Logius Standard - Definitive version

NLGov REST API Design Rules 2.0.0

Logius Standard
Definitive version March 07, 2024

This version:
https://gitdocumentatie.logius.nl/publicatie/api/adr/2.0.0

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/1.0

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

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

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

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

Abstract

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

§ 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*, and *SHOULD* 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.

Status of This Document

This is the definitive version of this document. Edits resulting from consultations have been applied.

Table of Contents

Organization / Committee	Version number	Official status	Date
Forum Standaardisatie	1.0	reported	15-10-2019
Forum Standaardisatie	1.0	'comply of explain' standard (mandatory open standard)	09-07-2020
Working group	2.0.0-rc.1	working version / final draft by 'Working Group'	05-09-2023
KP API Steering committee	2.0.0-rc.1	approved consultation version / adopted by 'KP API'	21-09-2024
MIDO programmeringstafel	2.0.0-rc.2	release candidate 2 / definitief concept	14-02-2024
MIDO PGDI Committee	2.0.0-rc.2	definitive version / approved by 'PGDI'	07-03-2024
Forum Standaardisatie	2.0.0-rc.2	reported	25-01-2024
Forum Standaardisatie	2.0.0	intake pending	18-04-2024
Forum Standaardisatie	2.0.0	definitive version / approved by Forum Standaardisatie	tbd

§ 1. Introduction

This section is non-normative.

§ 1.1 Goal

More and more governmental organizations 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 organizations designing new APIs, with the purpose of increasing developer experience (DX) and interoperability between APIs. Hopefully, many organizations 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)
Normative	Open API Specification (OAS)
Normative	NL GOV OAuth profiel
Normative	Digikoppeling REST API koppelvlak specificatie
Normative module	GEO module
Normative module	Transport Security module

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:

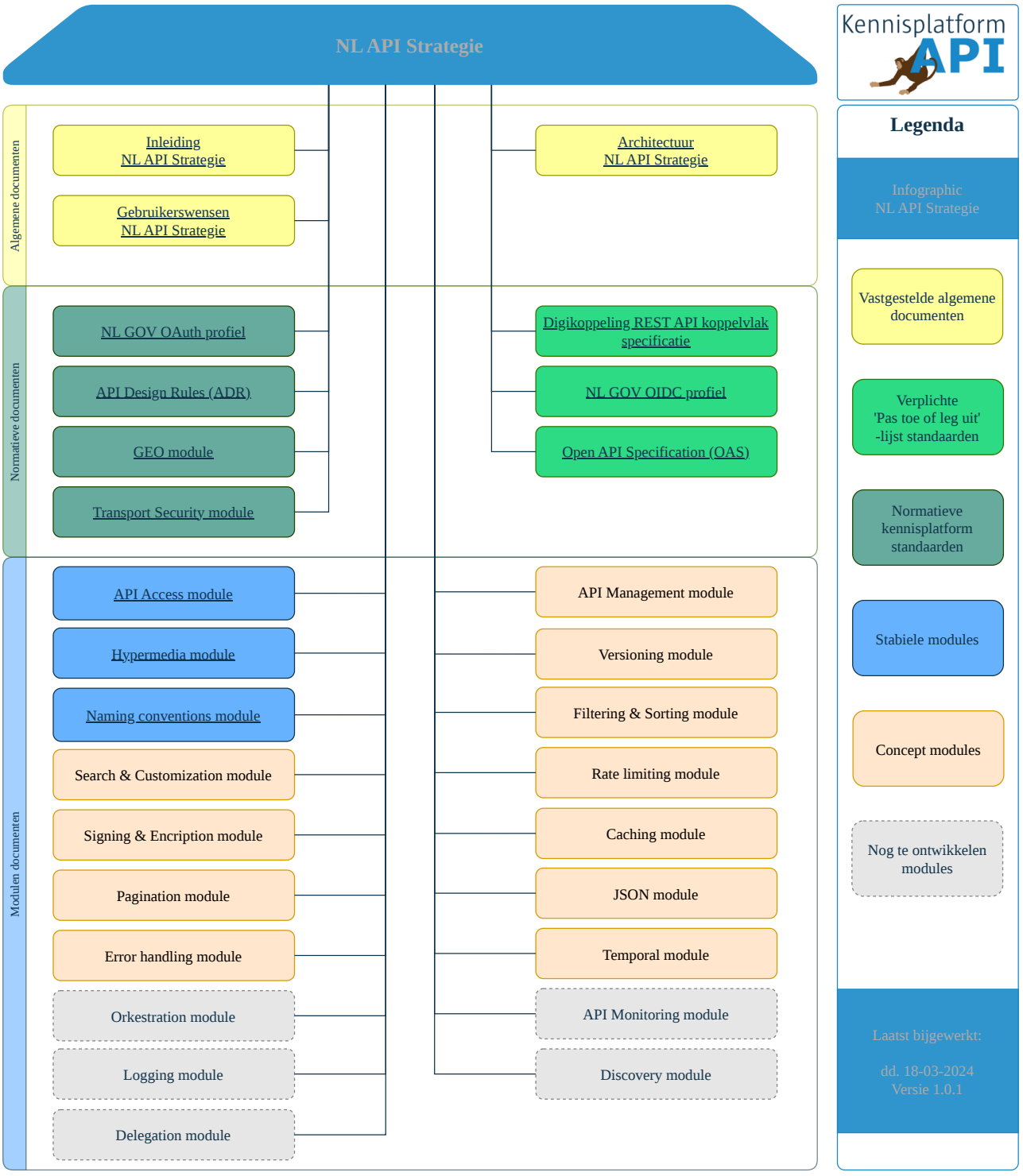

Figure 1 NL API Strategie Infographic

§ 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. Summary

2.1 Normative Design Rules

Design rules can be technical rules, which should be tested automatically and functional rules which should be considerd when designing and building the api.

2.1.1 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/http-safety: Adhere to HTTP safety and idempotency semantics for operations
- /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/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/geospatial: Apply the geospatial module for geospatial data

2.1.2 List of technical rules

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

- /core/http-methods: Only apply standard HTTP methods
- /core/doc-openapi: Use OpenAPI Specification for documentation
- /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-security: Apply the transport security module

§ 3. The core set of Design Rules

§ 3.1 Resources

The REST architectural style is centered around the concept of a resource. A resource is the key abstraction of information, where every piece of information is named by assigning a globally unique URI (Uniform Resource Identifier). Resources describe *things*, which can vary between physical objects (e.g. a building or a person) and more abstract concepts (e.g. a permit or an event).

Functional

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

Statement

Resources are referred to using nouns (instead of verbs) that are relevant from the perspective of the user of the API.

> 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.

Implications

Adherence to this rule needs to be manually verified.

A resource describing a single thing is called a singular resource. Resources can also be grouped into collections, which are resources in their own right and can typically be paged, sorted and filtered. Most often all collection members have the same type, but this is not necessarily the case. A resource describing multiple things is called a collection resource. Collection resources typically contain references to the underlying singular resources.

Functional

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

Statement

A collection resource represents multiple things.

Rationale

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

Example collection resources, describing a list of things:

```
https://api.example.org/v1/gebouwen
https://api.example.org/v1/vergunningen
```

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

Example singular resource, contained within a collection resource:

```
https://api.example.org/v1/gebouwen/3b9710c4-6614-467a-ab8
https://api.example.org/v1/vergunningen/d285e05c-6b01-45c3
```

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 singular resource describing the profile of the currently authenticated user:
>
> ```
> https://api.example.org/v1/gebruikersprofiel
> ```

Implications

Adherence to this rule needs to be manually verified.

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

The exact meaning of concepts is often lost in translation. Publishing an API for an international audience might also be a reason to define interfaces in English. Note that glossaries exist that define useful sets of attributes which *SHOULD* preferably be reused. Examples can be found at schema.org.

Implications

Adherence to this rule needs to be manually verified.

Technical

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

Statement

A URI *MUST* never 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 enforces API consumers to use the correct URI.

