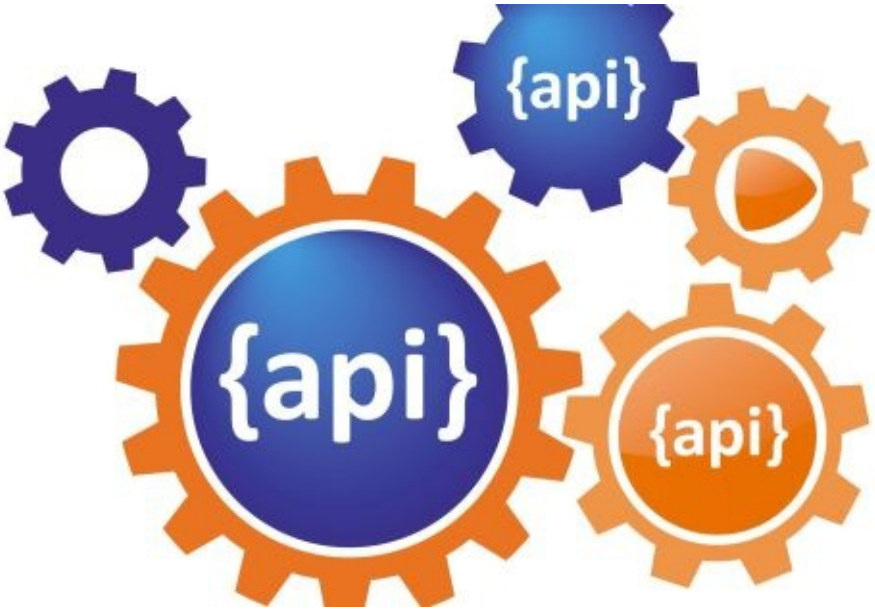
Table of Contents



Introduction

Zalando's software architecture centers around decoupled microservices that provide functionality via RESTful APIs with a JSON payload. Small engineering teams own, deploy and operate these microservices in their AWS (team) accounts. Our APIs most purely express what our systems do, and are therefore highly valuable business assets. Designing high-quality, long-lasting APIs has become even more critical for us since we started developing our new open platform strategy, which transforms Zalando from an online shop into an expansive fashion platform. Our strategy emphasizes developing lots of public APIs for our external business partners to use via third-party applications.

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 Zalando APIs will look like the same author created them.

Zalando specific information

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

Furthermore, teams should take part in API review process [internal link].

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

Table of Contents

Design Principles

To compare the interface design approaches of SOAP-based web services to those of REST services, the former tend to be centered around operations that are usually use-case specific and specialized. In contrast, REST is all about (business) entities found in the system and exposed as resource endpoints — leading to interfaces that are more broadly usable (here, clients sometimes have to filter out unnecessary information).

API Design Principles

1. We prefer REST-based APIs with JSON payloads
2. We prefer systems to be truly RESTful
3. We strive to build interoperating distributed systems that different teams can evolve in parallel

An important principle for (RESTful) API design and usage is Postel's Law, aka the Robustness Principle (RFC 1122): “Be liberal in what you accept, be conservative in what you send.”

Read the following to gain additional insight on the RESTful service architecture paradigm and general RESTful API design style:

- Fielding Dissertation: Architectural Styles and the Design of Network-Based Software Architectures
- Book: REST in Practice: Hypermedia and Systems Architecture
- Book: Build APIs You Won't Hate
- InfoQ eBook: Web APIs: From Start to Finish
- Lessons-learned blog: Thoughts on RESTful API Design

We apply the RESTful web service principles to all kind of application components, whether they provide functionality via the Internet or via the intranet as larger application elements. We strive to build interoperating distributed systems that different teams can evolve in parallel.

General Guidelines

The titles are marked with the corresponding labels: **Must:**, **Should:**, **Could:**.

Must: API First: Define APIs using OpenAPI

API First is one of our architecture principles, as per our Architecture Rules of Play. Define, document and review your APIs before delivering them (also see API review procedure). Whether you choose YAML or JSON as a format of your specification file is up to you.

Should: Provide External Documentation

In addition to defining the API with OpenAPI, it’s good practice to provide documentation that describes the API’s scope and purpose; architecture and use-case context (including figures); sequence flows; edge cases and possible error situations, etc. Include a link to this documentation using the “externalDocs” property in your RESTful API OpenAPI definition and post it online on GitHub Enterprise pages, on specific team web servers, or as a Google doc.

Must: Write APIs in U.S. English

Security

Must: Secure endpoints with OAuth 2.0

Every API endpoint needs to be secured using OAuth 2.0. Please refer to the official OpenAPI spec on how to specify security definitions in you API specification or take a look at the following example.

```
securityDefinitions:
  oauth2:
    type: oauth2
    flow: implicit
    authorizationUrl: https://auth.zalando.com/oauth2/access_token?realm=services
    scopes:
      uid: Unique identifier of the user accessing the service.
      fulfillment-order-service.read: Allows to read from the fulfillment order service.
```

In order to see how to define and assign scopes, please refer to the next section.

Must: Define and assign scopes

Every API needs to define scopes and every endpoint needs to have at least one scope assigned. Scopes are defined by a name and a description once per API specification, as shown in the previous section. Please refer to the following rules when creating scope names:

```
<scope> ::= <standard-scope> |           -- should be sufficient for majority of use cases
            <resource-specific-scope>    -- for special security access differentiation use cases
<standard-scope> ::= <application-id>.<access-type>
<resource-specific-scope> ::= <application-id>.<resource-id>.<access-type>

<application-id> ::= <as defined via STUPS>
<access-type> ::= read | write           -- might be extended in future
<resource-id> ::= <free identifier following application-id syntax>
```

APIs should stick to standard scopes by default -- for the majority of use cases, restricting access to specific APIs (with read vs. write differentiation) is sufficient for controlling access for client types like merchant or retailer business partners, customers or operational staff. We want to avoid too many, fine grained scopes increasing governance complexity without real value add. In some situations, where the API serves different types of resources for different owners, resource specific scopes may make sense.

