RESTful API and Event Guidelines

Table of Contents

1. Introduction

Origin

Our first attempts to compile a API guideline from scratch, made little sense when there are good open standards available in all sizes. We forked the [Zalando RESTful API guidelines](https://github.com/zalando/restful-api-guidelines) and adapted it to fit our needs.

Since this is not based on our use case exactly, this document is meant to be a living document which can be improved overtime to fit our needs.

Overview

Our architecture is centered around decoupled 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 our business.

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 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.

Purpose

The purpose of our "RESTful API guidelines" is to define standards to successfully establish "consistent API look and feel" quality. 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: Here are some recommended 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: Irresistible 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

As part of our API strategy, we encourage product and platform thinking. This should be expressed in the delivery of APIs as parts of a coherent platform.

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

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 API first principle

You must follow the API First Principle, more specifically:

- You must define APIs first, before coding its implementation, using OpenAPI as specification language
- You must design your APIs consistently with these guidelines; use of a linter for automated rule checks.
- You must call for early review feedback from peers and client developers for all component external APIs, i.e. all apis with `x-api-audience =/= component-internal` (see API Audience).

MUST provide API specification using OpenAPI

We use the OpenAPI specification as standard to define API specification files. API designers are required to provide the API specification using a single **self-contained YAML** file to improve readability. We encourage to use **OpenAPI 3.0** version, but still support **OpenAPI 2.0** (a.k.a. Swagger 2).

The API specification files should be subject to version control using a source code management system - best together with the implementing sources.

You must / should publish the component external / internal API specification with the deployment of the implementing service.

Hint: A good way to explore **OpenAPI 3.0/2.0** is to navigate through the OpenAPI specification mind map and use our Swagger Plugin for IntelliJ IDEA to create your first API. To explore and validate/evaluate existing APIs the Swagger Editor may be a good starting point.

Hint: We do not yet provide guidelines for GraphQL. We focus on resource oriented HTTP/REST API style (and related tooling and infrastructure support) for general purpose peer-to-peer microservice communication. Here, we think that GraphQL has no major benefits, but a couple of downsides compared to REST. However, GraphQL can provide a lot of value for specific target domain problems, especially backends for frontends (BFF) and mobile clients.

SHOULD provide API user manual

In addition to the API Specification, it is good practice to provide an 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:

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

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

MUST write APIs using U.K. English

MUST only use durable and immutable remote references

Normally, API specification files must be **self-contained**, i.e. files should not contain references to local or remote content, e.g. `../fragment.yaml#/element` or `$ref: 'https://github.com/zalando/zally/blob/master/server/src/main/resources/api/zally-api.yaml#/schemas/LintingRequest'`. The reason is, that the content referred to is *in general* **not durable** and **not immutable**. As a consequence, the semantic of an API may change in unexpected ways.

However, you may use remote references to resources accessible by the following service URLs:

- <code>https://github.com/zalando/restful-api-guidelines</code> – used to refer to user defined, immutable API specification revisions published via the internal API repository.
- <code><a href="https://github.com/zalando/restful-api-guidelines/{model.yaml}"

class="bare">https://github.com/zalando/restful-api-guidelines/{model.yaml}</code> – used to refer to guideline defined re-usable API fragments (see <code>{model.yaml}</code> files in restful-api-guidelines/models for details).

As we control these URLs, we ensure that their content is **durable** and **immutable**. This allows to define API specifications by using fragments published via this sources, as suggested in **MUST** specify success and error responses.

4. REST Basics - 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/contact/{name,url,email}` containing the responsible team

Following OpenAPI extension properties **must** be provided in addition:

- `#/info/x-api-id` unique identifier of the API (see rule 215)
- `#/info/x-audience` intended target audience of the API (see rule 219)

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
  <...>
```

MUST provide API identifiers

Each API specification must be provisioned with a globally unique and immutable API identifier. The API identifier is defined in the `info`-block of the OpenAPI specification and must conform to the following definition:

```
/info/x-api-id:
  type: string
  format: urn
  pattern: ^[a-z0-9][a-z0-9-:.]{6,62}[a-z0-9]$
  description: |
    Mandatory globally unique and immutable API identifier. The API
    id allows to track the evolution and history of an API specification
    as a sequence of versions.
```

API specifications will evolve and any aspect of an OpenAPI specification may change. We require API identifiers because we want to support API clients and providers with API lifecycle management features, like change trackability and history or automated backward compatibility checks. The immutable API identifier allows the identification of all API specification versions of an API evolution. By using API semantic version information or API publishing date as order criteria you get the **version** or **publication history** as a sequence of API specifications.

Note: While it is nice to use human readable API identifiers based on self-managed URNs, it is recommend to stick to UUIDs to relief API designers from any urge of changing the API identifier while evolving the API. Example:

```
openapi: 3.0.1
info:
  x-api-id: d0184f38-b98d-11e7-9c56-68f728c1ba70
  title: Parcel Service API
  description: API for <...>
  version: 1.5.8
  <...>
```

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 organisational and legal boundaries:

component-internal

This is often referred to as a *team internal API* or a *product internal API*. The API consumers with this audience are restricted to applications of the same **functional component** which typically represents a specific **product** with clear functional scope and ownership. All services of a functional component / product are owned by a specific dedicated owner and engineering team(s). Typical examples of component-internal APIs are APIs being used by internal helper and worker services or that support service operation.

business-unit-internal

The API consumers with this audience are restricted to applications of a specific product portfolio owned by the same business unit.

company-internal

The API consumers with this audience are restricted to applications owned by the business units of the same the organisation.

external-partner

The API consumers with this audience are restricted to applications of business partners of the organisation owning the API and the organisation itself.

external-public

APIs with this audience can be accessed by anyone with Internet access.

Note: a smaller audience group is intentionally included in the wider group and thus does not need to be declared additionally.

The API audience is provided as API meta information in the `info`-block of the OpenAPI specification and must conform to the following specification:

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

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: company-internal
  title: Parcel Helper Service API
  description: API for <...>
  version: 1.2.4
  <...>
```

For details and more information on audience groups see the API Audience narrative (internal link).

MUST/SHOULD use functional naming schema

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

A unique `functional-name` is assigned to each functional component serving an API. It is built of the domain name of the functional group the component is belonging to and a unique a short identifier for the functional component itself:

```
<functional-name>      ::= <functional-domain>-<functional-component>
<functional-domain>    ::= [a-z][a-z0-9-]* -- managed functional group of components
<functional-component> ::= [a-z][a-z0-9-]* -- name of API owning functional component
```

Depending on the API audience, you **must/should/may** follow the functional naming schema for hostnames and event names (and also permission names, in future) as follows:

Functional Naming	Audience
must	external-public, external-partner
should	company-internal, business-unit-internal
may	component-internal

Please see the following rules for detailed functional naming patterns: * **MUST** follow naming convention for hostnames * [213]

Internal Guideance: You *must* use the simple functional name registry (internal link) to register your functional name before using it. The registry is a centralized infrastructure service to ensure uniqueness of your functional names (and available domains — including subdomains) and to support hostname DNS resolution.
Hint: Due to lexicalic restrictions of DNS names there is no specific separator to split a functional name into (sub) domain and component; this knowledge is only managed in the registry.

MUST follow naming convention for hostnames

Hostnames in APIs must, respectively should conform to the functional naming depending on the audience as follows (see **MUST/SHOULD** use functional naming schema for details and `<functional-name>` definition):

```
<hostname>             ::= <functional-hostname> | <application-hostname>

<functional-hostname>  ::= <functional-name>.zalandoapis.com
```

Hint: The following convention (e.g. used by legacy STUPS infrastructure) is deprecated and **only** allowed for hostnames of component-internal APIs:

```
<application-hostname> ::= <application-id>.<organization-unit>.zalan.do
<application-id>       ::= [a-z][a-z0-9-]*  -- application identifier
<organization-id>      ::= [a-z][a-z0-9-]*  -- organization unit identifier, e.g. team
identifier
```

Exception: There are legacy hostnames used for APIs with `external-partner` audience which may not follow this rule due to backward compatibility constraints. The API Linter maintains a whitelist for this exceptions (including e.g. `api.merchants.zalando.com` and `api-sandbox.merchants.zalando.com`).

5. REST Basics - Security

MUST secure endpoints

Every API endpoint must be protected and armed with authentication and authorization. As part of the API definition you must specify how you protect your API using either the `http` typed `bearer` or `oauth2` typed security schemes defined in the OpenAPI Authentication Specification.

