API Producer Guidelines

These guidelines are provided to properly document, design and implement a REST API.

They are organized into 5 **themes** as point of views that will be the focus of distincts peoples.

Technical rules focus on **interoperability** from any kind of application able to consume a REST API, and that you handle **changes** in a way that **will not unexpectingly break your consumers**.

> **Hint**
>
> Not all these guidelines apply to your perimeter !
>
> see SG Group API Guidelines and SGM API Wholesale Guidelines first.

Principles¶

Here are a few overall principles to follow when developing an API. In a few words: make the value of your API, its behavior and its usage clear.

Design your API first¶

Follow these usages:

- Define APIs contract first, before coding its implementation, using OpenAPI as specification language. You can use a dedicated tool or do it from code. Keep in mind that working from code implies some cost (creating the project...) and will make participating more difficult for some less technical people.
- Design your APIs consistently with this guidelines; use our Swagger Design Check Service for automated rule checks.
- Call for early review feedback from peers and client developers.
- Once code is implemented, generate OpenAPI spec from code and continue to check its design.

API-100 manual **Use Design First approach**
Early Design with multiple contributors (as first consumers) and with internet usage in mind even if development not started — Principles

Design your API for consumers¶

The team exposing an API must completely master the exposed interface formalized by the OpenAPI specification, and ensure it is easily understandable and usable by any consumer using any technology.

Your consumer must not be impacted by, nor be aware of, the internal technologies you used to build your API.

This includes - but is not limited to:

- keeping up-to-date use documentation and information in the Developer Portal
- keeping the API model agnostic of internals
- following naming and JSON conventions
- abiding by REST design standards
- Ensure existing consumers wont break on changes

API-101 manual **Design for consumers** Interface is designed for consumer usage without exposing anything related to the internal implementation	Principles

Documentation¶

Document your API's data and operations (interface contract) using OpenAPI Specification.

To make it easier to understand your API, for functional or business people and for developers that are less experienced in using such API, you need to expose enough information on the various items composing your API.

Always match the running code¶

The OpenAPI specification is valuable if and only if it is always exactly matching the latest running code on the same environment.

Also the OpenAPI Specification will be used as an official contract for some security use cases (like internet exposition of an API).

So it is very important that the OpenAPI Specification always matches the running code.

To ensure this, best way it to generate the documentation from the code itself, and expose it at the same moment when new code is released to an environment.

See also Define Format for Type Number and Integer

> **Important**
>
> For automated rule API-249 to properly detect your usage of SGAPI Stack on swagger generation, you must :
>
> - Use SG Monitoring for APIs (zipkin) with the most recent SGAPI Stack In Java you should use the latest available version matching your Spring Boot minor version or (3.4.2, 3.9.2 or >= 3.13.0)
> - Connect to SGMON with your own API's credential = from the credential tab within your API Card in the API Catalog, not from a Client App
>   - As temporary solution, you may also well declare the apiCode and apiVersion fields in your SGMON/APIs stack config

API-248 manual **Expose OpenAPI Specification matching the code** Generate your OpenAPI specification from code and code annotation, expose it directly to Catalog for crawling, or at least push it at each change	Documentation
API-249 auto-catalog-check **Use the SG API stack to generate the OAS/Swagger json spec** For proper detection : use latest SGAPI stack (ex: java 3.11) and connect to SGMON with a credential from your API card's credential tab (not from a client app)	Documentation

Synchronize OpenAPI information with API catalog one¶

The information that are present both in the API catalog and in the OpenAPI Specification should be the same.

- version : major version must be same
- title : should be same
- contact : should match “Support contacts”

API-251 auto-swagger-check **Ensure similar information in API catalog and specification matches**
meta info (title, version, description, contact) should match between catalog and swagger if provided in swagger (API-242 must be GREEN, else this rule may stay Pending) Documentation

Versioning¶

Ensure as much as possible the backward compatibility of your API to reduce the need for new major versions (breaking changes), and make the breaking changes events very explicit and announced to your consumers.

Ensure existing consumers wont break on changes¶

When doing changes on your API, all existing consumers must continue to run as usual.

Consumers usually have independent release lifecycles, focus on stability, and avoid changes that do not provide additional value. APIs are contracts between API providers and API consumers that cannot be broken via unilateral decisions.

As long as there is no mandatory need to impact functionally all existing consumers, any changes to an API should be done in a backward compatible way as much as possible. This require a good initial design, meant to be stable as long as possible.

Just as mentioned in Use Semantic Versioning, such changes are considered a **minor** update.

There are two techniques to change APIs without breaking them:

- follow rules for compatible changes (see Apply only compatible changes on minor upgrades)
- If compatible change is not possible, introduce new API versions and still support older versions (see Support multiple major versions)

Put major version in URL¶

You have to put the major version of your API into URL. So that there will be no risk of impacts towards existing clients, and clients know explicitly which major version of your API they use.

As mentioned in Use Semantic Versioning, the major version of an API represents an incompatible change.

An incompatible change is a change that prevents an existing client to work, either expected items (fields/operations) do not exist anymore, or new mandatory items have to be managed by the client in its functional logic.

At the implementation level and SLA, a new major version can be considered as a completely different service. But the existing client still consider it to be the same one at functional level.

Good examples

```
https://foo.world.socgen/api/v1/items/
https://foo-api.world.socgen/v1/items/
https://foo-api-v1.world.socgen/items/
```

Version	Verb and Resource	Response	Description
v1.0	GET /api/**v1**/deals/12457	{ name:'foo' }	clients from v1.0 uses 'name'
v1.1	GET /api/**v1**/deals/12457	{ name:'foo', desc:'bar' }	client from v1.0 ignores 'desc', clients from v1.1 can use 'desc'
v2.0	GET /api/**v2**/deals/12457	{ deal: { name:'foo', desc:'bar' }, other:'qux' }	only clients from v2.0 can use it, clients from v1.0 and v1.1 must stay on v1 which is still alive parallely to v2, until the planned deprecation

Incorrect

```
openapi: 3.0.1
info:
  version: 1.2
servers:
  - url: "https://inter.net/api"
paths:
  /v1.0/pets/cats/{cat_id}: {} => v1.0 is not correct
  /v2/pets/cats/{cat_id}: {} => v2 is not coherent to info.version 1.2
```

Correct

```
openapi: 3.0.1
info:
  version: 1.2
servers:
  - url: "https://inter.net/api"
paths:
  /v1/pets/cats/{cat_id}: {}
  /pets/v1/cats/{cat_id}: {}
```

You may put version directly at server

```
openapi: 3.0.1
info:
  version: 1.2
servers:
  - url: "https://inter.net/api/v1"
paths:
  /pets/cats/{cat_id}: {}
```

API-257 auto-swagger-check **Put major version in url**
Present in OpenAPI Specification (server URL or all operations) (API-242 must be GREEN, Versioning else this rule may stay Pending)

Use Semantic Versioning¶

OpenAPI allows to specify the API specification version in #/info/version.

This version should be complied to Semantic Versioning 2.0 rules 1 to 8 and 11 restricted to the format <MAJOR>.<MINOR>.<PATCH> for versions as follows:

- Increment the MAJOR version when you make **incompatible** API changes after having aligned this changes with consumers,
- Increment the MINOR version when you add new functionality in a **backwards-compatible** manner, and

- Optionally increment the PATCH version when you make **backwards-compatible** bug fixes or editorial changes not affecting the functionality.

The MAJOR number must be the same as in the url as mentioned: Put major version in URL

This full version number should also be the same tracked in logs.

The other part of the version number provides information that can be useful for both client and API teams for diagnostics purposes.

Incorrect:

```
openapi: 3.0.1
info:
  version: 1 => not MAJOR.MINOR(.PATCH)

openapi: 3.0.1
info:
  version: v1.0 => no "v" before version

openapi: 3.0.1
info:
  version: version1 => no "version" before version and not MAJOR.MINOR(.PATCH)
```

Correct:

```
openapi: 3.0.1
info:
  version: 1.3.7
  <...>
```

This version should match the URL of your API.

API-258 auto-swagger-check **Keep versioning coherent**
Keep coherence between url version and version in metadata (API-242 must be GREEN, else this rule may stay Pending) Versioning

See also Put major version in URL

Limit frequency of major versions¶

When changing your APIs, do so in a compatible way and avoid generating additional API versions. Adding a new major version should occur rarely and only for explicit strong reasons.

A new major version is a distinct information within the API Catalog that you have to declare on your API. It will also impact your API endponints (see Put major version in URL). All your clients will have to subscribe to the new API version and adapt URL.

See also Design your API for consumers

API-260 manual **Limit frequency of major versions**
New major versions are done only when strictly needed Versioning

Support multiple major versions¶

As mentioned in Limit frequency of major versions, you should try your best to avoid major versions.

If sometimes you have no other choice but to upgrade your API on a new major version, do not force all your existing clients to switch to this new major version at same time. So you have to support at least two major versions of your API in parallel for a period of time.