Some examples for standard and resource-specific scopes:

Application ID	Resource ID	Access Type	Example
`fulfillment-order`		`read`	`fulfillment-order.read`
`fulfillment-order`		`write`	`fulfillment-order.write`
`sales-order`	`sales_order`	`read`	`sales-order.sales_order.read`
`sales-order`	`shipment_order`	`read`	`sales-order.shipment_order.read`

After scopes names are defined and the scope is declared in the security definition at the top of an API specification, it should be assigned to each API operation by specifying a `security` requirement like this:

```
paths:
  /sales-orders/{order-number}:
    get:
      summary: Retrieves a sales order
      security:
        - oauth2:
          - sales-order-service.sales_order.read
```

In very rare cases a whole API or some selected endpoints may not require specific access control. However, to make this explicit you should assign the `uid` scope in this case. It's available to every OAuth2 account by default.

Compatibility

Must: Don’t 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 service contracts 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. With Postel’s Law in mind, here are some rules for providers and consumers that allow us to make compatible changes without versioning:

Should: Prefer Compatible Extensions

Apply the following rules to evolve RESTful APIs in a backward-compatible way:

- Ignore unknown fields in the payload
- Add only optional, never mandatory fields
- Never change the meaning of a field.
- Enum ranges cannot not 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.
- 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 values can be reduced when used as output 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 (e.g. via casuistic with default behavior).
- Support redirection in case an URL has to change (301 Moved Permanently)

Must: Prepare Clients for Compatible API Extensions (the Robustness Principle)

How to do this:

- Ignore new and unknown fields in the payload (see also Fowler’s “TolerantReader” post)
- Be prepared for new enum values declared with x-extensible-enum (see above); provide default behavior for unknown values, if applicable
- Follow the redirect when the server returns an “HTTP 301 Moved Permanently” response code

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 suggest to only use the first two approaches.

Must: Use Media Type Versioning

When API versioning is unavoidable, you have to design your multi-version RESTful APIs using media type versioning (instead of URI versioning, see below). Media type versioning is less tightly coupled since it supports content negotiation and hence reduces complexity of release management.

Media type versioning: Here, version information and media type are provided together via the HTTP Content-Type header — e.g. application/x.zalando.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 content schema, not to URI or method semantics.

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

```
Content-Type: application/x.zalando.cart+json;version=2
Accept: application/x.zalando.cart+json;version=2
```

Using header versioning should:

- 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

Hint: OpenAPI currently doesn't support content negotiation, though a comment in this issue mentions a workaround (using a fragment identifier that gets stripped off).

Must: Do Not Use URI Versioning

With URI 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 URI versioning, and go instead with media type versioning and content negotiation (see above).

Should: Provide Version Information in OpenAPI Documentation

Only the documentation, not the API itself, needs version information.

Example:

```
"swagger": "2.0",
"info": {
  "title": "Parcel service API",
  "description": "API for <...>",
  "version": "1.0.0",
    <...>
}
```

During a (possibly) long-running API review phase you need different versions of the API description. These versions may include changes that are incompatible with earlier draft versions. So we apply the following version schema MAJOR.MINOR.DRAFT that increments the...

- MAJOR version, when you make incompatible API changes
- MINOR version, when you add functionality in a backwards-compatible manner
- DRAFT version, when you make changes during the review phase that are not related to production releases

We recommend using the DRAFT version only for unreleased API definitions that are still under review; for example:

```
version 1.4.0  -- current version
version 1.4.1  -- first draft and call for review of API extensions compatible with 1.4.0
version 1.4.2  -- second draft and call for review of API extensions that are still compatible
                  with 1.4.0 but possibly incompatible with 1.4.1
version 1.5.0  -- approved version for implementation and release
version 1.5.1  -- first draft for next review and API change cycle;
                  compatible with 1.4.0 and 1.5.0
```

Hint: This versioning scheme differs in the less strict DRAFT aspect from semantic version information used for released APIs and service applications.

JSON Guidelines

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

Property Naming

Must: Property names must be snake_case (and never camelCase).

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.

Must: Property names must be an ASCII subset

Property names are restricted to ASCII encoded strings. The first character must be a letter, an underscore or a dollar sign, and subsequent characters can be a letter, an underscore, a dollar sign, or a number.

Should: Reserved JavaScript keywords should be avoided

Most API content is consumed by non-JavaScript clients today, but for security and sanity reasons, JavaScript (strictly, ECMAScript) keywords are worth avoiding. A list of keywords can be found in the ECMAScript Language Specification.

Should: Array names should be pluralized

To indicate they contain multiple values prefer to pluralize array names. This implies that object names should in turn be singular.

Property Values

Must: Boolean property values must not be null

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.

Should: Null values should have their fields removed

Swagger/OpenAPI, which is in common use, doesn't support null field values (it does allow omitting that field completely if it is not marked as required). However that doesn't prevent clients and servers sending and receiving those fields with null values. Also, in some cases null may be a meaningful value - for example, JSON Merge Patch RFC 7382) using null to indicate property deletion.

Should: Empty array values should not be null

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

Should: Enumerations should be represented as Strings

Strings are a reasonable target for values that are by design enumerations.

Should: Date property values should conform to RFC 3399

Use the date and time formats defined by RFC 3339:

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

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

A zone offset may be used (both, in request and responses) -- this is simply defined by the standards. However, we encourage restricting dates to UTC and without offsets. For example `2015-05-28T14:07:17Z` rather than `2015-05-28T14:07:17+00:00`. From experience we have learned that zone offsets are not easy to understand and often not correctly handled. Note also that zone offsets are different from local times that might be including daylight saving time. Localization of dates should be done by the services that provide user interfaces, if required.