The majority of our APIs (especially the company internal APIs) are protected using JWT tokens provided by the platform IAM token service. In these situations you should use the `http` typed Bearer Authentication security scheme — it is based on OAuth2.0 RFC 6750 defining the standard header `Auhorization: Bearer <token>`. The following code snippet shows how to define the bearer security scheme.

```yaml
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

The bearer security schema can than be applied to all API endpoints, e.g. requiring the token to have `api-repository.read` scope for permission as follows (see also **MUST** define and assign permissions (scopes)):

```yaml
security:
  - BearerAuth: [ api-repository.read ]
```

In other, more specific situations e.g. with customer and partner facing APIs you may use other OAuth 2.0 authorization flows as defined by RFC 6749. Please consult the OpenAPI OAuth 2.0 Authentication section for details on how to define `oauth2` typed security schemes correctly.

Note: Do not use OpenAPI `oauth2` typed security scheme flows (e.g. `implicit`) if your service does not fully support it and implements a simple bearer token scheme, because it exposes authentication server address details and may make use of redirection.

MUST define and assign permissions (scopes)

APIs must define permissions to protect their resources. Thus, at least one permission must be assigned to each API endpoint.

The naming schema for permissions corresponds to the naming schema for hostnames and event type names. Please refer to **MUST** follow naming convention for permissions (scopes) for designing permission names and see the following examples.

Application ID	Resource ID	Access Type	Example
`order-management`	`sales-order`	`read`	`order-management.sales-order.read`
`order-management`	`shipment-order`	`read`	`order-management.shipment-order.read`
`fulfillment-order`		`write`	`fulfillment-order.write`
`business-partner-service`		`read`	`business-partner-service.read`

Note: APIs should stick to component specific permissions without resource extension to avoid the complexity of too many fine grained permissions. For the majority of use cases, restricting access for specific API endpoints using read or write is sufficient.

The defined permissions are than assigned to each API endpoint based on the security schema (see example in previous section) by specifying the security requirement as follows:

```yaml
paths:
 /business-partners/{partner-id}:
   get:
     summary: Retrieves information about a business partner
     security:
       - BearerAuth: [ business-partner-service.read ]
```

In some cases a whole API or selected API endpoints may not require speciifc permissions, e.g. if information is public or protected by object level authorization. To make this explicit you should assign the `uid` pseudo permission, that is always available as OAuth2 default scope.

```yaml
paths:
  /public-information:
   get:
     summary: Provides public information about ...
              Accessible by any user; no permissions needed.
     security:
       - BearerAuth: [ uid ]
```

Hint: Following a minimal a minimal API specification approach, the `Authorization`-header does not need to be defined on each API endpoint, since it is required and so to say implicitly defined via the security section.

MUST follow naming convention for permissions (scopes)

As long as the functional naming is not yet supported by our permission registry, permission names in APIs must conform to the following naming pattern:

```
<permission> ::= <standard-permission> |  -- should be sufficient for majority of use
cases
                 <resource-permission> |  -- for special security access
differentiation use cases
                 <pseudo-permission>      -- used to explicitly indicate that access
is not restricted

<standard-permission> ::= <application-id>.<access-mode>
<resource-permission> ::= <application-id>.<resource-name>.<access-mode>
<pseudo-permission>   ::= uid

<application-id>      ::= [a-z][a-z0-9-]*  -- application identifier
<resource-name>       ::= [a-z][a-z0-9-]*  -- free resource identifier
<access-mode>         ::= read | write    -- might be extended in future
```

This pattern is compatible with the previous definition.

6. REST Basics - Data formats

MUST use standard data formats

Open API (based on JSON Schema Validation vocabulary) defines formats from ISO and IETF standards for date/time, integers/numbers and binary data. You **must** use these formats, whenever applicable:

`OpenAPI type`	`OpenAPI format`	**Specification**	**Example**
`integer`	`int32`	4 byte signed integer between -2^{31} and 2^{31}-1	`7721071004`
`integer`	`int64`	8 byte signed integer between -2^{63} and 2^{63}-1	`772107100456824`
`integer`	`bigint`	arbitrarily large signed integer number	`77210710045682438959`
`number`	`float`	`binary32` single precision decimal number — see IEEE 754-2008/ISO 60559:2011	`3.1415927`
`number`	`double`	`binary64` double precision decimal number — see IEEE 754-2008/ISO 60559:2011	`3.141592653589793`
`number`	`decimal`	arbitrarily precise signed decimal number	`3.141592653589793238462643383279`
`string`	`byte`	`base64url` encoded byte following RFC 7493 Section 4.4	`"VA=="`
`string`	`binary`	`base64url` encoded byte sequence following RFC 7493 Section 4.4	`"VGVzdA=="`
`string`	`date`	RFC 3339 internet profile — subset of ISO 8601	`"2019-07-30"`
`string`	`date-time`	RFC 3339 internet profile — subset of ISO 8601	`"2019-07-30T06:43:40.252Z"`
`string`	`time`	RFC 3339 internet profile — subset of ISO 8601	`"06:43:40.252Z"`
`string`	`duration`	RFC 3339 internet profile — subset of ISO 8601	`"P1DT30H4S"`
`string`	`period`	RFC 3339 internet profile — subset of ISO 8601	`"2019-07-30T06:43:40.252Z/PT3H"`
`string`	`password`		`"secret"`
`string`	`email`	RFC 5322	`"example@example.com"`

OpenAPI type	OpenAPI format	Specification	Example
`string`	`idn-email`	RFC 6531	`"hello@bücher.example"`
`string`	`hostname`	RFC 1034	`"www.example.com"`
`string`	`idn-hostname`	RFC 5890	`"bücher.example"`
`string`	`ipv4`	RFC 2673	`"104.75.173.179"`
`string`	`ipv6`	RFC 2673	`"2600:1401:2::8a"`
`string`	`uri`	RFC 3986	`"https://www.example.com/"`
`string`	`uri-reference`	RFC 3986	`"/clothing/"`
`string`	`uri-template`	RFC 6570	`"/users/{id}"`
`string`	`iri`	RFC 3987	`"https://bücher.example/"`
`string`	`iri-reference`	RFC 3987	`"/woman-winter-jackets/"`
`string`	`uuid`	RFC 4122	`"e2ab873e-b295-11e9-9c02-…"`
`string`	`json-pointer`	RFC 6901	`"/items/0/id"`
`string`	`relative-json-pointer`	Relative JSON pointers	`"1/id"`

Note: Formats `bigint` and `decimal` have been added to the OpenAPI defined formats — see also **MUST** define a format for number and integer types and **MUST** use standard formats for date and time properties below.

We add further OpenAPI formats that are useful especially in an e-commerce environment e.g. `language code`, `country code`, and `currency` based other ISO and IETF standards. You **must** use these formats, whenever applicable:

OpenAPI type	format	Specification	Example
`string`	`iso-639`	two letter language code — see ISO 639-1	`"en"`
`string`	`bcp47`	multi letter language tag — see BCP 47. It is a compatible extension of ISO 639-1 optionally with additional information for language usage, like region, variant, script.	`"en-DE"`
`string`	`iso-3166`	two letter country code — see ISO 3166-1 alpha-2	`"GB"` **Hint:** It is `"GB"`, not `"UK"`.

OpenAPI type	format	Specification	Example
`string`	`iso-4217`	three letter currency code — see ISO 4217	`"EUR"`
`string`	`gtin-13`	Global Trade Item Number — see GTIN	`"5710798389878"`
`string`	`regex`	regular expressions as defined in ECMA 262	`"^[a-z0-9]+$"`

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

MUST define a format for number and integer types

In **MUST** use standard data formats we added `bigint` and `decimal` to the OpenAPI defined formats. As an implication, you must always provide one of the formats `int32`, `int64`, `bigint` or `float`, `double`, `decimal` when you define an API property of JSON type `number` or `integer`.

By this we prevent clients from guessing the precision incorrectly, and thereby changing the value unintentionally. The precision must be translated by clients and servers into the most specific language types; in Java, for instance, the `number` type with `decimal` format will translate into `BigDecimal` and `integer` type with `int32` format will translate to `int` or `Integer` Java types.

MUST encode binary data in `base64url`

You may expose binary data. You must use a standard media type and data format, if applicable — see Rule 168. If no standard is available, you must define the binary data as `string` typed property with `binary` format using `base64url` encoding — as also described in **MUST** use standard data formats.

MUST use standard formats for date and time properties

As a specific case of **MUST** use standard data formats, you must use the `string` typed formats `date`, `date-time`, `time`, `duration`, or `period` for the definition of date and time properties. The formats are based on the standard RFC 3339 internet profile -- a subset of ISO 8601

Exception: For passing date/time information via standard protocol headers, HTTP RFC 7231 requires to follow the date and time specification used by the Internet Message Format RFC 5322.

As defined by the standard, time zone offset may be used, however, we recommend to only use times based on UTC without local 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 which may include daylight saving time. 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.

Hint: We discourage using numerical timestamps. It typically creates 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 use standard formats for time duration and interval properties

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

MUST use standard formats for country, language and currency properties

As a specific case of **MUST** use standard data formats you must use the following standard formats:

- Country codes: ISO 3166-1-alpha2 two letter country codes indicated via OpenAPI format `iso-3166`
- Language codes: ISO 639-1 two letter language codes indicated via OpenAPI format `iso-639`
- Language variant tags: BCP 47 multi letter language tag indicated via OpenAPI format `bcp47`. (It is a compatible extension of ISO 639-1 with additional optional information for language usage, like region, variant, script)
- Currency codes: ISO 4217 three letter currency codes indicated via OpenAPI format `iso-4217`

SHOULD use content negotiation, if clients may choose from different resource representations

In some situations the API supports serving different representations of a specific resource (at the same URL) e.g. JSON, PDF, TEXT, or HTML representations for an invoice resource. You should use content negotiation to support clients specifying via the standard HTTP headers `Accept`, `Accept-Language`, `Accept-Encoding` which representation is best suited for their use case, for example, which language of a document, representation / content format, or content encoding. You **SHOULD** use standard media types like `application/json` or `application/pdf` for defining the content format in the `Accept` header.

SHOULD only use UUIDs if necessary

Generating IDs can be a scaling problem in high frequency and near real time use cases. UUIDs solve this problem, as they can be generated without collisions in a distributed, non-coordinated way and without additional server round trips.

However, they also come with some disadvantages:

- pure technical key without meaning; not ready for naming or name scope conventions that might be helpful for pragmatic reasons, e.g. we learned to use names for product attributes, instead of UUIDs

- less usable, because…
  - cannot be memorized and easily communicated by humans
  - harder to use in debugging and logging analysis
  - less convenient for consumer facing usage
- quite long: readable representation requires 36 characters and comes with higher memory and bandwidth consumption
- not ordered along their creation history and no indication of used id volume
- may be in conflict with additional backward compatibility support of legacy ids

UUIDs should be avoided when not needed for large scale id generation. Instead, for instance, server side support with id generation can be preferred (`POST` on id resource, followed by idempotent `PUT` on entity resource). Usage of UUIDs is especially discouraged as primary keys of master and configuration data, like brand-ids or attribute-ids which have low id volume but widespread steering functionality.

Please be aware that sequential, strictly monotonically increasing numeric identifiers may reveal critical, confidential business information, like order volume, to non-privileged clients.

In any case, we should always use string rather than number type for identifiers. This gives us more flexibility to evolve the identifier naming scheme. Accordingly, if used as identifiers, UUIDs should not be qualified using a format property.

Hint: Usually, random UUID is used - see UUID version 4 in RFC 4122. Though UUID version 1 also contains leading timestamps it is not reflected by its lexicographic sorting. This deficit is addressed by ULID (Universally Unique Lexicographically Sortable Identifier). You may favour ULID instead of UUID, for instance, for pagination use cases ordered along creation time.

7. REST Basics - URLs

Guidelines for naming and designing resource paths and query parameters.

SHOULD 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 pluralize resource names

Usually, a collection of resource instances is provided (at least the API should be ready here). The special case of a *resource singleton* must be modeled as a collection with cardinality 1 including definition of `maxItems` = `minItems` = 1 for the returned `array` structure to make the cardinality constraint explicit.

Exception: the *pseudo identifier* `self` used to specify a resource endpoint where the resource identifier is provided by authorization information (see **MUST** identify resources and sub-resources via path segments).

MUST use URL-friendly resource identifiers

To simplify encoding of resource IDs in URLs they must match the regex `[a-zA-Z0-9:._\-/]*`. Resource IDs only consist of ASCII strings using letters, numbers, underscore, minus, colon, period, and - on rare occasions - slash.

Note: slashes are only allowed to build and signal resource identifiers consisting of compound keys.

Note: to prevent ambiguities of unnormalized paths resource identifiers must never be empty. Consequently, support of empty strings for path parameters is forbidden.

MUST use kebab-case for path segments

Path segments are restricted to ASCII kebab-case strings matching regex `^[a-z][a-z\-0-9]*$`. The first character must be a lower case letter, and subsequent characters can be a letter, or a dash(`-`), or a number.

Example:

```
/shipment-orders/{shipment-order-id}
```

Hint: kebab-case applies to concrete path segments and not necessarily the names of path parameters.

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/{order-id}
GET /orders/{order-id}/
GET /orders//{order-id}
```

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 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 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 instance, if an application has to lock articles explicitly so that only one user may edit them, create an article lock with `PUT` or `POST` instead of using a lock action.

Request:

```
PUT /article-locks/{article-id}
```

The added benefit is that you already have a service for browsing and filtering article locks.

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.

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.

SHOULD model complete business processes

An API should contain the complete business processes containing all resources representing the process. This enables clients to understand the business process, foster a consistent design of the business process, allow for synergies from description and implementation perspective, and eliminates implicit invisible dependencies between APIs.

In addition, it prevents services from being designed as thin wrappers around databases, which normally tends to shift business logic to the clients.

MUST identify resources and sub-resources via path segments

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

```
/resources/{resource-id}/sub-resources/{sub-resource-id}
```

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/{partner-id}/addresses/{address-id}` is valid, then, in principle, also `/partners/{partner-id}/addresses`, `/partners/{partner-id}` 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 **MAY** expose compound keys as resource identifiers).

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/{empl-id}` or `/employees/{empl-id}/personal-details`.

MAY expose compound keys as resource identifiers

If a resource is best identified by a *compound key* consisting of multiple other resource identifiers, it is allowed to reuse the compound key in its natural form containing slashes instead of *technical* resource identifier in the resource path without violating the above rule **MUST** identify resources and sub-resources via path segments as follows:

```
/resources/{compound-key-1}[delim-1]...[delim-n-1]{compound-key-n}
```

Example paths:

```
/shopping-carts/{country}/{session-id}
/shopping-carts/{country}/{session-id}/items/{item-id}
/api-specifications/{docker-image-id}/apis/{path}/{file-name}
/api-specifications/{repository-name}/{artifact-name}:{tag}
/article-size-advices/{sku}/{sales-channel}
```

Warning: Exposing a compound key as described above limits ability to evolve the structure of the resource identifier as it is no longer opaque.

To compensate for this drawback, APIs must apply a compound key abstraction consistently in all requests and responses parameters and attributes allowing consumers to treat these as *technical resource identifier* replacement. The use of independent compound key components must be limited to search and creation requests, as follows:

```
# compound key components passed as independent search query parameters
GET /article-size-advices?skus=sku-1,sku-2&sales_channel_id=sid-1
=> { "items": [{ "id": "id-1", ...  },{ "id": "id-2", ...  }] }

# opaque technical resource identifier passed as path parameter
GET /article-size-advices/id-1
=> { "id": "id-1", "sku": "sku-1", "sales_channel_id": "sid-1", "size": ... }

# compound key components passed as mandatory request fields
POST /article-size-advices { "sku": "sku-1", "sales_channel_id": "sid-1", "size": ...
}
=> { "id": "id-1", "sku": "sku-1", "sales_channel_id": "sid-1", "size": ... }
```

Where `id-1` is representing the opaque provision of the compound key `sku-1/sid-1` as technical resource identifier.

Remark: A compound key component may itself be used as another resource identifier providing another resource endpoint, e.g `/article-size-advices/{sku}`.

MAY 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
```

SHOULD limit number of resource types

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

For example, the resources below would be counted as three resource types, one for customers, one for the addresses, and one for the customers' related addresses:

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

Note that:

- We consider `/customers/id/preferences` part of the `/customers` resource type because it has a one-to-one relation to the customer without an additional identifier.
- We consider `/customers` and `/customers/id/addresses` as separate resource types because `/customers/id/addresses/{addr}` also exists with an additional identifier for the address.
- We consider `/addresses` and `/customers/id/addresses` as separate resource types because there's no reliable way to be sure they are the same.

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

Nevertheless one API should hold all necessary resources to model complete business processes helping clients to understand these flows.

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.)

MUST use snake_case (never camelCase) for query parameters

See also **MUST** property names must be snake_case (and never camelCase).

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:

- `q`: default query parameter, e.g. used by browser tab completion; should have an entity specific alias, e.g. sku.
- `sort`: comma-separated list of fields (as defined by **MUST** define collection format of header and query parameters) to define the sort order. To indicate sorting direction, fields may be prefixed with `+` (ascending) or `-` (descending), e.g. /sales-orders?sort=+id.
- `fields`: field name expression to retrieve only a subset of fields of a resource. See **SHOULD** support partial responses via filtering below.
- `embed`: field name expression to expand or embedded sub-entities, e.g. inside of an article entity, expand silhouette code into the silhouette object. Implementing `embed` correctly is difficult, so do it with care. See **SHOULD** allow optional embedding of sub-resources below.
- `offset`: numeric offset of the first element provided on a page representing a collection request. See REST Design - Pagination section below.
- `cursor`: an opaque pointer to a page, never to be inspected or constructed by clients. It usually (encrypted) encodes the page position, i.e. the identifier of the first or last page element, the pagination direction, and the applied query filters to recreate the collection. See Cursor-based pagination in RESTful APIs or REST Design - Pagination section below.
- `limit`: client suggested limit to restrict the number of entries on a page. See REST Design - Pagination section below.

8. REST Basics - JSON payload

These guidelines provides recommendations for defining JSON data referring 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 JSON data to have an idiomatic form across teams and services.

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

Non-JSON media types may be supported, if you stick to a business object specific standard format for the payload data, for instance, image data format (JPG, PNG, GIF), document format (PDF, DOC, ODF, PPT), or archive format (TAR, ZIP).

Generic structured data interchange formats other than JSON (e.g. XML, CSV) may be provided, but only additionally to JSON as default format using content negotiation, for specific use cases where clients may not interpret the payload structure.

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/vendor.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.

Exception: Custom media type should be only used in situations where you need to provide API endpoint versioning (with content negotiation) due to incompatible changes.

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 snake_case (and never camelCase)

Property names are restricted to ASCII snake_case strings matching regex `^[a-z_][a-z_0-9]*$`. The first character must be a lower case letter, or an underscore, and subsequent characters can be a letter, an underscore, or a number.

Examples:

