ETSI GS NFV-SOL 015 V1.2.1 (2020-12)

Network Functions Virtualisation (NFV); Protocols and Data Models; Specification of Patterns and Conventions for RESTful NFV-MANO APIs

Disclaimer

The present document has been produced and approved by the Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
RGS/NFV-SOL015ed121

Keywords
API, data, MANO, model, NFV, protocol

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document defines patterns and conventions for RESTful NFV-MANO API specifications, gives recommendations on API versioning and provides an API specification template.

The present document defines provisions to be followed by the ETSI NFV Industry Specification Group (ISG) when creating RESTful NFV-MANO API specifications. The provisions do not apply to implementations.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI GS NFV-SOL 013: "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; Specification of common aspects for RESTful NFV MANO APIs".

[2] IETF RFC 5789: "PATCH Method for HTTP".

NOTE: Available from https://tools.ietf.org/html/rfc5789.

[3] IETF RFC 7396: "JSON Merge Patch".

NOTE: Available from https://tools.ietf.org/html/rfc7396.

[4] IETF RFC 7232: "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests".

NOTE: Available from https://tools.ietf.org/html/rfc7232.

[5] IETF RFC 3986: "Uniform Resource Identifier (URI): Generic Syntax".

NOTE: Available from https://tools.ietf.org/html/rfc3986.

[6] IETF RFC 7233: "Hypertext Transfer Protocol (HTTP/1.1): Range Requests".

NOTE: Available from https://tools.ietf.org/html/rfc7233.

[7] IETF RFC 7231: "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content".

NOTE: Available from https://tools.ietf.org/html/rfc7231.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GS NFV 003: "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV".

[i.2] PlantUML website.

NOTE: Available from http://plantuml.com/.

[i.3] PlantUML tool download.

NOTE: Available from https://sourceforge.net/projects/plantuml/files/plantuml.jar/download.

[i.4] PlantUML reference guide.

NOTE: Available from http://plantuml.com/PlantUML_Language_Reference_Guide.pdf.

[i.5] ETSI NFV repository of OpenAPI™ files.

NOTE: Available from https://forge.etsi.org/rep/nfv/.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI GS NFV 003 [i.1] apply.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Programming Interface
BWC	BackWard Compatible
CR	Change Request
CRUD	Create, Read, Update, Delete
CTC	Change Type Code
ETag	Entity Tag
ETSI	European Telecommunications Standards Institute
GS	Group Specification
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
IFA	InterFaces and Architecture
ISG	Industry Specification Group
JSON	JavaScript Object Notation

LCM	LifeCycle Management
MANO	MANagement and Orchestration
NBWC	Non-BackWard Compatible
NFV	Network Functions Virtualisation
REST	REpresentational State Transfer
RFC	Request For Comments
RPC	Remote Procedure Call
SOL	SOLutions
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
VNF	Virtualised Network Function

4 Conventions

4.1 Case conventions

The following case conventions for names and strings are available for potential use in the RESTful NFV-MANO API specifications.

1) ALLUPPER

All letters of a string shall be uppercase letters. Digits may be used except at the first position. Other characters shall not be used.

EXAMPLES 1 and 2:

a) MANAGEMENTINTERFACE

b) ETSINFVMANAGEMENT2

2) alllower

All letters of a string shall be lowercase letters. Digits may be used except at the first position. Other characters shall not be used.

EXAMPLES 3 and 4:

a) managementinterface

b) etsinfvmanagement2

3) UPPER_WITH_UNDERSCORE

All letters of a string shall be uppercase letters. Digits may be used except at the first position. Word boundaries shall be represented by the underscore "_" character. Other characters shall not be used.

EXAMPLES 5 and 6:

a) MANAGEMENT_INTERFACE

b) ETSI_NFV_MANAGEMENT_2

4) lower_with_underscore

All letters of a string shall be lowercase letters. Digits may be used except at the first position. Word boundaries shall be represented by the underscore "_" character. Other characters shall not be used.