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

Sometimes it can seem data is naturally represented using numerical timestamps, but this can introduce interpretation issues with precision - for example 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.

Could: Time durations and intervals could conform to ISO 8601

Schema based JSON properties that are by design durations and intervals could be strings formatted as recommended by ISO 8601 (Appendix A of RFC 3399 contains a grammar for durations).

Could: Standards could be used for Language, Country and Currency

- ISO 3166-1-alpha2 country
  - (It's "GB", not "UK", even though "UK" has seen some use at Zalando)
- ISO 639-1 language code
  - BCP-47 (based on ISO 639-1) for language variants
- ISO 4217 currency codes

API Naming

Must: Path segments must be lowercase separate words with hyphens

Example:

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

This applies to concrete path segments and not the names of path parameters. For example `{shipment_order_id}` would be ok as a path parameter.

Must: Query parameters must be snake_case (never camelCase)

Examples:

```
customer_number, order_id, billing_address
```

Must: You Must Hyphenate HTTP Headers

Should: Prefer Hyphenated-Pascal-Case for HTTP header Fields

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

Examples:

```
Accept-Encoding
Apply-To-Redirect-Ref
Disposition-Notification-Options
Original-Message-ID
```

See also: HTTP Headers are case-insensitive (RFC 7230).

Could: Use Standardized Headers

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

Must: Always Pluralize Resource Names

Usually, a collection of resource instances is provided (at least API should be ready here). The special case of a resource singleton is a collection with cardinality 1.

Could: First Path segment May be /api

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 — add “/api” as base path to clearly separate public and non-public API resources.

Must: Avoid Trailing Slashes

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

Could: Use Conventional Query Strings

If you provide query support for sorting, pagination, filtering functions or other actions, use the following standardized naming conventions:

- `q` — default query parameter (e.g. used by browser tab completion); should have an entity specific alias, like sku
- `limit` — to restrict the number of entries. See Pagination section below. Hint: You can use size as an alternate query string.
- `cursor` — key-based page start. See Pagination section below.
- `offset` — numeric offset page start. See Pagination section below. Hint: In combination with limit, you can use page as an alternative to offset.
- `sort` — comma-separated list of fields to sort. To indicate sorting direction, fields my prefixed with + (ascending) or - (descending, default), e.g. /sales-orders?sort=+id
- `fields` — to retrieve a subset of fields. See *Support Filtering of Resource Fields* below.
- `embed` — to expand embedded entities (ie.: inside of an article entity, expand silhouette code into the silhouette object). Implementing “expand” correctly is difficult, so do it with care. See *Embedding resources* for more details.

Resources

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: 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.

Must: Use Domain-Specific Resource Names

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

Must: Identify resources and Sub-Resources via Path Segments

Basic URL structure:

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

Examples:

```
/carts/1681e6b88ec1/items
/carts/1681e6b88ec1/items/1
```

Could: Consider Using (Non-) Nested URLs

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

```
/carts/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 is a collection for sales orders; however, sales orders have globally unique id and some services may choose to access the orders directly, for instance:

```
/customer/1681e6b88ec1
/sales-order/5273gh3k525a
```

Should: Limit of Resources

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 microservice API definition. In this sense the number of resources exposed via API should be limited - our experience is that a typical range for a functional focussed good API is some 4-8 different resources. 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.

Should: Limit 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)

HTTP

Must: Use HTTP Methods Correctly

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

GET

- reads a resource or set of resource instances, respectively
- individual resources will usually generate a 404 if the resource does not exist; collection resources may return either 200 or 404 if the listing is empty
- must NOT have request body payload

PUT:

- fully uploads an entity, i.e. provides a complete replacement by the resource representation passed as payload
- resource instance IDs are maintained by the client (either generated by the client or by the server in a previous POST request) and passed as a URL path segment
- PUT operations are usually only accepted by single resources, not collection resources, as PUT on a collection would imply replacing the entire collection
- usually robust against non-existence of the entity by implicit creation before update

PATCH:

- partial upload, i.e. only a specific subset of resource fields are replaced
- partial resource representation passed as payload has either resource content type with optional fields (to be updated) or a custom content type that also may include instructions of how to change the resource
- usually accepted only by single resources, because the semantics for PATCH on a collection resource are very hard to define
- usually not robust against non existence of the entity

DELETE:

- deletes a resource instance
- DELETE operations are usually only accepted by single resources, not collection resources, as DELETE on a collection would imply deleting the entire collection
- should return either status 404 (Not found) or 410 (Gone) if the resource does not exist

POST:

- creates a resource instance
- resource instance id(s) are created and maintained by server and returned with the output payload
- POST methods should be accepted by collection resources only
- more generally, POST should be used for scenarios that cannot be covered by the other methods. For instance, GET with complex (e.g. SQL like structured) query that needs to be passed as request body payload. In such cases, make sure to document the fact that POST is used as a workaround

HEAD

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

OPTIONS

- returns the available operations (methods) on a given endpoint (usually either as a comma separated list of methods or as a structured list of link templates)
- this operation is rarely implemented, though it could be used to self-describe the full functionality of a resource.

Must: HTTP Methods must Fulfill Safeness and Idempotency Properties

An operation can be...

- idempotent, i.e. operation will produce the same results if executed once or multiple times (note: this does not necessarily mean returning the same status code)
- safe, i.e. must not have side effects such as state changes

Method implementations must fulfill the following basic properties:

HTTP method	safe	idempotent
OPTIONS	Yes	Yes
HEAD	Yes	Yes
GET	Yes	Yes
PUT	No	Yes
POST	No	No
DELETE	No	Yes
PATCH	No	No

Please see also Best Practices [internal link] for further hints on how to support the different HTTP methods on resources.

Must: Use Meaningful HTTP Status Codes

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 Content-Location header. (More details found in the Common Headers section.) *Always* set the Location header.	POST, PUT
202	Accepted - The request was successful and will be processed asynchronously.	POST, PUT, DELETE, PATCH
204	No content - There is no response body	PUT, DELETE

Redirection Codes:

Code	Meaning	Methods
301	Moved Permamently - 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.	PATCH, POST, PUT, DELETE
304	Not Modified - resource has not been modified since the date or version passed via request headers If-Modified-Since or If-None-Match.	GET

Client Side Error Codes:

Code	Meaning	Methods
400	Bad request - generic / unknown error	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 - returned if, e.g. when two clients try to create the same resource or if there are concurrent, conflicting updates	PUT, DELETE, PATCH
412	Precondition Failed - returned for conditional requests, e.g. If-Match if the condition failed. Used for optimistic locking.	PUT, DELETE, PATCH
415	Unsupported Media Type - e.g. clients sends request body without content type	PUT, DELETE, PATCH
422	Unprocessable Entity - semantic error (as opposed to a syntax error which would usually trigger a 400)	POST, PUT, DELETE, PATCH
423	Locked - Pessimistic locking, e.g. processing states	PUT, DELETE, PATCH
428	Precondition Required - server requires the request to be conditional (e.g. to make sure that the “lost update problem” is avoided).	All
429	Too many requests - the client does not consider rate limiting and sent too many requests. See "Use 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 senseful)	All
501	Not Implemented - server cannot fulfill the request (usually implies future availability, e.g. new feature).	All
503	Service Unavailable - server is (temporarily) not available (e.g. due to overload) -- client retry may be senseful.	All

All error codes you can find in RFC7231 and Wikipedia or via https://httpstatuses.com/.

Must: Providing Error Documentation

APIs should define the functional, business view and abstract from implementation aspects. Errors become a key element providing context and visibility into how to use an API. The error object should be extended by an application-specific error identifier since the HTTP status code often is not specific enough to articulate the domain error situation. For this reason, we use a standardized error return object definition — see *Use Common Error Return Objects*.

The OpenAPI specification shall include definitions for error descriptions that will be returned; they are part of the interface definition and provide important information for service clients to handle exceptional situations and support troubleshooting. You should also think about a troubleshooting board — it is part of the associated online API documentation, provides information and handling guidance on application-specific errors and is referenced via links of the API definition. This can reduce service support tasks and contribute to service client and provider performance.

Must: Use 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 and therefore the request can't be fulfilled. 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.

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.

Performance

Should: Reducing Bandwidth Needs and Improving 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. (Zalando is a 'Mobile First' company, so be mindful of this point.)

Common techniques include:

- gzip compression
- querying field filters to retrieve a subset of resource attributes (see *Support Filtering of Resource Fields* below)
- paginate lists of data items (see *Pagination* below)
- `ETag` (and If-[None-]Match) headers to avoid refetch of unchanged resources
- pagination for incremental access of larger (result) lists

Each of these items is described in greater detail below.

Should: 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 Filtering of Resource Fields

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 determine the subset of fields he wants to receive via the fields query parameter — example see Google AppEngine API's partial response:

Unfiltered

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

HTTP/1.1 200 OK
Content-Type: application/x.person+json

{
  "id": "cddd5e44-dae0-11e5-8c01-63ed66ab2da5",
  "name": "John Doe",
  "address": "1600 Pennsylvania Avenue Northwest, Washington, DC, United States",
  "birthday": "1984-09-13",
  "partner": {
    "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/resources/123?fields=(name,partner(name)) HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/x.person+json;fields=(name,partner(name))

{
  "name": "John Doe",
  "partner": {
    "name": "Jane Doe"
  }
}
```

The approach we recommend for field filtering is a Zalando Github project, json-fields. It defines a formal grammar for the ANTLR parser generator and provides a ready-to use library for Java / Jackson based projects (Maven link). Teams that use other JSON serializers are encouraged to contribute to the open source project and create their own parser / framework based on this grammar.

Other approaches we have considered are JSONPath or GraphQL. While they have advantages, neither of them can easily be plugged into an existing serialization process, so they require an additional, manual serialization process, whereas the above solution addresses our main filter use cases and can easily be introduced with a minimum of effort.

Hint: OpenAPI doesn't allow you to formally specify whether depending on a given parameter will return different parts of the specified result schema. Explain this in English in the parameter description.

Could: Support the ETag Header

If a resource changes, the contents of the `ETag` header must also change. Combined with the `If-Match` and `If-None-Match` headers, the `ETag` header allows caching of entities.

Its possible contents:

- the entity’s version number
- hash of the response body
- hash of the entity’s last modified field

Also see *Allow Embedding of Complex Sub-Resources* below as contribution to performance optimization as to avoid multiple requests for sub-resources.

Pagination

Must: Support Pagination

Access to lists of data items must support pagination for best client side batch processing and iteration experience. This holds true for all lists that are (potentially) larger than just a few hundred entries.

There are two page iteration techniques:

- Offset/Limit-based pagination: numeric offset identifies the first page entry
- Cursor-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/previous page links. This favours cursor-based over offset-based pagination.

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 / limit based pagination is more 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
- Variability of data 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.
  - When using cursor-based pagination, paging cannot continue when the cursor entry has been deleted while fetching two pages
- Performance considerations - efficient server-side processing using offset-based pagination is hardly feasible for:
  - Higher data list volumes, especially if they do not reside in the database's main memory
  - Sharded or NoSQL databases
- Cursor-based navigation may not work if you need the total count of results and / or backward iteration support

Further reading:

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

Could: Use Pagination Headers Where Applicable

- Set `X-Total-Count` to send back the total count of entities.
- Set the link headers to provide information to the client about subsequent paging options.

