Abstract

This document contains a normative standard for designing APIs in the Dutch Public Sector.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current Geonovum publications and the latest revision of this document can be found via https://www.geonovum.nl/geo-standaarden/alle-standaarden(in Dutch).

Dit is een werkversie die op elk moment kan worden gewijzigd, verwijderd of vervangen door andere documenten. Het is geen door de werkgroep goedgekeurde consultatieversie.

Ten opzichte van de vorige versie van de API strategie (15-7-2019) zijn geen inhoudelijke aanpassingen gedaan, de wijzigingen zijn alleen redactioneel, het document is opgesplitst. De API designrules (voorheen Hoofdstuk 4) is een los document geworden. De inhoud van de API designrules in deze versie is volledig gelijk aan de inhoude die toegezonden is aan de expert groep van het forum standaardisatie.

Table of Contents

1. Introduction

Dit onderdeel is niet normatief.

This chapter contains a short introduction on the API Designrules standard.

1.1 Status of the API Designrules

THis version of the desgnrules 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 chapter 4 of the document "API Strategie voor de Nederlandse Overheid" which can be found at the following location: http://docs.geostandaarden.nl/api/vv-hr-API-Strategie-20190715

1.2 Authors

Despite the fact that only one author is mentioned in the list of authors, this document is the result of a collaborative effort by the API Designrules Working Group.

1.3 Reading Guide

This document is part of the "Nederlandse API Strategie"

Part	Description	Status	Link
I	General description of the API Strategy	Informative	https://docs.geostandaarden.nl/api/API-Strategie/
IIa	**Standard for designing APIs**	**Normative**	https://docs.geostandaarden.nl/api/API-Designrules/
IIb	Extension on the Standard for designing APIs	Informative	https://docs.geostandaarden.nl/api/API-Strategie-ext/

2. API designrules

> *This chapter aims to describe a set of design rules for the unambiguous provision of RESTful APIs (henceforth abbreviated as APIs). This achieves a predictable governments so developers can easily start consuming and combining APIs. Until now, this chapter does not include rules for other types of APIs, e.g. SOAP. In the addendum *API-principles*, the set of rules has been condensed into a number of core principles to keep in mind for the design and creation of APIs.*

2.1 Introduction

More and more government organisations implement and provide RESTful APIs. In many cases, these APIs provide access to data sets complementary to existing interfaces, e.g. SOAP and WFS. These APIs aim to be developer-friendly (see also paragraph 2.6 and chapter 3) and quick 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. A developer has to understand how every API can be used, but there should have to not be a difference in the technical implementation. The Knowlegde Platform APIs aims to provide a set of design rules or prinicples for APIs to align their technical operation across government organisations and to facilitate their implementation. This chapter describes the widely applicable set of design rules. Hopefully, government organisations will adopt these design rules in their own API strategies and provide feedback about exceptions and additions to subsequently improve these design rules.

Keep in mind, these design rules should be applied in the creation of an API only if the functionality described is desirable.

All paragraphs in this chapter, except for paragraph 4.5 are **Normative**. Paragraph 4.5 is **Informative**.

2.2 RESTful principles

The most important prinicple of REST is the seperation of the API in logical resources (*things*). The resources describe the information of the *thing*. These resources are manipulated using HTTP-requests and HTTP-operations. Each operation (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`) has a specific meaning.

> HTTP also defines operations, e.g. `HEAD`, `TRACE`, `OPTIONS` en `CONNECT`. In the context of REST, these operations are hardly ever used and have been excluded from the rest of this chapter.

Operation	CRUD	Description
`POST`	Create	Create resources that represent collections (i.e. `POST` adds a resource to a collection).
`GET`	Read	Retrieve a resource from the server. Data is only retrieved and not modified.
`PUT`	Update	Replace a specific resource. Is also used as a *create* " if the resource at the indicated identifier/URI does not exist yet.
`PATCH`	Update	Partially modify an existing resource. The request contains the data that have to be changed and the operations that modify the resource in the designated JSON merge patch format (RFC 7386).
`DELETE`	Delete	Remove the specific resource.

For each operation one has to specify whether it has to be *safe* and/or *idempotent*. This is important, because clients and middelware rely on this.

