Logius Standard - Draft

NLgov REST API Design Rules

Logius Standard
Draft October 29, 2026

This version:
https://logius-standaarden.github.io/API-Design-Rules/

Latest published version:
https://gitdocumentatie.logius.nl/publicatie/api/adr/

Latest editor's draft:
https://logius-standaarden.github.io/API-Design-Rules/

Previous version:
https://gitdocumentatie.logius.nl/publicatie/api/adr/2.2.1/

Editors:
Frank Terpstra (Geonovum)
Jan van Gelder (Geonovum)
Alexander Green (Logius)
Martin van der Plas (Logius)
Tim van der Lippe (Logius)

Authors:
Jasper Roes (Het Kadaster)
Joost Farla (Het Kadaster)

Participate:
GitHub Logius-standaarden/API-Design-Rules
All issues
File an issue
Commit history
Pull requests

This document is also available in these non-normative format: PDF

This document is licensed under
Creative Commons Attribution 4.0 International Public License

Status of This Document

This is a draft that could be altered, removed or replaced by other documents. It is not a recommendation approved by TO.

Table of Contents

Status of This Document

Abstract

1. Introduction

1.1 Goal

1.2 Status

1.3 Authors

1.4 Reading Guide

1.5 Extensions

2. The core set of Design Rules

2.1 Summary

2.2 Resources

2.3 Date and time

2.4 HTTP methods

2.5 Statelessness

2.6 Relationships

2.7 Operations

2.8 Error handling

2.9 Documentation

2.10 Versioning

2.11 Transport Security

2.11.1 HTTP-level Security

2.11.2 Browser-based applications

2.11.3 Validate content types

2.12 Normative modules

3. Glossary

A. Linter configuration

B. References

B.1 Normative references

B.2 Informative references

Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *NOT RECOMMENDED*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [*RFC2119*] [*RFC8174*] when, and only when, they appear in all capitals, as shown here.

Abstract

This document contains a normative standard for designing APIs in the Dutch Public Sector. The Governance of this standard is described by the API-Standaarden beheermodel published by Logius. This document is part of the *Nederlandse API Strategie*, which consists of a set of documents.

1. Introduction

This section is non-normative.

1.1 Goal

More and more governmental organisations offer REST APIs (henceforth abbreviated as APIs), in addition to existing interfaces like SOAP and WFS. These APIs aim to be developer-friendly and easy to implement. While this is a commendable aim, it does not shield a developer from a steep learning curve getting to know every new API, in particular when every individual API is designed using different patterns and conventions.

This document aims to describe a widely applicable set of design rules for the unambiguous provisioning of REST APIs. The primary goal is to offer guidance for organisations designing new APIs, with the purpose of increasing developer experience (DX) and interoperability between APIs. Hopefully, many organisations will adopt these design rules in their corporate API strategies and provide feedback about exceptions and additions to subsequently improve these design rules.

1.2 Status

This version of the design rules has been submitted to Forum Standaardisatie for inclusion on the Comply or Explain list of mandatory standards in the Dutch Public Sector. This document originates from the document API Strategie voor de Nederlandse Overheid, which was recently split into separate sub-documents.

1.3 Authors

Despite the fact that two authors are mentioned in the list of authors, this document is the result of a collaborative effort by the members of the *API Design Rules Working Group*.

1.4 Reading Guide

This document is part of the *Nederlandse API Strategie*.

The Nederlandse API Strategie consists of a set of distinct documents.

Status	Description & Link
Informative	Inleiding NL API Strategie
Informative	Architectuur NL API Strategie
Informative	Gebruikerswensen NL API Strategie
Normative	API Design Rules (ADR v2.0)
Normative	Open API Specification (OAS 3.0)
Normative	NLgov OAuth profiel
Normative	Digikoppeling REST API koppelvlak specificatie
Normative module	GEO module v1.0
Normative module	Signing module v1.0
Normative module	Encryption module v1.0

Before reading this document it is advised to gain knowledge of the informative documents, in particular the Architecture.

An overview of all current documents is available in this Dutch infographic:

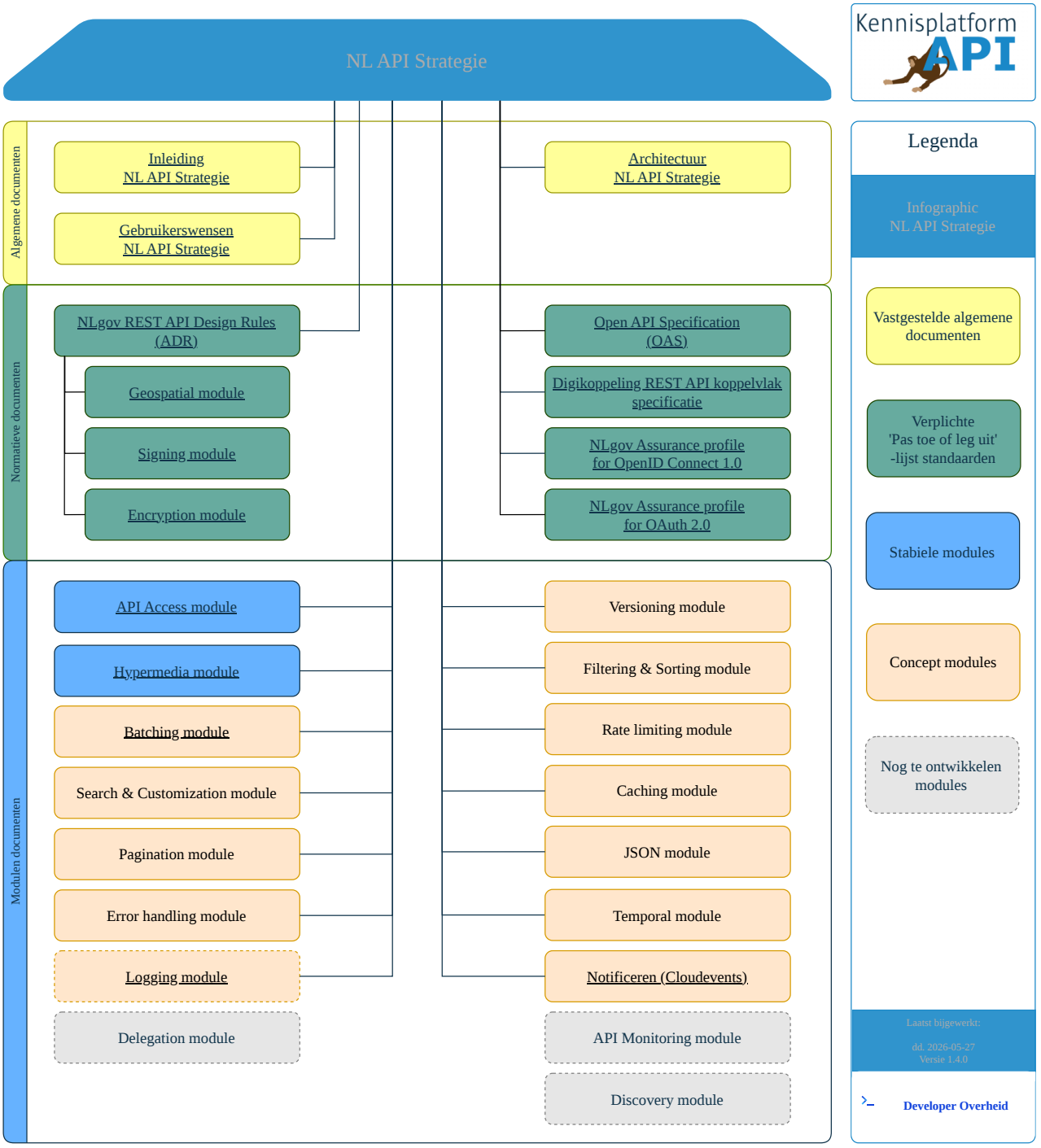


Figure 1 NL API Strategie Infographic

§1.5 Extensions

> NOTE
> In addition to this (normative) document, separate modules are being written to provide a set of extensions. These modules are all separate documents and exists in a latest editor's draft (*Werkversie* in Dutch). The latest editor's draft is actively being worked on and can be found on GitHub. It contains the most recent changes.

§2. The core set of Design Rules

§2.1 Summary

Design rules are technical rules, which should be tested automatically, and functional rules, which should be considered when designing and building the API.