For example:

```
Link: <http://catalog-service.zalando.net/articles?cursor=62726863268328&limit=100>;
      rel="next"; title="next chunk of articles"
```

or

```
Link: <http://catalog-service.zalando.net/articles?offset=5&limit=100>;
      rel="prev"; title="previous chunk of articles"
Link: <http://catalog-service.zalando.net/articles?offset=205&limit=100>;
      rel="next"; title="next chunk of articles"
```

Possible relations: `next` , `prev` , `last` , `first`

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.:

- Avoid Actions — Think About Resources
- Keep URLs Verb-Free
- Use HTTP Methods Correctly
- Use Meaningful 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.

Could: Use HATEOAS

Although we prefer REST Maturity Level 2, we do not forbid implementing Level 3, which is HATEOAS (Hypertext As the Engine Of Application State). But you should be aware of the shortcomings in our setup.

Because we are following API First principles, HATEOAS brings nothing new to the table in terms of API self-descriptiveness. Furthermore, generic HATEOAS clients which crawl and use APIs on their own are only a theoretical concept so far. Our whole internal tooling around APIs like Twintip isn't adjusted for HATEOAS neither. For now, we have not seen a good reason to implement HATEOAS in an API.

There are several other concerns regarding the promised advantages of HATEOAS (see RESTistential Crisis over Hypermedia APIs):

- Hypermedia does not prevent clients from required manual changes when domain model changes over time
- Hypermedia makes sense for humans, not machines
- Hypermedia does not prevent API clients to implement shortcuts and directly target resources without 'discovering' them

If you use HATEOAS please present your findings in the API Guild [internal link].

Could: Use URIs for Custom Link Relations

Related or even embedded (sub-) resources should be linked via “meta link attributes” within the response payload; use here the following HAL compliant syntax:

```
{
  ...
  “_links”: {
    “https://docs.team.zalan.do/rels/my-entity”: [{
      “href”: “https://service.team.zalan.do/my-entities/123”
    }]
  }
}
```

Or the JSON API compliant one where you also embed the entities directly:

```
{
  ...
  “relationships”: {
    "my-entities": {
      "data": [
        {"id": 1, "type": "my-entity"}
      ],
      "links": {
        "related": "/my-entities/123"
      }
    }
  }
}
```

Should: Use Standards for Hypermedia Link Types, HAL or JSON API recommended

To represent hypermedia links in payload results, we should consider using a standard format like HAL, JSON API, JSON-LD with Hydra, or Siren. (Hint: Read Kevin Sookocheff’s post, for instance, to learn more about the hypermedia types.)

We are in an ongoing discussion in the API guild which standard should be recommended for a certain use case. In the meantime we recommend to use HAL or JSON API.

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 *Conventional Query Strings* for naming.

Must: Modify the Content-Type for Embedded Resources

Embedded resources requires to change the media type of the response accordingly:

```
GET /order/123?embed=(items)
Accept: application/x.order+json
```

```
HTTP/1.1 200 OK
Content-Type: application/x.order+json;embed=(items)
```

Data Formats

Must: Use JSON as the Body Payload

JSON-encode the body payload. The JSON payload must follow RFC-7159 by having (if possible) a serialized object or array as the top-level object.

Must: Use Standard Date and Time Formats

JSON Payload

Read more about date and time format in Json Guideline.

HTTP headers

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

Could: Use Standards for Country, Language and Currency Codes

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

- ISO 3166-1-alpha2 country codes
  - (It is “GB”, not “UK”, even though “UK” has seen some use at Zalando)
- ISO 639-1 language code
  - BCP-47 (based on ISO 639-1) for language variants
- ISO 4217 currency codes

Could: Use Application-Specific Content Types

For instance, `application/x.zalando.article+json`. For complex types, it’s better to have a specific content type. For simple use cases this isn’t necessary. We can attach version info to media type names and support content negotiation to get different representations, e.g. `application/x.zalando.article+json;version=2`.

Common Data Objects

Definitions of data objects that are good candidates for wider usage:

Should: Use a Common Money Object

Use the following common money structure:

```
Money:
  type: object
  properties:
    amount:
      type: number
      format: decimal
      example: 99.95
    currency:
      type: string
      format: iso-4217
      example: EUR
  required:
    - amount
    - currency
```

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.

Should: Use Common Address Fields

Address structures play a role in different functional and use-case contexts, including country variances. The address structure below should be sufficient for most of our business-related use cases. Use it in your APIs — and compatible extend it if necessary for your API concerns:

```
address:
    description:
      a common address structure adequate for many use cases
    type: object
    properties:
      salutation:
        type: string
        description: |
          A salutation and/or title which may be used for personal contacts. Hint: not to be confused with the gender i
        example: Mr
      first_name:
        type: string
        description: given name(s) or first name(s) of a person; may also include the middle names
        example: Hans Dieter
      last_name:
        type: string
        description: family name(s) or surname(s) of a person
        example: Mustermann
      business_name:
        type: string
        description: company name of the business organization
        example: Consulting Services GmbH
      street:
        type: string
        description: full street address including house number and street name
        example: Schönhauser Allee 103
      additional:
        type: string
        description: further details like suite, apartment number, etc.
        example: 2. Hinterhof rechts
      city:
        type: string
        description: name of the city
        example: Berlin
      zip:
        type: string
        description: Zip code or postal code
        example: 14265
      country_code:
        type: string
        format: iso-3166-1-alpha-2
        example: DE
    required:
      - first_name
      - last_name
      - street
      - city
      - zip
      - country_code
```

Must: Use Common Error Return Objects

RFC 7807 defines the media type `application/problem+json` for a common error return object (and the format of the contained JSON object). Operations should return that (together with a suitable status code) when any problem occurred during processing and you can give more details than the status code itself can supply, whether it be caused by the client or the server (i.e. both for 4xx or 5xx errors).