It is highly recommended to not have more than 2 majors versions as it may be difficult to maintain. Multiple versions can significantly complicate understanding, testing, maintaining, evolving, operating and releasing our systems (supplementary reading).

The new major version may be :

- Supported in parallel with the old one within the same technical component (microservice)
    - Ideally the new major version interface then usually become the main one, and a mapping translates the old one to the new one
- As a distinct technical component (microservice), usually sharing same backend/database as previous one

API-261 manual **Support multiple major versions**
Leave time for consumers to switch to new latest major version — Versioning

Warn consumers of major version shutdown¶

If you are implementing a new major version of your API (even though you should try to avoid/limit major versions), make sure that all existing consumers know the deadline before which they have to upgrade.

At least even if a consumer is not impacted by the breaking changes and is not interested in using the newly provided features, it has to change the base url of your API in its configuration because the major version is always contained in url (see Put major version in URL).

Generally, a consumer requires time to prepare the upgrade because of the budget concerns or because it may have a different planning of develop and release cycle.

You should help existing consumers to migrate to the latest major version (i.e. by providing a migration manual) in order to make the period with parallel versions as short as possible.

So you must notify or warn your clients ahead of time. You can use the list of subscriptions on your API as an entry point for this.

Ex: at least one year between major versions seems to be a good period, allowing all your consumers to handle potential budget needs, when managed yearly.

Meanwhile you may have to support both previous and new major version in parallel.

See also: Support multiple major versions

API-262 manual **Warn early about deprecated version removal**
Your clients are warned of version decommission early enough so they can implement changes to support new version — Versioning

Apply only compatible changes on minor upgrades¶

API designers should apply the following rules to evolve RESTful APIs for services in a backward-compatible way:

- Do not remove/rename a field or query parameter.
- Add only optional fields, never mandatory ones.
- Do not remove existing resource/operation or change its path.
    - But you can add new resources/operations.
- Never change the semantic of fields (e.g. a creation datetime changing from user request datetime to backend operation datetime).
- Never change the validation/constraints logic to be more restrictive and make sure that all constraints are clearly defined in description.
- Enum range **cannot be reduced** when used as **input** parameters — as existing clients will provide old values that server will reject.

  But Enum range **can be extended** when used for **input** parameters — as server can accept more values than existing clients will provide.

- Enum range **cannot be extended** when used for **output** parameters — as existing clients cannot handle new values.

  Enum range **could be reduced** when used as **output** parameters as existing clients can handle more values that server will provide. This will not break the contract but it could impact the functional behavior of a client application depending on the context.

If an enum range is used for output parameters and likely to be extended but without impacting client logic, define instead a property of type string with x-extensible-enum to specify an open-ended list of explicit values as mentioned in Use enums carefully

API-263 manual **Apply only compatible changes on minor upgrades**
Only backward compatible API changes are present in minor updates
Versioning

Support compatible changes on client side

API clients (which could be APIs) should apply the robustness principle:

- Be conservative with API requests and data passed as input, e.g. avoid exploiting definition deficits like passing megabytes of strings with unspecified maximum length.
- Be tolerant in processing and reading data of API responses

API clients must be prepared for compatible API changes on APIs:

- Be tolerant with unknown fields in the payload, i.e. just ignore new fields.
- Be prepared that enum like (x-extensible-enum) return parameter may deliver new values; either be agnostic or provide default behavior for unknown values.
- Be sure that fields are accessed by their name rather than their hardcoded index (as a list item)

Potential issues:

- Ignoring new fields is often a technological issue due to serializers or too strong validators.
- Ignoring new value on enums style field may also be challenging, use plain string fields for such purpose if necessary.
- Seeking value by index instead of by the field name will break down even minor upgrade.

API-264 manual **Support compatible changes on client side**
As consumer of API, keep loose coupling and support changes.
Versioning

Always return an object on JSON responses

Even if the core feature of an endpoint/resource is to reply a list of items or a single scalar value, you must always return in the response body a JSON object (and not an array or scalar) as a top level data structure, to support future extensibility.

This allows you to easily extend your response without breaking backwards compatibility.

Incorrect

```
openapi: 3.0.1
paths:
  /articles:
    get:
      responses:
        200:
          content:
            application/json:
              schema:
                type: array          => Do not return an array directly
                items:
                  type: string
```

Correct

/

```
openapi: 3.0.1
paths:
  /articles:
    get:
      responses:
        200:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Articles'
components:
  schemas:
    Article:
      properties:
        name:
          type: string
    Articles:
      properties:
        articles:
          type: array
          items :
            $ref: '#/components/schemas/Article'
```

API-265 auto-swagger-check **Always return an object on JSON responses**
Responding a scalar or an array is less evolutive (API-242 must be GREEN, else this rule may stay Pending) Versioning

Use enums carefully¶

Enums are per definition closed sets of values, that are assumed to be complete and not intended for extension. This closed principle of enumerations imposes compatibility issues when an enumeration must be extended. Because common deserializers usually impose the knowledge of all possible values in advance and will throw an exception or force a null value when receiving an unknown value.

If you consider that the client should still behave correctly on an unexpected value, then you may declare the type as simple string. You can expose the values as a list in the description but we recommend to use an open-ended list of values as a string-typed value with **x-extensible-enum** marker.

To explicit an open-ended list of values use the marker **x-extensible-enum** as follows:

```
deliver_methods:
  type: string
  x-extensible-enum:
    - parcel
    - letter
    - email
```

Note

x-extensible-enum is not JSON Schema conform and will be ignored by most tools.

You can also consider Microsoft ms-enum approach or following a discussion on the enums in OpenAPI specification for more details.

API-266 manual **Use enums carefully on contract**
Use string type in specification if not a closed set Versioning

Avoid using media type versioning¶

Refrain from using media type versioning. Instead Put major version in URL.

Media type is a format accepted/produced by an API. All resources usually use the generic media type `application/json`. To version with media type you first need to use *vendor specific* media type like `application/x.sgcib.cart+json`. Then you can add a version parameter for media type like `application/x.sgcib.cart+json;version=2`.

This pattern allows you to version each resource instead of versioning the API. But the cost of doing this is more work on server side and more awareness - and also code - on client side. The versioning of resources is also less visible.

For these reasons Put major version in URL instead of using media type versioning.

See also blog reference (advocating for media type versioning)

API-267 manual **Avoid media type versioning** Use generic media type and URL versioning	Versioning

Naming conventions¶

Use naming conventions to ease quick and non ambiguous semantic understanding by developers (consumers and producers).

Put “api” keyword before version in URL¶

Inside SG, a normal URL will always be expected to provide a none-API component like web-ui. So if your service gives only specific operational support functions with a JSON response (see Use JSON as default format), it is highly recommended to begin with the keyword **“api”** followed by the major version in your API path:

```
https://foo.socgen/api/v1/items/
https://eqd-dealmanager.fr.world.socgen/eqd-dealmanager/api/v1/...
https://eqd-dealmanager.fr.world.socgen/api/v1/...
https://dsp2-api.fr.world.socgen/...
```

But sometimes you may have other needs:

```
https://calculator.fr.world.socgen/api/fic/v1/...
=> expose version 1 of "fic" api, and imply that there are other api perimeters belonging to
"calculator"
(for ex: https://calculator.fr.world.socgen/api/eqd/v1/...)

https://calculator.fr.world.socgen/fic/api/v1/...
=> expose version 1 of "fic" api, and imply that there is a none-API component like UI for "fic"
perimeter
(for ex: https://calculator.fr.world.socgen/fic/ui/...)
```

This will help clearly distinguish an API from none-API component and allow both human and tools to behave consistently.

See also Containing the major version within the path keeps unambiguous the selection by client without relaying on headers as mentioned Put major version in URL.

Incorrect:

```
openapi: 3.0.1
servers:
  - url: "https://inter.net"
  - url: "https://inter.org"
paths:
  /v3/pets/cats/{catId}: {} => no "api"
  /v3/pets/api/cats/{catId}: {} => "api" should before "v3"
```

In the above example, there will be 4 endpoints but not all of them have “api” before the major version (absent or after the major version):

```
https://inter.net/v3/pets/cats/{catId}
https://inter.org/v3/pets/cats/{catId}
https://inter.net/v3/pets/api/cats/{catId}
https://inter.org/v3/pets/api/cats/{catIid}
```

Correct:

```
openapi: 3.0.1
servers:
  - url: "https://inter.net"
  - url: "https://inter.org"
paths:
  /api/v3/pets/cats/{cat_id}: {}
  /api/pets/v3/cats/{cat_id}: {}
```

You may also put “api” in servers

```
openapi: 3.0.1
servers:
  - url: "https://inter.net/api"
  - url: "https://inter.org/api"
paths:
  /v3/pets/cats/{cat_id}: {}
  /pets/v3/cats/{cat_id}: {}
```

You could put “api” to a server or a path but guarantee that each endpoint (fullpath : server + path) respects this rule.