```
customer_number, sales_order_number, billing_address
```

Rationale: No established industry standard exists, but many popular Internet companies prefer snake_case: e.g. GitHub, Stack Exchange, Twitter. Others, like Google and Amazon, use both - but not only camelCase. It's essential to establish a consistent look and feel such that JSON looks as if it came from the same hand.

SHOULD declare enum values using UPPER_SNAKE_CASE string

Enumerations should 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 name date/time properties with `_at` suffix

Dates and date-time properties should end with `_at` to distinguish them from boolean properties which otherwise would have very similar or even identical names:

- `created_at` rather than `created`,
- `modified_at` rather than `modified`,
- `occurred_at` rather than `occurred`, and
- `returned_at` rather than `returned`.

Note: `created` and `modified` were mentioned in an earlier version of the guideline and are therefore still accepted for APIs that predate this rule.

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 use same semantics for `null` and absent properties

OpenAPI 3.x allows to mark properties as `required` and as `nullable` to specify whether properties may be absent (`{}`) or `null` (`{"example":null}`). If a property is defined to be not `required` and `nullable` (see 2nd row in Table below), this rule demands that both cases must be handled in the exact same manner by specification.

The following table shows all combinations and whether the examples are valid:

required	nullable	{}	{"example":null}
`true`	`true`	□ No	□ Yes
`false`	`true`	□ Yes	□ Yes
`true`	`false`	□ No	□ No
`false`	`false`	□ Yes	□ No

While API designers and implementers may be tempted to assign different semantics to both cases, we explicitly decide **against** that option, because we think that any gain in expressiveness is far outweighed by the risk of clients not understanding and implementing the subtle differences incorrectly.

As an example, an API that provides the ability for different users to coordinate on a time schedule, e.g. a meeting, may have a resource for options in which every user has to make a `choice`. The difference between *undecided* and *decided against any of the options* could be modeled as *absent* and `null` respectively. It would be safer to express the `null` case with a dedicated Null object, e.g. `{}` compared to `{"id":"42"}`.

Moreover, many major libraries have somewhere between little to no support for a `null`/absent pattern (see Gson, Moshi, Jackson, JSON-B). Especially strongly-typed languages suffer from this since a new composite type is required to express the third state. Nullable `Option`/`Optional`/`Maybe` types could be used but having nullable references of these types completely contradicts their purpose.

The only exception to this rule is JSON Merge Patch RFC 7396) which uses `null` to explicitly indicate property deletion while absent properties are ignored, i.e. not modified.

SHOULD not use `null` for empty arrays

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

MUST use common field names and semantics

You must use common field names and semantics whenever applicable. Common fields are idiomatic, create consistency across APIs and support common understanding for API consumers.

We define the following common field names:

- `id`: the identity of the object. If used, IDs must be opaque strings(UUIDv4) and not numbers. IDs are unique within some documented context, are stable and don't change for a given object once assigned, and are never recycled cross entities.
- `type`: the kind of thing this object is. If used, the type of this field should be a string. Types allow runtime information on the entity provided that otherwise requires examining the OpenAPI file.
- `ETag`: the ETag of an embedded sub-resource. It may be used to carry the `ETag` for subsequent `PUT`/`PATCH` calls (see `ETags` in result entities).

Further common fields are defined in **SHOULD** name date/time properties with `_at` suffix. The following guidelines define standard objects and fields:

- **SHOULD** use pagination response page object
- **MUST** use the common address fields
- **MUST** use the common money object

Example JSON schema:

```yaml
tree_node:
  type: object
  properties:
    id:
      description: the identifier of this node
      type: string
    created_at:
      description: when got this node created
      type: string
      format: 'date-time'
    modified_at:
      description: when got this node last updated
      type: string
      format: 'date-time'
    type:
      type: string
      enum: [ 'LEAF', 'NODE' ]
    parent_node_id:
      description: the identifier of the parent node of this node
      type: string
  example:
    id: '123435'
    created_at: '2017-04-12T23:20:50.52Z'
    modified_at: '2017-04-12T23:20:50.52Z'
    type: 'LEAF'
    parent_node_id: '534321'
```

MUST use the common address fields

Address structures play a role in different business and use-case contexts, including country variances. All attributes that relate to address information must follow the naming and semantics defined below.

```yaml
addressee:
  description: a (natural or legal) person that gets addressed
  type: object
  properties:
    salutation:
      description: |
        a salutation and/or title used for personal contacts to some
        addressee; not to be confused with the gender information!
      type: string
      example: Mr
    first_name:
      description: |
        given name(s) or first name(s) of a person; may also include the
        middle names.
      type: string
```

```yaml
      example: Hans Dieter
    last_name:
      description: |
        family name(s) or surname(s) of a person
      type: string
      example: Mustermann
    business_name:
      description: |
        company name of the business organization. Used when a business is
        the actual addressee; for personal shipments to office addresses, use
        `care_of` instead.
      type: string
      example: Consulting Services GmbH
  required:
    - first_name
    - last_name

address:
  description:
    an address of a location/destination
  type: object
  properties:
    care_of:
      description: |
        (aka c/o) the person that resides at the address, if different from
        addressee. E.g. used when sending a personal parcel to the
        office /someone else's home where the addressee resides temporarily
      type: string
      example: Consulting Services GmbH
    street:
      description: |
        the full street address including house number and street name
      type: string
      example: Schönhauser Allee 103
    additional:
      description: |
        further details like building name, suite, apartment number, etc.
      type: string
      example: 2. Hinterhof rechts
    city:
      description: |
        name of the city / locality
      type: string
      example: Berlin
    zip:
      description: |
        zip code or postal code
      type: string
      example: 14265
    country_code:
      description: |
```

```
        the country code according to
        [iso-3166-1-alpha-2](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2)
      type: string
      example: DE
  required:
    - street
    - city
    - zip
    - country_code
```

Grouping and cardinality of fields in specific data types may vary based on the specific use case (e.g. combining addressee and address fields into a single type when modeling an address label vs distinct addressee and address types when modeling users and their addresses).

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://opensource.zalando.com/restful-api-guidelines/models/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:

```
{
  "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.

9. REST Basics - HTTP requests

MUST use HTTP methods correctly

Be compliant with the standardized HTTP method semantics (see HTTP/1 RFC-7230 and RFC-7230 updates from 2014) summarized as follows:

GET

`GET` requests are used to **read** either a single or a collection resource.

- `GET` requests for individual resources will usually generate a 404 if the resource does not exist
- `GET` requests for collection resources may return either 200 (if the collection is empty) or 404 (if the collection is missing)
- `GET` requests must NOT have a request body payload (see `GET with body`)

Note: `GET` requests on collection resources should provide sufficient filter and REST Design - Pagination mechanisms.

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:

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

Note: It is no option to encode the lengthy structured request information using header parameters. From a conceptual point of view, the semantic of an operation should always be expressed by the resource names, as well as the involved path and query parameters. In other words by everything that goes into the URL. Request headers are reserved for general context information (see **SHOULD** use only the specified proprietary headers). In addition, size limits on query parameters and headers are not reliable and depend on clients, gateways, server, and actual settings. Thus, switching to headers does not solve the original problem.

Hint: As `GET with body` is used to transport extensive query parameters, the `cursor` cannot any longer be used to encode the query filters in case of cursor-based pagination. As a consequence, it is best practice to transport the query filters in the body payload, while using pagination links containing the `cursor` that is only encoding the page position and direction. To protect the pagination sequence the `cursor` may contain a hash over all applied query filters (See also **SHOULD** use pagination links where applicable).

PUT

`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, plus possibly server-generated fields like `modified_at`)
- 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)

Important: It is good practice to prefer `POST` over `PUT` for creation of (at least top-level) resources. This leaves the resource identifier management under control of the service and not the client, and focuses `PUT` on its usage for updates. However, in situations where all resource attributes including the identifier are under control of the client as input for the resource creation you should use `PUT` and pass the resource identifier via the URL path. 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) without data duplication in case of repetition.

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.

POST

`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"*.

- 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).

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.

PATCH

`PATCH` method extends HTTP via RFC-5789 standard to update parts of the resource objects where e.g. in contrast to `PUT` only a specific subset of resource fields should be changed. The set of changes is represented in a format called a *patch document* passed as payload and identified by a specific media type. The semantic is best described as *"please change the resource identified by the URL according to my patch document"*. The syntax and semantics of the patch document is not defined in RFC-5789 and must be described in the API specification by using specific media types.

- `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)

Note: since implementing PATCH correctly is a bit tricky, we strongly suggest to choose one and only one of the following patterns per endpoint (unless forced by a backwards compatible change). In preference order:

1. use PUT with complete objects to update a resource as long as feasible (i.e. do not use PATCH at all).
2. use PATCH with JSON Merge Patch standard, a specialized media type application/merge-patch+json for partial resource representation to update parts of resource objects.
3. use PATCH with JSON Patch standard, a specialized media type application/json-patch+json that includes instructions on how to change the resource.
4. use POST (with a proper description of what is happening) instead of PATCH, if the request does not modify the resource in a way defined by the semantics of the standard media types above.

In practice JSON Merge Patch quickly turns out to be too limited, especially when trying to update single objects in large collections (as part of the resource). In this cases JSON Patch is more powerful while still showing readable patch requests (see also JSON patch vs. merge). JSON Patch supports changing of array elements identified via its index, but not via (key) fields of the elements as typically needed for collections.

