SailPoint RESTful API Guidelines

Github Repository

Other formats: PDF, EPUB3

Table of Contents

1. Introduction

SailPoint's SaaS software architecture centers around microservices that provide functionality via RESTful APIs with a JSON payload. Small engineering teams own, deploy and operate these microservices. Our APIs most purely express what our systems do, and are therefore highly valuable business assets. Designing high-quality, long-lasting APIs has become even more critical

for us as we invest more in our SaaS platform and enabling customers and partners to build functionalty outside of our UI.

With this in mind, we've adopted "API First" as one of our key engineering principles. Microservices development begins with API definition outside the code and ideally involves ample peer-review feedback to achieve high-quality APIs. API First encompasses a set of quality-related standards and fosters a peer review culture including a lightweight review procedure. We encourage our teams to follow them to ensure that our APIs:

- are easy to understand and learn
- are general and abstracted from specific implementation and use cases
- are robust and easy to use
- have a common look and feel
- follow a consistent RESTful style and syntax
- are consistent with other teams' APIs and our global architecture

Ideally, all SailPoint APIs will look like the same author created them.

Conventions used in these guidelines

The requirement level keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" used in this document (case insensitive) are to be interpreted as described in RFC 2119.

SailPoint specific information

The purpose of our "RESTful API guidelines" is to define standards to successfully establish "consistent API look and feel" quality. The SailPoint API Guild drafted and owns this document. Teams are responsible to fulfill these guidelines during API development and are encouraged to contribute to guideline evolution via pull requests.

These guidelines will, to some extent, remain work in progress as our work evolves, but teams can confidently follow and trust them.

In case guidelines are changing, following rules apply:

- existing APIs don't have to be changed, but we recommend it
- clients of existing APIs have to cope with these APIs based on outdated rules
- new APIs have to respect the current guidelines

Furthermore you should keep in mind that once an API becomes public externally available, it has to be re-reviewed and changed according to current guidelines - for sake of overall consistency.

2. Principles

API design principles

Comparing SOA web service interfacing style of SOAP vs. REST, the former tend to be centered around operations that are usually use-case specific and specialized. In contrast, REST is centered around business (data) entities exposed as resources that are identified via URIs and can be manipulated via standardized CRUD-like methods using different representations, and hypermedia. RESTful APIs tend to be less use-case specific and come with less rigid client / server coupling and are more suitable for an ecosystem of (core) services providing a platform of APIs to build diverse new business services. We apply the RESTful web service principles to all kind of application (micro-) service components, independently from whether they provide functionality via the internet or intranet.

- We prefer REST-based APIs with JSON payloads
- We prefer systems to be truly RESTful [1]

An important principle for API design and usage is Postel's Law, aka The Robustness Principle (see also RFC 1122):

- Be liberal in what you accept, be conservative in what you send

Readings: Some interesting reads on the RESTful API design style and service architecture:

- Article: REST API Design - Resource Modeling
- Article: Richardson Maturity Model — Steps toward the glory of REST
- Book: Irresistable APIs: Designing web APIs that developers will love
- Book: REST in Practice: Hypermedia and Systems Architecture
- Book: Build APIs You Won't Hate
- Fielding Dissertation: Architectural Styles and the Design of Network-Based Software Architectures

API as a product

At SailPoint, we want to deliver products to our (internal and external) customers which can be consumed like a service. Platform products provide their functionality via (public) APIs; hence, the design of our APIs should be based on the API as a Product principle:

- Treat your API as product and act like a product owner
- Put yourself into the place of your customers; be an advocate for their needs
- Emphasize simplicity, comprehensibility, and usability of APIs to make them irresistible for client engineers
- Actively improve and maintain API consistency over the long term
- Make use of customer feedback and provide service level support

Embracing 'API as a Product' facilitates a service ecosystem, which can be evolved more easily and used to experiment quickly with new business ideas by recombining core capabilities. It makes the difference between agile, innovative product service business built on a platform of APIs and ordinary enterprise integration business where APIs are provided as "appendix" of existing products to support system integration and optimised for local server-side realization.

Understand the concrete use cases of your customers and carefully check the trade-offs of your API design variants with a product mindset. Avoid short-term implementation optimizations at the expense of unnecessary client side obligations, and have a high attention on API quality and client developer experience.

API as a Product is closely related to our API First principle (see next chapter) which is more focused on how we engineer high quality APIs.

API first

API First is one of our engineering and architecture principles. In a nutshell API First requires two aspects:

- define APIs first, before coding its implementation, using a standard specification language
- get early review feedback from peers and client developers

By defining APIs outside the code, we want to facilitate early review feedback and also a development discipline that focus service interface design on…

- profound understanding of the domain and required functionality
- generalized business entities / resources, i.e. avoidance of use case specific APIs
- clear separation of WHAT vs. HOW concerns, i.e. abstraction from implementation aspects — APIs should be stable even if we replace complete service implementation including its underlying technology stack

Moreover, API definitions with standardized specification format also facilitate…

- single source of truth for the API specification; it is a crucial part of a contract between service provider and client users
- infrastructure tooling for API discovery, API GUIs, API documents, automated quality checks

Elements of API First are also this API Guidelines and a standardized API review process as to get early review feedback from peers and client developers. Peer review is important for us to get high quality APIs, to enable architectural and design alignment and to supported development of client applications decoupled from service provider engineering life cycle.

It is important to learn, that API First is **not in conflict with the agile development principles** that we love. Service applications should evolve incrementally — and so its APIs. Of course, our API specification will and should evolve iteratively in different cycles; however, each starting with draft status and *early* team and peer review feedback. API may change and profit from implementation concerns and automated testing feedback. API evolution during development life cycle may include breaking changes for not yet productive features and as long as we have aligned the changes with

the clients. Hence, API First does *not* mean that you must have 100% domain and requirement understanding and can never produce code before you have defined the complete API and get it confirmed by peer review.

On the other hand, API First obviously is in conflict with the bad practice of publishing API definition and asking for peer review after the service integration or even the service productive operation has started. It is crucial to request and get early feedback — as early as possible, but not before the API changes are comprehensive with focus to the next evolution step and have a certain quality (including API Guideline compliance), already confirmed via team internal reviews.

3. General guidelines

The titles are marked with the corresponding labels: **MUST**, **SHOULD**, **MAY**.

MUST follow SailPoint API Guidelines

You must design your APIs consistently with these guidelines; use our API linter for automated rule checks, but not every rule can be automated.

SHOULD follow API first principle

You should follow the API First Principle, more specifically:

- You should define APIs first, before coding their implementation, using OpenAPI as the specification language
- You should call for early review feedback from peers and client developers

MUST provide API specification using OpenAPI

We use the OpenAPI specification as the standard to define API specification files. OpenAPI 3.0 must be supported, but you MAY support other versions, like Swagger 2.

The API specification files should be subject to version control using a source code management system.

You must publish the component API specification with the deployment of the implementing service, and, hence, make it discoverable for the appropriate group via our API Portal.

MUST provide detailed API description

Within the API specification, you must provide sufficient information in the description of the API to facilitate proper usage. This may include:

- API scope, purpose, and use cases
- Major edge cases
- Major dependencies

SHOULD provide API user manual

In addition to the API Specification, it is good practice to provide a separate API user manual to improve client developer experience, especially of engineers that are less experienced in using this API. A helpful API user manual typically describes the following API aspects:

- concrete examples of API usage
- edge cases, error situation details, and repair hints
- architecture context and dependencies - including figures and sequence flows

The user manual must be published online, e.g. via our documentation hosting platform service. Please do not forget to include a link to the API user manual into the API specification using the #/externalDocs/url property.

NOTE	This manual does not have to be created by engineering, but could be created by a documentation team, Developer Relations or by community effort. It is important to provide extra documentation for our developers to reduce the number of support related questions that come in.

MUST describe every parameter and property

Every query/path parameter and request/response property in the API specification must have a description

MUST provide an example for every parameter and property

Every query/path parameter and request/response property in the API specification must have an accurate example. An accurate example will show the API consumer what an input/output value will realistically look like, and could even be used in a real request/response. However, take care to not use personally identifiable information or secrets in examples.

Every operation (POST, PUT, PATCH, etc) may define one or more operation level examples.

SHOULD keep operation summaries at five or less words

Certain tools, like Postman, have a limit on how many words can be displayed within the operation summary. It is helpful for consumers to have short summaries that describe the operation at its most basic level to improve readability for consumers.

MUST write APIs using U.S. English

4. Meta information

MUST contain API meta information

API specifications must contain the following OpenAPI meta information to allow for API management:

- `#/info/title` as (unique) identifying, functional descriptive name of the API
- `#/info/version` to distinguish API specifications versions following semantic rules
- `#/info/description` containing a proper description of the API
- `#/info/x-audience` intended target audience of the API (see rule 219)

NOTE	We'll automatically generate `#/info/contact/*` when creating the public Open API spec.

MUST use semantic versioning

OpenAPI allows to specify the API specification version in `#/info/version`. To share a common semantic of version information we expect API designers to comply 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.

Additional Notes:

- **Pre-release** versions (rule 9) and **build metadata** (rule 10) must not be used in API version information.
- While patch versions are useful for fixing typos etc, API designers are free to decide whether they increment it or not.
- API designers should consider to use API version `0.y.z` (rule 4) for initial API design.

Example:

```
openapi: 3.0.1
info:
  title: Parcel Service API
  description: API for <...>
  version: 1.3.7
  <...>
```

NOTE We'll automatically generate the global `#/info/version` when creating public Open API spec. Each individual endpoint spec author need not worry about this attribute.

MUST provide API audience

Each API must be classified with respect to the intended target audience supposed to consume the API, to facilitate differentiated standards on APIs for discoverability, changeability, quality of design and documentation, as well as permission granting. We differentiate the following API audience groups with clear organizational and legal boundaries:

`internal-company` `external-public`

NOTE This is only for documentation generation purposes and not related to authorization—authz concerns must be addressed with normal policies.

```
/info/x-audience:
  type: string
  x-extensible-enum:
    - internal-company
    - external-public
  description: |
    Intended target audience of the API. Relevant for standards around
    quality of design and documentation, reviews, discoverability,
    changeability.
```

NOTE Exactly one audience per API specification is allowed. For this reason a smaller audience group is intentionally included in the wider group and thus does not need to be declared additionally. If parts of your API have a different target audience, we recommend to split API specifications along the target audience — even if this creates redundancies (rationale (internal link)).

Example:

```
openapi: 3.0.1
info:
  x-audience: internal-company
  title: Service to Service API
  description: API for <...>
  version: 1.2.4
  <...>
```

5. Security

MUST secure endpoints with OAuth 2.0

Every public API endpoint must be secured using OAuth 2.0.

The following code snippet shows how to define the authorization scheme using a bearer token (e.g. JWT token).

```
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

The next code snippet applies this security scheme to all API endpoints. The bearer token of the client must have additionally the scopes scope_1 and scope_2.