API-285 auto-swagger-check **'api' keyword is present in URL for production environment and before the version** 'api' keyword is present in URL for production environment and before the version.	Naming conventions
API-286 manual **Put "api" keyword at the right place in URL** It must be 'api' at the valid place in production environment path	Naming conventions

Use functional naming in hostname¶

Functional naming is a powerful, yet easy way to align global resources as host, permission within an the application landscape. It helps to preserve uniqueness of names while giving readers meaningful context information about the addressed component. Besides, the most important aspect is, that it allows to keep APIs stable in the case of technical and organizational changes.

Hostname in SGM APIs must, respectively should conform to the functional naming which do not contain just words from organizations or applications, because the functional naming is more sustainable and understandable:

```
<functional-hostname>  ::= <functional-name>.fr.world.socgen
```

In example below, the functional naming **equity-acquisition** is better than the application naming **heracles**:

```
Good example

https://equity-acquisition.fr.world.socgen/api/v1/items?...

Bad example

https://heracles.fr.world.socgen/api/v1/items?...
```

In case the hostname is hard to change (eg. due to a historic issue), you should at least put functional naming in path:

```
https://heracles.fr.world.socgen/equity-acquisition/api/v1/items?...
```

API-287 manual **Use functional naming in hostname** The hostname of the API gives functional meaning	Naming conventions
API-288 manual **FQDN should be expressive**	

The hostname of the API gives functional meaning — Naming conventions

Avoid character encoding in URL¶

In order to make it easy to read and understand an URL during both development or diagnostics phases, it is better to avoid those characters that may require un url encoding. You have to take care of the path (the resource IDs), the parameter name and the parameter value.

Characters that you can freely use:

```
ASCII letters [a-zA-Z]
digits [0-9]
- _ . ! ~ * ' ( )
```

Take care of reserved characters which have specific meanings inside specific urls parts:

```
; , / ? : @ & = + $ #
```

Any other character needing to be escaped should be avoided if possible, example:

```
space % { } [ ] | \ ~ ` ^
```

API-289 auto-swagger-check **Avoid character encoding in URL when possible**
URL should be easy to read and understand, put % encoded character only when no other choice (API-242 must be GREEN, else this rule may stay Pending) — Naming conventions

Constrain URL to less than 1024 chars¶

Some components as part of a web API flow (browsers, load balancer, api server, ...), may have limitations on the length of URL.

In order to prevent such issues, you should ensure the full URL length (including query string) will not exceed 1024 characters.

You could limit the URL length by :

- Limit number of Nested Resource Levels
- Avoid character encoding in URL (for path parameter values)
- using *POST* request with a body for some use cases (see Use GET correctly)

API-290 manual **Constrain url size below 1k**
There is no risk to have a too long url potentially reaching the max size supported by frameworks or network components — Naming conventions

Prefer Hyphenated-Pascal-Case for HTTP header Fields¶

This is for consistency in your documentation (most other headers follow this convention). Avoid camelCase (without hyphens). Exceptions are common abbreviations like “ID.”

Incorrect

```yaml
openapi: 3.0.1
components:
  headers:
    accept-Encoding: {}
    apply-to-redirect-ref: {}
    disposition-notification-options: {}
    original-message-id: {}
```

Correct

```
openapi: 3.0.1
components:
  headers:
    Accept-Encoding: {}
    Apply-To-Redirect-Ref: {}
    Disposition-Notification-Options: {}
    Original-Message-ID: {}
```

API-291 auto-swagger-check **Use Hyphenated-Pascal-Case for header** Ex: Content-Type, not content-type (API-242 must be GREEN, else this rule may stay Pending) Naming conventions

Prefer kebab-case for path segment¶

Use lowercase separate words with hyphens (foo-bar) for *path segments* by default as it is common url usage on websites. This is known as kebab-case or spinal-case. This applies to concrete path segments and not the names of path parameters. In some cases, using camelCase (fooBar) for path segment, may be accepted but should be avoided:

```
/structured-products/  => preferred
/structuredProducts/   => accepted
```

See also Use camelCase for property names , Prefer camelCase for query parameters , Prefer camelCase for path parameters

Incorrect

```
openapi: 3.0.1
paths:
  /partner_orders/{orderId}: {}            => rejected because partner_orders are snake_case
```

Correct

```
openapi: 3.0.1
paths:
  /partner-orders/{orderId}: {}
```

API-292 auto-swagger-check **Use kebab-case on path segment** Ex: /big-deals/{id}, not /bigDeals/ (API-242 must be GREEN, else this rule may stay Pending) Naming conventions

Prefer camelCase for path parameters¶

If parameters are part of path : compose words using 'camelCase' or 'snake_case' but not both.

Use camelCase separate words for *path parameters* by default

```
structuredProductId, appCode, orderId
```

But snack_case is also accepted:

```
structured_product_id, app_code, order_id
```

See also Use camelCase for property names , Prefer kebab-case for path segment , Prefer camelCase for query parameters

Incorrect

```
openapi: 3.0.1
paths:
  /articles/{article-id}:
    get:
      parameters:
        - name: article-id                                  => path parameter in hyphens-case
          in: path # a path parameter not a query parameter
```

Correct

```
openapi: 3.0.1
paths:
  /articles/{articleId}:
    get:
      parameters:
        - name: articleId
          in: query
```

API-293 | auto-swagger-check **Prefer camelCase for path parameters**
Like a javascript variable, ex: orderId ; order_id is allowed, but not order-id (API-242 must be GREEN, else this rule may stay Pending) Naming conventions

Prefer camelCase for query parameters¶

In query parameters : compose words using 'camelCase' or 'snake_case' but not both.

Use camelCase to separate words for *query parameters* by default

```
structuredProductId, appCode, orderId
```

But snack_case is also accepted:

```
structured_product_id, app_code, order_id
```

See also : Use camelCase for property names , Prefer kebab-case for path segment , Prefer camelCase for path parameters

Incorrect

```
openapi: 3.0.1
paths:
  /articles:
    get:
      parameters:
        - name: filter-expensive-articles                   => query parameter in hyphens-case
          in: query # a query parameter not a path parameter
```

Correct

```
openapi: 3.0.1
paths:
  /articles:
    get:
      parameters:
        - name: filterExpensiveArticles
          in: query
```

API-294 | auto-swagger-check **Prefer camelCase for query parameters**
Like a javascript variable, ex: orderId ; order_id is allowed, but not order-id (API-242 must be GREEN, else this rule may stay Pending) Naming conventions

Use common parameters for full text search¶

If you provide query support for searching, in order to make it understandable that this is about a full text search, you should stick to the following naming conventions:

/

```
/persons?q=foo  (Google like)
/persons?query=foo  (Google like)
/persons?search=foo  (more explicit)
```

API-295 manual **Use common parameters for full text search**
See documentation for examples

Naming conventions

Use common parameters for paging¶

Access to lists of data items must support pagination to protect the service against overload as well as for best client side iteration and batch processing experience.

There are two or even more well known page iteration techniques. But we encourage the Cursor/Limit-based one :

- A cursor to point to the current page - use keyword “page” starting from 1 (human understanding way rather than the computer one)
- A limit to specify the number of records per page - use keyword “size” or “pageSize” (when “size” may be a business field)

Please remember that the response must not be a direct array, as this prevent enhancements on the root model.

Examples

```
GET deals?user=john.doe&page=3&pageSize=50
```

Following keywords should be reserved as parameters names only for paging semantic : range, first, last, page, size, pageSize, skip, top, $skip, $top.

> **Note**
>
> Consistent paging depends on the creation rate of requested data. To ensure consistency, a sort criteria must always be considered (explicit in url or implicit in API documentation) and server side should rely on as-of data.
>
> Discuss this rule on Jive

API-296 manual **Use common parameters for paging**
See documentation for examples

Naming conventions

Use common parameters for filtering¶

Any searching criteria rather than global unique ID of a resource should never be present as a path segment as said in Use path segments only for resources, because a path segment does not allow evolutions of criteria and at the same time a path segment represents a composite relationship (see Do not use nested path if not composite).

So a search criteria can be present only as a query parameter. The parameter value is the direct field value while searching for strict equality.

