Hypatos RESTful API and Event Guidelines

Hypatos

Other formats: PDF, EPUB3

Table of Contents

1. Introduction

Hypatos'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 into our Kubernetes cluster. These APIs are exposed for internal and external use. However, our strategy emphasizes developing lots of public APIs

for our external business partners to use via third-party applications. In order to maximize the adoption of our public API, we pay a high attention to developer experience (DX) while designing the APIs.

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:

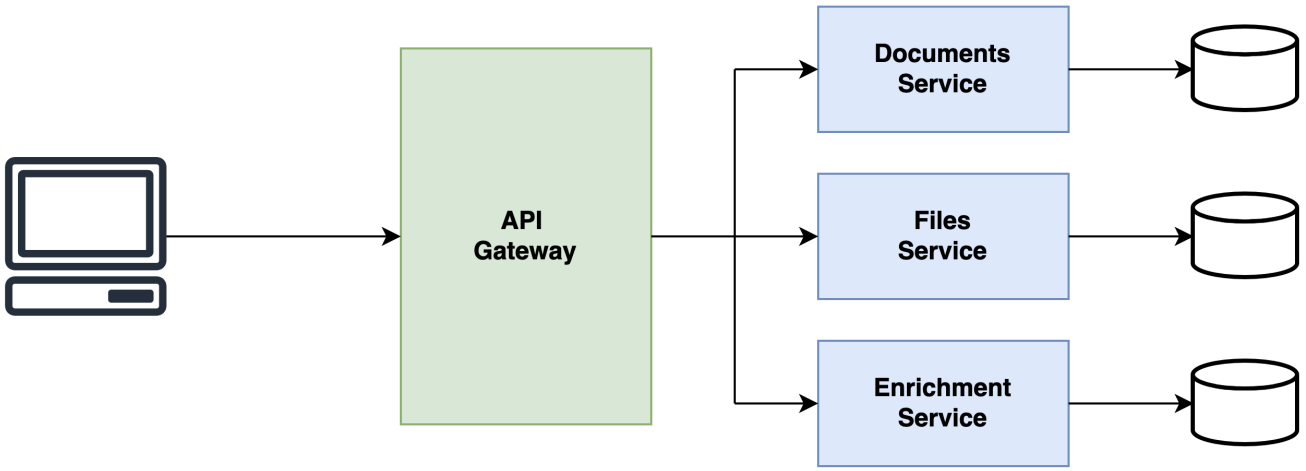
- are easy to understand and learn
- are general and abstracted from specific implementation and use cases
- are robust and easy to use
- have a common look and feel
- follow a consistent RESTful style and syntax
- are consistent with other teams' APIs and our global architecture

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

API audience

As mentioned above our APIs are exposed for internal and external clients (see more details on API audience. Internal clients of a microservice are other microservices, internal utility tools or automation processes. Even though these clients are internal, they must respect the data ownership principle and retrieve the data using official APIs. Retrieving data from the databases from our services is forbidden.

While the APIs of individual microservices are available to our internal clients, they are not exposed to the internet. These APIs are exposed to the clients via an API gateway which acts as a single entry point.



The API gateway insulates clients from internal details about how Hypatos architecture is partitioned into microservices by exposing a single "monolytic" Hypatos REST API. Our microservices may decide which of their endpoints are exposed for public access which are kept internal. This REST API can be accessed under the following base URL.

https://api.cloud.hypatos.ai/v2

Any API exposed to the internet must be exposed via this base URL.

Conventions used in these guidelines

The requirement level keywords used in this document (case insensitive) are to be interpreted as described in RFC 2119. We are using 3 keywords only. Here is a brief explanation of their meaning.

- **MUST** - A guideline is an absolute requirement that must not be omitted
- **SHOULD** - A guideline is recommended to be followed. If valid and objective reasons in particular circumstances exist, the particular guideline may be ignored
- **MAY** - A guideline is truly optional. It can be ignored without any reason.

Hypatos specific information

The purpose of our "RESTful API guidelines" is to define standards to successfully establish "consistent API look and feel" quality. This document was drafted by Zalando and adopted by the Hypatos API Governance guild. Our teams are responsible to fulfill these guidelines during API development and are encouraged to contribute to guideline evolution via pull requests.