A previous version of this guideline (before the publication of that RFC and the registration of the media type) told to return `application/x.problem+json` in these cases (with the same contents). Servers for APIs defined before this change should pay attention to the `Accept` header sent by the client and set the `Content-Type` header of the problem response correspondingly. Clients of such APIs should accept both media types.

APIs may define custom problems types with extension properties, according to their specific needs.

Here is the definition in Open API 2.0 YAML format in the case you don't need any extensions:

```
Problem:
 type: object
 properties:
   type:
     type: string
     format: uri
     description: |
       An absolute URI that identifies the problem type.  When dereferenced,
       it SHOULD provide human-readable documentation for the problem type
       (e.g., using HTML).
     example: http://httpstatus.es/503
   title:
     type: string
     description: |
       A short, summary of the problem type. Written in english and readable
       for engineers (usually not suited for non technical stakeholders and
       not localized); example: Service Unavailable
   status:
     type: integer
     format: int32
     description: |
       The HTTP status code generated by the origin server for this occurrence
       of the problem.
     example: 503
   detail:
     type: string
     description: |
       A human readable explanation specific to this occurrence of the
       problem.
     example: Connection to database timed out
   instance:
     type: string
     format: uri
     description: |
       An absolute URI that identifies the specific occurrence of the problem.
       It may or may not yield further information if dereferenced.
 required:
   - type
   - title
   - status
```

Must: An error message must not contain the stack trace.

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.

Common Headers

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

Content-Location

This header is used in the response of either a successful read (GET, HEAD) or successful write operation (PUT, POST or PATCH).

In the case of the GET requests it points to a location where an alternate representation of the entity in the response body can be found. In this case one has to set the Content-Type header 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
```

In the case of mutating HTTP methods, the Content-Location header can be used when there is a response body, and then it indicates that the included response body can be found at the location indicated in the header.

If the header value is the same as the location of the created resource (as indicated by the Location header after POST) or the modified resource (as indicated by the request URI after PUT / PATCH), then the returned body is indeed the current representation of the entity making a subsequent GET operation from the client side not necessary.

If your API returns the new representation after a PUT, PATCH, or POST you should include the Content-Location header to make it explicit, that the returned resource is an up-to-date version.

More details in rfc7231

Could: Use the Prefer header to indicate processing preferences

The `Prefer` header defined in RFC7240 allows clients to request processing behaviors from servers. RFC7240 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:

```
Prefer:
  name: Prefer
  description: |
    The RFC7240 Prefer header indicates that particular server
    behaviors are preferred by the client but are not required
    for successful completion of the request.