Good examples

```
GET deals?user=john.doe
// user is ('john.doe' or 'will.iam') and region is ('paris' or 'london')
GET deals?users=john.doe,will.iam&regions=paris,london
GET deals?users=john.doe&users=will.iam&regions=paris&regions=london
```

Using comma is a violation of rule ‘Avoid character encoding in URL’ but it can be a compromise to reduce the size of the URL in standard use cases (see also Constrain URL to less than 1024 chars)

Never

```
GET deals/john.doe/london/2014-10-09/2015-10-09
```

If you want to create complex filters you may use some normalized prefixes before the value like "**!**", "**not.**", "**lt.**", "**gt.**", "**gte.**", which are natural for developers

```
GET deals?user=!john.doe
GET deals?user=not.john.doe
GET deals?amount=gte.1000
GET deals?created=between.2014-10-09,2015-10-09
```

Using dot in prefix will avoid URL encoding, but using ':' is more natural to use when URL encoding is not an issue

```
GET deals?amount=gte:1000
```

Note that if you append normalized suffixes on parameter name, this will complicate the interface (eg: ?createdAfter=2014-10-09&createdSince=2015-10-09)

API-297 manual **Use common parameters for filtering**
See documentation for examples

Naming conventions

Use common parameters for sorting¶

You should provide query support for sorting especially when you provide query support for paging. Sorting criteria ensures that each page is idempotent. By the way, you should always consider an implicit sorting criteria like ascending creation date or ascending sequence number (something that should not change).

It is recommended to use only one query param containing comma-separated list of fields to define the sort order, avoiding ambiguities.

Prefix with '-' to indicate descending direction, default direction being ascending as usual.

Example

```
GET persons?sort=category,-name,city
```

Bad example

```
GET persons?sort=category&desc=name&sort=city
```

Using a comma is a violation of rule 'Avoid character encoding in URL' but it can be a compromise to allow more complex sorting.

API-298 manual **Use common parameters for sorting**
See documentation for examples

Naming conventions

Use common parameters for fields selection¶

Depending on your use case and payload size, you can ease client app developer efforts and significantly reduce network bandwidth needs by allowing to select the fields to return.

Monitoring these requested fields allows to know which data matters to clients.

Use a "fields" query parameter containing a comma-separated list of fields names or of predefined "fields set".

Examples

```
GET deals/123?fields=id,amount,currency
GET deals/123?fields=_summary
GET deals/123?fields=_summary,currency
```

> **Note**
>
> Some non REST frameworks like graphQL manage this natively, but it is not a SG standard (as of July 2019). So usage of graphQL is reserved for UI dedicated backends (BFF).

API-299 manual **Use common parameters for fields selection**
See documentation for examples
Naming conventions

Define Collection Format of Query Parameters and Headers¶

Sometimes, query parameters and headers allow to provide a list of values, either by providing a comma-separated list (csv) or by repeating the parameter multiple times with different values (multi).

The API specification should explicitly define one type as follows:

Description	OpenAPI 3.0	Swagger 2.0	Example
Comma separated values	style: form, explode: false	collectionFormat: csv	?param=value1,value2
Multiple parameters	style: form, explode: true	collectionFormat: multi	?param=value1¶m=value2

When choosing the collection format, take into account the tool support, the escaping of special characters and the maximal URL length.

We recommend the usage of comma separated values to reduce the footprint on url and to ensure the ordering of values.

Incorrect

```
openapi: 3.0.1
components:
  parameters:
    filters:
      in: query
      style: spaceDelimited             => rejected style
      schema:
        type: array
```

Correct

```
openapi: 3.0.1
components:
  parameters:
    filters:
      in: query
      style: form
      schema:
        type: array
```

API-300 auto-swagger-check **Use and Specify Explicitly the Form-Style Query Format for Collection Parameters**
Must specify format to help clients (API-242 must be GREEN, else this rule may stay Pending)
Naming conventions

JSON conventions¶

Use conventions on JSON content to ensure proper interoperability

Use JSON as default format¶

JSON is the primary format for APIs and should be available by default on each operation (both in request and response).

Some operations may also accept text structured data format like XML or CSV. And for some very special use cases, other text data formats like like YAML, HTML, positional are acceptable.

Additionnal binary serialization formats may be proposed, like Protobuf and Avro for performance reasons (but should not be imposed to clients who have no performance requirements).

Runtime dependent binary serialization must never be used (ex: java, dotnet, python) as they are not interoperable.

Some operation may still consume/produce files instead of pure data like PDF, images, Office docs.

Incorrect

```
swagger: '2.0'
paths:
  /api/v3/reports:
        post:
          consumes:         => no application/json or application/.*+json format
                - text/json
                - text/plain
          parameters:
                - name: category
                  in: body
                  required: false
        get:
          responses:
                200:
                  content:
                        "application/x.sg.contract+json;v=123": {} => invalid format
                        "application/vnd.api+json": {}
```

Correct

```
swagger: '2.0'
paths:
  /api/v3/reports:
        post:
          consumes:
                - application/json
                - text/json
                - application/*+json
          parameters:
                - name: category
                  in: body
                  required: false
        patch:
                consumes:
                - application/merge-patch+json => valid
        get:
          responses:
                200:
                  content:
                        "application/json": {}
```

API-301 manual **Use JSON as default format**

All operations support at least json format for request and response (API-242 must be GREEN, else this rule may stay Pending)

JSON conventions

Use camelCase for property names¶

Property names (in request or response body) are restricted to ASCII strings without accents: **a-zA-Z0-9**. By convention, they should use the same casing as traditional javascript convention: **camelCase**.

Incorrect

```
openapi: '3.0.1'
components:
  schemas:
    article:
      properties:
        megalaser-turbo-article:          => kebab-case
          type: boolean
        super_megalaser_turbo_article:    => snake_case
          type: boolean
```

Correct

```
openapi: '3.0.1'
components:
  schemas:
    article:
      properties:
        superMegalaserTurboArticle:
          type: boolean
```

API-302 auto-swagger-check **Use camelCase for property names**
On request and response body, the fields use camel casing as : bigDeal, following JSON conventions javascript convention (API-242 must be GREEN, else this rule may stay Pending)

Combine Property with Type (And Format) Correctly

Whenever an API tries to define a property served as Date or Date-Time (“^.*_?[Dd]ate$”), the format of this property must be “string” and the type must be “date” or “date-time”.

Whenever an API tries to define a property served as Boolean (“^is[A-Z].*$” or ““^has[A-Z].*$””), the type of this property must be “boolean”.

Incorrect

```
openapi: '3.0.1'
info:
  title: Awesome API
  version: 1.0.0
components:
  schemas:
        Pet:
          properties:
                birthDate:                      => seems to be a date
                  type: string
                isDog:           => seems to be a boolean
                  type: string
                hasTeeth:        => seems to be a boolean
                  type: string
```

Correct

```
openapi: '3.0.1'
info:
  title: Awesome API
  version: 1.0.0
components:
  schemas:
        Pet:
          properties:
                birthDate:
                  type: string
                  format: date
                isDog:
                  type: boolean
                hasTeeth:
                  type: boolean
```

/

API-303 auto-swagger-check **Ensure property's type (and format) is aligned with naming convention on suffix** Some names imply a type by standard convention (API-242 must be GREEN, else this rule may stay Pending)	JSON conventions

Use Standard Date and Time Formats¶

As JSON has no datetime format, by common convention like RFC 3339, use only the iso format as bellow :

- for “date” use strings matching **YYYY-MM-DD**
- for “date-time” use strings matching **YYYY-MM-DDTHH:MM:SSZ** (Z is for UTC timezone)

Avoid any other format to ensure easy and non-ambiguous date/date-time parsing.

```
Ex: "2016-09-09"
Ex: "2016-09-09T13:59:59Z", "2016-09-09T13:59:59+00:00"
```

The timezone is required to ensure that various systems from different locations in the world understand exactly the same time.

For the usage of timezone, we encourage **restricting your datetime to UTC (Z, +00:00) and without offsets (ex: +01:00, +02:00 ...)**. From experience we have learned that zone offsets are not easy to understand and often not correctly handled. Note also that zone offsets are different from local times that might include daylight saving time (ex: +01:00 or +02:00 for Paris during Daylight Saving Time period). Localization of dates should be done by the services that provide user interfaces, if required.

This will help human customers avoid converting timezones while comparing datetime from different systems, based on textual values from logs or other sources.

API-304 manual **Use Standard Date and Time Formats** Comply with standard for date and time and use UTC only	JSON conventions
API-305 manual **API is documenting the format of date and time properties and they follow the standard.** Using standards improves interoperability and reduces issues	Naming conventions

Pluralize The Name Of Array Property¶

To indicate an array contain multiple values or an object seen as an open dictionary, you must pluralize array names. This implies that object names should in turn be singular.

Incorrect

```
openapi: 3.0.1
components:
  schemas:
    car:
      properties:
        feature:                    => name is not pluralized
          type: array
          items:
            type: string
```

Correct

```
openapi: 3.0.1
components:
  schemas:
    car:
      properties:
        features:
          type: array
          items:
            type: string
```

API-306 auto-swagger-check **Pluralize the name of array property**
As a convention to quickly understand fields semantic ; like: {deals:[]} (API-242 JSON conventions must be GREEN, else this rule may stay Pending)

Define Format for Type Number and Integer¶

Whenever an API defines a property of type number or integer, the precision must be defined by the format as follows to prevent clients from guessing the precision incorrectly, and thereby changing the value unintentionally:

type	format	specified value range
integer	int32	integer between -2^31 and 2^31-1
integer	int64	integer between -2^63 and 2^63-1
integer	bigint	arbitrarily large signed integer number
number	float	IEEE 754-2008/ISO 60559:2011 binary64 decimal number
number	double	IEEE 754-2008/ISO 60559:2011 binary128 decimal number
number	decimal	arbitrarily precise signed decimal number

The precision must be translated by clients and servers into the most specific language types. E.G. a number type in decimal format must be translated into BigDecimal in Java and an integer type in int32 format must be translated into int or Integer in Java.