> **Safe (read-only)**
>
> Safe (read-only) in this case means that the semantics have been defined as read-only. This is important, because clients and middelware like to use caching.

> **Idempotent**
>
> Idempotent means that multiple, identical requests have the same effect as one request.

Operation	Safe	Idempotent
`POST`	No	No
`GET`, `OPTIONS`, `HEAD`	Yes	Yes
`PUT`	No	Yes
`PATCH`	No	Optional
`DELETE`	No	Yes

> API principle: operations are *Safe* and/or *Idempotent*

REST makes use of the client stateless server design principle derived from client server with the additional constraint that it is not allowed to maintain the state at the server. Each request from the client to the server has to contain all information required to process the request without the need to use state-information at the server.

> API principle: do not maintain state information at the server

2.2.1 What are resources?

A fundamental concept in every RESTful API is the resource. A resource is an object with a type, attributes, relation with other resources and a number of operations to modify them. Resources are referred to using nouns (not verbs) that are relevant from the perspective of the user of the API. Operations are actions applied to these resources. Operations are referred to using verbs that are relevant from the perspectie of the user of the API.

One can translate internal data models as-is to resources, but not by definition. The point is to not hide all relevant

implementation details. Some example resources are: *aanvragen* (applications), *activiteiten* (activities), *panden* (buildings), *rijksmonumenten* (national monuments), and *vergunningen* (permits).

Once the resources have been identified, one determines the operation that are applicable and how the API supports them. RESTful APIs perform CRUD (Create, Read, Update, Delete) operations using HTTP operations:

Request	Description
`GET /rijksmonumenten`	Retrieves a list of national monuments
`GET /rijksmonumenten/12`	Retrieves a specific national monument
`POST /rijksmonumenten`	Creates a new national monument
`PUT /rijksmonumenten/12`	Modifies national monument #12 completely

Request	Description
`PATCH /rijksmonumenten/12`	Modified national monument #12 partially
`DELETE /rijksmonumenten/12`	Deletes national monument #12

REST applies existing HTTP operations to implement functionality at one service endpoint. This removes the requirement for additional URI naming conventions and the URI structure remains clear.

> API principle: Only apply default HTTP operations

> API principle: Leave off trailing slashes from API endpoints

2.2.2 Language usage

Since the exact meaning of concepts are often lost in translation, resources and the underlying entities and attributes are defined in Dutch.

> API principle: Define interfaces in Dutch unless there is an official English glossary

2.2.3 Interface nomenclature: singular or plural?

Here, the *Keep It Simple Stupid* (KISS) rule is applicable. Although grammatically, it may feel wrong to request a single resource using the plural of the resource, it is a pragmatic choice to refer to endpoints consistently using plural. For the user it is much easier to not have to keep in mind singular and plural (*aanvraag/aanvragen, regel/regels*). Furthermore, this implementation is much more straightforward as most development frameworks are able to resolve both a single resource (`/aanvragen/12`) and multiple resources (`/aanvragen`) using one controller.

> API principle: Use plural nouns to indicate resources

2.2.4 How to deal with relations?

If a relation can only exist in the context of another resource (1 to n relation), then the dependent resource (child) can only be retrieved through the parent. The next example explains this. A status belongs to one application. Statuses can be retrieved through the endpoint `/aanvragen`:

retrieved through the endpoint `/aanvragen`.

Request	Description
`GET /aanvragen/12/statussen`	Retrieves a list of statuses of application #12
`GET /aanvragen/12/statussen/5`	Retrieves a specific status (#5) of application #12
`POST /aanvragen/12/statussen`	Creates a new status for application #12
`PUT /aanvragen/12/statussen/5`	Modifies status #5 of application #12 completely
`PATCH /aanvragen/12/statussen/5`	Modifies status #5 of application #12 partially
`DELETE /aanvragen/12/statussen/5`	Deletes status #5 of application #12

> API principle: Create relations of nested resources within the endpoint

In case of an n-to-n relation, there are various ways to retrieve a resource. The following requests respond identically:

Request	Description
`GET /aanvragen/12/activiteiten`	Retrieves a list of activities for application #12
`GET /activiteiten?aanvraag=12`	Retrieves a list of activities, filtered by application #12

In case of an n-to-m relation, the API supports the retrieval of individual resources anyway, at least providing the identifier of the related resource (relation). The user has to request the endpoint of the related resource (relation) to retrieve this one. This is referred to as *lazy loading*. The user decides whether to load the relation and when.