Note: Patching the same resource twice is **not** required to be idempotent (check **MUST** fulfill common method properties) and may result in a changing result. However, you **SHOULD** consider to design POST and PATCH idempotent to prevent this.

Hint: To prevent unnoticed concurrent updates when using PATCH 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 Optimistic locking in RESTful APIs and **SHOULD** consider to design POST and PATCH idempotent for details and options.

DELETE

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

- 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

`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

`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
`PATCH`	□ No	⚠️ No, but SHOULD consider to design <code>POST</code> and <code>PATCH</code> idempotent	□ No
`DELETE`	□ No	□ Yes	□ No

Method	Safe	Idempotent	Cacheable
`OPTIONS`	✓ Yes	✓ Yes	✗ No
`TRACE`	✓ Yes	✓ Yes	✗ No

Note: **MUST** document cachable `GET`, `HEAD`, and `POST` endpoints.

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 **SHOULD** 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.

SHOULD 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

Header and query parameters allow to provide a collection of values, either by providing a comma-separated list of values or by repeating the parameter multiple times with different values as follows:

Parameter Type	Comma-separated Values	Multiple Parameters	Standard
Header	`Header: value1,value2`	`Header: value1, Header: value2`	RFC 7230 Section 3.2.2
Query	`?param=value1,value2`	`?param=value1&param=value2`	RFC 6570 Section 3.2.8

As OpenAPI does not support both schemas at once, an API specification must explicitly define the collection format to guide consumers as follows:

Parameter Type	Comma-separated Values	Multiple Parameters
Header	`style: simple, explode: false`	not allowed (see RFC 7230 Section 3.2.2)
Query	`style: form, explode: false`	`style: form, explode: true`

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

SHOULD design simple query languages using query parameters

We prefer the use of query parameters to describe resource-specific query languages for the majority of APIs because it's native to HTTP, easy to extend and has excellent implementation support in HTTP clients and web frameworks.

Query parameters should have the following aspects specified:

- Reference to corresponding property, if any
- Value range, e.g. inclusive vs. exclusive
- Comparison semantics (equals, less than, greater than, etc)
- Implications when combined with other queries, e.g. *and* vs. *or*

How query parameters are named and used is up to individual API designers. The following examples should serve as ideas:

- `name=Alice`, to query for elements based on property equality
- `age=5`, to query for elements based on logical properties
  - Assuming that elements don't actually have an `age` but rather a `birthday`
- `max_length=5`, based on upper and lower bounds (`min` and `max`)
- `shorter_than=5`, using terminology specific e.g. to *length*
- `created_before=2019-07-17` or `not_modified_since=2019-07-17`
  - Using terminology specific e.g. to time: *before*, *after*, *since* and *until*

We don't advocate for or against certain names because in the end APIs should be free to choose the terminology that fits their domain the best.

SHOULD 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 response 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.
```

10. REST Basics - HTTP status codes

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 **MUST** only use durable and immutable remote references):

```
responses:
  ...
  default:
    description: error occurred - see status code and problem object for more
information.
    content:
      "application/problem+json":
        schema:
          $ref: 'https://opensource.zalando.com/restful-api-guidelines/models/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 official HTTP status codes

You must only use official HTTP status codes consistently with their intended semantics. Official HTTP status codes are defined via RFC standards and registered in the IANA Status Code Registry. Main RFC standards are RFC7231 and RFC 6585 (and there are upcoming new ones, e.g. draft legally-restricted-status). An overview on the official error codes provides Wikipedia: HTTP status codes (which also lists some unofficial status codes, e.g. defined by popular web servers like Nginx, that we do not suggest to use).

SHOULD only use most common HTTP status codes

The most commonly used codes are best understood and listed below as subset of the official HTTP status codes and consistent with their semantics in the RFCs. We avoid less commonly used codes that easily create misconceptions due to less familar semantics and API specific interpretations.

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 [standard-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>`
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>`

Code	Meaning	Methods
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.

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:

```yaml
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

RFC 7807 defines a Problem JSON object using the media type `application/problem+json` to provide an extensible human and machine readable failure information beyond the HTTP response status code to transports the failure kind (`type` / `title`) and the failure cause and location (`instant` / `detail`). To make best use of this additional failure information, every endpoints must be capable of returning a Problem JSON on client usage errors (4xx status codes) as well as server side processing errors (5xx status codes).

Note: Clients must be robust and **not rely** on a Problem JSON object being returned, since (a) failure responses may be created by infrastructure components not aware of this guideline or (b) service may be unable to comply with this guidelines in certain error situations.

Hint: The media type `application/problem+json` is often not implemented as a subset of `application/json` by libraries and services! Thus clients need to include `application/problem+json` in the `Accept`-Header to trigger delivery of the extended failure information.

The OpenAPI schema definition of the Problem JSON object can be found on GitHub. You can reference it by using:

```
responses:
  503:
    description: Service Unavailable
    content:
      "application/problem+json":
        schema:
          $ref: 'https://opensource.zalando.com/restful-api-guidelines/models/problem-
1.0.1.yaml#/Problem'
```

You may define custom problem types as extensions of the Problem JSON object if your API needs to return specific, additional, and more detailed error information.

Note: Problem `type` and `instance` identifiers in our APIs are not meant to be resolved. RFC 7807 encourages that problem types are URI references that point to human-readable documentation,

but we deliberately decided against that, as all important parts of the API must be documented using OpenAPI anyway. In addition, URLs tend to be fragile and not very stable over longer periods because of organizational and documentation changes and descriptions might easily get out of sync.

In order to stay compatible with RFC 7807 we proposed to use relative URI references usually defined by `absolute-path [ '?' query ] [ '#' fragment ]` as simplified identifiers in `type` and `instance` fields:

- `/problems/out-of-stock`
- `/problems/insufficient-funds`
- `/problems/user-deactivated`
- `/problems/connection-error#read-timeout`

Hint: The use of absolute URIs is not forbidden but strongly discouraged. If you use absolute URIs, please reference problem-1.0.0.yaml#/Problem instead.

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.

11. REST Basics - HTTP headers

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

Though we generally discourage usage of proprietary headers, they are useful to pass generic, service independent, overarching information relevant for our specific application architecture. We consistently define these proprietary headers in this section below. Whether services support these concerns or not is optional. Therefore, the OpenAPI API specification is the right place to make this explicitly visible — use the parameter definitions of the resource HTTP methods.

MAY use standard headers

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

SHOULD use kebab-case with uppercase separate words for HTTP headers

This convention is followed by (most of) the 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

`Content-Location` is an *optional* response header that 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 URL 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 URL 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:

```yaml
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, we do not want to change the header name and semantic, and do not name it like the proprietry headers below. The header addresses a generic REST concern and is different from the specific proprietary headers.

SHOULD use only the specified proprietary headers

As a general rule, proprietary HTTP headers should be avoided. From a conceptual point of view, the business semantics and intent of an operation should always be expressed via the URLs path and query parameters, the method, and the content, but not via proprietary headers. Headers are typically used to implement protocol processing aspects, such as flow control, content negotiation, and authentication, and represent business agnostic request modifiers that provide generic context information (RFC 7231).

However, the exceptional usage of proprietary headers is still helpful when domain-specific generic context information…

1. needs to be passed end to end along the service call chain (even if not all called services use it as input for steering service behavior e.g. `X-Sales-Channel` header) and/or…
2. is provided by specific gateway components, for instance, our Fashion Shop API or Merchant API gateway.

Below, we explicitly define the list of proprietary header exceptions usable for all services for passing through generic context information of our fashion domain (use case 1).

Per convention, non standardized, proprietary header names are prefixed with `X-`. (Due to backward compatibility, we do not follow the Internet Engineering Task Force's recommendation in RFC 6648 to deprecate usage of `X-` headers.) Remember that HTTP header field names are not case-sensitive:

Header field name	Type	Description	Header field value example
{X-Example}	String	For more information see above.	There once was a custom header

MUST propagate proprietary headers

All proprietary headers listed above are end-to-end headers [2] and must be propagated to the services down the call chain. The header names and values must remain unchanged.

For example, the values of the custom headers like `X-device-Type` can affect the results of queries by using device type information to influence recommendation results. Besides, the values of the custom headers can influence the results of the queries (e.g. the device type information influences the recommendation results).

Sometimes the value of a proprietary header will be used as part of the entity in a subsequent request. In such cases, the proprietary headers must still be propagated as headers with the subsequent request, despite the duplication of information.

12. REST Design - 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.:

- **MUST** avoid actions — think about resources
- **MUST** keep URLs verb-free
- **MUST** use HTTP methods correctly
- **SHOULD** only use most common 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 avoid REST maturity level 3 - HATEOAS

We do not generally recommend to implement REST Maturity Level 3. HATEOAS comes with additional API complexity without real value in our SOA context where client and server interact via REST APIs and provide complex business functions as part of our e-commerce SaaS platform.