Rationale

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

> URI without a trailing slash (correct):
>
> ```
> https://api.example.org/v1/gebouwen
> ```
>
> URI with a trailing slash (incorrect):
>
> ```
> https://api.example.org/v1/gebouwen/
> ```

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: The API *MUST* meet the prerequisets to be tested. These include that an OAS file is publicly available, parsable, all $refs are resolvable and paths are defined.
- Step 2: Check if paths are present in the OpenAPI Specification.
- Step 3: Loop al paths and check if it ends with a slash ("/").
- Step 4: Check all paths with a get request and without parameters. They *SHOULD* resolve in HTTP 404.

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.

Implications

- 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 serializations which might be cumbersome for consumption.

§ 3.2 HTTP methods

Although the REST architectural style does not impose a specific protocol, REST APIs are typically implemented using HTTP [rfc7231].

Technical

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

Statement

Resources *MUST* be retrieved or manipulated using standard HTTP methods (GET/POST/PUT/PATCH/DELETE).

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 standardized 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.
`PATCH`	Update	Partially updates an existing resource. The request only contains the resource modifications instead of

Method	Operation	Description
		the full resource representation.
`DELETE`	Delete	Remove a resource with the given URI.

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific testscripts are published in the [ADR-Validator] repository.

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 [rfc7231] 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.x Path Item Object also supports these methods, except for `CONNECT`.
According to RFC 7231 4.1 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`, etc.
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 7231 6.5.5.

How to test

Test case 1:

- Step 1: The API *MUST* meet the prerequisites to be tested. These include that an OAS file is publicly available, parsable, all $refs are resolvable and paths are defined.
- Step 2: Send an HTTP GET or HEAD request to any of the endpoints with a definition of a GET operation mentioned in the OAS file. The server *MUST* respond with a HTTP status code other than `405 Method Not Allowed`.

Test case 2:

- Step 1: The API *MUST* meet the prerequisites to be tested. These include that an OAS file is publicly available, parsable, all $refs are resolvable, and paths are defined.
- Step 2: Send a request to the API with an optional HTTP method that is supported by the API. The server *MUST* respond with an HTTP status code other than `405 Method Not Allowed`.

Test case 3:

- Step 1: The API *MUST* meet the prerequisites to be tested. These include that an OAS file is publicly available, parsable, all $refs are resolvable, and paths are defined.
- Step 2: Send a request to the API with an optional HTTP method that is not supported by the API. The server *MUST* respond with an HTTP status code `405 Method Not Allowed`. The response *MUST* contain an `Allow` header with a list of supported methods for the target resource.

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

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

Rationale

The HTTP protocol [rfc7231] 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.

Implications

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.

§ 3.3 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 memorize previous requests. Session state must therefore reside entirely on the client.

To properly understand this constraint, it's 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's 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 doesn't have to memorize 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 analyzed 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 doesn't 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, no session state *MUST* reside on the server. Session state *MUST* therefore reside entirely on the client.

Implications

Adherence to this rule needs to be manually verified.

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.

§ 3.4 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.

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 doesn't 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 counterintuitive 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.

Implications

Adherence to this rule needs to be manually verified.

§ 3.5 Operations

Functional

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

Statement

Model resource operations 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 of 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. 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 declined in an earlier stage?). Furthermore, this requires a fine-grained authorization 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 declination 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 on the background after adding a review. This attribute *SHOULD* however be read-only.
3. In exceptional cases, the approaches above still don't 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.

Implications

Adherence to this rule needs to be manually verified.

§ 3.6 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.

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: The API *MUST* meet the prerequisets to be tested. These include that an OAS file is publicly available, parsable, all $refs are resolvable and paths are defined.
- Step 2: Check the specification type.
- Step 3: All references *MUST* be publicly resolvable, including the external references.

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.

Implications

Adherence to this rule needs to be manually verified.

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

Clients (such as Swagger UI or ReDoc) *MUST* be able 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 *SHOULD* provide its own OAS document(s).

> 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
> ```

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: The API *MUST* meet the prerequisets to be tested. These include that an OAS file (openapi.json) is publicly available, parsable, all $refs are resolvable and paths are defined.
- Step 2: The openapi.yaml *MAY* be available. If available it *MUST* contain yaml, be readable and parsable.
- Step 3: The openapi.yaml *MUST* contain the same OpenAPI Specification as the openapi.json.
- Step 4: The CORS header Access-Control-Allow-Origin *MUST* allow all origins.