```
security:
  - bearerAuth: [ scope_1, scope_2 ]
```

MUST define and assign permissions (scopes)

Each endpoint must document every AMS right needed to access the endpoint. If creating a new AMS right that doesn't exist in the list, then it must be added to the list first.

To appropriately document AMS rights on an endpoint, use the following OpenAPI properties:

```
security:
  - bearerAuth:
     - 'idn:task-definition:read'
     - 'idn:task-definition:write'
```

A full example on an endpoint might look like this:

```yaml
get:
  tags:
    - tenants
  summary: Get Tenant
  description: Get tenant object based on current auth token
  operationId: getTenant
  responses:
    '200':
      description: successful operation
      content:
        application/json:
          schema:
            $ref: '../schemas/Tenant.yaml'
  security:
    - bearerAuth:
        - 'cam:general:read'
```

MUST Document necessary license add-on to use an API collection

If an API collection requires additional product licenses to enable the feature, then each required license must be documented in the API collection description.

MUST follow naming convention for permissions (scopes)

TBD

6. Compatibility

MUST not break backward compatibility

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

There are two techniques to change APIs without breaking them:

- follow rules for compatible extensions
- introduce new API versions and still support older versions

We strongly encourage using compatible API extensions and discourage versioning (see **SHOULD** avoid versioning and **MUST** use URI versioning below). The following guidelines for service providers (**SHOULD** prefer compatible extensions) and consumers (**MUST** prepare clients to accept

compatible API extensions) enable us (having Postel's Law in mind) to make compatible changes without versioning.

Note: There is a difference between incompatible and breaking changes. Incompatible changes are changes that are not covered by the compatibility rules below. Breaking changes are incompatible changes deployed into operation, and thereby breaking running API consumers. Usually, incompatible changes are breaking changes when deployed into operation. However, in specific controlled situations it is possible to deploy incompatible changes in a non-breaking way, if no API consumer is using the affected API aspects (see also Deprecation guidelines).

Hint: Please note that the compatibility guarantees are for the "on the wire" format. Binary or source compatibility of code generated from an API definition is not covered by these rules. If client implementations update their generation process to a new version of the API definition, it has to be expected that code changes are necessary.

SHOULD prefer compatible extensions

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

- Add only optional, never mandatory fields.
- Never change the semantic of fields (e.g. changing the semantic from customer-number to customer-id, as both are different unique customer keys)
- Input fields may have (complex) constraints being validated via server-side business logic. Never change the validation logic to be more restrictive and make sure that all constraints are clearly defined in description.
- Enum ranges can be reduced when used as input parameters, only if the server is ready to accept and handle old range values too. Enum range can be reduced when used as output parameters.
- Enum ranges cannot be extended when used for output parameters — clients may not be prepared to handle it. However, enum ranges can be extended when used for input parameters.
- Support redirection in case an URL has to change 301 (Moved Permanently).

MUST prepare clients to accept compatible API extensions

Service clients should apply the robustness principle:

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

Service clients must be prepared for compatible API extensions of service providers:

- Be tolerant with unknown fields in the payload (see also Fowler's "TolerantReader" post), i.e. ignore new fields but do not eliminate them from payload if needed for subsequent `PUT`

requests.

- Be prepared that `x-extensible-enum` return parameter may deliver new values; either be agnostic or provide default behavior for unknown values.
- Be prepared to handle HTTP status codes not explicitly specified in endpoint definitions. Note also, that status codes are extensible. Default handling is how you would treat the corresponding 2xx code (see RFC 7231 Section 6).
- Follow the redirect when the server returns HTTP status code 301 (Moved Permanently).

SHOULD design APIs conservatively

Designers of service provider APIs should be conservative and accurate in what they accept from clients:

- Unknown input fields in payload or URL should not be ignored; servers should provide error feedback to clients via an HTTP 400 response code.
- Be accurate in defining input data constraints (like formats, ranges, lengths etc.) — and check constraints and return dedicated error information in case of violations.
- Prefer being more specific and restrictive (if compliant to functional requirements), e.g. by defining length range of strings. It may simplify implementation while providing freedom for further evolution as compatible extensions.

Not ignoring unknown input fields is a specific deviation from Postel's Law (e.g. see also The Robustness Principle Reconsidered) and a strong recommendation. Servers might want to take different approach but should be aware of the following problems and be explicit in what is supported:

- Ignoring unknown input fields is actually not an option for `PUT`, since it becomes asymmetric with subsequent `GET` response and HTTP is clear about the `PUT` *replace* semantics and default roundtrip expectations (see RFC 7231 Section 4.3.4). Note, accepting (i.e. not ignoring) unknown input fields and returning it in subsequent `GET` responses is a different situation and compliant to `PUT` semantics.
- Certain client errors cannot be recognized by servers, e.g. attribute name typing errors will be ignored without server error feedback. The server cannot differentiate between the client intentionally providing an additional field versus the client sending a mistakenly named field, when the client's actual intent was to provide an optional input field.
- Future extensions of the input data structure might be in conflict with already ignored fields and, hence, will not be compatible, i.e. break clients that already use this field but with different type.

In specific situations, where a (known) input field is not needed anymore, it either can stay in the API definition with "not used anymore" description or can be removed from the API definition as long as the server ignores this specific parameter.

MUST always return JSON objects as top-level data structures

In a response body, you must always return a JSON object (and not e.g. an array) as a top level data structure to support future extensibility. JSON objects support compatible extension by additional attributes. This allows you to easily extend your response and e.g. add pagination later, without breaking backwards compatibility. See **MAY** use pagination links where applicable for an example.

Maps (see **SHOULD** define maps using `additionalProperties`), even though technically objects, are also forbidden as top level data structures, since they don't support compatible, future extensions.

MUST treat OpenAPI specification as open for extension by default

The OpenAPI specification is not very specific on default extensibility of objects, and redefines JSON-Schema keywords related to extensibility, like `additionalProperties`. Following our compatibility guidelines, OpenAPI object definitions are considered open for extension by default as per Section 5.18 "additionalProperties" of JSON-Schema.

When it comes to OpenAPI, this means an `additionalProperties` declaration is not required to make an object definition extensible:

- API clients consuming data must not assume that objects are closed for extension in the absence of an `additionalProperties` declaration and must ignore fields sent by the server they cannot process. This allows API servers to evolve their data formats.
- For API servers receiving unexpected data, the situation is slightly different. Instead of ignoring fields, servers *may* reject requests whose entities contain undefined fields in order to signal to clients that those fields would not be stored on behalf of the client. API designers must document clearly how unexpected fields are handled for `PUT`, `POST`, and `PATCH` requests.

API formats must not declare `additionalProperties` to be false, as this prevents objects being extended in the future.

Note that this guideline concentrates on default extensibility and does not exclude the use of `additionalProperties` with a schema as a value, which might be appropriate in some circumstances, e.g. see **SHOULD** define maps using `additionalProperties`.

SHOULD avoid versioning

When changing your RESTful APIs, do so in a compatible way and avoid generating additional API versions. Multiple versions can significantly complicate understanding, testing, maintaining, evolving, operating and releasing our systems (supplementary reading).

If changing an API can't be done in a compatible way, then proceed in one of these three ways:

- create a new resource (variant) in addition to the old resource variant
- create a new service endpoint — i.e. a new application with a new API (with a new domain

name)

- create a new API version supported in parallel with the old API by the same microservice

MUST use URI versioning

SailPoint uses URI versioning with the following structure:

/v{version number}

Ex. /v3, /v4, etc.

Note: Beta APIs fall under /beta.

MUST follow versioned API requirements

All versioned APIs must adhere to the following requirements

- Can be internal or external.
- NO BREAKING CHANGES! (barring necessary security fixes)
- Supported (sev-1!)
- Documented in OpenAPI specification format
- Minimum 2 year support obligation

Breaking changes include, but are not limited to, the following:

- Removal of fields on resource models
- Changing an existing field on a post/put/patch from optional to required (or not permitted)
- Removal of resources
- URI changes
- New or different response status codes
- Other semantic changes (including new values in enumerated types if those values are part of the API contract)
- Example: Consider a list API that returns multiple object types with a common standardized representation and the objects have a TYPE field. Adding a new TYPE value would represent a breaking change UNLESS the the contract for the API specifies that new types may be added. If there is no means of filtering based on TYPE in this example, this is probably an unacceptable term for the API contract.

If in doubt, ask! Call out any uncertainties during API review or during the design process.

Non-breaking changes may be added to an existing live version. Here are some examples of non-breaking changes:

- New fields on response models (that do not change the meaning of the model)
- Making a currently required field on an input model optional

- Adding new optional fields to input models (as long as the default value of the field preserves the previous meaning of the model)
- Net new resources
- New query parameters and filterable/sortable fields (as long as they are optional and the existing behavior is preserved if the new parameters are not passed)

MUST follow beta API requirements

- Beta route in styx (example: "beta/sim-integrations")
- Supported (during business hours, no sev-1s)
- Documented in OpenAPI specification format (and marked as beta)
- Breaking changes must be announced ahead of time to all stakeholders
- Can be internal or external.
- This is expected to be a stable, usable API that is available for external and internal integration.

7. Deprecation

Sometimes it is necessary to phase out an API endpoint, an API version, or an API feature, e.g. if a field or parameter is no longer supported or a whole business functionality behind an endpoint is supposed to be shut down. As long as the API endpoints and features are still used by consumers these shut downs are breaking changes and not allowed. To progress the following deprecation rules have to be applied to make sure that the necessary consumer changes and actions are well communicated and aligned using *deprecation* and *sunset* dates.

SHOULD Confer with clients on accepted deprecation time-span

Before shutting down an API, version of an API, or API feature the producer should make sure that all clients have given their consent on a sunset date. Producers should help consumers to migrate to a potential new API or API feature by providing a migration manual and clearly state the time line for replacement availability and sunset (see also **SHOULD** add `Deprecation` and `Sunset` header to responses). The producer should wait for all clients of a sunset API feature to migrate before shutting down the deprecated API.

MUST reflect deprecation in API specifications

The API deprecation must be part of the API specification.

If an API endpoint (operation object), an input argument (parameter object), an in/out data object (schema object), or on a more fine grained level, a schema attribute or property should be deprecated, the producers must set `deprecated: true` for the affected element and add further explanation to the `description` section of the API specification. If a future shut down is planned, the producer must provide a sunset date and document in details what consumers should use instead

and how to migrate.

MUST monitor usage of deprecated API scheduled for sunset

Owners of an API, API version, or API feature used in production that is scheduled for sunset must monitor the usage of the sunset API, API version, or API feature in order to observe migration progress and avoid uncontrolled breaking effects on ongoing consumers. See also **MUST** monitor API usage.

Must notify customers using deprecated APIs on a timely basis to help them move onto newer APIs, especially as the API moves closer to its sunset date.

SHOULD add `Deprecation` and `Sunset` header to responses

During the deprecation phase, the producer should add a `Deprecation: <date-time>` (see draft: RFC Deprecation HTTP Header) and - if also planned - a `Sunset: <date-time>` (see RFC 8594) header on each response affected by a deprecated element (see **MUST** reflect deprecation in API specifications).

The `Deprecation` header can either be set to `true` - if a feature is retired -, or carry a deprecation time stamp, at which a replacement will become/became available and consumers must not on-board any longer (see **MUST** not start using deprecated APIs). The optional `Sunset` time stamp carries the information when consumers latest have to stop using a feature. The sunset date should always offer an eligible time interval for switching to a replacement feature.

```
Deprecation: Tue, 31 Dec 2024 23:59:59 GMT
Sunset: Wed, 31 Dec 2025 23:59:59 GMT
```

If multiple elements are deprecated the `Deprecation` and `Sunset` headers are expected to be set to the earliest time stamp to reflect the shortest interval consumers are expected to get active.

Note: adding the `Deprecation` and `Sunset` header is not sufficient to gain client consent to shut down an API or feature.

Hint: In earlier guideline versions, we used the `Warning` header to provide the deprecation info to clients. However, `Warning` header has a less specific semantics, will be obsolete with draft: RFC HTTP Caching, and our syntax was not compliant with RFC 7234 — Warning header.

SHOULD add monitoring for `Deprecation` and `Sunset` header

Clients should monitor the `Deprecation` and `Sunset` headers in HTTP responses to get information about future sunset of APIs and API features (see **SHOULD** add `Deprecation` and `Sunset` header to responses). We recommend that client owners build alerts on this monitoring information to ensure alignment with service owners on required migration task.

Hint: In earlier guideline versions, we used the `Warning` header to provide the deprecation info (see hint in **SHOULD** add `Deprecation` and `Sunset` header to responses).

MUST not start using deprecated APIs

Clients must not start using deprecated APIs, API versions, or API features.

8. JSON guidelines

These guidelines provides recommendations for defining JSON data at SailPoint. JSON here refers to RFC 7159 (which updates RFC 4627), the "application/json" media type and custom JSON media types defined for APIs. The guidelines clarifies some specific cases to allow SailPoint JSON data to have an idiomatic form across teams and services.

The first some of the following guidelines are about property names, the later ones about values.

SHOULD pluralize array names

Names of arrays should be pluralized to indicate that they contain multiple values. This implies in turn that object names should be singular.

MUST property names must be ASCII camelCase

Property names are restricted to ASCII strings. The first character must be a lower case letter, there must not be any spaces, and new words should start with a capital letter instead of a space. For example, “new client table” would be “newClientTable” in camelCase.

“ID” is common in field names, and must be presented in camel case as follows:

- If “ID” appears as the first word, then it is entirely lowercase (ex. “id”).
- If “ID” appears after the first word, then the “I” is capitalized and the “d” is lowercase (ex. “userId”).

MUST declare enum values using UPPER_SNAKE_CASE string

Enumerations must be represented as `string` typed OpenAPI definitions of request parameters or model properties. Enum values (using `enum` or `x-extensible-enum`) need to consistently use the upper-snake case format, e.g. `VALUE` or `YET_ANOTHER_VALUE`. This approach allows to clearly distinguish values from properties or other elements.

Exception: This rule does not apply for case sensitive values sourced from outside API definition scope, e.g. for language codes from ISO 639-1, or when declaring possible values for a rule 137 [`sort` parameter].

SHOULD define maps using `additionalProperties`

A "map" here is a mapping from string keys to some other type. In JSON this is represented as an object, the key-value pairs being represented by property names and property values. In OpenAPI schema (as well as in JSON schema) they should be represented using additionalProperties with a schema defining the value type. Such an object should normally have no other defined properties.

The map keys don't count as property names in the sense of rule 118, and can follow whatever format is natural for their domain. Please document this in the description of the map object's schema.

Here is an example for such a map definition (the `translations` property):