List of technical rules

- /core/no-trailing-slash: Leave off trailing slashes from URIs
- /core/path-segments-kebab-case: Use kebab-case in path segments
- /core/query-keys-camel-case: Use camelCase in query keys
- /core/always-return-objects: Always return a response with an object
- /core/lang-code: Use standard language codes and field names for language content
- /core/date-time/format: Use standard format for date, datetime and time
- /core/date-time/date-omit-time-portion: Omit time portion for date fields
- /core/error-handling/problem-details: Use problem details for error responses
- /core/error-handling/invalid-input: Use status code 400 for invalid input
- /core/doc-openapi: Use OpenAPI Specification for documentation
- /core/doc-openapi-contact: Document contact information for publicly available APIs
- /core/publish-openapi: Publish OAS document at a standard location in JSON-format
- /core/uri-version: Include the major version number in the URI

- /core/semver: Adhere to the Semantic Versioning model when releasing API changes
- /core/version-header: Return the full version number in a response header
- /core/transport/tls: Secure connections using TLS
- /core/transport/security-headers: Use mandatory security headers in all API responses
- /core/transport/cors: Use CORS to control access

List of functional rules

- /core/naming-resources: Use nouns to name resources
- /core/naming-collections: Use plural nouns to name collection resources
- /core/interface-language: Define interfaces in Dutch unless there is an official English glossary available
- /core/hide-implementation: Hide irrelevant implementation details
- /core/date-time/timezone: Allow all time zone offsets in requests and use UTC in responses
- /core/http-methods: Only apply standard HTTP methods
- /core/http-safety: Adhere to HTTP safety and idempotency semantics for operations
- /core/http-response-code: Adhere to HTTP status codes to convey appropriate errors
- /core/stateless: Do not maintain session state on the server
- /core/nested-child: Use nested URIs for child resources
- /core/resource-operations: Model resource operations as a sub-resource or dedicated resource
- /core/error-handling/all-errors: Return all errors together for bad requests
- /core/doc-language: Publish documentation in Dutch unless there is existing documentation in English
- /core/deprecation-schedule: Include a deprecation schedule when deprecating features or versions
- /core/transition-period: Schedule a fixed transition period for a new major API version
- /core/changelog: Publish a changelog for API changes between versions
- /core/transport/no-sensitive-uris: No sensitive information in URIs
- /core/modules/geospatial: Apply the geospatial module for geospatial data
- /core/modules/signing: Apply the signing module for signing payloads
- /core/modules/encryption: Apply the encryption module for encrypting payloads

§2.2 Resources

The REST architectural style is centred around the concept of a resource. A resource is an abstraction of a conceptual entity, identified by a globally unique URI. It may correspond to anything from a physical object (e.g. a building or a person) to an abstract concept (e.g. a permit, an event or today's weather). Although a resource is not tied to any specific exchange format, its current state can be transferred to clients through one or more representations, such as JSON or XML.

Functional

/core/naming-resources: Use nouns to name resources

Statement

Resources *MUST* be referred to using nouns (instead of verbs) that represent entities meaningful to the API consumer.

EXAMPLE 1

A few correct examples of nouns as part of a URI:

- `Gebouw`
- `Vergunning`

This is different than RPC-style APIs, where verbs are often used to perform certain actions:

- `Opvragen`
- `Registreren`

Rationale

Resources describe objects, not actions.

A resource that corresponds to a single conceptual entity is referred to as a singular resource. Resources can also be logically grouped into collections, which are themselves resources and typically support operations like paging, sorting, and filtering. While collection members are often of the same type, this is not strictly required. A collection resource contains references (URIs) to the individual singular resources it includes.

Functional

/core/naming-collections: Use plural nouns to name collection resources

Statement

Collection resources *MUST* be referred to using plural nouns.

Rationale

The path segment describing the name of the collection resource *MUST* be written in the plural form.

Singular resources contained within a collection resource are generally named by appending a path segment for the identification of each individual resource.

Singular resources that stand on their own, i.e. which are not contained within a collection resource, *MUST* be named with a path segment that is written in the singular form.

> EXAMPLE 2
>
> Collection resources describe a list of things:
>
> ```
> https://api.example.org/v1/gebouwen
> https://api.example.org/v1/vergunningen
> ```
>
> Singular resource that is contained within a collection resource:
>
> ```
> https://api.example.org/v1/gebouwen/3b9710c4-6614-467a-ab82
> https://api.example.org/v1/vergunningen/d285e05c-6b01-45c3-
> ```
>
> Singular resource describing the profile of the currently authenticated user:
>
> ```
> https://api.example.org/v1/gebruikersprofiel
> ```

Functional

/core/interface-language: Define interfaces in Dutch unless there is an official English glossary available

Statement

Resources and the underlying attributes *SHOULD* be defined in the Dutch language unless there is an official English glossary available.

Rationale

If your API references terms used in law or official government communication for example, then these terms have a well-defined meaning. The exact meaning of concepts is often lost in translation, hence such terms *SHOULD* be defined in the Dutch language.

Publishing an API for an international (e.g. European) audience might be a reason to define interfaces in English instead.

Note that glossaries exist that define useful sets of attributes which *SHOULD* preferably be reused. Examples can be found at schema.org.

Technical

/core/no-trailing-slash: Leave off trailing slashes from URIs

Statement

A URI *MUST NOT* contain a trailing slash. When requesting a resource including a trailing slash, this *MUST* result in a `404` (not found) error response and not a redirect. This forces API consumers to use the correct URI.

> NOTE
> This rule does not apply to the root resource (append `/` to the service root URL).

Rationale

Leaving off trailing slashes, and not implementing a redirect, forces API consumers to use the correct URI. This avoids confusion and ambiguity.

> EXAMPLE 3
>
> URI without a trailing slash (correct):
>
> `https://api.example.org/v1/gebouwen`
>
> URI with a trailing slash (incorrect):
>
> `https://api.example.org/v1/gebouwen/`
>
> URI for the root resource is exempt (correct):
>
> `https://api.example.org/v1/`

How to test

Analyse all resource paths (except the root resource path) in the OpenAPI Description to confirm no resource paths end with a forward slash (`/`). This rule can be automatically checked and an example test is shown in the linter configuration.

Technical

/core/path-segments-kebab-case: Use kebab-case in path segments

Statement

Path segments of a URI *MUST* only contain lowercase letters, digits or hyphens. This is also known as kebab-case. Hyphens *MUST* only be used to delineate distinct words. This also implies that diacritics *MUST* be normalised and special characters *MUST* be omitted.

Another implication of this rule is that file extensions *MUST NOT* be used. Resources *SHOULD* use the `Accept` header for content negotiation.

The last path segment *MAY* start with `_`, which is used as a convention to implement operations

Rationale

Some web servers and frameworks do not handle case sensitivity or special characters of URIs well. The use of kebab-case path segments ensures compatibility with a broad range of systems. It is a more common implementation choice for path segments than camelCase or snake_case. Information (such as names of objects) that requires special characters can be part of the request body instead of being in the URI.

EXAMPLE 4

URI path segment using kebab-case (correct):

`https://api.example.org/v1/financiele-claims`

URI path segment not using hyphens to delineate words (incorrect):

`https://api.example.org/v1/financiele_claims`

URI path segment not lowercase characters (incorrect):

`https://api.example.org/v1/financieleClaims`

URI path segment ending with a hyphen (incorrect):

`https://api.example.org/v1/organisatie-`

URI path segment starting with a hyphen (incorrect):

`https://api.example.org/v1/-organisatie`

URI path segment using normalised diacritics (correct):

`https://api.example.org/v1/scenes`

URI path segment using diacritics (incorrect):

`https://api.example.org/v1/scènes`

URI path segment omitting special characters (correct):

`https://api.example.org/v1/schemas`

URI path segment using special characters (incorrect):

`https://api.example.org/v1/schema's`

URI path segment using file extensions (incorrect):

`https://api.example.org/v1/schema.txt`

Last URI path segment starting with _ (correct):

`https://api.example.org/v1/organisaties/_zoek`

How to test

Loop all resource paths in the OpenAPI Description and check that all resource path segments use lowercase letters, digits or hyphens (`-`). The last path segment is

allowed to start with a _.

> EXAMPLE 5
> You can use the following regular expression for each resource path:
>
> ```
> ^(\/|(\/_[a-z0-9]+|\/(([a-z0-9\-]+|{[^}]+})(\/([a-z0-9\-\.
> ```

This rule can be automatically checked and an example test is shown in the linter configuration.

Technical

/core/query-keys-camel-case: Use camelCase in query keys

Statement

Query keys in a URI *MUST* only contain letters and digits, where the first letter of each word is capitalised, except for the first letter (*MUST NOT* be a digit) of the entire compound word. This is also known as lower camelCase. This also implies that diacritics *MUST* be normalised and special characters *MUST* be omitted.

Rationale

Query keys are often converted to JSON object keys, where camelCase is the naming convention to avoid compatibility issues with JavaScript when deserialising objects.

> EXAMPLE 6
>
> URI query key using camelCase (correct):
>
> ```
> https://api.example.org/v1/gebouwen?typeGebouw=woning
> ```
>
> URI query key not using camelCase (incorrect):
>
> ```
> https://api.example.org/v1/gebouwen?type-gebouw=woning
> ```
>
> URI query key starts with digit (incorrect):
>
> ```
> https://api.example.org/v1/gebouwen?2ndReviewer=alice
> ```

How to test

Loop all resource paths in the OpenAPI Description and check that all query keys use letters, digits in camelCase. You can use the following regular expression for

each query key:

EXAMPLE 7

```
^\$?[a-z][a-z\d]*([A-Z][a-z\d]*)*$
```

This rule can be automatically checked and an example test is shown in the linter configuration.

