the following API layers have used:

- **sys** – to indicate a System API (i.e. an API that is abstracting connectivity to a particular system, i.e. Salesforce)
- **proc** – to indicate a Process/Orchestration API (i.e. an API that is performing a business process by orchestrating/composing other APIs)

NOTE 2: SnapLogic APIM automatically appends the configured API version in the Manager to the base URL, so there is no need to add a version to the API name. For example, using the first case above, the URL will look like: **/apim/siemensshs-salesforce-sys-api/v1**

2.11. API Performance Testing

As a best practice, SnapLogic recommends that all APIs undergo performance tests as part of the promotion to the Production environment. This is to ensure that each API fulfils its SLAs and the platform itself can deliver the performances required.

SnapLogic recommends using K6 for performance testing. K6 is a developer-centric, free and open-source load testing tool built for making performance testing a productive and enjoyable experience. Using K6, it is possible to catch performance regression and problems earlier, allowing to build resilient systems and robust applications.