```
components:
  schemas:
    Message:
      description:
        A message together with translations in several languages.
      type: object
      properties:
        message_key:
          type: string
          description: The message key.
        translations:
          description:
            The translations of this message into several languages.
            The keys are [IETF BCP-47 language
tags](https://tools.ietf.org/html/bcp47).
          type: object
          additionalProperties:
            type: string
            description:
              the translation of this message into the language identified by the key.
```

An actual JSON object described by this might then look like this:

```
{ "message_key": "color",
  "translations": {
    "de": "Farbe",
    "en-US": "color",
    "en-GB": "colour",
    "eo": "koloro",
    "nl": "kleur"
  }
}
```

MUST not use `null` for boolean properties

Schema based JSON properties that are by design booleans must not be presented as nulls. A boolean is essentially a closed enumeration of two values, true and false. If the content has a meaningful null value, strongly prefer to replace the boolean with enumeration of named values or statuses - for example accepted_terms_and_conditions with true or false can be replaced with terms_and_conditions with values yes, no and unknown.

MUST define a default value for boolean properties

All boolean properties must have a default value defined

SHOULD avoid using qualifying verbs

Avoid using qualifying verbs, especially on boolean fields, e.g.

- Discouraged: isEnabled
- Recommended: enabled

SHOULD use positive semantics for boolean fields

The name of a Boolean field should preferably express semantics such that true indicates a positive attribute, action, capability, etc.

- Discouraged: "disabled": true
- Recommended: "enabled": false

MUST use a field name that suggests the value type when referencing an object

When a field contains an ID or reference to an foreign object, not the parent object, the field name should suggest the value type:

- Discouraged: "owner": "2c90b0c06460804b016460f9f59b0015"

- Recommended: "ownerId": "2c90b0c06460804b016460f9f59b0015"

For example, the following request/response for an account object uses the proper naming for object references.

“id” refers to the account object being requested, and all other object references include the object reference name (i.e. sourceId, identityId, etc.)

```
{
    "id": "id12345",
    "name": "aName",
    "created": "2019-08-24T14:15:22Z",
    "modified": "2019-08-24T14:15:22Z",
    "sourceId": "2c9180835d2e5168015d32f890ca1581",
    "identityId": "2c9180835d2e5168015d32f890ca1581",
    "attributes": { },
    "authoritative": true,
    "description": "string",
    "disabled": true,
    "locked": true,
    "nativeIdentity": "string",
    "systemAccount": true,
    "uncorrelated": true,
    "uuid": "string",
    "manuallyCorrelated": true,
    "hasEntitlements": true
}
```

SHOULD name references to foreign objects as <objectName>Ref

- Discouraged: "launcher": "frank.dogs"
- Recommended: "launcherRef": {"resource": "identities", "type": "ALIAS", "value": "frank.dogs"}

Example

```
{
    "id": "2c9180857182305e0171993735622948",
    "name": "Alison Ferguso",
    "alias": "alison.ferguso",
    "email": "alison.ferguso@acme-solar.com",
    "status": "Active",
    "managerRef": {
        "type": "IDENTITY",
        "id": "2c9180a46faadee4016fb4e018c20639",
        "name": "Thomas Edison"
    },
    "attributes":[]
}
```

SHOULD avoid using nested objects

In general, we discourage nesting DTOs inside others. This has typically led to bloated DTOs and made it complicated to enforce authorization requirements and other business rules around those nested objects. It is preferable instead for the DTO to have a field containing an id or reference that allows the nested object to be separately fetched.

It is recognized, of course, that particular use cases may require nesting objects inside each other. For example, if a UI module needs to display data from a set of 100 IdentityRequests and their child IdentityRequestItems, it makes no sense to require the UI to make one API call to get the list of IdentityRequests and then 100 additional calls to get the IdentityRequestItems for each.

It is preferable in these cases to use a summary DTO for the nested objects that contains the minimum amount of detail required to support the known or plausible use case(s). For example, if the only reason I need to include the owner of an object is so the caller can display their first and last name, then it is better to do something like the following:

```
{
    ...
    "owner": {
        "type": "IDENTITY",
        "id": "2c90b0c06460804b016460f9f59b001",
        "firstName": "Frank",
        "lastName": "Dogs"
    }
}
```

One particular valid use of nested objects occurs when a DTO abstracts over a set of types that may have significantly different attributes. In this case the non-general fields of the DTO should be pushed down to a nested object, with a type field on the main object being used as a discriminator. For example, if a DTO could represent either an Access Profile or a Role, the former case could be implemented as follows:

```
{
     ...
     "type": "ACCESS_PROFILE",
     ...
     "accessProfileInfo": {
          "appRefs": ["app1", "app2"]
     },
     "roleInfo": null
}
```

MUST define a default for optional values

All properties must define a default value for optional properties. This must documented in the specification so clients know what value will be used should they ignore a property.

MUST define the “required” attribute for request/response objects and parameters

All request/response schemas MUST define the “required” attribute for each property and parameter per the OpenAPI specification. For request/response objects, see https://swagger.io/docs/specification/data-models/data-types/#required. For path and query parameters, see https://swagger.io/docs/specification/describing-parameters/

Generally, query parameters should be optional, but there are cases where a query parameter is required. In these cases, make sure to set the “required” attribute for the query parameters to true.

MUST use same semantics for `null` and absent properties

TBD

MUST use the “nullable” attribute for properties that can be null

If a property or parameter can return `null`, then it must have the `nullable: true` OpenAPI property.

MUST not use `null` for empty arrays

Empty array values can unambiguously be represented as the empty list, `[]`.

SHOULD define dates properties compliant with RFC 3339

Use the date and time formats defined by RFC 3339:

- for "date" use strings matching `date-fullyear "-" date-month "-" date-mday`, for example: `2015-05-28`
- for "date-time" use strings matching `full-date "T" full-time`, for example `2015-05-28T14:07:17Z`

Note that the OpenAPI format "date-time" corresponds to "date-time" in the RFC) and `2015-05-28` for a date (note that the OpenAPI format "date" corresponds to "full-date" in the RFC). Both are specific profiles, a subset of the international standard ISO 8601.

A zone offset may be used (both, in request and responses) — this is simply defined by the standards. However, we encourage restricting dates to UTC and without offsets. For example `2015-05-28T14:07:17Z` rather than `2015-05-28T14:07:17+00:00`. 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 be including daylight saving time. Localization of dates should be done by the services that provide user interfaces, if required.

When it comes to storage, all dates should be consistently stored in UTC without a zone offset. Localization should be done locally by the services that provide user interfaces, if required.

Sometimes it can seem data is naturally represented using numerical timestamps, but this can introduce interpretation issues with precision, e.g. whether to represent a timestamp as 1460062925, 1460062925000 or 1460062925.000. Date strings, though more verbose and requiring more effort to parse, avoid this ambiguity.

SHOULD define time durations and intervals properties conform to ISO 8601

Schema based JSON properties that are by design durations and intervals could be strings formatted as recommended by ISO 8601 (Appendix A of RFC 3339 contains a grammar for durations).

9. Data formats

MUST use JSON as payload data interchange format

Use JSON (RFC 7159) to represent structured (resource) data passed with HTTP requests and responses as body payload. The JSON payload must use a JSON object as top-level data structure (if possible) to allow for future extension. This also applies to collection resources, where you ad-hoc would use an array — see also **MUST** always return JSON objects as top-level data structures.

Additionally, the JSON payload must comply to the more restrictive Internet JSON (RFC 7493), particularly

- Section 2.1 on encoding of characters, and
- Section 2.3 on object constraints.

As a consequence, a JSON payload must

- use `UTF-8` encoding
- consist of valid Unicode strings, i.e. must not contain non-characters or surrogates, and
- contain only unique member names (no duplicate names).

MAY pass non-JSON media types using data specific standard formats

TBD

SHOULD use standard media types

You should use standard media types (defined in media type registry of Internet Assigned Numbers Authority (IANA)) as `content-type` (or `accept`) header information. More specifically, for JSON payload you should use the standard media type `application/json` (or `application/problem+json` for **MUST** support problem JSON).

You should avoid using custom media types like `application/x.sailpoint.article+json`. Custom media types beginning with `x` bring no advantage compared to the standard media type for JSON, and make automated processing more difficult.

MUST encode embedded binary data in `base64url`

Exposing binary data using an alternative media type is generally preferred. See the rule above.

If an alternative content representation is not desired then binary data should be embedded into the JSON document as a `base64url`-encoded string property following RFC 7493 Section 4.4.

MUST use standardized property formats

JSON Schema and OpenAPI define several data formats, e.g. `date`, `time`, `email`, and `url`, based on ISO and IETF standards. The following table lists these formats including additional formats useful in an e-commerce environment. You **should** use these formats, whenever applicable.

type	format	Specification	Example
integer	int32		7721071004
integer	int64		772107100456824
integer	bigint		77210710045682438959
number	float	IEEE 754-2008	3.1415927
number	double	IEEE 754-2008	3.141592653589793

type	format	Specification	Example
number	decimal		3.14159265358979323846264338327
string	bcp47	BCP 47	"en-DE"
string	byte	RFC 7493	"dGVzdA=="
string	date	RFC 3339	"2019-07-30"
string	date-time	RFC 3339	"2019-07-30T06:43:40.252Z"
string	email	RFC 5322	"example@sailpoint.de"
string	gtin-13	GTIN	"5710798389878"
string	hostname	RFC 1034	"www.sailpoint.de"
string	ipv4	RFC 2673	"104.75.173.179"
string	ipv6	RFC 2673	"2600:1401:2::8a"
string	iso-3166	ISO 3166-1 alpha-2	"DE"
string	iso-4217	ISO 4217	"EUR"
string	iso-639	ISO 639-1	"de"
string	json-pointer	RFC 6901	"/items/0/id"
string	password		"secret"
string	regex	ECMA 262	"^[a-z0-9]+$"
string	time	RFC 3339	"06:43:40.252Z"
string	uri	RFC 3986	"https://www.sailpoint.de/"
string	uri-template	RFC 6570	"/users/{id}"
string	uuid	RFC 4122	"e2ab873e-b295-11e9-9c02-…"

Remark: Please note that this list of standard data formats is not exhaustive and everyone is encouraged to propose additions.

MUST use standard date and time formats

JSON payload

Read more about date and time format in **SHOULD** define dates properties compliant with RFC 3339.

HTTP headers

Http headers including the proprietary headers use the HTTP date format defined in RFC 7231.

MUST use standards for country, language and currency codes

Use the following standard formats for country, language and currency codes:

- Country codes: ISO 3166-1-alpha2 two letter country codes
  - Hint: It is "GB", not "UK", even though "UK" has seen some use at sailpoint
- Language codes: ISO 639-1 two letter language codes
- Language variant tags: BCP 47
  - It is a compatible extension of ISO 639-1, providing additional information for language usage, like region (using ISO 3166-1), variant, script and others.
- Currency codes: ISO 4217 three letter currency codes

MUST define format for number and integer types

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 binary32 decimal number
number	double	IEEE 754-2008/ISO 60559:2011 binary64 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. for the following definitions the most specific language types in Java will translate to `BigDecimal` for `Money.amount` and `int` or `Integer` for the `OrderList.page_size`:

```yaml
components:
  schemas:
    Money:
      type: object
      properties:
        amount:
          type: number
          description: Amount expressed as a decimal number of major currency units
          format: decimal
          example: 99.95
        ...

    OrderList:
      type: object
      properties:
        page_size:
          type: integer
          description: Number of orders in list
          format: int32
          example: 42
```

10. Common data types

Definitions of data objects that are good candidates for wider usage. Below you can find a list of common data types used in the guideline:

- Money object
- Problem object
- Address object

MUST use the common money object

Use the following common money structure:

```yaml
Money:
  type: object
  properties:
    amount:
      type: number
      description: >
        The amount describes unit and subunit of the currency in a single value,
        where the integer part (digits before the decimal point) is for the
        major unit and fractional part (digits after the decimal point) is for
        the minor unit.
      format: decimal
      example: 99.95
    currency:
      type: string
      description: 3 letter currency code as defined by ISO-4217
      format: iso-4217
      example: EUR
  required:
    - amount
    - currency
```

APIs are encouraged to include a reference to the global schema for Money.