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

In case guidelines are changing, following rules apply:

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

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

2. Principles

API design principles

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

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

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

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

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

- Article: REST API Design - Resource Modeling
- Article: Richardson Maturity Model — Steps toward the glory of REST
- Book: 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

At Hypatos we consider APIs to be more than just an application programming interface. Our API objectives relate directly to our business objectives and help us to achieve our business goals. Hence, the design of our APIs should be based on the API as a Product principle:

- Treat your API as product and act like a product owner
- Put yourself into the place of your customers; be an advocate for their needs

- Emphasize simplicity, comprehensibility, and usability of APIs to make them irresistible for client engineers
- Actively improve and maintain API consistency over the long term
- Make use of customer feedback and provide service level support

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

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

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

API first

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

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

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

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

Moreover, API definitions with standardized specification format also facilitate…

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

checks

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

It is important to learn, that API First is **not in conflict with the agile development principles** that we love. Service applications should evolve incrementally — and so its APIs. Of course, our API specification will and should evolve iteratively in different cycles; however, each starting with draft status and *early* team and peer review feedback. API may change and profit from implementation concerns and automated testing feedback. API evolution during development life cycle may include breaking changes for not yet productive features and as long as we have aligned the changes with the clients. Hence, API First does *not* mean that you must have 100% domain and requirement understanding and can never produce code before you have defined the complete API and get it confirmed by peer review.

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

3. General guidelines

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

MUST follow 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
- You must call for early review feedback from peers and client developers

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 support OpenAPI starting from **OpenAPI 3.0** version only.

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

MUST publish API specification to SwaggerHub

All of our APIs are published to SwaggerHub.

SHOULD provide user manual for internal APIs

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, GHE 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 provide user manual for public APIs

While for internal APIs an API user manual is an optional best practise (see **SHOULD** provide user manual for internal APIs), for public APIs it is an absolute requirement.

MUST keep API docs up-to-date

Our public API docs must be always up-to-date. Whenever a new version of a particular API Specification of a microservice is going live, the API docs must be updated. The process for updating the API docs must be followed.

MUST write APIs using U.S. English

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 **MUST** provide API identifiers)
- `#/info/x-audience` intended target audience of the API (see **MUST** provide API audience)

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 the 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.

company-internal

The API consumers with this audience are restricted to applications owned by the teams in Hypatos

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
    - company-internal
    - 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.

Example:

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

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 a security scheme as defined in the OpenAPI Authentication Specification. So far, the only security scheme we are supporting is the Client Credentials Grant defined in the OAuth 2.0 authorization framework specification. However, we are planning to support the Authorization Code Flow defined in the OpenID Connect Core 1.0 specification.

Our IdP (Identity Provider) is issuing tokens in JWT format, so that our APIs need to use the `http` typed Bearer Authentication security scheme defining the standard header `Auhorization: Bearer <token>`. The following code snippet shows how to define the bearer security scheme.

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

The bearer security schema can then be applied to all API endpoints, e.g. requiring the access token to have `documents.read` scope for permission as follows (see also **MUST** follow naming convention for permissions (scopes)):

```
security:
  - OAuth2: [ documents.read ]
```

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.

Please refer to **MUST** secure endpoints for designing permission names and see the following examples.

Context ID	Resource ID	Access Type	Example
`documents`	`document`	`read`	`documents.document.read`
`documents`		`write`	`documents.write`
`files`	`file`	`read`	`files.file.read`
`enrichment`	`invoice`	`write`	`enrichment.invoice.write`

Context ID	Resource ID	Access Type	Example
enrichment		write	enrichment.write

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

```
paths:
 /documents/{id}:
    get:
      summary: Retrieves information about a document
      security:
        - BearerAuth: [ documents.read ]
```

In some cases a whole API or selected API endpoints may not require specific permissions, e.g. if information is public. To make this explicit you should use an empty array.