Technical

/core/always-return-objects: Always return a response with an object

Statement

A JSON or XML response *MUST* have a top-level object, regardless of request method. For collection resources, the object *MUST* contain a field (its key *MAY* be named `items`) with its value an array of items from that collection.

Rationale

For singular resources, objects are returned as part of RESTful design.

For collection resources, items can be retrieved all at once, or a subset thereof. In both cases, wrapping the returned items in an object allows metadata to be attached to the response. Examples of metadata are structured data to implement linked data concepts, or pagination data for performance purposes.

If a collection resource directly returns an array of object information from the collection, such metadata can only be provided using HTTP headers. While that is feasible, HTTP headers are generally more difficult to work with than content from a response body.

Even if currently no metadata is attached to a response, that could be the case in the future. Changing the response from an array to an object is a breaking change, hence it is future-proof to always return an object.

EXAMPLE 8

The following example shows a response from a collection resource containing all items

```
GET /organisations HTTP/1.1
```

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

{
   "someOtherField": {
   },
  "items": [
     {
       "id": "12345",
       "name": "Logius"
     },
     {
       "id": "67890",
       "name": "Geonovum"
     }
  ]
}
```

How to test

Analyse all JSON and XML responses for paths and check that the response contains a top-level object. This rule can be automatically checked and an example test is shown in the linter configuration.

Functional

/core/hide-implementation: Hide irrelevant implementation details

Statement

An API *SHOULD NOT* expose implementation details of the underlying application, development platforms/frameworks or database systems/persistence models.

Rationale

- The primary motivation behind this design rule is that an API design *MUST* focus on usability for the client, regardless of the implementation details under the hood.

- The API, application and infrastructure need to be able to evolve independently to ease the task of maintaining backwards compatibility for APIs during an agile development process.
- The API design of Convenience,- and Process API types (as described in Aanbeveling 2 of the NL API Strategie) *SHOULD NOT* be a 1-on-1 mapping of the underlying domain- or persistence model.
- The API design of a System API type (as described in Aanbeveling 2 of the NL API Strategie) *MAY* be a mapping of the underlying persistence model.
- The API *SHOULD NOT* expose information about the technical components being used, such as development platforms/frameworks or database systems.
- The API *SHOULD* offer client-friendly attribute names and values, while persisted data may contain abbreviated terms or serialisations which might be cumbersome for consumption.

Technical

/core/lang-code: Use standard language codes and field names for language content

Statement

A resource containing language content *MUST* follow BCP 47 [*RFC4647*] [*RFC5646*]. The Primary Language Subtag *MUST* be lowercase. Languages *MUST* have a Region subtag in uppercase.

Each field that contains language content in the OpenAPI specification *MUST* set `"type": "string"` and set `"format"` to the OpenAPI format `"language"`.

These fields *MAY* specify a schema that includes an `"enum"` with the list of possible values.

EXAMPLE 9