2.2.5 Custom representation

The user of an API does not alway require the complete representation (i.e. all attributes) of a resource. Providing the option to select the required attributes in the requests reduces network traffic (relevant for light-weight applications), simplifies the use of the API and makes it adjustable (fit-for-use). The query parameter `fields` supports this usage. The query parameter accepts a comma-separated list of field names. The result is a custom representation. For example, the following request retrieves sufficient information to show a sorted list of applications (*aanvragen*):

In the case of HAL, linked resources are embedded in the default representation. Applying the aforementioned `fields` parameter, the contents of the body can be customised as required.

```
GET /aanvragen?fields=id,onderwerp,aanvrager,wijzigDatum&status=open&sorteer=wijzigDatum
```

> API principle: Implement custom representation if supported

2.2.6 How to implement operations that do not fit the CRUD model?

There are resource operations that are not related to data manipulation (CRUD). Examples of this kind of operations are: changing the state (activate and deactivate) of a resource and marking (starring) a resource. Depending on the type of operation there are three approaches:

1. Restructure the operation to incorporate it into the resource. This approach applies if the operation does not require any parameters. For example, an activation operation can be assigned to a boolean field `geactiveerd` that can be modified by a `PATCH` to the resource.

2. Treat the operation as a sub-resource. For example, mark an application by `PUT /aanvragen/12/markeringen` and

remove a mark by `DELETE /aanvragen/12/markeringen`. To fully follow the REST principles, also provide the `GET` operation for this sub-resource.

3. Sometimes there is no logical way to link an operation to an existing resource. An example of this kind of operations is a search across multiple resources. This operation cannot be assigned to anyone specific resource. In this case, the creation of an independent service endpoint `/_zoek` is the most obvious solution. Use the imperative mood of a verb to distinguish these endpoints from *genuine* endpoints that use the indicative mood of a verb.

The Dutch API strategy prefers approach 2 and 3.

> API principle: Implement operations that do not fit the CRUD model as sub-resources

2.3 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 the implementation. Hiding the documentation in PDF documents and/or behind a login creates a barrier not only for developers but also for search engines. Specifications (documentation) are avaialble as Open API Specification (OAS) v3.0 or newer.

> API principle: Documentation conforms to OAS v3.0 or newer

> API principle: Publish documentation in Dutch unless there is existing documentation in English or there is an official English glossary available

The documentation should provide examples including full request and response cycles. Developers should be able to test (and perform) requests directly from within the documentation. Furthermore, each error should be described and labeled with a unique error code to trace errors.

Once an API is in production, the *contract* (interface) should not be changed without prior notice. The documentation should include a deprecation schedule and all details of the change. Changes should be published not only as a changelog on a publicly available blog but also through a mailing list, using the email addresses obtained when the API keys were issued.

> API principle: Include a deprecation schedule when publishing API changes

2.3.1 Best practice(s)

> API principle: Publish OAS at a base-URI in JSON-format

2.4 Versioning

APIs should always be versioned. Versioning facilitates the transition between changes. Old and new versions are offered during a limited (1 year) deprecation period. A maximum of 3 versions of the API should be supported. Users decide for themselves the moment they transition from the old to the new version of an API, as long as they do this prior to the end of the deprecation period.

> API principle: Allow for a (maximum) 1 year deprecation period to a new API version

> Provide old and new versions (maximum 3) of an API concurrently for a limited, maximum 1 year deprecation period.

The URI of an API should include the major version number only. This allows the exploration of multiple versions of an API in the browser.

The version number start at 1 and is raised with 1 for every major release that breaks the backwards compatibility of the interface. The minor and patch version numbers are always in the response header of the message in the `major.minor.patch` format.