Our major concerns regarding the promised advantages of HATEOAS (see also RESTistential Crisis over Hypermedia APIs, Why I Hate HATEOAS and others for a detailed discussion):

- We follow the API First principle with APIs explicitly defined outside the code with standard specification language. HATEOAS does not really add value for SOA client engineers in terms of API self-descriptiveness: a client engineer finds necessary links and usage description (depending on resource state) in the API reference definition anyway.
- Generic HATEOAS clients which need no prior knowledge about APIs and explore API capabilities based on hypermedia information provided, is a theoretical concept that we haven't seen working in practice and does not fit to our SOA set-up. The OpenAPI description format (and tooling based on OpenAPI) doesn't provide sufficient support for HATEOAS either.
- In practice relevant HATEOAS approximations (e.g. following specifications like HAL or JSON API) support API navigation by abstracting from URL endpoint and HTTP method aspects via link types. So, Hypermedia does not prevent clients from required manual changes when domain model changes over time.
- Hypermedia make sense for humans, less for SOA machine clients. We would expect use cases where it may provide value more likely in the frontend and human facing service domain.
- Hypermedia does not prevent API clients to implement shortcuts and directly target resources without 'discovering' them.

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:

```yaml
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`):

```json
{
  "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.

SHOULD use simple hypertext controls for pagination and self-references

For pagination and self-references a simplified form of the extensible common hypertext controls should be used to reduce the specification and cognitive overhead. It consists of a simple URI value in combination with the corresponding link relations, e.g. `next`, `prev`, `first`, `last`, or `self`.

See **MUST** use common hypertext controls and **SHOULD** use pagination links where applicable for more information and examples.

MUST use full, absolute URI for resource identification

Links to other resource must 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 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.

13. REST Design - Performance

SHOULD reduce bandwidth needs and improve responsiveness

APIs should support techniques for reducing bandwidth based on client needs. This holds for APIs that (might) have high payloads and/or are used in high-traffic scenarios like the public Internet and telecommunication networks. Typical examples are APIs used by mobile web app clients with (often) less bandwidth connectivity.

Common techniques include:

- compression of request and response bodies (see **SHOULD** use `gzip` compression)
- querying field filters to retrieve a subset of resource attributes (see **SHOULD** support partial responses via filtering below)
- `ETag` and `If-Match`/`If-None-Match` headers to avoid re-fetching of unchanged resources (see **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header)
- `Prefer` header with `return=minimal` or `respond-async` to anticipate reduced processing

requirements of clients (see **MAY** consider to support `Prefer` header to handle processing preferences)

- REST Design - Pagination for incremental access of larger collections of data items
- caching of master data items, i.e. resources that change rarely or not at all after creation (see **MUST** document cachable `GET`, `HEAD`, and `POST` endpoints).

Each of these items is described in greater detail below.

SHOULD use `gzip` compression

Compress the payload of your API's responses with gzip, unless there's a good reason not to — for example, you are serving so many requests that the time to compress becomes a bottleneck. This helps to transport data faster over the network (fewer bytes) and makes frontends respond faster.

Though gzip compression might be the default choice for server payload, the server should also support payload without compression and its client control via `Accept-Encoding` request header — see also RFC 7231 Section 5.3.4. The server should indicate used gzip compression via the `Content-Encoding` header.

SHOULD support partial responses via filtering

Depending on your use case and payload size, you can significantly reduce network bandwidth need by supporting filtering of returned entity fields. Here, the client can explicitly determine the subset of fields he wants to receive via the `fields` query parameter. (It is analogue to GraphQL `fields` and simple queries, and also applied, for instance, for Google Cloud API's partial responses.)

Unfiltered

```
GET http://api.example.org/users/123 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "cddd5e44-dae0-11e5-8c01-63ed66ab2da5",
  "name": "John Doe",
  "address": "1600 Pennsylvania Avenue Northwest, Washington, DC, United States",
  "birthday": "1984-09-13",
  "friends": [ {
    "id": "1fb43648-dae1-11e5-aa01-1fbc3abb1cd0",
    "name": "Jane Doe",
    "address": "1600 Pennsylvania Avenue Northwest, Washington, DC, United States",
    "birthday": "1988-04-07"
  } ]
}
```

Filtered

```
GET http://api.example.org/users/123?fields=(name,friends(name)) HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "John Doe",
  "friends": [ {
    "name": "Jane Doe"
  } ]
}
```

The `fields` query parameter determines the fields returned with the response payload object. For instance, `(name)` returns `users` root object with only the `name` field, and `(name,friends(name))` returns the `name` and the nested `friends` object with only its `name` field.

OpenAPI doesn't support you in formally specifying different return object schemes depending on a parameter. When you define the field parameter, we recommend to provide the following description: `Endpoint supports filtering of return object fields as described inRule #157

The syntax of the query `fields` value is defined by the following BNF grammar.

```
<fields>            ::= [ <negation> ] <fields_struct>
<fields_struct>     ::= "(" <field_items> ")"
<field_items>       ::= <field> [ "," <field_items> ]
<field>             ::= <field_name> | <fields_substruct>
<fields_substruct>  ::= <field_name> <fields_struct>
<field_name>        ::= <dash_letter_digit> [ <field_name> ]
<dash_letter_digit> ::= <dash> | <letter> | <digit>
<dash>              ::= "-" | "_"
<letter>            ::= "A" | ... | "Z" | "a" | ... | "z"
<digit>             ::= "0" | ... | "9"
<negation>          ::= "!"
```

Note: Following the principle of least astonishment, you should not define the `fields` query parameter using a default value, as the result is counter-intuitive and very likely not anticipated by the consumer.

SHOULD allow optional embedding of sub-resources

Embedding related resources (also know as *Resource expansion*) is a great way to reduce the number of requests. In cases where clients know upfront that they need some related resources they can instruct the server to prefetch that data eagerly. Whether this is optimized on the server, e.g. a database join, or done in a generic way, e.g. an HTTP proxy that transparently embeds resources, is up to the implementation.

See **MUST** stick to conventional query parameters for naming, e.g. "embed" for steering of embedded resource expansion. Please use the BNF grammar, as already defined above for filtering, when it comes to an embedding query syntax.

Embedding a sub-resource can possibly look like this where an order resource has its order items as sub-resource (/order/{orderId}/items):

```
GET /order/123?embed=(items) HTTP/1.1

{
  "id": "123",
  "_embedded": {
    "items": [
      {
        "position": 1,
        "sku": "1234-ABCD-7890",
        "price": {
          "amount": 71.99,
          "currency": "EUR"
        }
      }
    ]
  }
}
```

MUST document cachable `GET`, `HEAD`, and `POST` endpoints

Caching has to take many aspects into account, e.g. general cacheability of response information, our guideline to protect endpoints using SSL and OAuth authorization, resource update and invalidation rules, existence of multiple consumer instances. As a consequence, caching is in best case complex, e.g. with respect to consistency, in worst case inefficient.

As a consequence, client side as well as transparent web caching should be avoided, unless the service supports and requires it to protect itself, e.g. in case of a heavily used and therefore rate limited master data service, i.e. data items that rarely or not at all change after creation.

As default, API providers and consumers should always set the `Cache-Control` header set to `Cache-Control: no-store` and assume the same setting, if no `Cache-Control` header is provided.

Note: There is no need to document this default setting. However, please make sure that your framework is attaching this header value by default, or ensure this manually, e.g. using the best practice of Spring Security as shown below. Any setup deviating from this default must be sufficiently documented.

```
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
```

If your service really requires to support caching, please observe the following rules:

- Document all cacheable `GET`, `HEAD`, and `POST` endpoints by declaring the support of `Cache-Control`, `Vary`, and `ETag` headers in response. **Note:** you must not define the `Expires` header to prevent redundant and ambiguous definition of cache lifetime. A sensible default documentation of these headers is given below.
- Take care to specify the ability to support caching by defining the right caching boundaries, i.e. time-to-live and cache constraints, by providing sensible values for `Cache-Control` and `Vary` in your service. We will explain best practices below.
- Provide efficient methods to warm up and update caches, e.g. as follows:
  - In general, you should support `ETag` Together With `If-Match`/ `If-None-Match` Header on all cacheable endpoints.
  - For larger data items support `HEAD` requests or more efficient `GET` requests with `If-None-Match` header to check for updates.
  - For small data sets provide full collection `GET` requests supporting `ETag`, as well as `HEAD` requests or `GET` requests with `If-None-Match` to check for updates.
  - For medium sized data sets provide full collection `GET` requests supporting `ETag` together with REST Design - Pagination and `<entity-tag>` filtering `GET` requests for limiting the response to changes since the provided `<entity-tag>`. **Note:** this is not supported by generic client and proxy caches on HTTP layer.

Hint: For proper cache support, you must return 304 without content on a failed `HEAD` or `GET` request with `If-None-Match: <entity-tag>` instead of 412.

```yaml
components:
  headers:
  - Cache-Control:
      description: |
        The RFC 7234 Cache-Control header field is providing directives to
        control how proxies and clients are allowed to cache responses results
        for performance. Clients and proxies are free to not support caching of
        results, however if they do, they must obey all directives mentioned in
        [RFC-7234 Section 5.2.2](https://tools.ietf.org/html/rfc7234) to the
        word.

        In case of caching, the directive provides the scope of the cache
        entry, i.e. only for the original user (private) or shared between all
        users (public), the lifetime of the cache entry in seconds (max-age),
        and the strategy how to handle a stale cache entry (must-revalidate).
        Please note, that the lifetime and validation directives for shared
        caches are different (s-maxage, proxy-revalidate).

      type: string
      required: false
      example: "private, must-revalidate, max-age=300"

  - Vary:
      description: |
        The RFC 7231 Vary header field in a response defines which parts of
        a request message, aside the target URL and HTTP method, might have
        influenced the response. A client or proxy cache must respect this
        information, to ensure that it delivers the correct cache entry (see
        [RFC-7231 Section
        7.1.4](https://tools.ietf.org/html/rfc7231#section-7.1.4)).

      type: string
      required: false
      example: "accept-encoding, accept-language"
```

Hint: For `ETag` source see **MAY** consider to support `ETag` together with `If-Match`/`If-None-Match` header.