    # (indicate the preferences supported by the API)

  in: header
  type: string
  required: false
```

Supporting APIs may return the `Preference-Applied` header also defined in RFC7240 to indicate whether the preference was applied.

Proprietary Headers

This section shares definitions of proprietary headers that should be named consistently because they address overarching service-related concerns. 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.

Header field name	Type	Description	Header field value example
X-Flow-ID	String	The flow id of the request, which is written into the logs and passed to called services. Helpful for operational troubleshooting and log analysis.	GKY7oDhpSiKY_gAAAABZ_A
X-UID	String	Generic user id of OpenId account that owns the passed (OAuth2) access token. E.g. additionally provided by OpenIG proxy after access token validation -- may save additional token validation round trips.	w435-dker-jdh357
X-Tenant-ID	String	The tenant id for future platform multitenancy support. *Should not be used unless new platform multitenancy is truly supported. But should be used by New Platform Prototyping services.* Must be validated for external retailer, supplier, etc. tenant users via OAuth2; details in clarification. Currently only used by New Platform Prototyping services.	9f8b3ca3-4be5-436c-a847-9cd55460c495
X-Sales-Channel	String	Sales channels are owned by retailers and represent a specific consumer segment being addressed with a specific product assortment that is offered via CFA retailer catalogs to consumers (see platform glossary [internal link])	101
X-Frontend-Type	String	Consumer facing applications (CFAs) provide business experience to their customers via different frontend application types, for instance, mobile app or browser. Info should be passed-through as generic aspect -- there are diverse concerns, e.g. pushing mobiles with specific coupons, that make use of it. Current range is mobile-app, browser, facebook-app, chat-app	mobile-app
X-Device-Type	String	There are also use cases for steering customer experience (incl. features and content) depending on device type. Via this header info should be passed-through as generic aspect. Current range is smartphone, tablet, desktop, other	tablet
X-Device-OS	String	On top of device type above, we even want to differ between device platform, e.g. smartphone Android vs. iOS. Via this header info should be passed-through as generic aspect. Current range is iOS, Android, Windows, Linux, MacOS	Android
X-App-Domain	Integer	The app domain (i.e. shop channel context) of the request. Note, app-domain is a legacy concept that will be replaced in new platform by combinations of main CFA concerns like retailer, sales channel, country	16

Remember that HTTP header field names are not case-sensitive.

API Discovery

Must: Applications Must Provide Online Access to Their API (Swagger) Definitions

In our dynamic and complex service infrastructure, it is important to provide a central place with online access to the API definitions of all running applications. All service applications must provide the following two API endpoints:

- endpoint(s) for GET access on its API OpenAPI definition(s), for instance `https://example.com/swagger.json` or `https://example.com/swagger.yaml` .
- “Twintip” discovery endpoint `https://example.com/.well-known/schema-discovery` that delivers the OpenAPI definition endpoint(s) above (see the link below for a description of its format).

Note, these discovery endpoints have to be supported but need not be part of the OpenAPI definition as there is no API specific information in their description.

Background: Twintip is an API definition crawler of the STUPS infrastructure; it checks all running applications via the endpoint above and stores the discovered API definitions. Twintip itself provides a RESTful API as well as an API Viewer (Swagger-UI) for central access to all discovered API definitions.

For the time being, this document is an appropriate place to mention this rule, even though it is not a RESTful API definition rule or related to our STUPS infrastructure for application service management.

Events

Zalando's architecture centers around decoupled microservices and in that context we favour asynchronous event driven approaches. The Nakadi framework provides an event publish and subscribe system, abstracting exchanges through a RESTful API.

Nakadi is a key element for data integration in the architecture, and the standard system for inter-service event messaging. The guidelines in this section are for use with the Nakadi framework and focus on how to design and publish events intended to be shared for others to consume. Critically, once events pass service boundaries they are considered part of the service API and are subject to the API guidelines.

Must: Treat Events as part of the service interface

Events are part of a service's interface to the outside world equivalent in standing to a service's REST API. Services publishing data for integration must treat their events as a first class design concern, just as they would an API. For example this means approaching events with the "API first" principle in mind as described in the Introduction and making them available for review.

Must: Events should define useful business resources

Events are intended to be used by other services including business process/data analytics and monitoring. They should be based around the resources and business processes you have defined for your service domain and adhere to its natural lifecycle (see also "Should: Define useful resources" in the General Guidelines).

As there is a cost in creating an explosion of event types and topics, prefer to define event types that are abstract/generic enough to be valuable for multiple use cases, and avoid publishing event types without a clear need.

Must: Event Types must conform to Nakadi's API

Nakadi defines a structure called an *EventType*, which describes details for a particular kind of event. The EventType declares standard information, such as a name, an owning application (and by implication, an owning team), a well known event category (business process or data change), and a schema defining the event payload. It also allows the declaration of validation and enrichment strategies for events, along with supplemental information such as how events are partitioned in the stream.

An EventType is registered with Nakadi via its *Schema Registry API*. Once the EventType is created, individual events that conform to the type and its payload schema can be published, and consumers can access them as stream of Events. Nakadi's Open API definitions include the definitions for two main categories, business events (BusinessEvent), and data change events (DataChangeEvent), as well as a generic 'undefined' type.

The service specific data defined for an event is called the *payload*. Only this non-Nakadi defined part of the event is expected to be submitted with the EventType schema. When defining an EventType's payload schema as part of your API and for review purposes, it's ok to declare the payload as an object definition using Open API, but please note that Nakadi currently expects the EventType's schema to be submitted as JSON Schema and not as Open API JSON or YAML. Further details on how to register EventTypes are available in the Nakadi project's documentation.

Must: Use the Business Events to signal steps and arrival points in business processes

Nakadi defines a specific event type for business processes, called BusinessEvent. When publishing events that represent steps in a business process, event types must be based on the BusinessEvent type.

All your events of a single business process will conform to the following rules:

- Business events must contain a specific identifier field (a business process id or "bp-id") similar to flow-id to allow for efficient aggregation of all events in a business process execution.
- Business events must contain a means to correctly order events in a business process execution. In distributed settings where monotically increasing values (such as a high precision timestamp that is assured to move forwards) cannot be obtained, Nakadi provides a `parent_eids` data structure that allows causal relationships to be declared between events.
- Business events should only contain information that is new to the business process execution at the specific step/arrival point.
- Each business process sequence should be started by a business event containing all relevant context information.
- Business events must be published reliably by the service.

At the moment we cannot state whether it's best practice to publish all the events for a business process using a single event type and represent the specific steps with a state field, or whether to use multiple event types to represent each step. For now we suggest assessing each option and sticking to one for a given business process.

Must: Use the Data Change Event structure to signal mutations

Nakadi defines an event for signalling data changes, called a DataChangeEvent. When publishing events that represents created, updated, or deleted data, change event types must be based on the DataChangeEvent category.

- Change events must identify the changed entity to allow aggregation of all related events for the entity.
- Change events should contain a means of ordering events for a given entity (such as created or updated timestamps that are assured to move forwards with respect to the entity, or version identifiers provided by some databases). Note that basing events on data structures that can be converged upon (such as CRDTs or logical clocks) in a distributed setting are outside the scope of this guidance.