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

MUST follow naming convention for permissions (scopes)

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

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

MUST not introduce custom claims in JWT or userinfo endpoint

As mentioned above, the access token issued by our IdP is a JWT. This JWT contains claims registered at the IANA JSON Web Token Claims Registry and a few custom claims

introduced by IdP. Hypatos introduced another claim called `company_id` to be used for authorization. This claim holds a company identifiers the current client has access to. The following example is demonstrating the `company_id` claim.

```
{
  "iat": 1671453749,
  "sub": "715e2b46-2a86-499c-b3be-fd2c85b101ff",
  "company_id": "637ca7c2bb9ccb1655fd6f30"
}
```

Please note that it is forbidden to introduce any other custom claims, neither in the access token (JWT) nor in the response of the `/userinfo` endpoint. If you need more information to perform authorization in your API, it must be persisted in the service behind the API.

MAY support service accounts

A service account is a special type of API client representing a non-human user that needs to authenticate and be authorized to access data in our APIs.

Granting access to a service account to access a resource is similar to granting access to any other client. However, there are special cases in which a service account needs to have access to all companies. To support such cases an API may accept a wildcard, as shown in the following example.

```
{
  "iat": 1671453749,
  "sub": "715e2b46-2a86-499c-b3be-fd2c85b101ff",
  "company_id": "*"
}
```

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

OpenAPI type	OpenAPI format	Specification	Example
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	3.14159265358979323846264338 3279
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@hypatos.ai"
string	idn-email	RFC 6531	"hello@bücher.example"
string	hostname	RFC 1034	"www.hypatos.ai"
string	idn-hostname	RFC 5890	"bücher.example"
string	ipv4	RFC 2673	"104.75.173.179"

OpenAPI type	OpenAPI format	Specification	Example
string	ipv6	RFC 4291	"2600:1401:2::8a"
string	uri	RFC 3986	"https://www.hypatos.ai/"
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	"/documents/"
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"
string	regex	regular expressions as defined in ECMA 262	"^[a-z0-9]+$"

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 might be useful 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-639-1	two letter language code — see ISO 639-1. Hint: In the past we used `iso-639` as format.	"en"

OpenAPI type	format	Specification	Example
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-alpha-2	two letter country code — see ISO 3166-1 alpha-2. Hint: In the past we used iso-3166 as format.	"GB" **Hint:** It is "GB", not "UK".
string	iso-4217	three letter currency code — see ISO 4217	"EUR"
string	gtin-13	Global Trade Item Number — see GTIN	"5710798389878"

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 use specific media types for binary data

If you want to receive or expose binary data, you must use a media type that is specific to the data format. You must not encode binary data into a JSON response. Instead, use media types such as `application/pdf`, `image/png`, etc. Please also see [168].

When receiving binary data you must not use `multipart/form-data` media type. Since the size of binary data we receive is frequently very big, we must provide streaming interfaces only in order to guarantee the reliability of our systems under high load. Since `multipart/form-data` requires loading the entire data into memory, usage if it is forbidden.

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

Properties and that are by design durations and time intervals should be represented as strings formatted as defined by ISO 8601 (RFC 3339 Appendix A contains a grammar for `durations` and `periods` - the latter called time intervals in ISO 8601). ISO 8601:1-2019 defines an extension (`..`) to express open ended time intervals that are very convenient in searches and are included in the below ABNF grammar:

```
    dur-second        = 1*DIGIT "S"
    dur-minute        = 1*DIGIT "M" [dur-second]
    dur-hour          = 1*DIGIT "H" [dur-minute]
    dur-time          = "T" (dur-hour / dur-minute / dur-second)
    dur-day           = 1*DIGIT "D"
    dur-week          = 1*DIGIT "W"
    dur-month         = 1*DIGIT "M" [dur-day]
    dur-year          = 1*DIGIT "Y" [dur-month]
    dur-date          = (dur-day / dur-month / dur-year) [dur-time]
    duration          = "P" (dur-date / dur-time / dur-week)

    period-explicit   = iso-date-time "/" iso-date-time
    period-start      = iso-date-time "/" (duration / "..")
    period-end        = (duration / "..") "/" iso-date-time
```

```
    period              = period-explicit / period-start / period-end
```

A time interval query parameters should use `<time-property>_between` instead of the parameter tuple `<time-property>_before`/`<time-property>_after`, while properties providing a time interval should be named `<time-property>_interval`.