```
"enumLanguage": {
   "type": "string",
   "format": "language",
   "enum": ["nl-NL", "fy-NL", "nl-BE", "en-GB"]
}
```

⚠ Warning

[*ISO3166-1*] concerns identifiers of countries and *MUST NOT* be used to denote languages, since countries and languages are not equivalent.

NOTE

Following [*RFC4647*] a language code in [*ISO-639-1*] format matches a language tag in [*RFC5646*] format regardless of region subtag.

EXAMPLE 10

The following table shows some examples of language codes and their respective subtags in corresponding capitalisation.

Language code	Textual description	Primary Language Subtag	Region Subtag
`nl-NL`	The Dutch language in The Netherlands	`nl`	`NL`
`nl-BE`	The Dutch language in Belgium	`nl`	`BE`
`fy-NL`	The Western Frisian language in The Netherlands	`fy`	`NL`
`en-GB`	The English language in the United Kingdom	`en`	`GB`

Rationale

Standardised language codes removes ambiguity in language handling between systems, potentially present in separate regions with different (spoken) languages.

How to test

Confirm each field that represents a language has a value in [*RFC5646*] format. This rule can be automatically checked and an example test is shown in the linter configuration.

§2.3 Date and time

Handling date and time is tricky and can lead to confusion among clients. The date-time rules remove ambiguity and provide clarity in the API contract between servers and clients.

EXAMPLE 11

A child is born on March 20th 2025 in The Netherlands. If a client sends a request with value `2025-03-20T00:00:00+01:00`, time zone conversion would result in `2025-03-19T23:00:00Z`. When the client receives this value in a response and incorrectly converts it to a date (by removing the time portion), this would result in `2025-03-19`.

Ambiguous date and time handling can therefore lead to misinterpretation and changes of days/months/years depending on which component performs which incorrect conversion. Clients could incorrectly remove a time portion from a datetime value if the value should have been a date in the first place. By specifying which formats are allowed in which fields, the odds of invalid conversion are reduced.

Technical

/core/date-time/format: Use standard format for date, datetime and time

Statement

All date, datetime and time fields in requests and responses *MUST* adhere to [*RFC9557*] and [*ISO8601-1*] format. Each field in the OpenAPI specification *MUST* set `"type": "string"` and set `"format"` to the OpenAPI format as listed in the following table:

Field type	ISO8601 format	OpenAPI format
Date	full-date	`"format": "date"`
Datetime	date-time	`"format": "date-time"`
Time	partial-time	`"format": "time-local"`

Rationale

Implementing RFC9557 and ISO 8601 removes ambiguity in date handling between systems and time zones.

NOTE

RFC9557 is a profile on ISO8601, but is not a strict subset of allowed notations. Practically, to adhere to both, the following limitations *MUST* be applied to RFC9557:

- In a field with a date-time value, the date and time component *MUST* be separated by a "T" in uppercase.
- The time zone offset "Z" *MUST* be uppercase.
- "-00:00" *MUST NOT* be used as time zone offset.

How to test

Analyse all fields and if the field represents a date, date-time or time, ensure it has the correct format according to the table above.

Functional

/core/date-time/timezone: Allow all time zone offsets in requests and use UTC in responses

Statement

APIs *MUST* accept any time zone offset in fields in requests containing a datetime. Fields in responses containing a datetime *SHOULD* be in UTC (e.g. Z as time zone offset).

EXAMPLE 12

A field "`meetingStartTime`" containing a datetime value to denote the start time of a meeting. Depending on the local time zone of the client, the UTC datetime value is converted to that local time zone.

For example, a Dutch government worker travels to Ottowa in Canada and has an online meeting with their Dutch colleagues in The Netherlands. Instead of showing the meeting start time in the time zone of The Netherlands, it is shown in the relevant local time zone in Ottowa.

If the original time zone is relevant for users (such as the time zone in which a value is registered), the time zone offset *MUST* be stored and published as a separate field in [*ISO8601-1*] format `time-offset`.

EXAMPLE 13

A response containing a field "`timeOfBirth`" with a datetime value to denote the time of birth of a child in The Netherlands also has a field "`timeOfBirthTimezone`" with the relevant time zone offset (`+01:00` or `+02:00`).

Rationale

Allowing clients to use any time zone offset in requests results in flexibility and less complexity for users. Using UTC in responses results in clarity and removes ambiguity.

NOTE

While storage formats are outside the scope of this specification, it is recommended to use appropriate temporal data types (such as `DATE` and `TIMESTAMPTZ`). Many database systems store these values internally in UTC and handle time zone conversion automatically on read/write.

Technical

/core/date-time/date-omit-time-portion: Omit time portion for date fields

Statement

If the time portion is not relevant, `date` format *MUST* be used instead of `date-time` format.

Rationale

Appending a default or irrelevant time portion to a date field can lead to interpretation errors. A publish date of `2025-07-24T00:00:00Z` could for instance be rendered as July 23 in Ireland. A default time of 23:59 would in turn cause date confusion east of Greenwich.

NOTE

To prevent conversion errors between storage and API, it is recommended to also store date values without a time portion (i.e. not as a timestamp).

EXAMPLE 14

A response containing a field "`birthDate`" with a date value to denote the date of birth of a child in The Netherlands.

How to test

Analyse all fields that set format to "date-time" and ensure that the fields do not represent solely a date. This rule can be automatically checked and an example test is shown in the linter configuration.

§2.4 HTTP methods

Although the REST architectural style does not impose a specific protocol, REST APIs are typically implemented using HTTP [*RFC9110*].

Functional

/core/http-methods: Only apply standard HTTP methods

Statement

An API *MUST* adhere to the HTTP method semantics defined in [*RFC9110*] and [*RFC5789*].

Rationale

The HTTP specifications offer a set of standard methods, where every method is designed with explicit semantics. Adhering to the HTTP specification is crucial, since HTTP clients and middleware applications rely on standardised characteristics.

Method	Operation	Description
`GET`	Read	Retrieve a resource representation for the given URI. Data is only retrieved and never modified.
`POST`	Create	Create a subresource as part of a collection resource. This operation is not relevant for singular resources. This method can also be used for exceptional cases.
`PUT`	Create/update	Create a resource with the given URI or replace (full update) a resource when the resource already exists.

Method	Operation	Description
`PATCH`	Update	Partially updates an existing resource. The request only contains the resource modifications instead of the full resource representation.
`DELETE`	Delete	Remove a resource with the given URI.

EXAMPLE 15

The following table shows some examples of the use of standard HTTP methods:

Request	Description
`GET /rijksmonumenten`	Retrieves a list of national monuments.
`GET /rijksmonumenten/12`	Retrieves an individual national monument.
`POST /rijksmonumenten`	Creates a new national monument.
`PUT /rijksmonumenten/12`	Modifies national monument #12 completely.
`PATCH /rijksmonumenten/12`	Modifies national monument #12 partially.
`DELETE /rijksmonumenten/12`	Deletes national monument #12.

NOTE

The HTTP specification [*RFC9110*] and the later introduced `PATCH` method specification [*RFC5789*] offer a set of standard methods, where every method is designed with explicit semantics. HTTP also defines other methods, e.g. `HEAD`, `OPTIONS`, `TRACE`, and `CONNECT`.
The OpenAPI Specification 3.0 Path Item Object also supports these methods, except for `CONNECT`.
According to RFC 9110 9.1 the `GET` and `HEAD` HTTP methods *MUST* be supported by the server, all other methods are optional.
In addition to the standard HTTP methods, a server may support other optional methods as well, e.g. `PROPFIND`, `COPY`, `PURGE`, `VIEW`, `LINK`, `UNLINK`, `LOCK`, `UNLOCK`, etcetera.
If an optional HTTP request method is sent to a server and the server does not support that HTTP method for the target resource, an HTTP status code `405 Method Not Allowed` shall be returned and a list of allowed methods for the target resource shall be provided in the `Allow` header in the response as stated in RFC 9110 15.5.6.

How to test

Analyse the OpenAPI Description to confirm all supported methods are either `post`, `put`, `get`, `delete`, or `patch`.

Functional

/core/http-safety: Adhere to HTTP safety and idempotency semantics for operations

Statement

The following table describes which HTTP methods *MUST* behave as safe and/or idempotent:

Method	Safe	Idempotent
`GET`	Yes	Yes
`HEAD`	Yes	Yes
`OPTIONS`	Yes	Yes
`POST`	No	No
`PUT`	No	Yes
`PATCH`	No	No
`DELETE`	No	Yes

Rationale

The HTTP protocol [*RFC9110*] specifies whether an HTTP method *SHOULD* be considered safe and/or idempotent. These characteristics are important for clients and middleware applications, because they *SHOULD* be taken into account when implementing caching and fault tolerance strategies.

Request methods are considered *safe* if their defined semantics are essentially read-only; i.e., the client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource. A request method is considered *idempotent* if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request.

Functional

/core/http-response-code: Adhere to HTTP status codes to convey appropriate errors

Statement

An API *MUST* use the semantically appropriate HTTP status code ([*RFC9110*]) for the response.

Rationale

The server *SHOULD NOT* only use `200` for success and `404` for error states. Use the semantically appropriate status code for success or failure.

In case of an error, the server *SHOULD NOT* pass technical details (e.g. call stacks or other internal hints) to the client. The error message *SHOULD* be generic to avoid revealing additional details and expose internal information which can be used with malicious intent.

§2.5 Statelessness

One of the key constraints of the REST architectural style is stateless communication between client and server. It means that every request from client to server must contain all of the information necessary to understand the request. The server cannot take advantage of any stored session context on the server as it didn't memorise previous requests. Session state must therefore reside entirely on the client.

To properly understand this constraint, it is important to make a distinction between two different kinds of state:

- *Session state*: information about the interactions of an end user with a particular client application within the same user session, such as the last page being viewed, the login state or form data in a multi-step registration process. Session state must reside entirely on the client (e.g. in the user's browser).
- *Resource state*: information that is permanently stored on the server beyond the scope of a single user session, such as the user's profile, a product purchase or information about a building. Resource state is persisted on the server and must be exchanged between client and server (in both directions) using representations as part of the request or response payload. This is actually where the term *REpresentational State Transfer (REST)* originates from.

> NOTE
>
> It is a misconception that there should be no state at all. The stateless communication constraint should be seen from the server's point of view and states that the server should not be aware of any *session state*.

Stateless communication offers many advantages, including:

- *Simplicity* is increased because the server does not have to memorise or retrieve session state while processing requests
- *Scalability* is improved because not having to incorporate session state across multiple requests enables higher concurrency and performance
- *Observability* is improved since every request can be monitored or analysed in isolation without having to incorporate session context from other requests
- *Reliability* is improved because it eases the task of recovering from partial failures since the server does not have to maintain, update or communicate session state. One failing request does not influence other requests (depending on the nature of the failure of course).

> Functional
>
> **/core/stateless**: Do not maintain session state on the server
>
> **Statement**
>
> In the context of REST APIs, the server *MUST NOT* maintain or require any notion of the functionality of the client application and the corresponding end user interactions.
>
> **Rationale**
>
> To achieve full decoupling between client and server, and to benefit from the advantages mentioned above, session state *MUST NOT* reside on the server.

Session state *MUST* therefore reside entirely on the client.

NOTE

The client of a REST API could be a variety of applications such as a browser application, a mobile or desktop application and even another server serving as a backend component for another client. REST APIs should therefore be completely client-agnostic.

§2.6 Relationships

Resources are often interconnected by relationships. Relationships can be modelled in different ways depending on the cardinality, semantics and more importantly, the use cases and access patterns the REST API needs to support.

Functional

/core/nested-child: Use nested URIs for child resources

Statement

When having a child resource which can only exist in the context of a parent resource, the URI *SHOULD* be nested.

Rationale

In this use case, the child resource does not necessarily have a top-level collection resource. The best way to explain this design rule is by example.

EXAMPLE 16

When modelling resources for a news platform including the ability for users to write comments, it might be a good strategy to model the collection resources hierarchically:

```
https://api.example.org/v1/articles/123/comments
```

The platform might also offer a photo section, where the same commenting functionality is offered. In the same way as for articles, the corresponding sub-collection resource might be published at:

```
https://api.example.org/v1/photos/456/comments
```

These nested sub-collection resources can be used to post a new comment (`POST` method) and to retrieve a list of comments (`GET` method) belonging to the parent resource, i.e. the article or photo. An important consideration is that these comments could never have existed without the existence of the parent resource.

From the consumer's perspective, this approach makes logical sense, because the most obvious use case is to show comments below the parent article or photo (e.g. on the same web page) including the possibility to paginate through the comments. The process of posting a comment is separate from the process of publishing a new article. Another client use case might also be to show a global *latest comments* section in the sidebar. For this use case, an additional resource could be provided:

```
https://api.example.org/v1/comments
```

If this would have not been a meaningful use case, this resource should not exist at all. Because it does not make sense to post a new comment from a global context, this resource would be read-only (only `GET` method is supported) and may possibly provide a more compact representation than the parent-specific sub-collections.

The singular resources for comments, referenced from all 3 collections, could still be modelled on a higher level to avoid deep nesting of URIs (which might increase complexity or problems due to the URI length):

```
https://api.example.org/v1/comments/123
https://api.example.org/v1/comments/456
```

Although this approach might seem counter-intuitive from a technical perspective (we simply could have modelled a single `/comments` resource with optional filters for article and photo) and might introduce partially

redundant functionality, it makes perfect sense from the perspective of the consumer, which increases developer experience.

§2.7 Operations

Functional

/core/resource-operations: Model resource operations as a sub-resource or dedicated resource

Statement

Resource operations *MUST* be modelled as a sub-resource or dedicated resource.

Rationale

There are resource operations which might not seem to fit well in the CRUD interaction model. For example, approving a submission or notifying a customer. Depending on the type of the operation, there are three possible approaches:

1. Re-model the resource to incorporate extra fields supporting the particular operation. For example, an approval operation can be modelled in a boolean attribute `goedgekeurd` that can be modified by issuing a `PATCH` request against the resource. A drawback of this approach is that the resource does not contain any metadata about the operation (when and by whom was the approval given? Was the submission rejected in an earlier stage?). Furthermore, this requires a fine-grained authorisation model, since approval might require a specific role.
2. Treat the operation as a sub-resource. For example, model a sub-collection resource `/inzendingen/12/beoordelingen` and add an approval or rejection by issuing a `POST` request. To be able to retrieve the review history (and to consistently adhere to the REST principles), also support the `GET` method for this resource. The `/inzendingen/12` resource might still provide a `goedgekeurd` boolean attribute (same as approach 1) which gets automatically updated in the background after adding a review. This attribute *SHOULD* however be read-only.
3. In exceptional cases, the approaches above still do not offer an appropriate solution. An example of such an operation is a global search across multiple resources. In this case, the creation of a dedicated resource, possibly nested under an existing resource, is the most obvious solution. Use the imperative mood of a verb, maybe even prefix it with a underscore to distinguish these

resources from regular resources. For example: `/search` or `/_search`. Depending on the operation characteristics, `GET` and/or `POST` method *MAY* be supported for such a resource.

§2.8 Error handling

Technical

/core/error-handling/problem-details: Use problem details for error responses

Statement

Error responses with HTTP status codes `4xx` or `5xx` *MUST* use either `application/problem+json` or `application/problem+xml` as the `Content-Type` header, and the response body *MUST* conform to the structure defined in [*RFC9457*].

The following fields *MUST* be present: `status`, `title`, and `detail`.

Rationale

Providing problem details in a machine-readable format aids automation and debugging. By using a common error format, APIs do not need to define their own or misuse existing HTTP status codes.

EXAMPLE 17

The following example shows the head and body of a detailed error response.

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "status": 404,
  "title": "Resource Not Found",
  "detail": "No building found with id 12345."
}
```