Incorrect

```
openapi: '3.0.1'
info:
  title: Bad API
  version: 1.0.0
components:
  schemas:
    Pet:
      properties:
        age:
          type: number            => Miss format
```

Correct

```yaml
openapi: '3.0.1'
info:
  title: Good API
  version: 1.0.0
components:
  schemas:
    Pet:
      properties:
        age:
          type: number
          format: decimal
```

Note: if you generate swagger documentation from code (e.g. to follow Always match the running code) then you should know that current tools (swagger java library) does not support specilying a format for type BigDecimal. Therefore you should may not be able to comply with this rule in some cases due to this limitation. For simpler types this is natively generated.

API-307 auto-swagger-check **Explicit the format for Integer and Number types**
It provides valuable information to clients (API-242 must be GREEN, else this rule may stay Pending)
JSON conventions

Explicit default value

In your OpenAPI specification, when a property of a JSON schema is optional and is not of string type, you must make it very clear to client that if this means a null value or a default value like 0 for numeric, false for boolean, first value of an enum for enums.

> **Warning**
>
> Changing the default value of a field is considered backward-incompatible changes as seen by consumers! See Ensure existing consumers wont break on changes.

Here are some good practices:

- **Empty array values should not be null**
  Empty array values can unambiguously be represented as the empty list, [ ].
- **Use standard default value if possible**
  Values such as *false* or *0* are common and should usually be used.
  Specify if the behavior is different from null (especially for string or objects)
- **Specify the default for specific string**
  Some common objects such as dates are passed as string with a specific format.
  Default value can be the current day or the epoch day. This is context specific and deserves being explicit.

API-308 manual **Explicit default value**
Default value or behavior specified for each optional parameter
Naming conventions

Do not expose runtime dependent types

Using a code-first approach, your OAS/Swagger is automatically infered and some platform-specific naming may appear in the generated document.

To be sure that every client whatever the technology it use will be able to produce and parse the JSON data, you must not introduce any specific extra convention on types or deserialization, because such types are understood only by specific technology stacks.

Good examples in json:

```
"someDateField":"2016-09-21T13:48:33.160Z",

"strings":["a","b"]
```

Bad examples in json:

```
"someDateField":{
  "DateTime":{"$date":"2016-09-21T13:48:33.160Z"},
  "Offset":0,
  "Ticks":{"$numberLong":"636100625131600582"}
},

"strings":{_t:"System.Collections.Generic.List`1[System.String]",_v:["a","b"]}
```

API-309 _ manual **Do not expose runtime dependent types**
See documentation for examples such as dates and exception language internal model — Naming conventions

REST Design¶

Apply REST Design to ease quick understanding and reliable usage from consumers developers

Use Standard HTTP Status Codes¶

Use standards HTTP status codes consistently with their intended semantics. Do not create any new HTTP status codes.

On each endpoint exposed by the API, use HTTP status code semantic as most as possible to allow your clients to better handle each use case and to have some generic behaviour depending on status code. It also eases diagnostics by quickly focusing on the kind of error and the origin of it (client or server).

Below we list the most commonly used and best understood HTTP status codes, consistent with their simple semantic in the RFCs. APIs should only use these to prevent misconceptions that arise from less commonly used HTTP status codes.

Generally you dont need to describe your HTTP status code if their usage is well covered by the semantic. But if the HTTP status code is not in the list below or its usage requires additional information aside the well defined semantic, the API specification must provide a clear description of the HTTP status code in the response.

API-311 _ auto-swagger-check **Use Standard HTTP Status Codes**
Use only referenced status codes, do not create your own (API-242 must be GREEN, else this rule may stay Pending) — REST design

To know more about HTTP status codes, see some external references :

- List of HTTP status codes
- Use Standard HTTP Status Codes
- https://tools.ietf.org/html/rfc7231#page-47
- https://developer.mozilla.org/fr/docs/Web/HTTP/Response_codes
- http://www.restapitutorial.com/httpstatuscodes.html

Use success codes correctly¶

Reply 2xx when request behaves correctly as expected and documented :

Code	Message	Semantic
200	OK	Usual OK, also to use for empty list and pagination. Must contain a body.

Code	Message	Semantic
201	Created	Resource created from a POST, or also a PUT when the identifier can be pre-defined
202	Accepted	Asynch request with further treatment, polling or webhook may be used to get result
204	No content	There is no data to reply so response body is empty (not to use for empty list)
206	Partial Content	Unable to return all expected data for known reasons described in documentation

Do not use 206 for paging : when client request a page and you return it, you fully replied to its request.

API-312 manual **Use 2xx statuses correctly**
2xx OK statuses are used accurately and response body is documented — REST design

Examples:

Verb and Resource	Body	Status code	Description
GET deals/123		200 OK	deal has been retrieved
PUT deals/123	{ ... }	200 OK	deal has been updated
GET deals?user=john.doe		200 OK	JSON object with empty array inside
PUT users/john.doe	{ ... }	201 Created	user has been created
POST deals	{ ... }	201 Created	deal has been synchronously created
POST deals	{ ... }	202 Accepted	request has been secured and will be processed
GET stats?day=_today		206 Partial Content	got only part of the response, as today is not yet finished

Use client error codes correctly¶

The client app did a wrong request and must correct its client code or better verify the input data. The API team should analyze such errors to improve its documentation and help its client to better consume it.

Code	Message	Semantic

Code	Message	Semantic
400	Bad request	Url does not match an endpoint, invalid parameter name or value, missing parameter, anything within body input fields/values that does not comply with documentation
401	Unauthorized	Client app or end-user authentication is missing, invalid, expired, incomplete or at insufficient level, and must be re-done properly. More generally the system was not able to validate the requester.
403	Forbidden	The rights of the Client app or end-user are not sufficient as per the IAM system Warning: cases when end-user authentication is required but not possible because the received token is not containing end user information will result in 403 error
404	Not found	Requested resource does not exists (only for a request on a specific resource id)
405	Method Not Allowed	HTTP verb is not allowed on this resource endpoint
406	Not Acceptable	Client ask for an invalid response content-type
410	Gone	Client ask to remove a resource that was already removed and cannot be recreated
409	Conflict	There is data conflict on a modification request (eg: old data version, lock, already exists), that the client (user or program) can solve somehow, he then can retry same or similar modif
429	Too Many Requests	A quota/throttling logic detected too many requests from client in current period

Since the status code may match several use cases you should Use standardized error body to give sufficient information.

API-313 manual **Use 4xx statuses correctly**
4xx Client Error statuses are used accurately and related response body are documented — REST design

Examples:

Verb and Resource	Body	Status Code	Comment
GET deal/123		400 Bad Request	should have use deals/123

Verb and Resource	Body	Status Code	Comment
GET deal/123	Autz: Bearer foo	401 Unauthorized	the token '5451..234' has expired
GET deal/123	Autz: Bearer foo	403 Forbidden	authenticated user have no sufficient rights to see this deal
GET deals/124		404 Not Found	
DELETE deals/123		405 Method Not Allowed	deleting this deal is not allowed only PATCH { 'status':'ended' }

Use server error codes correctly¶

Something unexpected happen on API logic that must be corrected and prevented. Each 500 error must be analyzed and an action plan defined asap to remediate. The API team support team must help the client app team as much as possible to handle such event and remediate.

Code	Message	Semantic
500	Internal Server Error	Server failed to process for unexpected reason, usually an untrapped exception, there must be a log with the exception message and stacktrace, but this information must not be in response body
501	Not Implemented	Server explicitely was unable to execute this operation, whereas it should have
502	Bad Gateway	Server response failed to be forwared to client by an intermediate component which raised this error to the client, such error must also be logged and be accessible to API team
503	Service Unavailable	Server explicitely can not reply at all, whereas it should have such response can be emitted during anticipated downtime periods
504	Gateway Time-out	Server was too long to respond and an intermediate component raised this error to the client, such error must also be logged and be accessible to API team

> **Important**
>
> Do not expose internal data as stacktrace or inner exception message, Do not expose internal info on error

Since the status code may match several use cases you should Use standardized error body to give sufficient information.

API-314 manual **Use 5xx statuses correctly**
5xx Server Error statuses are used accurately and response body are documented if present REST design

Avoid HTTP Status Codes 3xx¶

Statuses in the 3xx class are sent when additional action is required on the part of the client in order to complete the request. This is most commonly used in redirecting one URL to another, though not always.