The header (both request and response) should be implemented as follows:

HTTP header	Description
HTTP header	**Description**
`API-Version`	Indicates a specific API version in the context of a specific request. For example: `API-version: 1.2.56`

Using an optional request header one minor/patch version can be addressed. This means, that the client can send a request header to not only access versions v1 and v2 (the designated versions that are addressed in the URIs) but also access one *older* or *newer* version of API in the (pre-) production or acceptance test environment. For example, the following URIs point to the designated production release of the API that can be accessed in the URI:

`https://service.omgevingswet.overheid.nl/publiek/catalogus/api/raadplegen/v1`

`API-version: 1.0.2` (response header)

`https://service.omgevingswet.overheid.nl/publiek/catalogus/api/raadplegen/v2`

`API-version: 2.1.0` (response header)

Leaving off the request-header (`API-version: x.y.z`), one addresses always the *designated* production version. In case there is one other designated version available, e.g. v2.1.1, then it can be provided and addressed at the same base endpoint passing the correct request parameter:

`API-version: 2.1.1` (request header)

`https://service.omgevingswet.overheid.nl/publiek/catalogus/api/raadplegen/v2`

`API-version: 2.1.1` (response header)

Examples of backward compatible changes are the addition of an endpoint or an optional attribute to the payload.

> [API principle: Include only the major version number in the URI]
>
> One should only include the major version number. Minor version number and patch version number are included in the header of the message. Minor and patch versions have no impact on existing code, but major version do.

An API will never be fully stable. Change is inevitable. Managing change is important. In general, well documented and timely communicated deprecation schedules are the most importand for API users.

2.5 Extensions

Dit onderdeel is niet normatief.

The 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" 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

3. Normative API Principles

> NOTE
> API principles have unique numbers, depricated principles are removed from the list, new principles will get a new and higher nunmber. Thus gaps in the sequence can occur

3.1 API-01: Operations are *Safe* and/or *Idempotent*

Operations of an API are guaranteed to be *safe* and/or *idempotent* if that has been specified.

3.2 API-02: Do not maintain state information at the server

The client state is tracked fully at the client.

3.3 API-03: Only apply default HTTP operations

A RESTful API is an application programming interface that supports the default HTTP operations GET, PUT, POST, PATCH and DELETE.

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

Define resources and the underlying entities, fields and so on (the information model ad the external interface) in Dutch. English is allowed in case there is an official English glossary.

3.5 API-05: Use plural nouns to indicate resources

Names of resources are nouns and always in the plural form, e.g. *aanvragen* , *activiteiten*, *vergunningen*, even when it applies to single resources.

3.6 API-06: Create relations of nested resources within the endpoint

Preferrably, create relation within the endpoint if a relation can only exist with another resource (nested resource). In that case, the dependent resource does not have its own endpoint.

3.7 API-09: Implement custom representation if supported

Provide a comma-separated list of field names using the query parameter `fields` te retrieve a custom representation. In case non-existent field names are passed, a 400 Bad Request error message is returned.

3.8 API-10: Implement operations that do not fit the CRUD model as sub-resources

"Operations that do not fit the CRUD model are implemented as follows:

- Treat an operation as a sub-resource.
- Only in exceptional cases, an operator is implemented as an endpoint."

3.9 API-16: Use OAS 3.0 for documentation

Publish specifications (documentation) as Open API Specification (OAS) 3.0 or higher.

3.10 API-17: Publish documentation in Dutch unless there is existing documentation in English or there is an official English glossary available

Publish API documentation in Dutch. You may refer to existing documentation in Engelish and in case there is an official English glossary avaialble.

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

API changes and a deprecation schedule should be published not only as a changelog on a publicly available blog but also through a mailing list.

3.12 API-19: Allow for a maximum 1 year transition period to a new API version

Old and new versions (maximum 3) of an API should be provided concurrently for a limited, maximum 1 year transition period.

3.13 API-20: Include the major version number only in ihe URI

The URI of an API should include the major version number only. The minor and patch version numbers are in the response header of the message. Minor and patch versions have no impact on existing code, but major version do.

3.14 API-48: Leave off trailing slashes from API endpoints

URIs to retrieve collections of resources or individual resources don't include a trailing slash. A resource is only available at one endpoint/path. Resource paths end without a slash.

3.15 API-51: Publish OAS at the base-URI in JSON-format

Publish up-to-date documentation in the Open API Specification (OAS) at the publicly accessible root endpoint of the API in JSON format:

`https://service.omgevingswet.overheid.nl/publiek/catalogus/api/raadplegen/v1`

Makes the OAS relevant to v1 of the API available.

Thus, the up-to-date documentation is linked to a unique location (that is always concurrent with the features available in the API).

↥