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-alpha-2 two letter country codes indicated via format `iso-3166-alpha-2` in the OpenAPI specification.
- Language codes: ISO 639-1 two letter language codes indicated via format `iso-639-1` in the OpenAPI specification.
- Language variant tags: BCP 47 multi letter language tag indicated via format `bcp47` in the OpenAPI specification. (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 format `iso-4217` in the OpenAPI specification.

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 [172] 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.

MUST pluralize resource names

All resources must be pluralized regardless whether it returns a collection of a single object.

```
/documents
```

```
/documents/{id}
```

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.

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: 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:

```
/documents/{id}/external-id
```

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

MUST use camelCase (never snake_case) for query parameters

See also **SHOULD** use standard formats for time duration and interval properties.

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.

Note: 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 /documents/{id}
GET /documents/{id}/
GET /documents//{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.

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 use intent-based endpoints

Very often, REST APIs are built to mimic a CRUD-like interface which require a very coarse-grained design. For example, a `PUT` interface is typically provided to update a resource. The example below demonstrates an endpoint for updating a document.

```
PUT /documents/{id}
{
  "fileId": "5349b4ddd2781d08c09890f4",
  "entities": { ... },
  "externalId": "doc-0001",
  "metadata": {
    "internalSystemId": "sap-01"
  },
  "projectId": "6040dc9680b782b365ea77d5",
  "state": "done",
  "title": "scan-doc-1.jpg"
}
```

If the client intents to update document's state only, he is still expected to provide the entire document object because of the endpoint's coarse-grained nature. This forces the client to have knowledge of business logic behind the API. Even if the update endpoint were designed as `PATCH`, the risk of moving the business logic to the client side exists, because the client would need to know about the business logic triggered upon changing a property of a resource.

So, instead of designing very coarse-grained resources, API designer are encouraged to come up with intent-based REST APIs. The following example demonstrates an intent-based endpoint for updating document's external id.

```
POST /documents/{id}/external-id
doc-0001
```

8. REST Basics - JSON payload

These guidelines provides recommendations for defining JSON data at Hypatos. 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.

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** provide API audience.

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 [176]).

You should avoid using custom media types like `application/x.hypatos.invoice+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 camelCase (and never snake_case)

Property names are restricted to ASCII camelCase strings. The first character must be a lower case letter. The words are combined by capitalizing all words following the first word and removing the space.

Examples:

```
invoiceNumber, salesOrderNumber, billingAddress
```

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:

- `createdAt` rather than `created`,
- `modifiedAt` rather than `modified`,
- `occurredAt` rather than `occurred`, and
- `returnedAt` rather than `returned`.

Hint: Use `format: date-time` (or as `format: date`) as required in **MUST** use standard formats for date and time properties.

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 **SHOULD** pluralize array names, 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 differentiate between absent properties and `nullable` properties

OpenAPI 3.x allows to mark properties as required or optional using the `required` keyword. If a property is specified as `required:false`, it can be omitted in the payload if it is absent. However, absent properties must not be used with `null` value. For example, the following example demonstrates how not to use an absent property.

```
{
  "name": "Jon Doe",
  "email": null
}
```

If an API designer decided to allow `nullable` properties, it must explicitly define these as `nullable:true`.

The following table shows all combinations and whether the examples are valid. Please note that the column `{}` is representing an object holding a property. It is not representing the property itself.

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

MUST not use `null` for boolean properties

Schema based JSON properties that are by design booleans must not be presented as `null`.

A boolean is essentially a closed enumeration of two values, `true` and `false`. If the content has a meaningful null value, we strongly prefer to replace the boolean with enumeration of named values or statuses - for example `accepted_terms_and_conditions` with enumeration values `YES`, `NO`, `UNDEFINED`.

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 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.
- `xyzId`: an attribute within one object holding the identifier of another object must use a name that corresponds to the type of the referenced object or the relationship to the referenced object followed by `Id` (e.g. `companyId`).

Further common fields are defined in **SHOULD** name date/time properties with `At` suffix.

```
<!-- 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. 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.