The default setting for `Cache-Control` should contain the `private` directive for endpoints with standard OAuth authorization, as well as the `must-revalidate` directive to ensure, that the client does not use stale cache entries. Last, the `max-age` directive should be set to a value between a few seconds (`max-age=60`) and a few hours (`max-age=86400`) depending on the change rate of your master data and your requirements to keep clients consistent.

```
Cache-Control: private, must-revalidate, max-age=300
```

The default setting for `Vary` is harder to determine correctly. It highly depends on the API endpoint, e.g. whether it supports compression, accepts different media types, or requires other request specific headers. To support correct caching you have to carefully choose the value. However, a good first default may be:

```
Vary: accept, accept-encoding
```

Anyhow, this is only relevant, if you encourage clients to install generic HTTP layer client and proxy caches.

Note: generic client and proxy caching on HTTP level is hard to configure. Therefore, we strongly recommend to attach the (possibly distributed) cache directly to the service (or gateway) layer of your application. This relieves from interpreting the `Vary` header and greatly simplifies interpreting the `Cache-Control` and `ETag` headers. Moreover, is highly efficient with respect to caching performance and overhead, and allows to support more advanced cache update and warm up patterns.

Anyhow, please carefully read RFC 7234 before adding any client or proxy cache.

14. REST Design - 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 (See **SHOULD** use pagination links where applicable). This favours cursor-based over offset-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.

SHOULD prefer cursor-based pagination, avoid offset-based pagination

Cursor-based pagination is usually better and more efficient when compared to offset-based pagination. Especially when it comes to high-data volumes and/or storage in NoSQL databases.

Before choosing cursor-based pagination, consider the following trade-offs:

- Usability/framework support:
  - Offset-based pagination is more widely known than cursor-based pagination, so it has more framework support and is easier to use for API clients
- Use case - jump to a certain page:
  - If jumping to a particular page in a range (e.g., 51 of 100) is really a required use case, cursor-based navigation is not feasible.
- Data changes may lead to anomalies in result pages:
  - Offset-based pagination may create duplicates or lead to missing entries if rows are inserted or deleted between two subsequent paging requests.
  - If implemented incorrectly, cursor-based pagination may fail when the cursor entry has been deleted before fetching the pages.
- Performance considerations - efficient server-side processing using offset-based pagination is hardly feasible for:
  - Very big data sets, especially if they cannot reside in the main memory of the database.
  - Sharded or NoSQL databases.
- Cursor-based navigation may not work if you need the total count of results.

The `cursor` used for pagination is an opaque pointer to a page, that must never be **inspected** or **constructed** by clients. It usually encodes (encrypts) the page position, i.e. the identifier of the first or last page element, the pagination direction, and the applied query filters - or a hash over these - to safely recreate the collection (see also Cursor-based pagination in RESTful APIs).

SHOULD use pagination response page object

For iterating over collections (result sets) we propose to either use cursors (see **SHOULD** prefer cursor-based pagination, avoid offset-based pagination) or simple hypertext controls (see **SHOULD** use pagination links where applicable). To implement these in a consistent way, we have defined a response page object pattern with the following field semantics:

- `self`:the link or cursor in a pagination response or object pointing to the same collection object or page.
- `first`: the link or cursor in a pagination response or object pointing to the first collection object or page.
- `prev`: the link or cursor in a pagination response or object pointing to the previous collection object or page.

- `next`: the link or cursor in a pagination response or object pointing to the next collection object or page.
- `last`: the link or cursor in a pagination response or object pointing to the last collection object or page.

Pagination responses should contain the following additional array field to transport the page content:

- `items`: array of resources, holding all the items of the current page (`items` may be replaced by a resource name).

To simplify user experience, the applied query filters may be returned using the following field (see also `GET with body`):

- `query`: object containing the query filters applied in the search request to filter the collection resource.

As Result, the standard response page using cursors or pagination links may be defined as follows:

```yaml
ResponsePage:
  type: object
  required:
    - items
    - next
  properties:
    self:
      description: Pagination link|cursor pointing to the current page.
      type: string
      format: uri|cursor
    first:
      description: Pagination link|cursor pointing to the first page.
      type: string
      format: uri|cursor
    prev:
      description: Pagination link|cursor pointing to the previous page.
      type: string
      format: uri|cursor
    next:
      description: Pagination link|cursor pointing to the next page.
      type: string
      format: uri|cursor
    last:
      description: Pagination link|cursor pointing to the last page.
      type: string
      format: uri|cursor

    query:
      description: >
        Object containing the query filters applied to the collection resource.
      type: object
      properties: ...

    items:
      description: Array of collection items.
      type: array
      required: false
      items:
        type: ...
```

Note: While you may support cursors for `next`, `prev`, `first`, `last`, and `self`, it is best practice to replace these with links in favor of **SHOULD** use pagination links where applicable.

SHOULD 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": "http://my-service.zalandoapis.com/resources?cursor=<self-position>",
  "first": "http://my-service.zalandoapis.com/resources?cursor=<first-position>",
  "prev": "http://my-service.zalandoapis.com/resources?cursor=<previous-position>",
  "next": "http://my-service.zalandoapis.com/resources?cursor=<next-position>",
  "last": "http://my-service.zalandoapis.com/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.

15. REST Design - 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 media type 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.
- Use `x-extensible-enum`, if range is used for output parameters and likely to be extended with growing functionality. It defines an open list of explicit values and clients must be agnostic to new values.
- Support redirection in case an URL has to change 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 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 srvice clients must be prepared for compatible API extensions of response data:
  - 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).

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

As we discourage versioning by all means because of the manifold disadvantages, we strongly recommend to only use the first two approaches.

Tip: versioning should always be the last resort and decided with peers.

MUST use media type versioning

However, when API versioning is unavoidable, you have to design your multi-version RESTful APIs using media type versioning (see **MUST** not use URL versioning). Media type versioning is less tightly coupled since it supports content negotiation and hence reduces complexity of release management.

Version information and media type are provided together via the HTTP `Content-Type` header — e.g. `application/vendor.cart+json;version=2`. For incompatible changes, a new media type version for the resource is created. To generate the new representation version, consumer and producer can do content negotiation using the HTTP `Content-Type` and `Accept` headers.

> **NOTE** This versioning only applies to the request and response content schema, not to URI or method semantics.

Custom media type format

Custom media type format should have the following pattern:

```
application/x.<custom-media-type>+json;version=<version>
```

- `custom-media-type` is a custom type name, e.g. `vendor.cart`
- `version` is a number, e.g. `2`

Example

In this example, a client wants only the new version of the response:

```
Accept: application/vendor.cart+json;version=2
```

A server responding to this, as well as a client sending a request with content should use the `Content-Type` header, declaring that one is sending the new version:

```
Content-Type: application/vendor.cart+json;version=2
```

Using media type versioning should:

- Use a custom media type, e.g. `application/vendor.cart+json`
- Include versions in request and response headers to increase visibility
- Include `Content-Type` in the `Vary` header to enable proxy caches to differ between versions

```
Vary: Content-Type
```

TIP

You **SHOULD** avoid media type versioning. Until an incompatible change is necessary, it is recommended to stay with the standard `application/json` media type.

Further reading: API Versioning Has No "Right Way" provides an overview on different versioning approaches to handle breaking changes without being opinionated.

MUST not use URL versioning

With URL versioning a (major) version number is included in the path, e.g. `/v1/customers`. The consumer has to wait until the provider has been released and deployed. If the consumer also supports hypermedia links — even in their APIs — to drive workflows (HATEOAS), this quickly becomes complex. So does coordinating version upgrades — especially with hyperlinked service dependencies — when using URL versioning. To avoid this tighter coupling and complexer release management we do not use URL versioning, instead we **MUST** use media type versioning with content negotiation.

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 **SHOULD** 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.

SHOULD used open-ended list of values (`x-extensible-enum`) for enumerations

Enumerations are per definition closed sets of values, that are assumed to be complete and not intended for extension. This closed principle of enumerations imposes compatibility issues when an enumeration must be extended. To avoid these issues, we strongly recommend to use an open-ended list of values instead of an enumeration unless:

1. the API has full control of the enumeration values, i.e. the list of values does not depend on any external tool or interface, and
2. the list of value is complete with respect to any thinkable and unthinkable future feature.

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

```yaml
delivery_methods:
  type: string
  x-extensible-enum:
    - PARCEL
    - LETTER
    - EMAIL
```

Note: `x-extensible-enum` is not JSON Schema conform but will be ignored by most tools.

See **SHOULD** declare enum values using UPPER_SNAKE_CASE string about enum value naming conventions.

16. REST Design - 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.

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 obtain approval of clients before API shut down

Before shutting down an API, version of an API, or API feature the producer must 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). Once all clients of a sunset API feature are migrated, the producer may shut down the deprecated API feature.