In the case of GET requests, the browser will generally carry out the second request without any input or additional interaction from the user. In other cases, additional user intervention is required.

But as a API producer, you are required to offer an OpenAPI Specification, so you must guarantee that all your API operations respond always in a direct way :

- a 2xx status code for success
- a 4xx status code for client error
- a 5xx status code for server error

If one day you need to redirect an endpoint (have to give a 3xx status code) because you need to replace the implementation or because you make an incompatible change, you must introduce a new API major version. Even in such case, you don't need a 3xx status code.

See also Ensure existing consumers wont break on changes

API-315 auto-swagger-check **Avoid Status Codes 3xx** Don't use 3xx status codes (API-242 must be GREEN, else this rule may stay Pending)	REST design

Use GET correctly¶

GET requests are used to read either a single or a collection of resources.

- GET requests must **NOT** have an input body payload (not compliant with browsers)
- GET requests on collection resources should provide sufficient filter and pagination mechanisms. See Use common parameters for paging and Use common parameters for filtering
- GET must be **SAFE**, meaning it must NOT have intended side effects (changes to the system state).
    - There can be some "write" operations within the server during a HTTP GET, like audit or cache update but no changes of functional data. See REST Cook Book.
    - If for some reasons you have to use GET for a non safe operation, use a verbal form on the url to explicit your are not in a REST mode, as in Use verbal form for non REST operations.
- POST to GET (non-REST)
    - When you need significant volume of input data for your GET request, that may conflict with the URL size limits from various HTTP components, then use a non-REST POST as in Use verbal form for non REST operations
    - Avoid forcing usage of a two-steps RESTful sequence, doing a POST to formalize a query then a GET to run-it, as it is a "complex" statefull pattern, to use only when your query cannot be run synchronously (see 202 below)
- GET requests usually returns:
    - 200 OK for individual resource if it exists, returning the resource in output body
    - 404 Not Found with an error response if the individual resource does not exist
    - 204 No content in rare situations where it is normal to not have any result, like when polling for an already identified resource still being created, or checking for a typed sub-resource that can be empty as per documentation. See Be Responsive for details on polling

- 200 OK on collections, with a response body even for empty results (do not use 204) and for pagination (do not use 206)
  - Do not reply directly an array but an object, as in Always return an object on JSON responses
- 202 Accepted if getting results takes time and is to be retrieved asynchronously;
  - the direct response gives an identifier to some temporary resource that will contain the result, as in Be Responsive
- 206 Partial if not all requested data can be retrieve for documented reasons (not pagination), as in Use success codes correctly

API-316 manual **Use GET correctly**
GET verb is used only to retrieve data without changing system state — REST design

Use POST correctly¶

POST requests are usually used to create a single resource, but other semantics are possible.

- On creation, data comes from the JSON input body (and not parameters in the url) and response body contains the identifier of the created resource.
- POST with an empty input body can also be used to allocate a new identifying id as empty resource (like getting an auto-incremented id), and then a PUT is used on this url with the related input body. This help to avoid the “double-click” duplicates creations.
- POST is also usable to trigger some treatment/computation with a response that may not be a persisted resource. In such situation use a verb in url (see Use verbal form for non REST operations).
- POST used on non restful operation, like a complex search, returns 200 for success (see Use verbal form for non REST operations).

Successful POST requests may generate:

- 200 if the treatment/computation was done OK with some response content returned for information
- 201 if resource has been created, with response content containing the identifier of created resource
- 202 if the request was accepted but has not been finished yet (Asynch)
- 204 if the treatment/computation was done OK and no response content is needed
- 206 if the treatment/computation was done partially OK with some response content returned for information

Examples

Verb and Resource	Input Body	Status Code	Semantic
POST do-sanitize-all	empty	204 No content	Sanitize done OK
POST trades	empty	200 OK	Returns an allocated id
POST do-search	{...}	200 OK	Results of complex search
POST big-prices	{...}	202 Accepted	Compute in-progress

Verb and Resource	Input Body	Status Code	Semantic
POST prices	{...}	206 Partial	Not all prices where computed for expected reasons

API-317 manual **Use POST correctly**
POST verb is used only to create a resource or trigger treatments — REST design

Use PUT correctly

PUT requests are used to update (and create) a full resource.

- PUT requests are applied to single resource, and not collections, as this would imply replacing the entire collection
- on successful PUT requests, the server will replace the entire resource with the content from input body
- UPSERT: We may also CREATE a new resource using PUT when the path (id) can be pre-defined
    - If resource does not exists, it's created, else it's replaced
    - Ensure there is no possible conflicts on such id between multiple clients

Successful PUT requests will usually generate :

- 200 on update – with actual content returned
- 204 on update – without actual content returned
- 201 on creation, returning again the same identifier as in path

If you want to use code 200 without returning the data (to simplify the work for clients) then you must make it explicit in the documentation.

API-318 manual **Use PUT correctly**
PUT verb is used only to fully replace a resource or create one with predefined identifier — REST design

Use PATCH correctly

PATCH requests are used to update parts of single resources, i.e. where only a specific subset of resource fields should be replaced. To fully update the resource, use PUT instead.

- The semantic of the input body is not defined in standards and must be described in the API specification.
    - To be explicit, you may use the simple `application/merge-patch+json` (RFC)
    - Avoid using meta-data formats as JSON patch , as they are verbose and over complex for usual needs
- PATCH requests are usually applied to single resources as patching entire collection is challenging
- successful PATCH requests will usually generate
    - 200 on update – with actual content returned
    - 204 on update – without actual content returned

Example

```
POST /items
  Content-Type: application/json
  { "title":"foo", "summary":"before"}
201 Created
  Content-Type: application/json
  { "itemId":"123", "title":"foo", "summary":"before"}

PATCH /items/123
  Content-Type: application/merge-patch+json
  { "summary":"change!"}
200 OK
  Content-Type: application/json
  { "itemId":"123", "title":"foo", "summary":"change!"}
```

API-320 manual **Use PATCH correctly**
PATCH verb is used only to partially update an existing resource

REST design

Use DELETE correctly¶

DELETE requests are used to permanently delete resources.

- DELETE requests are usually applied to single resources, not on collection resources, as this would imply deleting the entire collection
- The parameter id of the deleted resource should not be reused by another resource. When resource ids are built using a unique auto-incremented numerical id from the database, there is no problem. But when ids are build from a word (like file name or user name), it may be reused by potential homonym. This should be very rare on core business resources, but still possible on short lived resources.
- successful DELETE requests will usually generate
  - 200 if information on the deleted resource is returned
  - 204 if no content is returned
- failed DELETE requests will usually generate
  - 404 if the resource cannot be found
  - 410 if the resource was already deleted before
- Despite having different status codes, DELETE requests are idempotent (see Fulfill “idempotent” property on HTTP requests).

If you want to differ from this standard then you must make it explicit in the documentation and justify it.

API-319 manual **Use DELETE correctly**
DELETE verb is used only to permanently remove a resource

REST design

Fulfill “idempotent” property on HTTP requests¶

All HTTP verbs except POST/PATCH should fulfill the idempotent property - the operation has the same intended effect on the system **state**, independently whether it is executed once or multiple times. Because it is easier to control the potential impact of operations on the server in order to maintain reliability.

This apply to GET, PUT and DELETE

This also applies to HEAD and OPTIONS - some HTTP verbs which are not “action” verbs.

External references :

- REST Cook Book

Even if it is not required, but it is also recommended to design POST and PATCH idempotent if possible.

Examples:

- GET /deals/42 is a safe operation that read information and does not modify the system state
- PUT /deals/42 updates the resource. It is unsafe because it modifies the system. It modifies the data of deal 42 to set the provided data so it is idempotent: several calls with the same arguments will end with the system in the same state (audit trails are not considered in this definition).
- POST /deals/ creates a new deal for each call. Therefore it is not idempotent.
- DELETE /deals/42 deletes the resource if it exist. It is unsafe because it modifies the system. Yet it is idempotent because several calls will end with the system in the same state (despite returning different status codes).

API-321 manual **Ensure idempotency on GET/PUT/DELETE**
Multiple identical calls to GET/PUT/DELETE always have same result — REST design

Use verbal form for non REST operations¶

Some situations are not eligible to REST semantic, eg:

- a query using POST because of too many parameters for an URL (as Constrain URL to less than 1024 chars)
- a pure computation or a trigger of high level process, using POST but not creating resources
- using a simple GET for un-SAFE operations due to constrains from client app or needs to use browser address bar

In such cases :

- Use a verb in the url, to clearly distinguish with other REST operations.
- Make it explicit in operation's description in swagger by writing `non-REST` within.
- It is recommended to make it fully explicit in url by using the `do-` prefix.