- Change events must be published reliably by the service.

Should: Use the hash partition strategy for Data Change Events

The `hash` partition strategy allows a producer to define which fields in an event are used as input to compute the partition the event should be added to.

The `hash` option is particulary useful for data changes as it allows all related events for an entity to be consistently assigned to a partition, providing an ordered stream of events for that entity as they arrive at Nakadi. This is because while each partition has a total ordering, ordering across partitions is not assured, thus it is possible for events sent across partitions to appear in a different order to consumers that the order they arrived at the server. Note that as of the time of writing, random is the default option in Nakadi and thus the `hash` option must be declared when creating the event type.

There may be exceptional cases where data change events could have their partition strategy set to be the producer defined or random options, but generally `hash` is the right option - that is while the guidelines here are a "should", they can be read as "must, unless you have a very good reason".

Should: Data Change Events should match API representations

A data change event's representation of an entity should correspond to the REST API representation.

There's value in having the fewest number of published structures for a service. Consumers of the service will be working with fewer representations, and the service owners will have less API surface to maintain. In particular, you should only publish events that are interesting in the domain and abstract away from implementation or local details - there's no need to reflect every change that happens within your system.

There are cases where it could make sense to define data change events that don't directly correspond to your API resource representations. Some examples are -

- Where the API resource representations are very different from the datastore representation, but the physical data are easier to reliably process for data integration.
- Publishing aggregated data. For example a data change to an individual entity might cause an event to be published that contains a coarser representation than that defined for an API
- Events that are the result of a computation, such as a matching algorithm, or the generation of enriched data, and which might not be stored as entity by the service.

Must: Event Types must indicate ownership

Event definitions must have clear ownership - this can be indicated via the `owning_application` field of the EventType.

Typically there is one producer application, which owns the EventType and is responsible for its definition, akin to how RESTful API definitions are managed. However, the owner may also be a particular service from a set of multiple services that are producing the same kind of event. Please note indicating that a specific application is producing or consuming a specific event type is not yet defined explicitly, but a subject of Nakadi service discovery support functions.

Must: Event Payloads must be defined in accordance with the overall Guidelines

Events must be consistent with other API data and the API Guidelines in general.

Everything expressed in the Introduction to these Guidelines is applicable to event data interchange between services. This is because our events, just like our APIs, represent a commitment to express what our systems do and designing high-quality, useful events allows us to develop new and interesting products and services.

What distinguishes events from other kinds of data is the delivery style used, asynchronous publish-subscribe messaging. But there is no reason why they could not be made available using a REST API, for example via a search request or as a paginated feed, and it will be common to base events on the models created for the service's REST API.

The following existing guidelines for API data are applicable to events -

General Guidelines -

- Must: API First: Define APIs using OpenAPI YAML or JSON
- Must: Write in U.S. English

Naming -

- Must: JSON field names must be snake_case (never camelCase)

Data Formats -

- Must: Use JSON as the Body Payload
- Must: Use Standard Date and Time Formats
- Could: Use Standards for Country, Language and Currency Codes
- Could: Use Application-Specific Content Types

Common Data Objects -

- Should: Use a Common Money Object
- Should: Use Common Address Fields

Hypermedia -

- Should: Use HATEOAS. The use of hypermedia techniques are equally applicable to events.
- Could: Use URIs for Custom Link Relations
- Should: Consider Using a Standard for linked/embedded resources.

There are a couple of technical considerations to bear in mind when defining event schema -

Note that Nakadi currently requires Open API event definitions to be submitted in JSON Schema syntax, and does not yet support Open API YAML. It's ok to define your events for peer review in YAML to avoid maintaining duplicate representations. This is best understood as point in time guidance - as the project develops it may support consuming Open API YAML (or other) structures directly, and the guidance here will be updated.

Must: Maintain backwards compatibility for Events

Changes to events must be based around making additive and backward compatible changes. This follows the guideline, "Must: Don’t Break Backward Compatibility" from the Compatibility guidelines.

In the context of events, compatibility issues are complicated by the fact that producers and consumers of events are highly asynchronous and can’t use content-negotiation techniques that are available to REST style clients and servers. This places a higher bar on producers to maintain compatibility as they will not be in a position to serve versioned media types on demand.

For event schema, these are considered backward compatible changes, as seen by consumers -

- Adding new optional fields to JSON objects.
- Changing the order of fields (field order in objects is arbitrary).
- Changing the order of values with same type in an array.
- Removing optional fields.
- Removing an individual value from an enumeration.

These are considered backwards-incompatible changes, as seen by consumers -

- Removing required fields from JSON objects.
- Changing the default value of a field.
- Changing the type of a field, object, enum or array.
- Changing the order of values with different type in an array (also known as a tuple).
- Adding a new optional field to redefine the meaning of an existing field (also known as a co-occurrence constraint).
- Adding a value to an enumeration (note that `x-extensible-enum` is not available in JSON Schema).

Must: Event identifiers must be unique

The `eid` (event identifier) value of an event must be unique.

The `eid` property is part of the standard metadata for an event and gives the event an identifier. Producing clients must generate this value when sending an event and it must be guaranteed to be unique from the perspective of the owning application. In particular events within a given event type's stream must have unique identifiers. This allows consumers to process the `eid` to assert the event is unique and use it as an idempotency check.

Note that uniqueness checking of the `eid` is not enforced by Nakadi at the event type or global levels and it is the responsibility of the producer to ensure event identifiers do in fact distinctly identify events. A straightforward way to create a unique identifier for an event is to generate a UUID value.

Must: Event Type names must follow conventions

Event types can follow these naming conventions (each convention has its own should, must or could conformance level) -

- Event type names must be url-safe. This is because the event type names are used by Nakadi as part of the URL for the event type and its stream.
- Event type names should be lowercase words and numbers, using hypens, underscores or periods as separators.

References

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

OpenAPI Specification

- OpenAPI Specification

Publications, specifications and standards

- RFC 3339: Date and Time on the Internet: Timestamps
- RFC 5988: Web Linking
- 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 7807: Problem Details for HTTP APIs
- 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

Tooling

API First Integrations

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

- Connexion: OpenAPI First framework for Python on top of Flask
- Friboo: utility library to write microservices in Clojure with support for Swagger and OAuth
- Play Swagger: build RESTful Play services from OpenAPI specification
- Swagger Codegen: template-driven engine to generate client code in different languages by parsing Swagger Resource Declaration
- Swagger Codegen Tooling: plugin for Maven that generates pieces of code from OpenAPI specification
- Swagger Plugin for IntelliJ IDEA: plugin to help you easily edit Swagger specification files inside IntelliJ IDEA

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):

- Problem: Java library that implements application/problem+json
- Problems for Spring Web MVC: library for handling Problems in Spring Web MVC
- Jackson Datatype Money: extension module to properly support datatypes of javax.money
- JSON fields: framework for limiting fields of JSON objects exposed by Rest APIs
- Tracer: call tracing and log correlation in distributed systems
- TWINTIP: API definition crawler for the STUPS ecosystem
- TWINTIP Spring Integration: API discovery endpoint for Spring Web MVC