MUST collect external partner consent on deprecation time span

If the API is consumed by any external partner, the API owner must define a reasonable time span that the API will be maintained after the producer has announced deprecation. All external partners must state consent with this after-deprecation-life-span, i.e. the minimum time span between official deprecation and first possible sunset, **before** they are allowed to use the API.

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 **SHOULD** monitor API usage.

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.

17. REST 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.

Note: To publish an API, it is still necessary to deploy the artifact successful, as we focus the discovery experience on APIs supported by running services.

SHOULD 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.

18. EVENT Basics - Event Types

WIP: At this point in time we do not have strict standards for event driven services.

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 frameworks were specifically designed to support the API First workflow with OpenAPI YAML files (sorted alphabetically):

- **FastAPI**: FastAPI framework

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

Support libraries

These utility libraries support you in implementing various parts of our RESTful API guidelines (sorted alphabetically):

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 **SHOULD** prefer cursor-based pagination, avoid offset-based pagination, 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:

```yaml
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.
```

```yaml
        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": "00000042" },
>     { "id": "00000043" }
>   ]
> }

< GET /orders/BO0000042

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

< PUT /orders/00000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { "id": "00000042", ... }

> 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": "00000042", "etag": "osjnfkjbnkq3jlnksjnvkjlsbf", "foo": 42, "bar": true
},
>     { "id": "00000043", "etag": "kjshdfknjqlowjdsljdnfkjbkn", "foo": 24, "bar":
false }
>   ]
> }

< PUT /orders/00000042
< If-Match: osjnfkjbnkq3jlnksjnvkjlsbf
< { "id": "00000042", "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/00000042`) 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 2016.

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

MUST publish OpenAPI specification, Removed Zalando specific part

- `2021-11-02`: document restructuring
- `2021-06-22`: **MUST** use standard data formats changed from **SHOULD** to **MUST**; consistency for rules around standards for data.
- `2021-06-03`: **MUST** secure endpoints with clear distinction of OpenAPI security schemes, favoring `bearer` to `oauth2`.
- `2021-06-01`: resolve uncertainties around 'occurred_at' semantics of event metadata.
- `2021-05-25`: **SHOULD** use standard media types with API endpoint versioning as only custom media type usage exception.
- `2021-05-05`: define usage on resource-ids in `PUT` and `POST` in **MUST** use HTTP methods correctly.
- `2021-04-29`: improve clarity of **MAY** use standard headers.
- `2021-03-19`: clarity on **MUST** use JSON as payload data interchange format.
- `2021-03-15`: [242] changed from **SHOULD** to **MUST**; improve clarity around event ordering.
- `2021-03-19`: best practice section Cursor-based pagination in RESTful APIs
- `2021-02-16`: define how to reference models outside the api in **MUST** only use durable and immutable remote references.
- `2021-02-15`: improve guideline **MUST** support problem JSON — clients must be prepared to not receive problem return objects.
- `2021-01-19`: more details for `GET with body` and `DELETE with body` (**MUST** use HTTP methods correctly).
- `2020-09-29`: include models for headers to be included by reference in API definitions (**SHOULD** use only the specified proprietary headers)
- `2020-09-08`: add exception for legacy host names to **MUST** follow naming convention for hostnames
- `2020-08-25`: change **SHOULD** declare enum values using UPPER_SNAKE_CASE string from **MUST** to **SHOULD**, explain exceptions
- `2020-08-25`: add exception for `self` to **MUST** identify resources and sub-resources via path segments and **MUST** pluralize resource names.
- `2020-08-24`: change "**MUST** avoid trailing slashes" to **MUST** use normalized paths without empty path segments and trailing slashes.
- `2020-08-20`: change **SHOULD** use only the specified proprietary headers from **MUST** to **SHOULD**, mention gateway-specific headers (which are not part of the public API).
- `2020-06-30`: add details to **MUST** use media type versioning

- 2020-05-19: new sections about DELETE with query parameters and DELETE with body in **MUST** use HTTP methods correctly.
- 2020-02-06: new rule **MAY** expose compound keys as resource identifiers
- 2020-02-05: add Sunset header, clarify deprecation producedure (**MUST** obtain approval of clients before API shut down, **MUST** collect external partner consent on deprecation time span, **MUST** reflect deprecation in API specifications, **MUST** monitor usage of deprecated API scheduled for sunset, **SHOULD** add Deprecation and Sunset header to responses, **SHOULD** add monitoring for Deprecation and Sunset header, **MUST** not start using deprecated APIs)
- 2020-01-21: new rule **SHOULD** declare enum values using UPPER_SNAKE_CASE string (as MUST, changed later to SHOULD)
- 2020-01-15: change "Warning" to "Deprecation" header in **SHOULD** add Deprecation and Sunset header to responses, **SHOULD** add monitoring for Deprecation and Sunset header.
- 2019-10-10: remove never-implemented rule "**MUST** Permissions on events must correspond to API permissions"
- 2019-09-10: remove duplicated rule "**MAY** Standards could be used for Language, Country and Currency", upgrade **MUST** use standard formats for country, language and currency properties from **MAY** to **SHOULD**.
- 2019-08-29: new rule **MUST** encode binary data in base64url, extend **MUST** use JSON as payload data interchange format pointing to RFC-7493
- 2019-08-29: new rules **SHOULD** design simple query languages using query parameters, **SHOULD** design complex query languages using JSON
- 2019-07-30: new rule **MUST** use standard data formats
- 2019-07-30: change **MUST** use the common money object from **SHOULD** to **MUST**
- 2019-07-30: change "**SHOULD** Null values should have their fields removed to" **MUST** use same semantics for null and absent properties.
- 2019-07-25: new rule **SHOULD** name date/time properties with _at suffix.
- 2019-07-18: improved cursor guideline for GET with body.
- 2019-06-25: change **MUST** define collection format of header and query parameters from **SHOULD** to **MUST**, use OpenAPI 3 syntax
- 2019-06-13: remove X-App-Domain from **SHOULD** use only the specified proprietary headers.
- 2019-05-17: add X-Mobile-Advertising-Id to **SHOULD** use only the specified proprietary headers.
- 2019-04-09 New rule **MUST** only use durable and immutable remote references
- 2019-02-19: New rule [233] extracted + expanded from **SHOULD** use only the specified proprietary headers.
- 2019-01-24: Improve guidance on caching (**MUST** fulfill common method properties, **MUST** document cachable GET, HEAD, and POST endpoints).
- 2019-01-21: Improve guidance on idempotency, introduce idempotency-key (**SHOULD** consider to design POST and PATCH idempotent, **SHOULD** use secondary key for idempotent POST design).
- 2019-01-16: Change **SHOULD** not use /api as base path from **MAY** to {SHOULD NOT}

- 2018-10-19: Add ordering_key_field to event type definition schema ([197], [203])
- 2018-09-28: New rule **MUST** use URL-friendly resource identifiers
- 2018-09-13: replaced OpenAPI 2.0 syntax with OpenAPI 3.0 in the example snippets
- 2018-08-10: New rule **MUST** document implicit response filtering
- 2018-07-12: Add audience field to event type definition ([197])
- 2018-06-11: Introduced new naming guidelines for host, permission, and event names.
- 2018-01-10: Moved meta information related aspects into new chapter REST Basics - Meta information.
- 2018-01-09: Changed publication requirements for API specifications (**MUST** publish OpenAPI specification).
- 2017-12-07: Added best practices section including discussion about optimistic locking approaches.
- 2017-11-28: Changed OAuth flow example from password to client credentials in REST Basics - Security.
- 2017-11-22: Updated description of X-Tenant-ID header field
- 2017-08-22: Migration to Asciidoc
- 2017-07-20: Be more precise on client vs. server obligations for compatible API extensions.
- 2017-06-06: Made money object guideline clearer.
- 2017-05-17: Added guideline on query parameter collection format.
- 2017-05-10: Added the convention of using RFC2119 to describe guideline levels, and replaced book.could with book.may.
- 2017-03-30: Added rule that permissions on resources in events must correspond to permissions on API resources
- 2017-03-30: Added rule that APIs should be modelled around business processes
- 2017-02-28: Extended information about how to reference sub-resources and the usage of composite identifiers in the **MUST** identify resources and sub-resources via path segments part.
- 2017-02-22: Added guidance for conditional requests with If-Match/If-None-Match
- 2017-02-02: Added guideline for batch and bulk request
- 2017-02-01: **SHOULD** use Location header instead of Content-Location header
- 2017-01-18: Removed "Avoid Javascript Keywords" rule
- 2017-01-05: Clarification on the usage of the term "REST/RESTful"
- 2016-12-07: Introduced "API as a Product" principle
- 2016-12-06: New guideline: "Should Only Use UUIDs If Necessary"
- 2016-12-04: Changed OAuth flow example from implicit to password in REST Basics - Security.
- 2016-10-13: **SHOULD** use standard media types
- 2016-10-10: Introduced the changelog. From now on all rule changes on API guidelines will be recorded here.

```
<!-- 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 REST Design - 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.

[2] HTTP/1.1 standard (RFC 7230) defines two types of headers: end-to-end and hop-by-hop headers. End-to-end headers must be transmitted to the ultimate recipient of a request or response. Hop-by-hop headers, on the contrary, are meaningful for a single connection only.