```yaml
SalesOrder:
  properties:
    grand_total:
      $ref: 'https://sailpoint-oss.github.io/sailpoint-api-guidelines/money-
1.0.0.yaml#/Money'
```

Please note that APIs have to treat Money as a closed data type, i.e. it's not meant to be used in an inheritance hierarchy. That means the following usage is not allowed:

```json
{
  "amount": 19.99,
  "currency": "EUR",
  "discounted_amount": 9.99
}
```

Cons

- Violates the Liskov Substitution Principle
- Breaks existing library support, e.g. Jackson Datatype Money
- Less flexible since both amounts are coupled together, e.g. mixed currencies are impossible

A better approach is to favor composition over inheritance:

```
{
    "price": {
      "amount": 19.99,
      "currency": "EUR"
    },
    "discounted_price": {
      "amount": 9.99,
      "currency": "EUR"
    }
}
```

Pros

- No inheritance, hence no issue with the substitution principle
- Makes use of existing library support
- No coupling, i.e. mixed currencies is an option
- Prices are now self-describing, atomic values

Notes

Please be aware that some business cases (e.g. transactions in Bitcoin) call for a higher precision, so applications must be prepared to accept values with unlimited precision, unless explicitly stated otherwise in the API specification.

Examples for correct representations (in EUR):

- `42.20` or `42.2` = 42 Euros, 20 Cent
- `0.23` = 23 Cent
- `42.0` or `42` = 42 Euros
- `1024.42` = 1024 Euros, 42 Cent
- `1024.4225` = 1024 Euros, 42.25 Cent

Make sure that you don't convert the "amount" field to `float` / `double` types when implementing this interface in a specific language or when doing calculations. Otherwise, you might lose precision. Instead, use exact formats like Java's `BigDecimal`. See Stack Overflow for more info.

Some JSON parsers (NodeJS's, for example) convert numbers to floats by default. After discussing the pros and cons we've decided on "decimal" as our amount format. It is not a standard OpenAPI format, but should help us to avoid parsing numbers as float / doubles.

MUST use common field names and semantics

TBD

11. API naming

MUST/SHOULD use functional naming schema

TBD

MUST use lowercase separate words with hyphens for path segments

- Use kebab-case for path segments
- Use {camelCase} with surrounding brackets to indicate path parameters

Example:

```
/shipment-orders/{shipmentOrderId}
```

MUST camelCase for query parameters

Examples:

```
customerNumber, orderId, billingAddress
```

We need to have a consistent look and feel for our APIs. In the case of query parameters, which can reference actual properties in the response object, camelCase preserves a consistent look and feel.

MUST pluralize resource names

When defining a path segment for a collection, the resource name must be plural to indicate it is a collection of resources.

Example:

`/users`, `/companies/{companyId}/employees/{employeeId}`

MUST not use `/api` as base path

In most cases, all resources provided by a service are part of the public API, and therefore should be made available under the root "/" base path.

If the service should also support non-public, internal APIs — for specific operational support functions, for example — we encourage you to maintain two different API specifications and provide API audience. For both APIs, you should not use `/api` as base path.

We see API's base path as a part of deployment variant configuration. Therefore, this information has to be declared in the server object.

MUST use normalized paths without empty path segments and trailing slashes

You must not specify paths with duplicate or trailing slashes, e.g. `/customers//addresses` or `/customers/`. As a consequence, you must also not specify or use path variables with empty string values.

Reasoning: Non standard paths have no clear semantics. As a result, behavior for non standard paths varies between different HTTP infrastructure components and libraries. This may leads to ambiguous and unexpected results during request handling and monitoring.

We recommend to implement services robust against clients not following this rule. All services **should** normalize request paths before processing by removing duplicate and trailing slashes. Hence, the following requests should refer to the same resource:

```
GET /orders/{orderId}
GET /orders/{orderId}/
GET /orders//{orderId}
```

Note: path normalization is not supported by all framework out-of-the-box. Services are required to support at least the normalized path while rejecting all alternatives paths, if failing to deliver the same resource.

MUST stick to conventional query parameters

If you provide query support for searching, sorting, filtering, and paginating, you must stick to the following naming conventions:

Pagination

- `limit`: Integer that specifies the maximum number of results to return. If not specified a default limit will be used.
- `offest`: Integer that specifies the offset of the first result from the beginning of the collection
- `count`: Boolean that indicates whether a total count will be returned, factoring in any filter

parameters, in the X-Total-Count response header.

`filters`: an item will only be included in the returned array if the filters expression evaluates to true for that item. Each endpoint that implements filters must clearly define in the API spec what operations and fields are supported.

`sorters`: a set of comma-separated field names. Each field name may be optionally prefixed with a "-" character, which indicates the sort is descending based on the value of that field. Otherwise, the sort is ascending. Each endpoint that implements sorters must clearly define which fields are supported.

See https://developer.sailpoint.com/docs/standard_collection_parameters.html#standard-collection-parameters for implementation details for each of the above parameters.

Note: Additional query parameters are allowed, but effort should be made to fit them within the five listed above.

MUST Customer org name must never appear in the path of public APIs

The customer organization is provided in the session context that is generated on the back end, and therefore does not need to be in the URL.

12. Resources

SHOULD avoid actions — think about resources

REST is all about your resources, so consider the domain entities that take part in web service interaction, and aim to model your API around these using the standard HTTP methods as operation indicators. For example, rather than creating a specific action for completing a certification campaign, prefer to use PATCH to update the completed status of the campaign.

Request

```
PATCH v1/campaigns/{campaignId}
```

Body

```
{
    "completed": true
}
```

Sometimes, standard HTTP methods aren't specific enough to indicate the action you wish to perform on a resource, or there is complex business logic on the back end that can't be satisfied by PATCHing a single field. In these cases, it is advisable to use the following URI format for specific

resource actions:

Request

```
POST v1/campaigns/{campaignId}/remediation-scan
```

SHOULD model complete business processes

TBD

SHOULD define *useful* resources

As a rule of thumb resources should be defined to cover 90% of all its client's use cases. A *useful* resource should contain as much information as necessary, but as little as possible. A great way to support the last 10% is to allow clients to specify their needs for more/less information by supporting filtering and embedding.

SHOULD keep URLs verb-free

The API describes resources, so the only place where actions should appear is in the HTTP methods. In URLs, use only nouns. Instead of thinking of actions (verbs), it's often helpful to think about putting a message in a letter box: e.g., instead of having the verb *cancel* in the url, think of sending a message to cancel an order to the *cancellations* letter box on the server side.

MUST use domain-specific resource names

API resources represent elements of the application's domain model. Using domain-specific nomenclature for resource names helps developers to understand the functionality and basic semantics of your resources. It also reduces the need for further documentation outside the API definition. For example, "sales-order-items" is superior to "order-items" in that it clearly indicates which business object it represents. Along these lines, "items" is too general.

MUST use URL-friendly resource identifiers: [a-zA-Z0-9:._\-/]*

TBD

MUST identify resources and sub-resources via path segments

Some API resources may contain or reference sub-resources. Sub-resources should be referenced by their name and identifier in the path segments as follows:

```
/resources/{resourceId}/sub-resources/{subResourceId}
```

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 instance, if `/partners/{partnerId}/addresses/{addressId}` is valid, then, in principle, also `/partners/{partnerId}/addresses`, `/partners/{partnerId}` and `/partners` must be valid. Examples of concrete url paths:

```
/shopping-carts/de:1681e6b88ec1/items/1
/shopping-carts/de:1681e6b88ec1
/content/images/9cacb4d8
/content/images
```

Note: resource identifiers may be build of multiple other resource identifiers (see [241]).

Exception: In some situations the resource identifier is not passed as a path segment but via the authorization information, e.g. an authorization token or session cookie. Here, it is reasonable to use `self` as *pseudo-identifier* path segment. For instance, you may define `/employees/self` or `/employees/self/personal-details` as resource paths — and may additionally define endpoints that support identifier passing in the resource path, like define `/employees/{emplId}` or `/employees/{emplId}/personal-details`.

SHOULD consider using (non-)nested URLs

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

```
/shoping-carts/de/1681e6b88ec1/cart-items/1
```

However, if the resource can be accessed directly via its unique id, then the API should expose it as a top-level resource. For example, customer has a collection for sales orders; however, sales orders have globally unique id and some services may choose to access the orders directly, for instance:

```
/customers/1637asikzec1
/sales-orders/5273gh3k525a
```

MUST not use sequential, numerical IDs

Numerical, sequential IDs are considered a security risk because malicious actors can enumerate through the IDs to obtain unauthorized information. API producers must use GUIDs or natural keys that aren't sequential.

SHOULD limit number of resource types

TBD

SHOULD limit number of sub-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 subresource entities. You should use ⇐ 3 sub-resource (nesting) levels — more levels increase API complexity and url path length. (Remember, some popular web browsers do not support URLs of more than 2000 characters.)

13. HTTP requests and responses

MUST use HTTP methods correctly

Be compliant with the standardized HTTP method semantics summarized as follows:

Creating an Object

`POST` requests are idiomatically used to **create** single resources on a collection resource endpoint, but other semantics on single resources endpoint are equally possible. The semantic for collection endpoints is best described as *"please add the enclosed representation to the collection resource identified by the URL"*. The semantic for single resource endpoints is best described as *"please execute the given well specified request on the resource identified by the URL"*.

- Modeled as `POST /⋯/plural-noun`, where `plural-noun` indicates the type of object being created.
- on a successful POST request, the server will create one or multiple new resources and provide their URI/URLs in the response
- successful POST requests will usually generate 200 (if resources have been updated), 201 (if resources have been created), 202 (if the request was accepted but has not been finished yet), and exceptionally 204 with Location header (if the actual resource is not returned).
- If the POST is used to create or update a resource, then the response payload needs to include every field from the request, and may include additional fields.
- If the POST is used as an action, then the response may be different from the request schema.
- Specifying a value for a system-generated field in the input results in a 400 Bad Request response.

Note: By using `POST` to create resources the resource ID must not be passed as request input date by the client, but created and maintained by the service and returned with the response payload.

Apart from resource creation, `POST` should be also used for scenarios that cannot be covered by the other methods sufficiently. However, in such cases make sure to document the fact that `POST` is used as a workaround (see e.g. `GET with body`).

Hint: Posting the same resource twice is **not** required to be idempotent (check **MUST** fulfill common method properties) and may result in multiple resources. However, you **SHOULD** consider to design `POST` and `PATCH` idempotent to prevent this.

Reading a Single Object

- Modeled as `GET /⋯/plural-noun/{nounId}`
- On success, returns a 200 with the DTO.
- A 404 is returned if the referenced object is not found.
- Query params are allowed, for example, to return the object at different levels of detail.
- Is free of side-effects.

Reading a List of Objects

- Modeled as `GET /⋯/plural-noun`.
- On success, returns a 200 with a JSON array of objects enveloped inside an object. By returning an object as the top level for all responses, we allow our APIs to extend without breaking backwards compatibility. If there was ever a need to add an additional field to a response that returns an array (ex. pagination links in the body), then we would need to break the API by wrapping it in an object. By requiring objects at the top level from the start, we avoid this in the future.
- A response object for a list of results must contain an array of `results`. Other properties are optional.