How to test

Verify all responses with status code `4xx` or `5xx` have `Content-Type` set to `application/problem+json` or `application/problem+xml` and the body

contains fields `status`, `title`, and `detail`. This rule can be automatically checked and an example test is shown in the linter configuration.

Technical

/core/error-handling/invalid-input: Use status code 400 for invalid input

Statement

API requests containing invalid input *MUST* result in HTTP status code `400` (Bad Request). Invalid input includes syntax errors, missing or invalid query parameters.

The request payload *SHOULD* be validated with a schema. A request payload with schema validation errors *MUST* be treated as invalid input.

Rationale

The semantics of status code `400` ("the server cannot or will not process the request due to something that is perceived to be a client error") match validation failures more closely than status code `422`, which historically originates from WebDAV and introduces no added interoperability benefit.

How to test

Verify that operations accepting query parameters and/or a request body contain a response with status code `400`. This rule can be automatically checked and an example test is shown in the linter configuration.

Functional

/core/error-handling/all-errors: Return all errors together for bad requests

Statement

API requests with HTTP status code `400` (Bad Request) *SHOULD* include all applicable schema validation errors and *MAY* include additional errors.

Rationale

To reduce the amount of round trips between client and server, all applicable schema validation errors *SHOULD* be returned together. This allows a client to present validation errors to a user in one go, reducing user friction with multiple retries.

It depends on a validation technique whether this is possible or not. For example, when a client provides a date in the weekend, where only dates on weekdays are allowed, it depends on which service performs these validation checks. In these

cases, include the additional validation errors together with other errors whenever feasible.

§2.9 Documentation

An API is as good as the accompanying documentation. The documentation has to be easily findable, searchable and publicly accessible. Most developers will first read the documentation before they start implementing. Hiding the technical documentation in PDF documents and/or behind a login creates a barrier for both developers and search engines.

Technical

/core/doc-openapi: Use OpenAPI Specification for documentation

Statement

API documentation *MUST* be provided in the form of an OpenAPI definition document which conforms to the OpenAPI Specification (from v3 onwards).

Rationale

The OpenAPI Specification (OAS) [*OPENAPIS*] defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. API documentation *MUST* be provided in the form of an OpenAPI definition document which conforms to the OpenAPI Specification (from v3 onwards). As a result, a variety of tools can be used to render the documentation (e.g. Swagger UI or ReDoc) or automate tasks such as testing or code generation. The OAS document *SHOULD* provide clear descriptions and examples.

How to test

Parse the resource at the provided location as an OpenAPI Description and confirm all `$refs` are resolvable and paths are defined. This rule can be automatically checked and an example test is shown in the linter configuration.

Technical

/core/doc-openapi-contact: Document contact information for publicly available APIs

Statement

OpenAPI definition document *SHOULD* include the `info.contact` object for publicly available APIs. The `info.contact` object *MUST* include the fields `name`, `url` and `email`. Contact information *SHOULD NOT* be a generic contact address for the whole organisation.

Rationale

The OpenAPI Specification (OAS) [*OPENAPIS*] can include contact information to make clear how to reach out to API owners in case of issues or questions. This is relevant for publicly available APIs (such as OData) where no pre-existing communication channel exists between provider and consumer of the API. For internal APIs (where communication channels such as chat or issue trackers are likely already known), the `info.contact` *MAY* be provided.

EXAMPLE 18

Relevant contact information can include an email address and issue tracker.

```
{
  "name": "Gebouwen API beheerder",
  "url": "https://www.github.com/ministerie/gebouwen/issue
  "email": "teamgebouwen@ministerie.nl"
}
```

How to test

- Step 1. Parse the OpenAPI Description to confirm the `info.contact` object is present.
- Step 2. Confirm `info.contact` contains the fields `name`, `url` and `email`.

This rule can be automatically checked and an example test is shown in the linter configuration.

Functional

/core/doc-language: Publish documentation in Dutch unless there is existing documentation in English

Statement

You *SHOULD* write the OAS document in Dutch.

Rationale

In line with design rule /core/interface-language, the OAS document (e.g. descriptions and examples) *SHOULD* be written in Dutch. If relevant, you *MAY* refer to existing documentation written in English.

Technical

/core/publish-openapi: Publish OAS document at a standard location in JSON-format

Statement

To make the OAS document easy to find and to facilitate self-discovering clients, there *SHOULD* be one standard location where the OAS document is available for download.

Rationale

It *MUST* be possible for clients (such as Swagger UI or ReDoc) to retrieve the document without having to authenticate. Furthermore, the CORS policy for this URI *MUST* allow external domains to read the documentation from a browser environment.

The standard location for the OAS document is a URI called `openapi.json` or `openapi.yaml` within the base path of the API. This can be convenient, because OAS document updates can easily become part of the CI/CD process.

At least the JSON format *MUST* be supported. When having multiple (major) versions of an API, every API version *SHOULD* provide its own OAS document(s).

> EXAMPLE 19
> An API having base path `https://api.example.org/v1` *MUST* publish the OAS document at:
>
> `https://api.example.org/v1/openapi.json`
>
> Optionally, the same OAS document *MAY* be provided in YAML format:
>
> `https://api.example.org/v1/openapi.yaml`

How to test

- Step 1: The API *MUST* meet the prerequisites to be tested. These include that an OAS file (openapi.json) is publicly available, parseable, all `$refs` are resolvable and paths are defined.

- Step 2: The openapi.yaml document *MAY* be available. If available it *MUST* contain YAML, be readable and parseable.
- Step 3: The openapi.yaml document *MUST* contain the same OpenAPI Description as the openapi.json document.
- Step 4: The CORS header Access-Control-Allow-Origin *MUST* allow all origins.

This rule can be automatically checked and an example test is shown in the linter configuration.

§2.10 Versioning

Changes in APIs are inevitable. APIs should therefore always be versioned, facilitating the transition between changes.

Functional

/core/deprecation-schedule: Include a deprecation schedule when deprecating features or versions

Statement

The API owner *SHOULD* implement well-documented deprecation schedules that are communicated in a timely fashion.

Rationale

Managing change is important. In general, good documentation and timely communication regarding deprecation schedules are the most important for API users. When deprecating features or versions, a deprecation schedule *MUST* be published. This document *SHOULD* be published on a public web page. Furthermore, active clients *SHOULD* be informed by e-mail once the schedule has been updated or when versions have reached end-of-life.

Functional

/core/transition-period: Schedule a fixed transition period for a new major API version

Statement

Old versions *MUST* remain available for a limited and fixed deprecation period.

Rationale

When releasing a new major API version, the old version *MUST* remain available for a limited and fixed deprecation period. Offering a deprecation period allows clients to carefully plan and execute the migration from the old to the new API version, as long as they do this prior to the end of the deprecation period. A maximum of 2 major API versions *MAY* be published concurrently.

Technical

/core/uri-version: Include the major version number in the URI

Statement

The URI of an API *MUST* include the major version number.

Rationale

The URI of an API (base path) *MUST* include the major version number, prefixed by the letter `v`. This allows the exploration of multiple versions of an API in the browser. The minor and patch version numbers are not part of the URI and *MAY* not have any impact on existing client implementations.

EXAMPLE 20

An example of an `openapi.yaml` for an API with a base path `https://api.example.org/v1` and current version 1.0.2:

```
openapi: 3.0.0
   info:
      version: '1.0.2'
   servers:
      - description: test environment
      url: https://api.test.example.org/v1
      - description: production environment
      url: https://api.example.org/v1
```

How to test

Parse the `url` field in the `servers` mentioned in the OpenAPI Description to confirm that a version number is present with prefix `v` and only contains the *major* version number. This rule can be automatically checked and an example test is shown in the linter configuration.