Examples

```
POST deals/do-get
  { dealIds:[12454,124545,57564,...] }

POST deals/do-search
  { region:"asia", amount:"lt:100", ..}

POST mailboxes/john.doe/outbox/123/send

POST compute-eligibility?deal=123&scope=seoul

POST deals/123/do-compute-elig?scope=seoul
```

Bad examples

POST deals/query Looks like REST but is not, and query is both a noun and a verb which is ambiguous.

API-322 manual **Use verbal form for non REST operations**
When an operation can not support REST semantic, use a verbal form in path segment — REST design

Pluralize Resource Names¶

In RESTful mode, the resource of an endpoint must use the plural noun form. This allows developers to predict and guess the method calls when they learn to work with your API.

You should stick to this rule as long as the represented resources has no matter what kind of identifier.

Incorrect:

```
swagger: '2.0'
paths:
  /cat:                                          => "cat" is singular
          ...
  /cats/{cat_id}/tail/{tail_id}/strands:         => "tail" is singular
          ...
```

Correct

```
swagger: '2.0'
paths:
  /cats:
          ...
  /cats/{cat_id}/tails/{tail_id}/strands:
          ...
```

The special case of a resource singleton is a collection with cardinality 1 or the resource of which kind you don't need to specify the ID but a common singular name to find it out.

Here is a common example. As a normal single user (not an administrator), you have only one single login. When you want to get the information about yourself, you don't want to specify your user-ID because it is not meaningful for you. So you could use "GET /users/_me" to achieve your goal. In this case, "/_me" will be exempted.

In order to make it different from a real resource in plural, the name of a resource in the above special case should start with "_"

```
GET /users/_me
GET /tasks/_latest
```

API-323 auto-swagger-check **Pluralize resource names**
By convention REST resource are plural, if not on a REST path segment, use 'do-' prefix in segment name (API-242 must be GREEN, else this rule may stay Pending) REST design

Avoid Trailing Slashes¶

The trailing slash must not have specific semantics. Resource paths must deliver the same results whether they have the trailing slash or not.

Incorrect:

```
openapi: '3.0.0'
paths:
  /api/test-api/: {}
  /long/bad/path/with/slash/: {}
```

Correct:

```
openapi: '3.0.0'
paths:
  /api/test-api: {}
  /long/bad/path/with/slash: {}
```

API-324 auto-swagger-check **Avoid trailing slashes on path**
Resource paths must deliver the same results whether they have the trailing slash or not (API-242 must be GREEN, else this rule may stay Pending) REST design

Use path segments only for resources¶

Resources should always be referenced by their name and identifier in the path segments but never by a search criteria. In other words, only the name and identifier of resources (or sub-resources as mentioned in Do not use nested path if not composite) can appear in the path segments. So a path segment placeholder is reserved to define the resource identifier.

Any kind of criteria used to search one or more items of a given resource (HTTP GET in most cases), must be exposed as query string parameters instead of in path segments.

In some cases you may have many criteria making the URL potentially too huge. In these cases you may want to use a **POST** request and pass criteria as the request body. You may also want to use another technology such as GraphQL.

Good examples

```
GET items/12345
GET items?startDate=2016-01-01&endDate=2016-02-01
```

Bad examples

```
GET items/start-date/2016-01-01/end-date/2016-02-01
GET items/2016-01-01/2016-02-01
```

See also Identify (sub-)resources via Path Segments , Use common parameters for filtering , Constrain URL to less than 1024 chars

API-326 manual **Use path segments only for resources**
Search criteria are query params, and path params are used only to identify a resource REST design

Identify (sub-)resources via Path Segments¶

Some API resources may contain or reference sub-resources. Embedded sub-resources, which are not top-level resources, are parts of a higher-level resource and cannot be used outside of its scope. Sub-resources should be referenced by their name and identifier in the path segments.

Basic URL structure:

```
/{resources}/[resourceId]/{subResources}/[subResourceId]
/{resources}/[partialId1][separator][partialId2]
```

Examples:

```
/carts/1681e6b88ec1/items
/carts/1681e6b88ec1/items/1
/customers/12ev123bv12v/addresses/DE_100100101
/content/images/9cacb4d8
```

Composite identifiers must not contain / as a separator. In order to improve the consumer experience, you should aim for intuitively understandable URLs, where each sub-path is a valid reference to a resource or a set of resources. For example, if **/customers/12ev123bv12v/addresses/DE_100100101** is a valid path of your API, then **/customers/12ev123bv12v/addresses**, **/customers/12ev123bv12v** and **/customers** must be valid as well in principle.

Incorrect

```
openapi: 3.0.1
paths:
  /merchants/{merchantId}/{addressId}:  => successive path parameter
    get:
      responses:
        200: {}
```

Correct

```
openapi: 3.0.1
paths:
  /merchants/{merchantId}/addresses/{addressId}:
    get:
      responses:
        200: {}
```

See also Do not use nested path if not composite in order to avoid abusive usage of sub-resources

API-325 auto-swagger-check **Identify resources via path segments**
There should be no consecutive path parameters (API-242 must be GREEN, else this rule may stay Pending) REST design

Do not use nested path if not composite¶

If a sub-resource is only accessible via its parent resource and may not exists without parent resource, you may consider using a nested URL structure, for instance:

```
POST /carts/{cart_id}/cart-items
GET, PUT, DELETE, PATCH /carts/{cart_id}/cart-items/{cart_item_id}
```

This nested resource "/cart-items/{cart_item_id}" is part of "/carts/{cart_id}" - which is a composition relationship.

A composition relationship implies :

- sub-resources can be accessed and searched only within the scope of its specific parent resource
- sub-resources can not be searched in a global mode
- sub-resources will be deleted when its parent resource is deleted
- sub-resources should not be impacted while no action on its parent resource (no side effect)

However, if the resource can be accessed directly via its unique id, or it is required to be independently searched out of its parent's scope, or it has complex relationships with other resources rather than its "parent" resource, then the API should expose it as a top-level resource - which is not a composition relationship.

For example, country has a collection for cities; however, cities have globally unique id and some services may choose to access the cities directly.

Good examples

For flexibility of usage, it is better to expose as top-level resources:

```
/countries/fr
/cities/paris => { country:"fr", ... } ;
/cities?country=fr
/buildings/granite => { city:"paris", ... };
/buildings?country=fr
/floors/granite-06
/floors?capacity_min=160
/desks/granite-06-129
/desks?is_available=true
/desks?user=john.doe
```

A resource associated with others:

```
/clients/foo
/providers/bar
/contracts?client=foo&provider=bar
```

Bad examples :

Following is bad example, even it seems like a logical way to represent a hierarchy of parent/child items, this prevents to search for available desks whatever floor of a building:

```
/countries/fr/cities/paris/buildings/granite/floors/6/desks/129
```

Another bad example is a resource as an association between two other resources : client - contract - provider:

```
/clients/foo/contracts?provider=bar
/providers/bar/contracts?client=foo
```

to be discussed on Jive

> API-327 manual **Use nested path only for composite relation**
> Only sub-resource strictly belonging to its parent is managed as nested path, and also if there is no need for a global search on it REST design

Limit number of Resource types¶

To keep maintenance and service evolution manageable, we should follow “functional segmentation” and “separation of concern” design principles and do not mix different business functionalities in same API definition. In practice this means that the number of resource types exposed via an API should be limited. In this context a resource type is defined as a set of highly related resources such as a collection, its members and any direct sub-resources.

Consider the following example:

```
/customers
/customers/{id}
/customers/{id}/preferences
/customers/{id}/addresses
/customers/{id}/addresses/{addr}
/addresses
/addresses/{addr}
```

This example contains 3 resource types with this definition: customers, addresses and customers’ related addresses

- **“/customers/{id}/preferences”** is part of the **/customers** resource type because it has a one-to-one relation to the customer without an additional identifier. That means you cannot dissociate preferences from the user.
- **/customers** and **/customers/{id}/addresses** as separate resource types because **/customers/{id}/addresses/{addr}** also exists with an additional identifier for the address.
- **/addresses** and **/customers/{id}/addresses** as separate resource types because there’s no reliable way to be sure they are the same.

> API-328 auto-swagger-check **Limit number of resources**
> API should have less than 8 resources (API-242 must be GREEN, else this rule may stay Pending) REST design

Given this definition, our experience is that well defined APIs involve no more than 4 to 8 resource types. There may be exceptions with more complex business domains that require more resources, but you should first check if you can split them into separate subdomains with distinct APIs.

See also Limit number of operations and Limit number of Nested Resource Levels

Limit number of Nested Resource Levels¶

There are main resources (with root url paths) and sub-resources (or nested resources with non-root urls paths). Use sub-resources if their life cycle is (loosely) coupled to the main resource, i.e. the main resource works as collection resource of the sub-resource entities. You should use <= 3 nested resource levels — more levels increase API complexity and url path length.

Remember, some popular web browsers do not support URLs of more than 2000 characters and some load balancers have a limitation at 1024 characters by default.