```
{
  "results": [...],
  "count": ...
}
```

- Supports pagination via `limit` and `offset` query parameters unless the back end data store makes this impossible or prohibitively expensive.
- The default value for `limit` is 250 unless the endpoint documentation states otherwise.
- The standard `filters` query parameter is preferred over custom filtering query params.
- If `filters` are used, the supported fields and operations are whitelisted. Unsupported filters should result in an error response.
- If `filters` refer to fields in nested objects, then "." notation is used, for example `filters=owner.name eq "leah.pierce"`
- Custom query params are allowed if filters cannot be used.
- If at all possible, supports reading a list of objects by their ids, either in the form of a filter, i.e. `filters=id in (id0, id1, ⋯, idN)` or a custom query param.
- Use of single boolean-valued params should be avoided; strings, enumerated values, or comma-separated values are preferred.

- The standard `sorters` query parameter may be used for sorting.
- If `sorters` are used, the supported fields are whitelisted.
- Results are not implicitly filtered or scoped based on the current logged in user. If such filtering is required it is via an explicit query param taking an identity id. By convention, me can stand in for the currently logged in user's identity id as the value for such a param.
- Is free of side-effects.

Get with Body Payload

APIs sometimes face the problem, that they have to provide extensive structured request information with GET, that may conflict with the size limits of clients, load-balancers, and servers. As we require APIs to be standard conform (request body payload in GET must be ignored on server side), API designers have to check the following two options:

1. GET with URL encoded query parameters: when it is possible to encode the request information in query parameters, respecting the usual size limits of clients, gateways, and servers, this should be the first choice. The request information can either be provided via multiple query parameters or by a single structured URL encoded string.
2. POST with body payload content: when a GET with URL encoded query parameters is not possible, a POST request with body payload must be used, and explicitly documented with a hint like in the following example:

```
paths:
  /products:
    post:
      description: >
        [GET with body payload](https://sailpoint-oss.github.io/sailpoint-api-
guidelines/#get-with-body) - no resources created:
        Returns all products matching the query passed as request input payload.
      requestBody:
        required: true
        content:
          ...
```

Updating an Object by Full Replacement

Modeled as `PUT /…/plural-noun/{nounId}`

`PUT` requests are used to **update** (and sometimes to create) **entire** resources – single or collection resources. The semantic is best described as *"please put the enclosed representation at the resource mentioned by the URL, replacing any existing resource."*.

- `PUT` requests are usually applied to single resources, and not to collection resources, as this would imply replacing the entire collection
- `PUT` requests are usually robust against non-existence of resources by implicitly creating the resource before updating

- on successful `PUT` requests, the server will **replace the entire resource** addressed by the URL with the representation passed in the payload (subsequent reads will deliver the same payload)
- successful `PUT` requests will usually generate 200 or 204 (if the resource was updated – with or without actual content returned), and 201 (if the resource was created)
- Returns a 404 if the object does not exist and the endpoint does not support PUT as a means of creation.
- Does a complete replacement of the referenced object and does not attempt to merge the input DTO with the existing object.

Important: It is good practice to prefer `POST` over `PUT` for creation of (at least top-level) resources. This leaves the resource ID management under control of the service and not the client, and focus `PUT` on its usage for updates. However, in situations where `PUT` is used for resource creation, the resource IDs are maintained by the client and passed as a URL path segment. Putting the same resource twice is required to be idempotent and to result in the same single resource instance (see **MUST** fulfill common method properties).

Hint: To prevent unnoticed concurrent updates and duplicate creations when using `PUT`, you **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header to allow the server to react on stricter demands that expose conflicts and prevent lost updates. See also Optimistic locking in RESTful APIs for details and options.

Updating an Object by Targeted Modification

`PATCH` requests are used to **update parts** of single resources, i.e. where only a specific subset of resource fields should be replaced. The semantic is best described as *"please change the resource identified by the URL according to my change request"*. The semantic of the change request is not defined in the HTTP standard and must be described in the API specification by using suitable media types.

- Modeled as `PATCH /../plural-noun/{nounId}`
- `PATCH` requests are usually applied to single resources as patching entire collection is challenging
- `PATCH` requests are usually not robust against non-existence of resource instances
- on successful `PATCH` requests, the server will update parts of the resource addressed by the URL as defined by the change request in the payload
- successful `PATCH` requests will usually generate 200 or 204 (if resources have been updated with or without updated content returned)
- Returns a 404 if the object does not exist.
- If synchronous, and the patch cannot be successfully applied, returns a 400.
- Mutable DTO fields are documented.

Deleting an Object

`DELETE` requests are used to **delete** resources. The semantic is best described as *"please delete the resource identified by the URL"*.

- Modeled as `DELETE /../plural-noun/{nounId}`
- `DELETE` requests are usually applied to single resources, not on collection resources, as this would imply deleting the entire collection.
- `DELETE` request can be applied to multiple resources at once using query parameters on the collection resource (see DELETE with query parameters).
- successful `DELETE` requests will usually generate 200 (if the deleted resource is returned) or 204 (if no content is returned).
- failed `DELETE` requests will usually generate 404 (if the resource cannot be found) or 410 (if the resource was already deleted before).

Important: After deleting a resource with `DELETE`, a `GET` request on the resource is expected to either return 404 (not found) or 410 (gone) depending on how the resource is represented after deletion. Under no circumstances the resource must be accessible after this operation on its endpoint.

DELETE with query parameters

`DELETE` request can have query parameters. Query parameters should be used as filter parameters on a resource and not for passing context information to control the operation behavior.

```
DELETE /resources?param1=value1&param2=value2...&paramN=valueN
```

Note: When providing `DELETE` with query parameters, API designers must carefully document the behavior in case of (partial) failures to manage client expectations properly.

The response status code of `DELETE` with query parameters requests should be similar to usual `DELETE` requests. In addition, it may return the status code 207 using a payload describing the operation results (see **MUST** use code 207 for batch or bulk requests for details).

DELETE with body payload

In rare cases `DELETE` may require additional information, that cannot be classified as filter parameters and thus should be transported via request body payload, to perform the operation. Since RFC-7231 states, that `DELETE` has an undefined semantic for payloads, we recommend to utilize `POST`. In this case the POST endpoint must be documented with the hint `DELETE with body` analog to how it is defined for `GET with body`. The response status code of `DELETE with body` requests should be similar to usual `DELETE` requests.

HEAD (Optional)

`HEAD` requests are used to **retrieve** the header information of single resources and resource collections.

- `HEAD` has exactly the same semantics as `GET`, but returns headers only, no body.

Hint: `HEAD` is particular useful to efficiently lookup whether large resources or collection resources have been updated in conjunction with the `ETag`-header.

OPTIONS (Optional)

`OPTIONS` requests are used to **inspect** the available operations (HTTP methods) of a given endpoint.

- `OPTIONS` responses usually either return a comma separated list of methods in the `Allow` header or as a structured list of link templates

Note: `OPTIONS` is rarely implemented, though it could be used to self-describe the full functionality of a resource.

MUST fulfill common method properties

Request methods in RESTful services can be…

- safe - the operation semantic is defined to be read-only, meaning it must not have *intended side effects*, i.e. changes, to the server state.
- idempotent - the operation has the same *intended effect* on the server state, independently whether it is executed once or multiple times. **Note:** this does not require that the operation is returning the same response or status code.
- cacheable - to indicate that responses are allowed to be stored for future reuse. In general, requests to safe methods are cachable, if it does not require a current or authoritative response from the server.

Note: The above definitions, of *intended (side) effect* allows the server to provide additional state changing behavior as logging, accounting, pre- fetching, etc. However, these actual effects and state changes, must not be intended by the operation so that it can be held accountable.

Method implementations must fulfill the following basic properties according to RFC 7231:

Method	Safe	Idempotent	Cacheable
`GET`	✓ Yes	✓ Yes	✓ Yes
`HEAD`	✓ Yes	✓ Yes	✓ Yes
`POST`	✗ No	⚠️ No, but SHOULD consider to design <code>POST</code> and <code>PATCH</code> idempotent	⚠️ May, but only if specific <code>POST</code> endpoint is safe. Hint: not supported by most caches.
`PUT`	✗ No	✓ Yes	✗ No

Method	Safe	Idempotent	Cacheable
PATCH	☐ No	⚠️ No, but SHOULD consider to design <code>POST</code> and <code>PATCH</code> idempotent	☐ No
DELETE	☐ No	☐ Yes	☐ No
OPTIONS	☐ Yes	☐ Yes	☐ No
TRACE	☐ Yes	☐ Yes	☐ No

Note: [227].

SHOULD consider to design POST and PATCH idempotent

In many cases it is helpful or even necessary to design POST and PATCH idempotent for clients to expose conflicts and prevent resource duplicate (a.k.a. zombie resources) or lost updates, e.g. if same resources may be created or changed in parallel or multiple times. To design an idempotent API endpoint owners should consider to apply one of the following three patterns.

- A resource specific **conditional key** provided via If-Match header in the request. The key is in general a meta information of the resource, e.g. a *hash* or *version number*, often stored with it. It allows to detect concurrent creations and updates to ensure idempotent behavior (see **MAY** consider to support ETag together with If-Match/If-None-Match header).
- A resource specific **secondary key** provided as resource property in the request body. The *secondary key* is stored permanently in the resource. It allows to ensure idempotent behavior by looking up the unique secondary key in case of multiple independent resource creations from different clients (see **MAY** use secondary key for idempotent POST design).
- A client specific **idempotency key** provided via Idempotency-Key header in the request. The key is not part of the resource but stored temporarily pointing to the original response to ensure idempotent behavior when retrying a request (see **MAY** consider to support Idempotency-Key header).

Note: While **conditional key** and **secondary key** are focused on handling concurrent requests, the **idempotency key** is focused on providing the exact same responses, which is even a *stronger* requirement than the idempotency defined above. It can be combined with the two other patterns.

To decide, which pattern is suitable for your use case, please consult the following table showing the major properties of each pattern:

	Conditional Key	Secondary Key	Idempotency Key
Applicable with	`PATCH`	`POST`	`POST`/`PATCH`
HTTP Standard	✓ Yes	✗ No	✗ No
Prevents duplicate (zombie) resources	✓ Yes	✓ Yes	✗ No
Prevents concurrent lost updates	✓ Yes	✗ No	✗ No
Supports safe retries	✓ Yes	✓ Yes	✓ Yes
Supports exact same response	✗ No	✗ No	✓ Yes
Can be inspected (by intermediaries)	✓ Yes	✗ No	✓ Yes
Usable without previous `GET`	✗ No	✓ Yes	✓ Yes

Note: The patterns applicable to `PATCH` can be applied in the same way to `PUT` and `DELETE` providing the same properties.

If you mainly aim to support safe retries, we suggest to apply conditional key and secondary key pattern before the Idempotency Key pattern.

MAY use secondary key for idempotent `POST` design

The most important pattern to design `POST` idempotent for creation is to introduce a resource specific **secondary key** provided in the request body, to eliminate the problem of duplicate (a.k.a zombie) resources.

The secondary key is stored permanently in the resource as *alternate key* or *combined key* (if consisting of multiple properties) guarded by a uniqueness constraint enforced server-side, that is visible when reading the resource. The best and often naturally existing candidate is a *unique foreign key*, that points to another resource having *one-on-one* relationship with the newly created resource, e.g. a parent process identifier.

A good example here for a secondary key is the shopping cart ID in an order resource.

Note: When using the secondary key pattern without `Idempotency-Key` all subsequent retries should fail with status code 409 (conflict). We suggest to avoid 200 here unless you make sure, that the delivered resource is the original one implementing a well defined behavior. Using 204 without content would be a similar well defined option.

MUST define collection format of header and query parameters

TBD

SHOULD design simple query languages using query parameters

TBD

MUST design complex query languages using JSON

Minimalistic query languages based on query parameters are suitable for simple use cases with a small set of available filters that are combined in one way and one way only (e.g. *and* semantics). Simple query languages are generally preferred over complex ones.

Some APIs will have a need for sophisticated and more complex query languages. Dominant examples are APIs around search (incl. faceting) and product catalogs.

Aspects that set those APIs apart from the rest include but are not limited to:

- Unusual high number of available filters
- Dynamic filters, due to a dynamic and extensible resource model
- Free choice of operators, e.g. `and`, `or` and `not`

APIs that qualify for a specific, complex query language are encouraged to use nested JSON data structures and define them using OpenAPI directly. The provides the following benefits:

- Data structures are easy to use for clients
  - No special library support necessary
  - No need for string concatenation or manual escaping
- Data structures are easy to use for servers
  - No special tokenizers needed
  - Semantics are attached to data structures rather than text tokens
- Consistent with other HTTP methods
- API is defined in OpenAPI completely
  - No external documents or grammars needed
  - Existing means are familiar to everyone

JSON-specific rules and most certainly needs to make use of the `GET`-with-body pattern.

Example

The following JSON document should serve as an idea how a structured query might look like.