Functional

/core/changelog: Publish a changelog for API changes between versions

Statement

A changelog *MUST* be published for every API version.

Rationale

When releasing new (major, minor or patch) versions, all API changes *MUST* be documented properly in a publicly available changelog.

Technical

/core/semver: Adhere to the Semantic Versioning model when releasing API changes

Statement

Semantic Versioning *MUST* be used for API versioning.

Rationale

Version numbering *MUST* follow the Semantic Versioning [*SemVer*] model to prevent breaking changes when releasing new API versions. Release versions are formatted using the `major.minor.patch` template (examples: `1.0.2`, `1.11.0`). Pre-release versions *MAY* be denoted by appending a hyphen and a series of dot separated identifiers (examples: `1.0.2-rc.1`, `2.0.0-beta.3`). When releasing a new version which contains backwards-incompatible changes, a new major version *MUST* be released. Minor and patch releases *MUST* only contain backwards compatible changes (e.g. the addition of an endpoint or an optional attribute).

How to test

Parse the `info.version` field in the OpenAPI Description to confirm it adheres to the Semantic Versioning format.

Technical

/core/version-header: Return the full version number in a response header

Statement

The API-Version header *MUST* be returned in a response header in every response.

Rationale

Since the URI only contains the major version, it is useful to provide the full version number in the response headers for every API call. This information could then be used for logging, debugging or auditing purposes. In cases where an intermediate networking component returns an error response (e.g. a reverse proxy enforcing access policies), the version number *MAY* be omitted.

The version number *MUST* be returned in an HTTP response header named `API-Version` (case-insensitive) and *SHOULD NOT* be prefixed.

EXAMPLE 21

An example of an API version response header:

```
API-Version: 1.0.2
```

How to test

A response includes a header "API-Version" with a number matching the version number set in the `info.version` field of the OpenAPI Description. This rule can be automatically checked and an example test is shown in the linter configuration.

2.11 Transport Security

This section describes security principles, concepts and technologies to apply when working with APIs. Controls need to be applied for the security objectives of integrity, confidentiality and availability of the API (which includes the services and data provided thereby). The architecture section of the API strategy contains architecture patterns for implementing transport security.

The scope of this section is limited to generic security controls that directly influence the visible parts of an API. Effectively, only security standards directly applicable to interactions are discussed here.

In order to meet the complete security objectives, every implementer *MUST* also apply a range of controls not mentioned in this section.

Note: security controls for signing and encrypting of application level messages are part of separate extensions: Signing and Encryption.

Technical

/core/transport/tls: Secure connections using TLS

Statement

Information exchanged by APIs *MUST* be secured using TLS-based connections. No exceptions, so everywhere and always. This is required by law.

One *MUST* follow the latest NCSC guidelines [*NCSC 2025*].

Rationale

Since the connection is always secured, the access method can be straightforward. This allows the application of basic access tokens instead of encrypted access tokens.

How to test

The usage of TLS is machine testable. Follow the latest NCSC guidelines on what is required to test. The server-side is what will be tested, only control over the server is assumed for testing. A testing client will be employed to test adherence of the server. Supporting any protocols, algorithms, key sizes, options or ciphers that are deemed insufficient or phased out by NCSC will lead to failure on the automated test. Both positive and negative scenarios are part of the test: testing that a subset of *Good* and *Sufficient* configurations are supported and configurations deemed *Insufficient* or marked for *Phase out*. A manual exception to the automated test results can be made when configurations designated for *Phase out* are supported; The API provider will have to provide clear documentation regarding the phase out schedule.

Functional

/core/transport/no-sensitive-uris: No sensitive information in URIs

Statement

Sensitive information *MUST NOT* be part of URIs

> NOTE
>
> The term sensitive is deliberately left undefined in this document.

In case of REST-API's for system to system communication on a closed network, this rule applies only when there is logging involved in systems that are not under control of the organisations involved in the exchange

Rationale

When using TLS connections, the path and query information in URIs are secured just like the message headers and body. However, URIs can be cached and logged, as can headers and bodies in the following situations:

- before the TLS connection starts on the server
- after the TLS connection ends on the client
- whenever the TLS protocol is terminated and newly initiated in between

> NOTE
>
> Be aware that queries (anything after the '?' in a URI) are also part of a URI.

> For REST API's that are accessed directly from user devices, like web browsers, do not put client secrets used for authentication and other sensitive information in the URI. These are directly visible to users, are stored in the web browser's history and cache and can be bookmarked and sent to others.
>
> For REST API's that are only used for system-to-system integration on closed networks where all systems are under control of the organisations involved in the exchange, do not put client secrets used for authentication in the URI and be careful to put sensitive information in the URI. Intermediate network components that terminate and newly initiate TLS could log or otherwise store URIs. Consider the consequences, advantages and disadvantages of using sensitive information in the URI and be deliberate about which information is logged, for which purposes and who has access.

§ 2.11.1 HTTP-level Security

The guidelines and principles defined in this section are client agnostic. When implementing a client agnostic API, one *SHOULD* at least facilitate that multi-purpose generic HTTP-clients like browsers are able to securely interact with the API. When implementing an API for a specific client it may be possible to limit measures as long as it ensures secure access for this specific client. Nevertheless it is advised to review the following security measures, which are mostly inspired by the OWASP REST Security Cheat Sheet.

Even while remaining client agnostic, clients can be classified in four major groups. This is in line with common practice in *The OAuth 2.0 Authorization Framework*. The groups are:

1. Web applications.
2. Native applications.
3. Browser-based applications.
4. System-to-system applications.

This section contains elements that apply to the generic classes of clients listed above. Although not every client implementation has a need for all the specifications referenced below, a client agnostic API *SHOULD* provide these to facilitate any client to implement relevant security controls.

Most specifications referenced in this section are applicable to the first three classes of clients listed above. Security considerations for native applications are provided in *OAuth 2.0 for Native Apps*, much of which can help non-OAuth2 based implementations as well. For browser-based

applications a subsection is included with additional details and information. System-to-system (sometimes called machine-to-machine) may have a need for the listed specifications as well. Note that different usage patterns may be applicable in contexts with system-to-system clients, see above under Client Authentication.

Realisations may rely on internal usage of HTTP-Headers. Information for processing requests and responses can be passed between components, that can have security implications. For instance, this is common practice between a reverse proxy or TLS-offloader and an application server. Additional HTTP headers are used in such example to pass an original IP-address or client certificate.

Implementations *MUST* consider filtering both inbound and outbound traffic for HTTP-headers used internally. The primary focus of inbound filtering is to prevent injection of malicious headers on requests. For outbound filtering, the main concern is leaking of information.

Technical

/core/transport/security-headers: Use mandatory security headers in all API responses

Statement

API security headers *MUST* be returned in all server responses to instruct the client to act in a secure manner

Rationale

There are a number of security related headers that can be returned in the HTTP responses to instruct browsers to act in specific ways. However, some of these headers are intended to be used with HTML responses, and as such may provide little or no security benefits on an API that does not return HTML. The following headers *SHOULD* be included in all API responses:

Header	Rationale
`Cache-Control: no-store`	Prevent sensitive information from being cached.
`Content-Security-Policy: frame-ancestors 'none'`	To protect against drag-and-drop style clickjacking attacks.
`Content-Type`	To specify the content type of the response. This *SHOULD* be `application/json` for JSON responses.
`Strict-Transport-Security`	To require connections over HTTPS and to protect against spoofed certificates.

Header	Rationale
`X-Content-Type-Options: nosniff`	To prevent browsers from performing MIME sniffing, and inappropriately interpreting responses as HTML.
`X-Frame-Options: DENY`	To protect against drag-and-drop style clickjacking attacks.
`Access-Control-Allow-Origin`	To relax the 'same origin' policy and allow cross-origin access. See /core/transport/cors for more information.

The headers below are only intended to provide additional security when responses are rendered as HTML. As such, if the API will never return HTML in responses, then these headers may not be necessary. You *SHOULD* include the headers as part of a defence-in-depth approach if there is any uncertainty about the function of the headers, the types of information that the API returns or information it may return in the future.

Header	Rationale
`Content-Security-Policy: default-src 'none'`	The majority of CSP functionality only affects pages rendered as HTML.
`Feature-Policy: 'none'`	Feature policies only affect pages rendered as HTML.
`Referrer-Policy: no-referrer`	Non-HTML responses should not trigger additional requests.

In addition to the above listed HTTP security headers, web- and browser-based applications *SHOULD* apply *Subresource Integrity*. When using third-party hosted contents, e.g. using a Content Delivery Network, this is even more relevant. While this is primarily a client implementation concern, it may affect the API when it is not strictly segregated or for example when shared supporting libraries are offered.

How to test

The presence of the mandatory security headers can be tested in an automated way. A test client makes a call to the API root. The response is tested for the presence of mandatory headers.