EXAMPLES 7 and 8:

a) management_interface

b) etsi_nfv_management_2

5) UpperCamel

The string shall be formed by concatenating words as follows: Each word shall start with an uppercase letter (this implies that the string starts with an uppercase letter). All other letters shall be lowercase letters. Digits may be used except at the first position. Other characters shall not be used. Words that are abbreviations shall follow the same scheme (i.e. first letter uppercase, all other letters lowercase).

EXAMPLES 9 and 10:

a) ManagementInterface

b) EtsiNfvManagement2

6) lowerCamel

The string shall follow the provisions defined for UpperCamel with the following difference: The first letter shall be lowercase (i.e. the first word starts with a lowercase letter).

EXAMPLES 11 and 12:

a) managementInterface

b) etsiNfvManagement2

4.2 Conventions for URI parts

Based on IETF RFC 3986 [5], the parts of the URI syntax that are relevant in the context of the RESTful NFV-MANO API specifications are as follows:

- *Path*, consisting of *segments*, separated by "/" (e.g. segment1/segment2/segment3)
- *Query*, consisting of pairs of parameter name and value (e.g. ?org=nfv&group=sol)

1) Path segment naming

a) The path segments of a resource URI which represent a constant string shall use the "lower_with_underscore" convention.

EXAMPLE 1: vnf_instances

b) If a resource represents a collection of entities, the last path segment of that resource's URI shall be a word in plural.

EXAMPLE 2: …/prefix/API/1_0/users

c) For resources that are not task resources, the last path segment of the resource URI should be a (composite) noun.

EXAMPLE 3: …/prefix/API/1_0/users

d) For resources that are task resources, the last path segment of the resource URI should be a verb, or start with a verb.

EXAMPLES 4 and 5:

- …/vnf_instances/{vnfInstanceId}/scale
- …/vnf_instances/{vnfInstanceId}/scale_to_level

e) A variable name which represents a single path segment or a sequence of one or more path segments of a resource URI shall use the "lowerCamel" convention and shall be surrounded by curly brackets. A variable in the path part of a resource URI represents a single path segment unless it is explicitly specified that it represents zero, one or more path segments.

EXAMPLE 6: {vnfInstanceId}

f) Once a variable is replaced at runtime by an actual string, the string shall follow the rules for a single path segment or one or more path segments defined in IETF RFC 3986 [5]. IETF RFC 3986 [5] disallows certain characters from use in a path segment. Each actual RESTful NFV-MANO API specification shall define this restriction to be followed when generating values for variables that represent a single path segment or one or more path segments, or propose a suitable encoding (such as percent-encoding according to IETF RFC 3986 [5]), to escape such characters if they can appear in input strings intended to be substituted for a path segment variable.

2) Query naming

a) Parameter names in queries shall use the "lower_with_underscore" convention.

EXAMPLE 7: ?working_group=SOL

b) Variables that represent actual parameter values in queries shall use the "lowerCamel" convention and shall be surrounded by curly brackets.

EXAMPLE 8: ?working_group={chooseAWorkingGroup}

c) Once a variable is replaced at runtime by an actual string, the convention defined in 1.f. shall apply to that string.

4.3 Conventions for names in data structures

The following syntax conventions shall be obeyed when defining the names for attributes and parameters in the RESTful NFV-MANO API data structures.

a) Names of attributes/parameters shall be represented using the "lowerCamel" convention.

EXAMPLE 1: vnfName

b) Names of arrays (i.e. those with cardinality 1..N or 0..N) shall be plural rather than singular.

EXAMPLES 2 and 3: users, extVirtualLinks

c) Identifiers of information elements defined in the corresponding "interface and information model" design stage specification (typically, ETSI NFV-IFA specification) using the name syntax "xyzStructureId" shall be represented using the name "id."

NOTE: In the aforementioned specifications, the identifiers in scope of this convention are attributes with names using the syntax "xyzStructureId" which are embedded in an information element named "XyzStructure" for the purpose of identifying and/or externally referencing an instance of that information element.

d) Each value of an enumeration types shall be represented using the "UPPER_WITH_UNDERSCORE" convention.

EXAMPLE 4: NOT_INSTATIATED

e) The names of data types shall be represented using the "UpperCamel" convention.

