Logius - Standaard

API Design Rules (Nederlandse API Strategie IIa) 1.0

Logius Standard
Definitive version 09 juli 2020

Overzicht standaarden

This version:
https://publicatie.centrumvoorstandaarden.nl/api/adr/1.0

Latest published version:
https://publicatie.centrumvoorstandaarden.nl/api/adr/

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

Editors:
Frank Terpstra (Geonovum)
Jan van Gelder (Geonovum)

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 this 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.

Status of This Document

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

Het OBDO heeft op advies van het Forum Standaardisatie deze versie vastgesteld.

Table of Contents

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 three distinct documents.

1.5 Extensions

In addition to this (normative) document, a separate document has been written providing a set of informative extensions. This extensions document exists in a *latest published version* (*Gepubliceerde versie* in Dutch) and a *latest editors draft* (*Werkversie* in Dutch). The *latest editor's draft*s is actively being worked on and can be found on GitHub. It contains the most recent changes.

The documents can be found here:

- Extensions Gepubliceerde versie
- Extensions Werkversie

2. Summary

> NOTE
> Design rules have unique and permanent numbers. In the event of design rules being deprecated or restructured, they are removed from the list. Therefore, gaps in the sequence can occur. New design rules will always get a new and higher number.

2.1 Normative Design Rules

API-01: Adhere to HTTP safety and idempotency semantics for operations

API-02: Do not maintain session state on the server

API-03: Only apply standard HTTP methods

API-04: Define interfaces in Dutch unless there is an official English glossary available

API-05: Use nouns to name resources

API-06: Use nested URIs for child resources

API-10: Model resource operations as a sub-resource or dedicated resource

API-16: Use OpenAPI Specification for documentation

API-17: Publish documentation in Dutch unless there is existing documentation in English

API-18: Include a deprecation schedule when publishing API changes

API-19: Schedule a fixed transition period for a new major API version

API-20: Include the major version number in the URI

API-48: Leave off trailing slashes from URIs

API-51: Publish OAS document at a standard location in JSON-format

API-53: Hide irrelevant implementation details

API-54: Use plural nouns to name collection resources

API-55: Publish a changelog for API changes between versions

API-56: Adhere to the Semantic Versioning model when releasing API changes

API-57: Return the full version number in a response header

3. The Design Rules

3. The 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).

> **API-05**: Use nouns to name resources
>
> Because resources describe things (and thus not actions), 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

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.

> **API-54**: Use plural nouns to name collection resources
>
> Because a collection resource represents multiple things, 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-ab82-36822cf48db1
> > https://api.example.org/v1/vergunningen/d285e05c-6b01-45c3-92d8-5e19a946b66f
> > ```

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
> ```

API-04: Define interfaces in Dutch unless there is an official English glossary available

Since the exact meaning of concepts is often lost in translation, resources and the underlying attributes should be defined in the Dutch language unless there is an official English glossary available. 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.

API-48: Leave off trailing slashes from URIs

According to the URI specification [rfc3986], URIs may contain a trailing slash. However, for REST APIs this is considered as a bad practice since a URI including or excluding a trailing slash might be interpreted as different resources (which is strictly speaking the correct interpretation).

To avoid confusion and ambiguity, 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.

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

API-53: Hide irrelevant implementation details

An API should not expose implementation details of the underlying application. 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.

A few examples of implementation details:

- The API design should not necessarily be a 1-on-1 mapping of the underlying domain- or 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 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].

API-03: Only apply standard HTTP methods

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. Adhering to the HTTP specification is crucial, since HTTP clients and middleware applications rely on standardized characteristics. Therefore, resources must be retrieved or manipulated using standard HTTP methods.

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

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.

HTTP also defines other methods, e.g. `HEAD`, `OPTIONS` and `TRACE`. For the purpose of this design rule, these operations are left out of scope.

API-01: Adhere to HTTP safety and idempotency semantics for operations

> 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.
>
> 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.
>
> 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

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.

> 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).

> **API-02**: Do not maintain session state on the server
>
> 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. 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.
>
> > 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.

> **API-06**: Use nested URIs for child resources
>
> When having a child resource which can only exist in the context of a parent resource, the URI should be nested. In that 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.

3.5 Operations

API-10: Model resource operations as a sub-resource or dedicated resource

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.

In this design rule, approach 2 and 3 are preferred.

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.

API-16: Use OpenAPI Specification for documentation

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.

API-17: Publish documentation in Dutch unless there is existing documentation in English

In line with design rule API-04, the OAS document (e.g. descriptions and examples) should be written in Dutch. If relevant, you may refer to existing documentation written in English.

API-51: Publish OAS document at a standard location in JSON-format

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. 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
> ```

3.7 Versioning

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

API-56: Adhere to the Semantic Versioning model when releasing API changes

Version numbering must follow the Semantic Versioning [SemVer] model to prevent breaking changes when releasing new API versions. Versions are formatted using the `major.minor.patch` template. 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).

API-20: Include the major version number in the URI

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/
> ```

API-57: Return the full version number in a response header

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
> ```

API-55: Publish a changelog for API changes between versions

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

API-18: Include a deprecation schedule when deprecating features or versions

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.

API-19: Schedule a fixed transition period for a new major API version

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.

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.

A. References

A.1 Informative references

[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/

[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

[rfc5789]

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

[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

[SemVer]

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

↑