/core/transport/cors: Use CORS to control access

Technical

Statement

CORS *MUST* be used to restrict access from other domains for applicable resources

Rationale

Different resources can have different uses, as some resources are publicly available whereas others are restricted to several domains. Modern web browsers use Cross-Origin Resource Sharing (CORS) to minimise the risk associated with cross-site HTTP-requests.

By default browsers only allow 'same origin' access to resources. This means that responses on requests to another `[scheme]://[hostname]:[port]` than the `Origin` request header of the initial request will not be processed by the browser. To enable cross-site requests APIs can return a `Access-Control-Allow-Origin` response header.

An allowlist *SHOULD* be used to determine the validity of different cross-site requests. To do this, check the `Origin` header of the incoming request and check if the domain in this header is on the allowlist. If this is the case, set the incoming `Origin` header in the `Access-Control-Allow-Origin` response header.

> NOTE
> Using a wildcard * in the `Access-Control-Allow-Origin` response header is *NOT RECOMMENDED*, because it disables CORS-security measures. However, if the resource has to be accessed by numerous other origins that are not known up front (such as all resources in an open API, or the `openapi.json` as required by /core/publish-openapi), you *MAY* use *.

How to test

Tests of this design rule can only be performed when the intended client is known to the tester. A test can be performed when this information is provided by the API provider. Otherwise no conclusive test result can be reached.

§ 2.11.2 Browser-based applications

A specific subclass of clients are browser-based applications, that require the presence of particular security controls to facilitate secure implementation. Clients in this class are also known as *user-agent-based* or *single-page-applications* (SPA). All browser-based applications *SHOULD* follow the best practices specified in OAuth 2.0 for Browser-Based Apps. These applications can be split into three architectural patterns:

- JavaScript applications with a backend; with this class of applications, the backend is the confidential client and should intermediate any interaction, with tokens never ending up in the browser. Effectively, these are not different from regular web-application for this security facet, even though they leverage JavaScript for implementation.
- JavaScript applications that share a domain with the API (resource server); these can leverage cookies marked as HTTP-Only, Secure and SameSite.
- JavaScript applications without a backend; these clients are considered public clients, and are potentially more vulnerable to several types of attacks, including Cross-Site Scripting (`XSS`), Cross Site Request Forgery (`CSRF`) and OAuth token theft. In order to support these clients, the Cross-Origin Resource Sharing (`CORS`) policy mentioned above is critical and *MUST* be supported.

§ 2.11.3 Validate content types

A REST request or response body *SHOULD* match the intended content type in the header. Otherwise this could cause misinterpretation at the consumer/producer side and lead to code injection/execution.

- Requests containing unexpected or missing content type headers *MUST* be rejected with HTTP response status `406 Not Acceptable` or `415 Unsupported Media Type`.
- Accidentally exposing unintended content types *MUST* be avoided by explicitly defining content types e.g. Jersey (Java) `@consumes("application/json"); @produces("application/json")`. This avoids XML external entity injection attack vectors for example.

It is common for REST services to allow multiple response types (e.g. `application/xml` or `application/json`) in which case then the client specifies the preferred order of response types by the Accept header in the request.

- Do not simply copy the `Accept` header to the `Content-type` header of the response.
- Reject the request (ideally with a `406 Not Acceptable` response) if the Accept header does not specifically contain one of the allowable types.

Services (potentially) including script code (e.g. JavaScript) in their responses *MUST* be especially careful to defend against header injection attacks.

- Ensure the intended Content-Type headers are sent in the response, matching the body content, e.g. `application/json` and not `application/javascript`.

§2.12 Normative modules

The following modules are normative for all REST API's.

Functional

/core/modules/geospatial: Apply the geospatial module for geospatial data

Statement

The *API Design Rules Module: Geospatial* version 1.0.x *MUST* be applied when providing geospatial data or functionality.

> NOTE
>
> Geospatial data refers to information that is associated with a physical location on Earth, often expressed by its 2D/3D coordinates.

Rationale

The *API Design Rules Module: Geospatial* formalises as set of rules regarding:

1. How to encode geospatial data in request and response payloads.
2. How resource collections can be filtered by a given bounding box.
3. How to deal with different coordinate systems (CRS).

Functional

/core/modules/signing: Apply the signing module for signing payloads

Statement

The *API Design Rules Module: Signing* version 1.0.x *MUST* be applied when signing payloads.

> NOTE
>
> This rule does not dictate signing. Instead, it only applies in situations where there is a need for assurance of end to end message integrity and authenticity between client application and server application. In those situations, *API Design Rules Module: Signing* specifies how to sign.

Rationale

The *API Design Rules Module: Signing* formalises as set of rules regarding:

1. How to sign data in request and response payloads.

2. Which header to specify the signature.

Functional

/core/modules/encryption: Apply the encryption module for encrypting payloads

Statement

The *API Design Rules Module: Encryption* version 1.0.x *MUST* be applied when encrypting payloads.

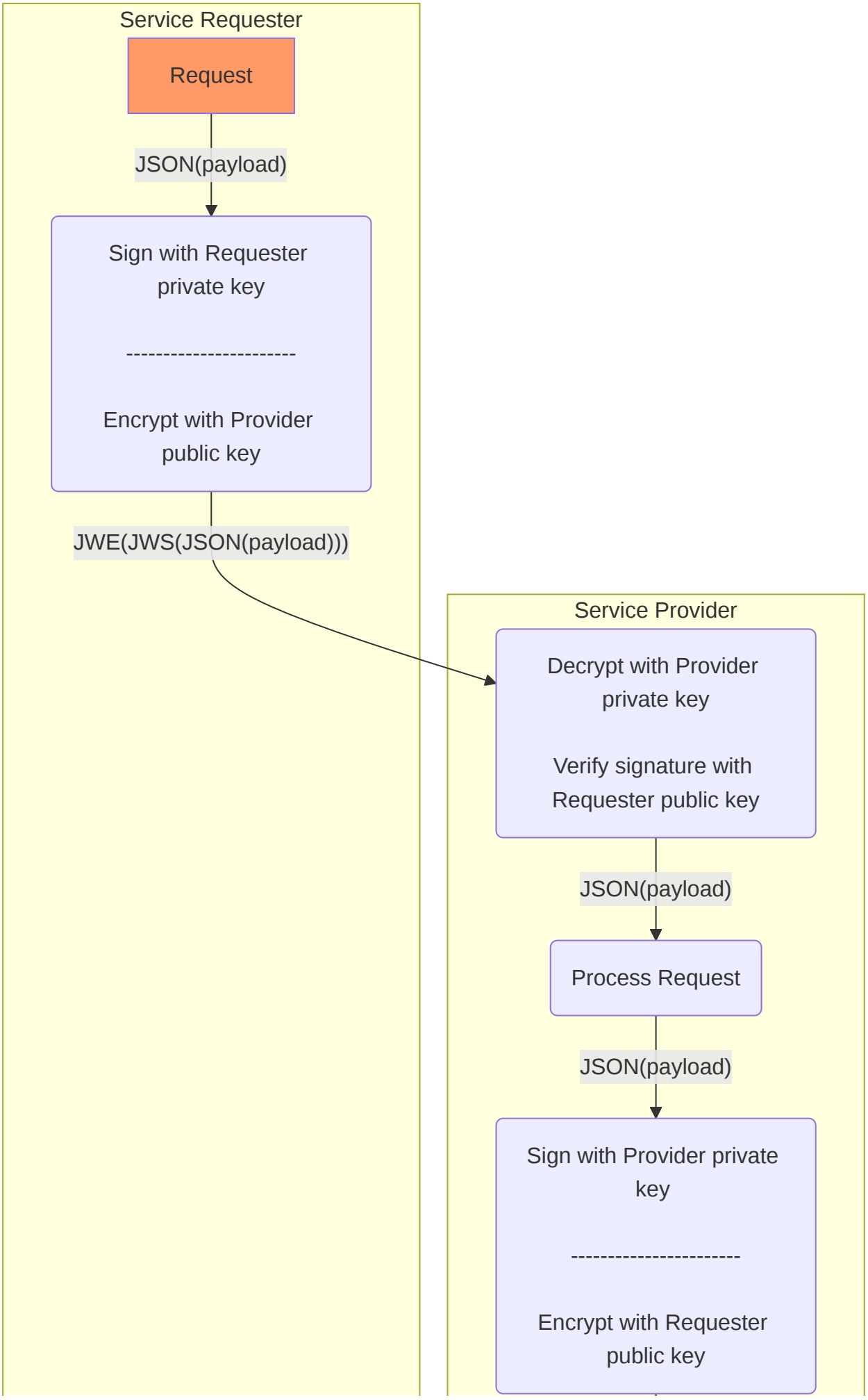
> NOTE
>
> This rule does not dictate encryption. Instead, it only applies in situations where there is a need for end to end message payload confidentiality between client application and server application. In those situations, *API Design Rules Module: Encryption* specifies how to encrypt.