```
{
  "and": {
    "name": {
      "match": "Alice"
    },
    "age": {
      "or": {
        "range": {
          ">": 25,
          "<=": 50
        },
        "=": 65
      }
    }
  }
}
```

Feel free to also get some inspiration from:

- Elastic Search: Query DSL
- GraphQL: Queries

MUST document implicit filtering

Sometimes certain collection resources or queries will not list all the possible elements they have, but only those for which the current client is authorized to access.

Implicit filtering could be done on:

- the collection of resources being returned on a `GET` request
- the fields returned for the detail information of the resource

In such cases, the fact that implicit filtering is applied must be documented in the API specification's endpoint description. Consider caching aspects when implicit filtering is provided. Example:

If an employee of the company *Foo* accesses one of our business-to-business service and performs a `GET /business-partners`, it must, for legal reasons, not display any other business partner that is not owned or contractually managed by her/his company. It should never see that we are doing business also with company *Bar*.

Response as seen from a consumer working at `FOO`:

```
{
    "items": [
        { "name": "Foo Performance" },
        { "name": "Foo Sport" },
        { "name": "Foo Signature" }
    ]
}
```

Response as seen from a consumer working at BAR:

```
{
    "items": [
        { "name": "Bar Classics" },
        { "name": "Bar pour Elle" }
    ]
}
```

The API Specification should then specify something like this:

```
paths:
  /business-partner:
    get:
      description: >-
        Get the list of registered business partner.
        Only the business partners to which you have access to are returned.
```

14. HTTP status codes and errors

MUST specify success and error responses

APIs should define the functional, business view and abstract from implementation aspects. Success and error responses are a vital part to define how an API is used correctly.

Therefore, you must define **all** success and service specific error responses in your API specification. Both are part of the interface definition and provide important information for service clients to handle standard as well as exceptional situations.

Hint: In most cases it is not useful to document all technical errors, especially if they are not under control of the service provider. Thus unless a response code conveys application-specific functional semantics or is used in a none standard way that requires additional explanation, multiple error response specifications can be combined using the following pattern (see also [234]):

```
responses:
  ...
  default:
    description: error occurred - see status code and problem object for more
information.
    content:
      "application/problem+json":
        schema:
          $ref: 'https://sailpoint-oss.github.io/sailpoint-api-guidelines/problem-
1.0.1.yaml#/Problem'
```

API designers should also think about a **troubleshooting board** as part of the associated online API documentation. It provides information and handling guidance on application-specific errors and is referenced via links from the API specification. This can reduce service support tasks and contribute to service client and provider performance.

MUST use a standard error response object

All error responses must use the standard error response DTO object defined at https://github.com/sailpoint/cloud-api-client-common/blob/master/api-specs/src/main/yaml/v3/schemas/ErrorResponseDto.yaml. This provides a consistent error response structure that can be easily consumed by clients.

MAY define application specific codes for the standard error response object

If using the `detailCode` within the https://github.com/sailpoint/cloud-api-client-common/blob/master/api-specs/src/main/yaml/v3/schemas/ErrorResponseDto.yaml, then the service owner is free to create their own logic for detail codes that will aid them in debugging issues with the service.

MUST use the most accurate response example for each endpoint

Each endpoint must define a response example for every success and error response that can be returned. These examples must accurately reflect what can be returned by the endpoint. Default examples shared across multiple endpoints may be used as long as they accurately reflect what can be returned by the endpoint. If a default example doesn't accurately reflect what can be returned by an endpoint, then that endpoint must override the default example with one that is accurate.

MUST use standard HTTP status codes

You must only use standardized HTTP status codes consistently with their intended semantics. You must not invent new HTTP status codes.

RFC standards define ~60 different HTTP status codes with specific semantics (mainly RFC7231 and RFC 6585) — and there are upcoming new ones, e.g. draft legally-restricted-status. See overview on all error codes on Wikipedia or via https://httpstatuses.com/) also inculding 'unofficial codes', e.g. used by popular web servers like Nginx.

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

Important: As long as your HTTP status code usage is well covered by the semantic defined here, you should not describe it to avoid an overload with common sense information and the risk of inconsistent definitions. Only 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.

Success codes

Code	Meaning	Methods
200	OK - this is the standard success response	`<all>`
201	Created - Returned on successful entity creation. You are free to return either an empty response or the created resource in conjunction with the Location header. (More details found in the [common-headers].) *Always* set the Location header.	`POST`, `PUT`
202	Accepted - The request was successful and will be processed asynchronously.	`POST`, `PUT`, `PATCH`, `DELETE`
204	No content - There is no response body.	`PUT`, `PATCH`, `DELETE`
207	Multi-Status - The response body contains multiple status informations for different parts of a batch/bulk request (see **MUST** use code 207 for batch or bulk requests).	`POST`, (`DELETE`)

Redirection codes

Code	Meaning	Methods
301	Moved Permanently - This and all future requests should be directed to the given URI.	`<all>`
303	See Other - The response to the request can be found under another URI using a `GET` method.	`POST`, `PUT`, `PATCH`, `DELETE`
304	Not Modified - indicates that a conditional GET or HEAD request would have resulted in 200 response if it were not for the fact that the condition evaluated to false, i.e. resource has not been modified since the date or version passed via request headers If-Modified-Since or If-None-Match.	`GET`, `HEAD`

Client side error codes

Code	Meaning	Methods
400	Bad request - generic / unknown error. Should also be delivered in case of input payload fails business logic validation.	`<all>`

Code	Meaning	Methods
401	Unauthorized - the users must log in (this often means "Unauthenticated").	<all>
403	Forbidden - the user is not authorized to use this resource.	<all>
404	Not found - the resource is not found.	<all>
405	Method Not Allowed - the method is not supported, see OPTIONS.	<all>
406	Not Acceptable - resource can only generate content not acceptable according to the Accept headers sent in the request.	<all>
408	Request timeout - the server times out waiting for the resource.	<all>
409	Conflict - request cannot be completed due to conflict, e.g. when two clients try to create the same resource or if there are concurrent, conflicting updates.	POST, PUT, PATCH, DELETE
410	Gone - resource does not exist any longer, e.g. when accessing a resource that has intentionally been deleted.	<all>
412	Precondition Failed - returned for conditional requests, e.g. If-Match if the condition failed. Used for optimistic locking.	PUT, PATCH, DELETE
415	Unsupported Media Type - e.g. clients sends request body without content type.	POST, PUT, PATCH, DELETE
423	Locked - Pessimistic locking, e.g. processing states.	PUT, PATCH, DELETE
428	Precondition Required - server requires the request to be conditional, e.g. to make sure that the "lost update problem" is avoided (see **MAY** consider to support Prefer header to handle processing preferences).	<all>
429	Too many requests - the client does not consider rate limiting and sent too many requests (see **MUST** use code 429 with headers for rate limits).	<all>

Server side error codes:

Code	Meaning	Methods
500	Internal Server Error - a generic error indication for an unexpected server execution problem (here, client retry may be sensible)	<all>
501	Not Implemented - server cannot fulfill the request (usually implies future availability, e.g. new feature).	<all>
503	Service Unavailable - service is (temporarily) not available (e.g. if a required component or downstream service is not available) — client retry may be sensible. If possible, the service should indicate how long the client should wait by setting the Retry-After header.	<all>

MUST use most specific HTTP status codes

You must use the most specific HTTP status code when returning information about your request processing status or error situations. See the below table for examples of when to use the generic 400 vs a more specific 4xx

See https://github.com/sailpoint/cloud-api-client-common/blob/master/design-docs/v3/definition.md#response-codes-and-headers for a list of response codes that SailPoint prefers to use.

If you encounter a scenario where two or more response codes are appropriate, prefer to use the response code that preserves the security of the system and does not hand out too much information to unauthorized users.

MUST use code 207 for batch or bulk requests

Some APIs are required to provide either *batch* or *bulk* requests using `POST` for performance reasons, i.e. for communication and processing efficiency. In this case services may be in need to signal multiple response codes for each part of an batch or bulk request. As HTTP does not provide proper guidance for handling batch/bulk requests and responses, we herewith define the following approach:

- A batch or bulk request **always** responds with HTTP status code 207 unless a non-item-specific failure occurs.
- A batch or bulk request **may** return 4xx/5xx status codes, if the failure is non-item-specific and cannot be restricted to individual items of the batch or bulk request, e.g. in case of overload situations or general service failures.
- A batch or bulk response with status code 207 **always** returns as payload a multi-status response containing item specific status and/or monitoring information for each part of the batch or bulk request.

Note: These rules apply *even in the case* that processing of all individual parts *fail* or each part is executed *asynchronously*!

The rules are intended to allow clients to act on batch and bulk responses in a consistent way by inspecting the individual results. We explicitly reject the option to apply 200 for a completely successful batch as proposed in Nakadi's `POST /event-types/{name}/events` as short cut without inspecting the result, as we want to avoid risks and expect clients to handle partial batch failures anyway.

The bulk or batch response may look as follows:

```
BatchOrBulkResponse:
  description: batch response object.
  type: object
  properties:
    items:
      type: array
      items:
        type: object
        properties:
          id:
            description: Identifier of batch or bulk request item.
            type: string
          status:
            description: >
              Response status value. A number or extensible enum describing
              the execution status of the batch or bulk request items.
            type: string
            x-extensible-enum: [...]
          description:
            description: >
              Human readable status description and containing additional
              context information about failures etc.
            type: string
        required: [id, status]
```

Note: while a *batch* defines a collection of requests triggering independent processes, a *bulk* defines a collection of independent resources created or updated together in one request. With respect to response processing this distinction normally does not matter.

MUST use code 429 with headers for rate limits

APIs that wish to manage the request rate of clients must use the 429 (Too Many Requests) response code, if the client exceeded the request rate (see RFC 6585). Such responses must also contain header information providing further details to the client. There are two approaches a service can take for header information:

- Return a `Retry-After` header indicating how long the client ought to wait before making a follow-up request. The Retry-After header can contain a HTTP date value to retry after or the number of seconds to delay. Either is acceptable but APIs should prefer to use a delay in seconds.
- Return a trio of `X-RateLimit` headers. These headers (described below) allow a server to express a service level in the form of a number of allowing requests within a given window of time and when the window is reset.

The `X-RateLimit` headers are:

- `X-RateLimit-Limit`: The maximum number of requests that the client is allowed to make in this window.

- `X-RateLimit-Remaining`: The number of requests allowed in the current window.
- `X-RateLimit-Reset`: The relative time in seconds when the rate limit window will be reset. **Beware** that this is different to Github and Twitter's usage of a header with the same name which is using UTC epoch seconds instead.

The reason to allow both approaches is that APIs can have different needs. Retry-After is often sufficient for general load handling and request throttling scenarios and notably, does not strictly require the concept of a calling entity such as a tenant or named account. In turn this allows resource owners to minimise the amount of state they have to carry with respect to client requests. The 'X-RateLimit' headers are suitable for scenarios where clients are associated with pre-existing account or tenancy structures. 'X-RateLimit' headers are generally returned on every request and not just on a 429, which implies the service implementing the API is carrying sufficient state to track the number of requests made within a given window for each named entity.

MUST support problem JSON

TBD

MUST not expose stack traces

Stack traces contain implementation details that are not part of an API, and on which clients should never rely. Moreover, stack traces can leak sensitive information that partners and third parties are not allowed to receive and may disclose insights about vulnerabilities to attackers.

15. Pagination

MUST support pagination

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. This holds true for all lists that are (potentially) larger than just a few hundred entries.

There are two well known page iteration techniques:

- Offset/Limit-based pagination: numeric offset identifies the first page entry
- Cursor/Limit-based — aka key-based — pagination: a unique key element identifies the first page entry (see also Facebook's guide)

The technical conception of pagination should also consider user experience related issues. As mentioned in this article, jumping to a specific page is far less used than navigation via `next`/`prev` page links.

We currently prefer to use Offset/Limit-based pagination.

Note: To provide a consistent look and feel of pagination patterns, you must stick to the common query parameter names defined in **MUST** stick to conventional query parameters.

MAY use pagination links where applicable

To simplify client design, APIs should support simplified hypertext controls for paginating over collections whenever applicable as follows (see also [pagination-fields] for details):

```
{
  "self": "https://myorg.api.identitynow.com/v3/resources?cursor=<self-position>",
  "first": "https://myorg.api.identitynow.com/v3/resources?cursor=<first-position>",
  "prev": "https://myorg.api.identitynow.com/v3/resources?cursor=<previous-position>",
  "next": "https://myorg.api.identitynow.com/v3/resources?cursor=<next-position>",
  "last": "https://myorg.api.identitynow.com/v3/resources?cursor=<last-position>",
  "query": {
    "query-param-<1>": ...,
    "query-param-<n>": ...
  },
  "items": [...]
}
```

Remark: You should avoid providing a total count unless there is a clear need to do so. Very often, there are significant system and performance implications when supporting full counts. Especially, if the data set grows and requests become complex queries and filters drive full scans. While this is an implementation detail relative to the API, it is important to consider the ability to support serving counts over the life of a service.

16. Hypermedia

MUST use REST maturity level 2

We strive for a good implementation of REST Maturity Level 2 as it enables us to build resource-oriented APIs that make full use of HTTP verbs and status codes. You can see this expressed by many rules throughout these guidelines, e.g.:

- **SHOULD** avoid actions — think about resources
- **SHOULD** keep URLs verb-free
- **MUST** use HTTP methods correctly
- **MUST** use standard HTTP status codes

Although this is not HATEOAS, it should not prevent you from designing proper link relationships in your APIs as stated in rules below.

SHOULD use full, absolute URI

Links to other resource should always use full, absolute URI.

Motivation: Exposing any form of relative URI (no matter if the relative URI uses an absolute or relative path) introduces avoidable client side complexity. It also requires clarity on the base URI, which might not be given when using features like embedding subresources. The primary advantage of non-absolute URI is reduction of the payload size, which is better achievable by following the recommendation to use gzip compression

MUST use common hypertext controls

When embedding links to other resources into representations you must use the common hypertext control object. It contains at least one attribute:

- `href`: The URI of the resource the hypertext control is linking to. All our API are using HTTP(s) as URI scheme.

In API that contain any hypertext controls, the attribute name `href` is reserved for usage within hypertext controls.

The schema for hypertext controls can be derived from this model:

```
HttpLink:
  description: A base type of objects representing links to resources.
  type: object
  properties:
    href:
      description: Any URI that is using http or https protocol
      type: string
      format: uri
  required:
    - href
```

The name of an attribute holding such a `HttpLink` object specifies the relation between the object that contains the link and the linked resource. Implementations should use names from the IANA Link Relation Registry whenever appropriate. As IANA link relation names use hyphen-case notation, while this guide enforces snake_case notation for attribute names, hyphens in IANA names have to be replaced with underscores (e.g. the IANA link relation type `version-history` would become the attribute `version_history`)

Specific link objects may extend the basic link type with additional attributes, to give additional information related to the linked resource or the relationship between the source resource and the linked one.

E.g. a service providing "Person" resources could model a person who is married with some other person with a hypertext control that contains attributes which describe the other person (`id`, `name`) but also the relationship "spouse" between the two persons (`since`):

```
{
  "id": "446f9876-e89b-12d3-a456-426655440000",
  "name": "Peter Mustermann",
  "spouse": {
    "href": "https://...",
    "since": "1996-12-19",
    "id": "123e4567-e89b-12d3-a456-426655440000",
    "name": "Linda Mustermann"
  }
}
```

Hypertext controls are allowed anywhere within a JSON model. While this specification would allow HAL, we actually don't recommend/enforce the usage of HAL anymore as the structural separation of meta-data and data creates more harm than value to the understandability and usability of an API.

MUST not use link headers with JSON entities

For flexibility and precision, we prefer links to be directly embedded in the JSON payload instead of being attached using the uncommon link header syntax. As a result, the use of the `Link` Header defined by RFC 8288 in conjunction with JSON media types is forbidden.

17. Standard headers

This section describes a handful of standard headers, which we found raising the most questions in our daily usage, or which are useful in particular circumstances but not widely known.

MAY use standardized headers

Use this list and explicitly mention its support in your OpenAPI definition.

SHOULD use uppercase separate words with hyphens for HTTP headers

This convention is followed by most standard headers e.g. as defined in RFC 2616 and RFC 4229. Examples:

```
If-Modified-Since
Accept-Encoding
Content-ID
Language
```

Note, HTTP standard defines headers as case-insensitive (RFC 7230, p.22). However, for sake of readability and consistency you should follow the convention when using standard or proprietary

headers. Exceptions are common abbreviations like `ID`.

MUST use `Content-*` headers correctly

Content or entity headers are headers with a `Content-` prefix. They describe the content of the body of the message and they can be used in both, HTTP requests and responses. Commonly used content headers include but are not limited to:

- `Content-Disposition` can indicate that the representation is supposed to be saved as a file, and the proposed file name.
- `Content-Encoding` indicates compression or encryption algorithms applied to the content.
- `Content-Length` indicates the length of the content (in bytes).
- `Content-Language` indicates that the body is meant for people literate in some human language(s).
- `Content-Location` indicates where the body can be found otherwise (**MAY** use `Content-Location` header for more details]).
- `Content-Range` is used in responses to range requests to indicate which part of the requested resource representation is delivered with the body.
- `Content-Type` indicates the media type of the body content.

SHOULD use `Location` header instead of `Content-Location` header

As the correct usage of `Content-Location` response header (see below) with respect to caching and its method specific semantics is difficult, we *discourage* the use of `Content-Location`. In most cases it is sufficient to inform clients about the resource location in create or re-direct responses by using the `Location` header while avoiding the `Content-Location` specific ambiguities and complexities.

More details in RFC 7231 7.1.2 Location, 3.1.4.2 Content-Location

MAY use `Content-Location` header

The `Content-Location` header is *optional* and can be used in successful write operations (`PUT`, `POST`, or `PATCH`) or read operations (`GET`, `HEAD`) to guide caching and signal a receiver the actual location of the resource transmitted in the response body. This allows clients to identify the resource and to update their local copy when receiving a response with this header.

The Content-Location header can be used to support the following use cases:

- For reading operations `GET` and `HEAD`, a different location than the requested URI can be used to indicate that the returned resource is subject to content negotiations, and that the value provides a more specific identifier of the resource.
- For writing operations `PUT` and `PATCH`, an identical location to the requested URI can be used to explicitly indicate that the returned resource is the current representation of the newly created

or updated resource.

- For writing operations `POST` and `DELETE`, a content location can be used to indicate that the body contains a status report resource in response to the requested action, which is available at provided location.

Note: When using the `Content-Location` header, the `Content-Type` header has to be set as well. For example:

```
GET /products/123/images HTTP/1.1

HTTP/1.1 200 OK
Content-Type: image/png
Content-Location: /products/123/images?format=raw
```

MAY consider to support `Prefer` header to handle processing preferences

The `Prefer` header defined in RFC 7240 allows clients to request processing behaviors from servers. It pre-defines a number of preferences and is extensible, to allow others to be defined. Support for the `Prefer` header is entirely optional and at the discretion of API designers, but as an existing Internet Standard, is recommended over defining proprietary "X-" headers for processing directives.

The `Prefer` header can defined like this in an API definition:

```
components:
  headers:
  - Prefer:
      description: >
        The RFC7240 Prefer header indicates that a particular server behavior
        is preferred by the client but is not required for successful completion
        of the request (see [RFC 7240](https://tools.ietf.org/html/rfc7240).
        The following behaviors are supported by this API:

        # (indicate the preferences supported by the API or API endpoint)
        * **respond-async** is used to suggest the server to respond as fast as
          possible asynchronously using 202 - accepted - instead of waiting for
          the result.
        * **return=<minimal|representation>** is used to suggest the server to
          return using 204 without resource (minimal) or using 200 or 201 with
          resource (representation) in the response body on success.
        * **wait=<delta-seconds>** is used to suggest a maximum time the server
          has time to process the request synchronously.
        * **handling=<strict|lenient>** is used to suggest the server to be
          strict and report error conditions or lenient, i.e. robust and try to
          continue, if possible.

      type: array
      items:
        type: string
      required: false
```

Note: Please copy only the behaviors into your `Prefer` header specification that are supported by your API endpoint. If necessary, specify different `Prefer` headers for each supported use case.

Supporting APIs may return the `Preference-Applied` header also defined in RFC 7240 to indicate whether a preference has been applied.

MAY consider to support `ETag` together with `If-Match` /`If-None-Match` header

When creating or updating resources it may be necessary to expose conflicts and to prevent the 'lost update' or 'initially created' problem. Following RFC 7232 "HTTP: Conditional Requests" this can be best accomplished by supporting the `ETag` header together with the `If-Match` or `If-None-Match` conditional header. The contents of an `ETag: <entity-tag>` header is either (a) a hash of the response body, (b) a hash of the last modified field of the entity, or (c) a version number or identifier of the entity version.

To expose conflicts between concurrent update operations via `PUT`, `POST`, or `PATCH`, the `If-Match: <entity-tag>` header can be used to force the server to check whether the version of the updated entity is conforming to the requested `<entity-tag>`. If no matching entity is found, the operation is supposed a to respond with status code 412 - precondition failed.

Beside other use cases, `If-None-Match: *` can be used in a similar way to expose conflicts in resource creation. If any matching entity is found, the operation is supposed a to respond with status code 412 - precondition failed.

The `ETag`, `If-Match`, and `If-None-Match` headers can be defined as follows in the API definition:

```
components:
  headers:
  - ETag:
      description: |
        The RFC 7232 ETag header field in a response provides the entity-tag of
        a selected resource. The entity-tag is an opaque identifier for versions
        and representations of the same resource over time, regardless whether
        multiple versions are valid at the same time. An entity-tag consists of
        an opaque quoted string, possibly prefixed by a weakness indicator (see
        [RFC 7232 Section 2.3](https://tools.ietf.org/html/rfc7232#section-2.3).

      type: string
      required: false
      example: W/"xy", "5", "5db68c06-1a68-11e9-8341-68f728c1ba70"

  - If-Match:
      description: |
        The RFC7232 If-Match header field in a request requires the server to
        only operate on the resource that matches at least one of the provided
        entity-tags. This allows clients express a precondition that prevent
        the method from being applied if there have been any changes to the
        resource (see [RFC 7232 Section
        3.1](https://tools.ietf.org/html/rfc7232#section-3.1).

      type: string
      required: false
      example: "5", "7da7a728-f910-11e6-942a-68f728c1ba70"

  - If-None-Match:
      description: |
        The RFC7232 If-None-Match header field in a request requires the server
        to only operate on the resource if it does not match any of the provided
        entity-tags. If the provided entity-tag is `*`, it is required that the
        resource does not exist at all (see [RFC 7232 Section
        3.2](https://tools.ietf.org/html/rfc7232#section-3.2).

      type: string
      required: false
      example: "7da7a728-f910-11e6-942a-68f728c1ba70", *
```

Please see Optimistic locking in RESTful APIs for a detailed discussion and options.

MAY consider to support `Idempotency-Key` header

When creating or updating resources it can be helpful or necessary to ensure a strong idempotent behavior comprising same responses, to prevent duplicate execution in case of retries after timeout and network outages. Generally, this can be achieved by sending a client specific *unique request key* – that is not part of the resource – via `Idempotency-Key` header.

The *unique request key* is stored temporarily, e.g. for 24 hours, together with the response and the request hash (optionally) of the first request in a key cache, regardless of whether it succeeded or failed. The service can now look up the *unique request key* in the key cache and serve the response from the key cache, instead of re-executing the request, to ensure idempotent behavior. Optionally, it can check the request hash for consistency before serving the response. If the key is not in the key store, the request is executed as usual and the response is stored in the key cache.

This allows clients to safely retry requests after timeouts, network outages, etc. while receive the same response multiple times. **Note:** The request retry in this context requires to send the exact same request, i.e. updates of the request that would change the result are off-limits. The request hash in the key cache can protection against this misbehavior. The service is recommended to reject such a request using status code 400.

Important: To grant a reliable idempotent execution semantic, the resource and the key cache have to be updated with hard transaction semantics – considering all potential pitfalls of failures, timeouts, and concurrent requests in a distributed systems. This makes a correct implementation exceeding the local context very hard.

The `Idempotency-Key` header must be defined as follows, but you are free to choose your expiration time:

```yaml
components:
  headers:
  - Idempotency-Key:
      description: |
        The idempotency key is a free identifier created by the client to
        identify a request. It is used by the service to identify subsequent
        retries of the same request and ensure idempotent behavior by sending
        the same response without executing the request a second time.

        Clients should be careful as any subsequent requests with the same key
        may return the same response without further check. Therefore, it is
        recommended to use an UUID version 4 (random) or any other random
        string with enough entropy to avoid collisions.