§ 3.7 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

Implement well documented and timely communicated deprecation schedules.

Rationale

Managing change is important. In general, well documented and timely communicated 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.

Implications

Adherence to this rule needs to be manually verified.

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.

Implications

Adherence to this rule needs to be manually verified.

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.

An example of a base path for an API with current version 1.0.2:

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

```
version: '1.0.2'
```

```
servers:
                  - description: test environment
                  url: https://api.test.example.org/v1/
                  - description: production environment
                  url: https://api.example.org/v1/
```

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: The base path *MUST* contain a version number.
- Step 2: Each url of the server object of the OpenAPI Specification *MUST* include a version number.
- Step 3: The version in the OAS file *MUST* be the same as the version in the base path.

Functional

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

Statement

Publish a changelog.

Rationale

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

Implications

Adherence to this rule needs to be manually verified.

Technical

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

Statement

Implement Semantic 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 *MAY* only contain backwards compatible changes (e.g. the addition of an endpoint or an optional attribute).

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: The API *MUST* meet the prerequisets to be tested. These include that an OAS file (openapi.json) is publicly available, parsable, all $refs are resolvable and paths are defined.
- Step 2: In the open api specification the info and version object *MUST* be available.
- Step 3: The version *MUST* comply with Semantic Versioning.

Technical

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

Statement

Return the API-Version header.

Rationale

Since the URI only contains the major version, it's 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.

> An example of an API version response header:
>
> ```
> API-Version: 1.0.2
> ```

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here. The specific tests are published in the [ADR-Validator] repository.

How to test

- Step 1: A request to the base url *MUST* give a response and include the header "API-Version".
- Step 2: The value of the header "API-Version" *MUST* have a valid Semantic Versioning number.

§ 3.8 Transport Security

Transport security is essential to safeguard the confidentiality, integrity, and authenticity of data during its transmission.

Technical

/core/transport-security: Apply the transport security module

Statement

The *API Design Rules Module: Transport Security* *MUST* be applied.

Rationale

The *API Design Rules Module: Transport Security* formalizes three rules to apply to APIs:

1. Secure connections using TLS
2. No sensitive information in URIs
3. Use CORS to control access

Furthermore, the module describes best practices for security headers, browser-based applications, and other HTTP configurations. These best practices *MUST* be

considered and the considerations *SHOULD* be published in the API documentation. Transport security is the baseline for REST API resources and the data concerned is a vital asset of the government. The rules and best practices are considered the minimal security principles, concepts and technologies to apply.

Implications

This rule is included in the automatic tests on developer.overheid.nl. The source code of the technical test can be found here.

§ 3.9 Geospatial

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

Functional

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

Statement

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

Rationale

The *API Design Rules Module: Geospatial* formalizes 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).

Implications

Adherence to this rule needs to be manually verified.

§ 4. Glossary

Resource

A resource is the key abstraction of information, where every piece of information is identified by a globally unique URI.

Singular resource

A singular resource is a resource describing a single thing (e.g. a building, person or event).

Collection resource

A collection resource is a resource describing multiple things (e.g. a list of buildings).

URI

A URI [rfc3986] (Uniform Resource Identifier) is a globally unique identifier for a resource.

OGC

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

A. References

A.1 Normative references

[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-TS]

API Design Rules Module: Transport Security. . Kennisplatform API's. March 07, 2024. URL: https://gitdocumentatie.logius.nl/publicatie/api/mod-ts/

[ADR-Validator]

Technical ADR Validation rule testset 0.5.0. H. Stijns. Geonovum. November 2023. URL: https://gitlab.com/commonground/don/adr-validator/-/tree/v0.5.0/pkg/rulesets

[OPENAPIS]

OpenAPI Specification. Darrell Miller; Jeremy Whitlock; Marsh Gardiner; Mike Ralphson; Ron Ratovsky; Uri Sarid; Tony Tam; Jason Harmon. 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/rfc/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/rfc/rfc3986

[rfc7231]

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.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/rfc/rfc8174

[SemVer]

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

§ A.2 Informative references

[rfc5789]

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

↑