Rationale

The *API Design Rules Module: Encryption* formalises as set of rules regarding:

1. How to encrypt data in request and response payloads.
2. The flow of operations between client and server.

If both the signing and encryption modules apply, use the following flow of operations:


Service Requester
Request
JSON(payload)
Sign with Requester private key

Encrypt with Provider public key
JWE(JWS(JSON(payload)))
Service Provider
Decrypt with Provider private key
Verify signature with Requester public key
JSON(payload)
Process Request
JSON(payload)
Sign with Provider private key

Encrypt with Requester public key

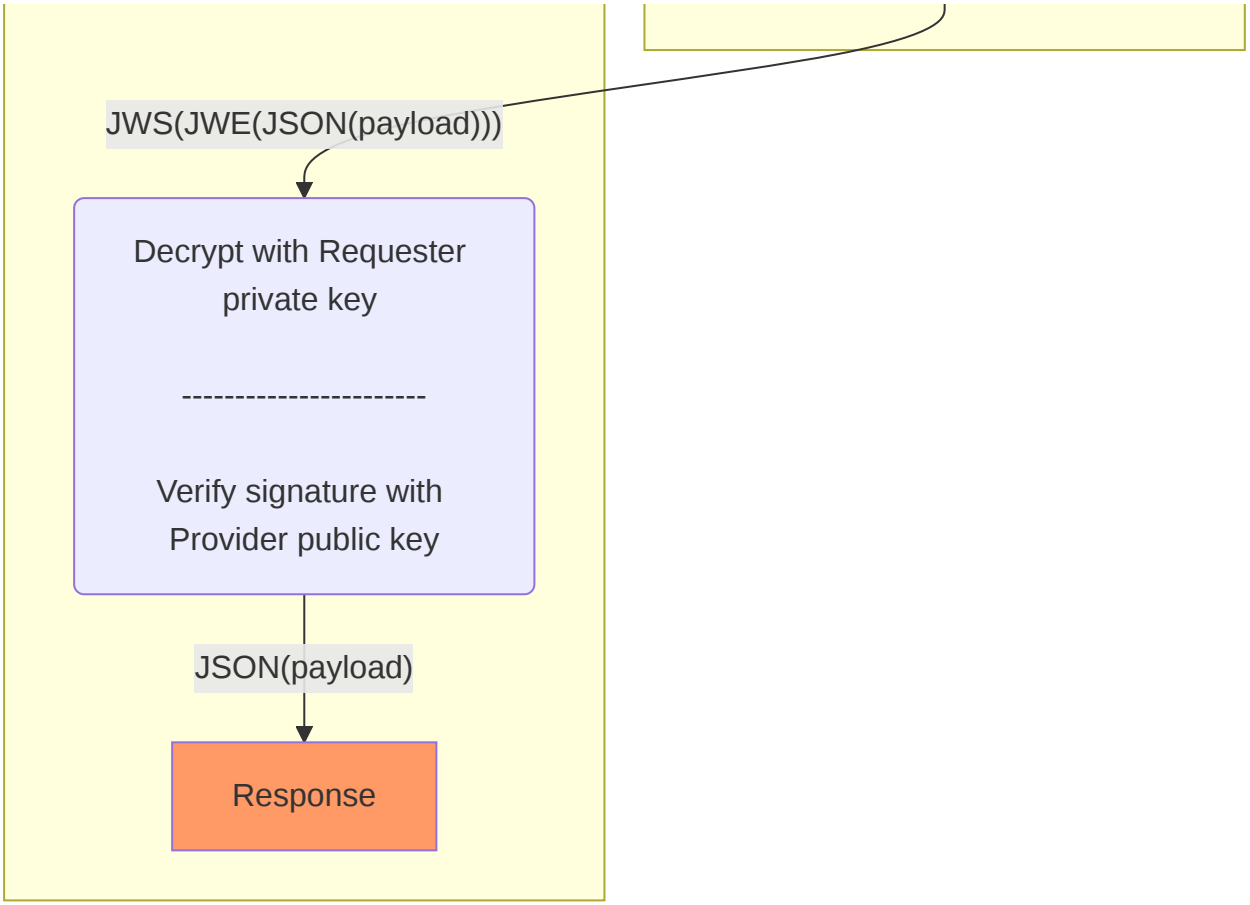


Figure 2 Signing in combination with encryption

§3. Glossary

Resource

A resource is an abstraction of a conceptual entity, identified by a globally unique URI, whose current state can be transferred to clients through one or more representations.

Singular resource

A singular resource is a resource that corresponds to a single conceptual entity (e.g. a building, person or event).

Collection resource

A collection resource is a resource that corresponds to a logical grouping of related resources and contains references (URIs) to the individual singular resources it includes. (e.g. a list of buildings).

URI

A URI [*RFC3986*] (Uniform Resource Identifier) is a string that identifies a resource. URIs are intended to be unique across the web, allowing resources to be unambiguously referenced.

OGC
: The Open Geospatial Consortium (OGC) is a consortium of experts committed to improving access to geospatial, or location information.

§A. Linter configuration

This section is non-normative.

▶ Details

Open the above details to inspect the linter configuration. You can also open the configuration file in separate tab

§B. References

§B.1 Normative references

[ADR-encryption]
: *API Design Rules Module: Encryption*. P. Haasnoot. Logius. URL: https://gitdocumentatie.logius.nl/publicatie/api/mod-encryption/1.0/

[ADR-GEO]
: *API Design Rules Module: Geospatial*. L. van den Brink, P. Bresters, P. van Genuchten, G. Mathijssen, M. Strijker. Geonovum. March 07, 2024. URL: https://gitdocumentatie.logius.nl/publicatie/api/mod-geo/

[ADR-signing]
: *API Design Rules Module: Signing*. P. Haasnoot. Logius. URL: https://gitdocumentatie.logius.nl/publicatie/api/mod-signing/1.0/

[ISO8601-1]
: *Date and time — Representations for information interchange — Part 1: Basic rules. ISO 8601-1:2019.*. International Organization for Standardization (ISO). 2019. ISO 8601-1:2019. URL: https://www.iso.org/standard/70907.html

[NCSC 2025]
: *Transport Layer Security (TLS) richtlijnen 2025-05*. NCSC. June 2025. URL: https://www.ncsc.nl/wat-kun-je-zelf-doen/documenten/publicaties/2025/juni/01/ict-beveiligingsrichtlijnen-voor-transport-layer-security-2025-05

[OPENAPIS]

OpenAPI Specification. Darrell Miller; Jason Harmon; Jeremy Whitlock; Marsh Gardiner; Mike Ralphson; Ron Ratovsky; Tony Tam; Uri Sarid. OpenAPI Initiative. URL: https://www.openapis.org/

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc2119/

[RFC3986]

Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://www.rfc-editor.org/info/rfc3986/

[RFC4647]

Matching of Language Tags. A. Phillips, Ed.; M. Davis, Ed. IETF. September 2006. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc4647/

[RFC5646]

Tags for Identifying Languages. A. Phillips, Ed.; M. Davis, Ed. IETF. September 2009. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc5646/

[RFC5789]

PATCH Method for HTTP. L. Dusseault; J. Snell. IETF. March 2010. Proposed Standard. URL: https://httpwg.org/specs/rfc5789.html

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc8174/

[RFC8252]

OAuth 2.0 for Native Apps. W. Denniss; J. Bradley. IETF. October 2017. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc8252/

[RFC9110]

HTTP Semantics. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed. IETF. June 2022. Internet Standard. URL: https://httpwg.org/specs/rfc9110.html

[RFC9457]

Problem Details for HTTP APIs. M. Nottingham; E. Wilde; S. Dalal. IETF. July 2023. Proposed Standard. URL: https://www.rfc-editor.org/info/rfc9457/

[RFC9557]

Date and Time on the Internet: Timestamps with Additional Information. U. Sharma; C. Bormann. IETF. April 2024. Proposed Standard. URL: https://www.rfc-editor.org/info/rfc9557/

[SemVer]

Semantic Versioning 2.0.0. T. Preston-Werner. June 2013. URL: https://semver.org

[SRI]

Subresource Integrity. Frederik Braun. W3C. 20 March 2026. W3C Working Draft. URL: https://www.w3.org/TR/sri-2/

§B.2 Informative references

[ISO-639-1]

Codes for the representation of names of languages -- Part 1: Alpha-2 code. ISO/TC 37/SC 2. ISO. 2002. International Standard. URL: https://www.iso.org/standard/22109.html

[ISO3166-1]

Codes for the representation of names of countries and their subdivisions — Part 1: Country code. International Organization for Standardization (ISO). August 2020. Published. URL: https://www.iso.org/standard/72482.html

[OAuth2]

The OAuth 2.0 Authorization Framework. D. Hardt. The Internet Engineering Task Force. October 2012. URL: https://tools.ietf.org/html/rfc6749

↥