Incorrect

```
openapi: "3.0.0"
paths:  => rejected because 4 nested resource levels (5 levels in total)
  /worlds/{world_id}/countries/{country_id}/states/{state_id}/cities/{city_id}/streets/{street_id}: {}
```

Correct

```
openapi: "3.0.0"
paths:
  /worlds
  /worlds/{world_id}
  /countries
  /countries/{country_id}                                    => is moved to root
  /states/{state_id}/cities/{city_id}/streets/{street_id}: {} => limit to 3 nested resource levels (4
levels in total)
```

If a nested resource can have an identifier, it is always possible to move it to root resource.

API-329 auto-swagger-check **Limit number of nested resource levels**
Less than 3 nested resource levels (API-242 must be GREEN, else this rule may stay Pending) REST design

Limit number of operations¶

The number of operations exposed via an API should be limited in order to keep maintenance and service evolution manageable.

There are some ways to limit the number of operations:

- Limit number of Resource types
- Limit number of Nested Resource Levels

For many resources you will have to support several GET requests, POST, PUT, DELETE and maybe PATCH.

API-330 auto-swagger-check **Avoid more than 100 operations in same API**
API should have less than 100 operations to be manageable by both producer and clients (API-242 must be GREEN, else this rule may stay Pending) REST design

Use standardized error body¶

API developer should use a common error object as the response through all its API because the general HTTP status code alone is not sufficient to explicit functional or technical issues.

Operations should return it (together with a suitable status code) when any problem occurred during processing and you can give more details than the status code itself can supply, whether it be caused by the client or the server (i.e. both for 4xx or 5xx error codes).

With this common error object, clients could be able to behave more according to error details if necessary. It is recommended that the error structure should be common, thus makes it possible to react to error with a common logic, in both client and API developers side.

For example, based on the error information, client application could be able to build some meaningful messages to the end user, like listing the fields having invalid value (4xx errors) or having a specific behavior depending on error content.

There is no yet common proposal on GBIS, please see working draft : What should be the json response body when on 400 error ?. Current discussions target the RFC problem-details in order to build upon the existing standard.

Incorrect

```
openapi: 3.0.1
paths:
  '/pets':
    get:
      responses:
        '200':
          <...>
        '400':                                                                  => rejected no error
body
          description: Bad request. User ID must be an integer and larger than 0.
        '401':
=> rejected no error body
          description: Authorization information is missing or invalid.
        '404':
=> rejected no error body
          description: An user with the specified ID was not found.
        '5XX':
=> rejected no error body
          description: Unexpected error.
```

Correct

```
openapi: 3.0.1
paths:
  '/pets':
    get:
      responses:
        '200':
          <...>
        '400':
          description: Bad request. User ID must be an integer and larger than 0.
          content:
            application/json:
              schema:
                errorDetail:
                  type: object
                  <...>
        '401':
          description: Authorization information is missing or invalid.
          content:
            application/json:
              schema:
                errorDetail:
                  type: object
                  <...>
        '404':
          description: An user with the specified ID was not found.
          content:
            application/json:
              schema:
                errorDetail:
                  type: object
                  <...>
        '5XX':
          description: Unexpected error.
          content:
            application/json:
              schema:
                errorDetail:
                  type: object
                  <...>
```

Or you can put a default response, it will be used when no content is defined:

```
openapi: 3.0.1
paths:
  '/pets':
    get:
      responses:
        default:
          content:
            application/json:
              schema:
                errorDetail:
                  type: object
        '200':
          <...>
        '400':
          description: Bad request. User ID must be an integer and larger than 0.
        '401':
          description: Authorization information is missing or invalid.
        '404':
          description: An user with the specified ID was not found.
        '5XX':
          description: Unexpected error.
```

API-331 auto-swagger-check **Use json object on error (4xx 5xx)** It provides valuable information to clients (API-242 must be GREEN, else this rule may stay Pending)	REST design
API-332 manual **Use problem details format to report errors** See documentation for examples such as dates and exception language internal model	Rest Design

Allow CORS Headers¶

Your API may support a series of different HTTP Headers which you could find each definition in this list. Don't forget to mention the supported HTTP Headers in your OpenAPI definition.

Your API may have a direct JavaScript based Web-UI client in browsers. It might be an application you support of simply the way for some of your clients to call your API.

To support this use case you have to configure your API to reply to the Cross-origin resource sharing (CORS) headers and allow all (or all SG) origins and related methods/headers.

Here are the HTTP headers that relate to CORS.

- Origin
- Access-Control-Request-Method
- Access-Control-Request-Headers

Response headers

- Access-Control-Allow-Origin
- Access-Control-Allow-Credentials
- Access-Control-Expose-Headers
- Access-Control-Max-Age
- Access-Control-Allow-Methods
- Access-Control-Allow-Headers

See this external reference Cross-origin resource sharing for more information.

See our tutorials.

Example of actual calls

A direct call from a browser and expected response

```
GET /api/v1/positions HTTP/1.1
Host: equity-deals
Origin: http://posttradeconsole.fr.world.socgen
...

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
...
```

Example of actual calls

A pre-check call from a browser, because of using non basic methods/contents/headers

```
OPTIONS /api/v1/items/123 HTTP/1.1
Host: equity-deals
Origin: http://posttradeconsole.fr.world.socgen
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: X-PINGOTHER
...

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, DELETE, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER
...

DELETE /api/v1/items/123 HTTP/1.1
Host: equity-deals
Origin: http://posttradeconsole.fr.world.socgen
X-PINGOTHER: pingpong

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
...
```

API-333 manual **Allow CORS headers to let any SPA call you**
Any SPA (browser) client is legitimate to call your API, as long as its client_id is authorized — Design

Do not Use Hypermedia/HATEOAS¶

If you don't know HATEOAS or you are not interested to use it, you could just skip this rule!

Hypermedia As The Engine Of Application State (HATEOAS) is a component of the REST application architecture that distinguishes it from other network application architectures.

With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through hypermedia. A REST client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia.

In HATEOAS, the server provides all future actions in URL format that the client may take within resource representations of a media type (usually application/json, but can be also application/xml etc...) The client then through its states can select an action (an url) directly or manipulate the response in other ways afforded by its media type without needing to build the url with parameters as per OpenAPI documentation.

But we do not generally recommend to implement HATEOAS (also known as REST Maturity Level 3) for several reasons:

- HATEOAS comes with additional API complexity without real value in our SOA context
- The theoretical concept of HATEOAS that client needs little to no prior knowledge hasn't been seen working in practice.
- HATEOAS has low meaning for client app program coded by developers

Do not use the following things as mentioned in JSON Hypertext Application Language.

Incorrect:

```
* Content-Type / Accepted : application/hal+json
* property name : _links, links, link, _link, _embedded
```

Please find detailed rationals here : When to use Hateoas/Hypermedia?.

If you think having a good use case about hypermedia, please propose it to the API Architect

API-335 _ auto-swagger-check **Avoid using Hypermedia/HATEOAS links** Property names pattern and content types using HATEOAS convention are detected and rejected, see why in details (API-242 must be GREEN, else this rule may stay Pending)	REST design

Do Not Use Link Headers with JSON entities¶

We don't allow the use of the Link Header defined by RFC 5988 in conjunction with JSON media types. We prefer links directly embedded in JSON payloads to the uncommon link header syntax.

Incorrect:

```
swagger: 2.0
info:
  title: Clean Swagger API
paths:
  /foo:
    get:
      parameters:
        - name: Authorization
          in: header
          type: string
        - name: Link                => Do not use Link in Header
          in: header
          type: string
      responses:
        202:
          description: Lorem Ipsum
          headers:
            Location:
              type: string
              format: url
    post:
      responses:
        202:
          description: Lorem Ipsum
          headers:
            Link:                   => Do not use Link in Header
              type: string
              format: url
```

Correct:

```yaml
swagger: 2.0
info:
  title: Clean Swagger API
paths:
  /foo:
    get:
      description: Lorem Ipsum
      responses:
        202:
          description: Lorem Ipsum
          headers:
            Location: # should not violate since not called `Link`
              type: string
              format: url
parameters:
  FlowId: # should not violate since not named `Link`
    name: X-Flow-Id
    in: header
    type: string
  Link: # should not violate since not a header
    name: Link
    in: query
    type: string
  ProductId: # should not violate since not a header nor named `Link`
    name: product_id
    in: path
    type: string
```

API-336 auto-swagger-check **Do not use Link headers with JSON entities**
For request and response body keep coherence within the bank (API-242 must be GREEN, else this rule may stay Pending)
REST design

Do not use OData¶

OData has been tested within GBIS and then rejected as not enough interoperable by any kind of clients, not enough supported, too far from usual REST conventions.

Any usage of it will be limited only to support some dedicated client tools (like PowerBI), and not considered as the official way to expose this data to any client app.

API-337 manual **Do not impose usage of OData**
You do not force your clients to support OData for data retrieval
REST design