        Idempotency keys expire after 24 hours. Clients are responsible to stay
        within this limits, if they require idempotent behavior.

      type: string
      format: uuid
      required: false
      example: "7da7a728-f910-11e6-942a-68f728c1ba70"
```

Hint: The key cache is not intended as request log, and therefore should have a limited lifetime, else it could easily exceed the data resource in size.

Note: The `Idempotency-Key` header unlike other headers in this section is not standardized in an RFC. Our only reference are the usage in the Stripe API. However, as it fit not into our section about [proprietary-headers], and we did not want to change the header name and semantic, we decided to treat it as any other common header.

18. API Operation

MUST publish OpenAPI specification

All service applications must publish OpenAPI specifications of their external APIs. While this is optional for internal APIs, i.e. APIs marked with the **component-internal** API audience group, we still recommend to do so to profit from the API management infrastructure.

IDN APIs are published from an internal GitHub repository. All other APIs will need to be submitted to the Developer Relations team for publishing.

MUST monitor API usage

Owners of APIs used in production should monitor API service to get information about its using clients. This information, for instance, is useful to identify potential review partner for API changes.

Hint: A preferred way of client detection implementation is by logging of the client-id retrieved from the OAuth token.

Appendix A: References

This section collects links to documents to which we refer, and base our guidelines on.

OpenAPI specification

- **OpenAPI specification**
- **OpenAPI specification mind map**

Publications, specifications and standards

- **RFC 3339**: Date and Time on the Internet: Timestamps
- **RFC 4122**: A Universally Unique IDentifier (UUID) URN Namespace
- **RFC 4627**: The application/json Media Type for JavaScript Object Notation (JSON)
- **RFC 8288**: Web Linking
- **RFC 6585**: Additional HTTP Status Codes
- **RFC 6902**: JavaScript Object Notation (JSON) Patch
- **RFC 7159**: The JavaScript Object Notation (JSON) Data Interchange Format
- **RFC 7230**: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
- **RFC 7231**: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
- **RFC 7232**: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests
- **RFC 7233**: Hypertext Transfer Protocol (HTTP/1.1): Range Requests
- **RFC 7234**: Hypertext Transfer Protocol (HTTP/1.1): Caching
- **RFC 7240**: Prefer Header for HTTP
- **RFC 7396**: JSON Merge Patch
- **RFC 7807**: Problem Details for HTTP APIs
- **RFC 4648**: The Base16, Base32, and Base64 Data Encodings
- **ISO 8601**: Date and time format
- **ISO 3166-1 alpha-2**: Two letter country codes
- **ISO 639-1**: Two letter language codes
- **ISO 4217**: Currency codes

- **BCP 47**: Tags for Identifying Languages

Dissertations

- **Roy Thomas Fielding - Architectural Styles and the Design of Network-Based Software Architectures**: This is the text which defines what REST is.

Books

- **REST in Practice: Hypermedia and Systems Architecture**
- **Build APIs You Won't Hate**
- **InfoQ eBook - Web APIs: From Start to Finish**

Blogs

- **Lessons-learned blog: Thoughts on RESTful API Design**

Appendix B: Tooling

This is not a part of the actual guidelines, but might be helpful for following them. Using a tool mentioned here doesn't automatically ensure you follow the guidelines.

API first integrations

The following software was specifically designed to support the API First workflow with OpenAPI YAML files (sorted alphabetically):

- **API Linter**: SailPoint's API linter using Spectral

The Swagger/OpenAPI homepage lists more Community-Driven Language Integrations, but most of them do not fit our API First approach.

Appendix C: Best practices

The best practices presented in this section are not part of the actual guidelines, but should provide guidance for common challenges we face when implementing RESTful APIs.

Cursor-based pagination in RESTful APIs

Cursor-based pagination is a very powerful and valuable technique (see also [160], that allows to efficiently provide a stable view on changing data. This is obtained by using an anchor element that allows to retrieve all page elements directly via an ordering combined-index, usually based on `created_at` or `modified_at`. Simple said, the cursor is the information set needed to reconstruct the database query to retrieves the minimal page information from the data storage.

The `cursor` itself is an opaque string, transmitted forth and back between service and clients, that must never be **inspected** or **constructed** by clients. Therefore, it is good practice to encode (encrypt) its content in a non-human-readable form.

The `cursor` content usually consists of a pointer to the anchor element defining the page position in the collection, a flag whether the element is included or excluded into/from the page, the retrieval direction, and a hash over the applied query filters (or the query filter itself) to safely re-create the collection. It is important to note, that a `cursor` should be always defined in relation to the current page to anticipate all occurring changes when progressing.

The `cursor` is usually defined as an encoding of the following information:

```
Cursor:
  descriptions: >
    Cursor structure that contains all necessary information to efficiently
    retrieve a page from the data store.
  type: object
  properties:
    position:
      description: >
        Object containing the keys pointing to the anchor element that is
        defining the collection resource page. Normally the position is given
        by the first or the last page element. The position object contains all
        values required to access the element efficiently via the ordered,
        combined index, e.g `modified_at`, `id`.
      type: object
      properties: ...

    element:
      description: >
        Flag whether the anchor element, which is pointed to by the `position`,
        should be *included* or *excluded* from the result set. Normally, only
        the current page includes the pointed to element, while all others are
        exclude it.
      type: string
      enum: [ INCLUDED, EXCLUDED ]

    direction:
      description: >
        Flag for the retrieval direction that is defining which elements to
        choose from the collection resource starting from the anchor elements
        position. It is either *ascending* or *descending* based on the
        ordering combined index.
      type: string
      enum: [ ASCENDING, DESCENDING ]

    query_hash:
      description: >
        Stable hash calculated over all query filters applied to create the
        collection resource that is represented by this cursor.
```

```
      type: string

    query:
      description: >
        Object containing all query filters applied to create the collection
        resource that is represented by this cursor.
      type: object
      properties: ...

  required:
    - position
    - element
    - direction
```

Note: In case of complex and long search requests, e.g. when `GET with body` is already required, the `cursor` may not be able to include the `query` because of common HTTP parameter size restrictions. In this case the `query` filters should be transported via body - in the request as well as in the response, while the pagination consistency should be ensured via the `query_hash`.

Remark: It is also important to check the efficiency of the data-access. You need to make sure that you have a fully ordered stable index, that allows to efficiently resolve all elements of a page. If necessary, you need to provide a combined index that includes the `id` to ensure the full order and additional filter criteria to ensure efficiency.

Further reading

- Twitter
- Use the Index, Luke
- Paging in PostgreSQL

Optimistic locking in RESTful APIs

Introduction

Optimistic locking might be used to avoid concurrent writes on the same entity, which might cause data loss. A client always has to retrieve a copy of an entity first and specifically update this one. If another version has been created in the meantime, the update should fail. In order to make this work, the client has to provide some kind of version reference, which is checked by the service, before the update is executed. Please read the more detailed description on how to update resources via `PUT` in the HTTP Requests Section.

A RESTful API usually includes some kind of search endpoint, which will then return a list of result entities. There are several ways to implement optimistic locking in combination with search endpoints which, depending on the approach chosen, might lead to performing additional requests to get the current version of the entity that should be updated.

`ETag` with `If-Match` header

An `ETag` can only be obtained by performing a `GET` request on the single entity resource before the update, i.e. when using a search endpoint an additional request is necessary.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { "id": "O0000042" },
>     { "id": "O0000043" }
>   ]
> }

< GET /orders/BO0000042

> HTTP/1.1 200 OK
> ETag: osjnfkjbnkq3jlnksjnvkjlsbf
> { "id": "BO0000042", ... }

< PUT /orders/O0000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { "id": "O0000042", ... }

> HTTP/1.1 204 No Content
```

Or, if there was an update since the `GET` and the entity's `ETag` has changed:

```
> HTTP/1.1 412 Precondition failed
```

Pros

- RESTful solution

Cons

- Many additional requests are necessary to build a meaningful front-end

`ETags` in result entities

The ETag for every entity is returned as an additional property of that entity. In a response containing multiple entities, every entity will then have a distinct `ETag` that can be used in subsequent `PUT` requests.

In this solution, the `etag` property should be `readonly` and never be expected in the `PUT` request

payload.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { "id": "O0000042", "etag": "osjnfkjbnkq3jlnksjnvkjlsbf", "foo": 42, "bar": true
},
>     { "id": "O0000043", "etag": "kjshdfknjqlowjdsljdnfkjbkn", "foo": 24, "bar":
false }
>   ]
> }

< PUT /orders/O0000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { "id": "O0000042", "foo": 43, "bar": true }

> HTTP/1.1 204 No Content
```

Or, if there was an update since the `GET` and the entity's `ETag` has changed:

```
> HTTP/1.1 412 Precondition failed
```

Pros

- Perfect optimistic locking

Cons

- Information that only belongs in the HTTP header is part of the business objects

Version numbers

The entities contain a property with a version number. When an update is performed, this version number is given back to the service as part of the payload. The service performs a check on that version number to make sure it was not incremented since the consumer got the resource and performs the update, incrementing the version number.

Since this operation implies a modification of the resource by the service, a `POST` operation on the exact resource (e.g. `POST /orders/O0000042`) should be used instead of a `PUT`.

In this solution, the `version` property is not `readonly` since it is provided at `POST` time as part of the payload.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> {
>   "items": [
>     { "id": "O0000042", "version": 1,  "foo": 42, "bar": true },
>     { "id": "O0000043", "version": 42, "foo": 24, "bar": false }
>   ]
> }

< POST /orders/O0000042
< { "id": "O0000042", "version": 1, "foo": 43, "bar": true }

> HTTP/1.1 204 No Content
```

or if there was an update since the `GET` and the version number in the database is higher than the one given in the request body:

```
> HTTP/1.1 409 Conflict
```

Pros

- Perfect optimistic locking

Cons

- Functionality that belongs into the HTTP header becomes part of the business object
- Using `POST` instead of PUT for an update logic (not a problem in itself, but may feel unusual for the consumer)

`Last-Modified` / `If-Unmodified-Since`

In HTTP 1.0 there was no `ETag` and the mechanism used for optimistic locking was based on a date. This is still part of the HTTP protocol and can be used. Every response contains a `Last-Modified` header with a HTTP date. When requesting an update using a `PUT` request, the client has to provide this value via the header `If-Unmodified-Since`. The server rejects the request, if the last modified date of the entity is after the given date in the header.

This effectively catches any situations where a change that happened between `GET` and `PUT` would be overwritten. In the case of multiple result entities, the `Last-Modified` header will be set to the latest date of all the entities. This ensures that any change to any of the entities that happens between `GET` and `PUT` will be detectable, without locking the rest of the batch as well.

Example:

```
< GET /orders

> HTTP/1.1 200 OK
> Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
> {
>   "items": [
>     { "id": "O0000042", ... },
>     { "id": "O0000043", ... }
>   ]
> }

< PUT /block/O0000042
< If-Unmodified-Since: Wed, 22 Jul 2009 19:15:56 GMT
< { "id": "O0000042", ... }

> HTTP/1.1 204 No Content
```

Or, if there was an update since the `GET` and the entities last modified is later than the given date:

```
> HTTP/1.1 412 Precondition failed
```

Pros

- Well established approach that has been working for a long time
- No interference with the business objects; the locking is done via HTTP headers only
- Very easy to implement
- No additional request needed when updating an entity of a search endpoint result

Cons

- If a client communicates with two different instances and their clocks are not perfectly in sync, the locking could potentially fail

Conclusion

We suggest to either use the `ETag` *in result entities* or `Last-Modified` / `If-Unmodified-Since` approach.

Appendix D: Changelog

This change log only contains major changes made after October 2021.

Non-major changes are editorial-only changes or minor changes of existing guidelines, e.g. adding new error code. Major changes are changes that come with additional obligations, or even change an existing guideline obligation. The latter changes are additionally labeled with "Rule Change" here.

To see a list of all changes, please have a look at the commit list in Github.

(Note that recent changes might be missing, as we update this list only occasionally, not with each pull request, to avoid merge commits.)

Rule Changes

```
<!-- Adds rule id as a postfix to all rule titles -->
<script>
var ruleIdRegEx = /(\d)+/;
var h3headers = document.getElementsByTagName("h3");
for (var i = 0; i < h3headers.length; i++) {
    var header = h3headers[i];
    if (header.id && header.id.match(ruleIdRegEx)) {
        var a = header.getElementsByTagName("a")[0]
        a.innerHTML += " [" + header.id + "]";
    }
}
</script>

<!-- Add table of contents anchor and remove document title -->
<script>
var toc = document.getElementById('toc');
var div = document.createElement('div');
div.id = 'table-of-contents';
toc.parentNode.replaceChild(div, toc);
div.appendChild(toc);
var ul = toc.childNodes[3];
ul.removeChild(ul.childNodes[1]);
</script>

<!-- Adds sidebar navigation -->
<script>
var header = document.getElementById('header');
var nav = document.createElement('div');
nav.id = 'toc';
nav.classList.add('toc2');
var title = document.createElement('div');
title.id = 'toctitle';

var link = document.createElement('a');
link.innerText = 'API Guidelines';
link.href = '#';

title.append(link);
nav.append(title);

var ul = document.createElement('ul');
ul.classList.add('sectlevel1');
```

```
var link = document.createElement('a');
link.innerHTML = 'Table of Contents';
link.href = '#table-of-contents';
li = document.createElement('li');
li.append(link);
ul.append(li);

var link, li;
var h2headers = document.getElementsByTagName('h2');
for (var i = 1; i < h2headers.length; i++) {
    var a = h2headers[i].getElementsByTagName("a")[0];
    if (a !== undefined) {
        link = document.createElement('a');
        link.innerHTML = a.innerHTML;
        link.href = a.hash;
        li = document.createElement('li');
        li.append(link);
        ul.append(li);
    }
}

document.body.classList.add('toc2');
document.body.classList.add('toc-left');
nav.append(ul);
header.prepend(nav);
</script>
```

[1] Per definition of R.Fielding REST APIs have to support HATEOAS (maturity level 3). Our guidelines do not strongly advocate for full REST compliance, but limited hypermedia usage, e.g. for pagination (see Hypermedia). However, we still use the term "RESTful API", due to the absence of an alternative established term and to keep it like the very majority of web service industry that also use the term for their REST approximations — in fact, in today's industry full HATEOAS compliant APIs are a very rare exception.