EXAMPLES 5 and 6: ResourceHandle, VnfInstance

4.4 Conventions for documenting the API data model

4.4.1 Overview

A RESTful NFV-MANO API specification shall follow the provisions in this clause to document the API data model. clause X.6 in Annex A provides a related data model template.

The data model shall be defined using a tabular format as described in the following clauses. The name of the data type shall be documented appropriately in the heading of the clause and in the caption of the table, preferably as shown in clause X.6.2 in Annex A.

4.4.2 Structured data types

Structured data types shall be documented in tabular format, as in table 4.4.2-1 (one table per named data type).

Table 4.4.2-1: Template for a table defining a named structured data type

Attribute name	Data type	Cardinality	Description

The following provisions apply to the content of the table:

1) "Attribute name" shall provide the name of the attribute in lowerCamel.

2) "Data type" shall provide one of the following:

   a) The name of a **named data type** (structured, simple or enum) that is defined elsewhere in the present document where the data type is specified, or in a referenced document. In case of a referenced type from another document, a reference to the defining document should be included in the "Description" column unless included in a global clause.

   b) An indication of the definition of an **inlined** nested **structure**. In case of inlining a structure, the "Data type" column shall contain the string "Structure (inlined)", and all attributes of the inlined structure shall be prefixed with one or more closing angular brackets ">", where the number of brackets represents the level of nesting.

   c) An indication of the definition of an **inlined enumeration type**. In case of inlining an enumeration type, the "Data type" column shall contain the string "Enum (inlined)", and the "Description" column shall contain the allowed values and their meanings.

3) If the maximum cardinality is greater than one, "Data type" may indicate the format of the list of values. If it is an array, the format of that list may be indicated by using the key word "array(<type>)". If it is a map, the format shall be indicated by using the key word "map(<type>)". In both cases, <type> indicates the data type of the individual list entries. In case neither "map" nor "array" is given and the maximum cardinality is greater than one, "array" shall be assumed as default. The presence or absence of the indication of "array" shall be consistent between all data types in the scope of an API.

4) "Cardinality" defines the allowed number of occurrences, either as a single value, or as two values indicating lower bound and upper bound, separated by "..". A value shall be either a non-negative integer number or an uppercase letter that serves as a placeholder for a variable number (e.g. N).

5) "Description" describes the meaning and use of the attribute and may contain normative statements. In case of an inlined enumeration type, the "Description" column shall define the allowed values and (optionally) their meanings, as follows: "Permitted values:" on one line, followed by one paragraph of the following format for each value: "- <VAL>: <Meaning of the value>".

Typically, named data types are used for a structure that is intended to be re-used for referencing from many data types, or when modularizing big data types into smaller ones, e.g. for exposure using sub-resources. Inline data types are used if the same inlined data type only appears in one or very few data types.

Table 4.4.2-2 provides an example.

Table 4.4.2-2: Example of a structured data type definition

Attribute name	Data type	Cardinality	Description
type	FooBarType	1	Indicates whether this is a foo, boo or hoo stream.
entryIdx	array(UnsignedInt)	0..N	The index of the entry in the signalling table for correlation purposes, starting at 0.
fooBarType	Enum (inlined)	1	Signals the type of the foo bar. Permitted values: - BIG_FOOBAR: Signals a big foobar. - SMALL_FOOBAR: Signals a small foobar.
fooBarColor	Enum (inlined)	1	Signals the colour of the foo bar. Permitted values: - RED_FOOBAR: Signals a red foobar. - GREEN_FOOBAR: Signals a green foobar.
firstChoice	MyChoiceOneType	0..1	First choice. See note.
secondChoice	map(MyChoiceTwoType)	0..N	Second choice. See note.
nestedStruct	Structure (inlined)	0..1	A structure that is inlined, rather than referenced via an external type.
>someId	String	1	An identifier. The level of nesting is indicated by ">".
>myNestedStruct	array(Structure(inlined))	0..N	Another nested structure, one level deeper.
>>child	String	1	Child node at nesting level 2, indicated by ">>".
NOTE: One of "firstChoice" or at least one of "secondChoice" but not both shall be present.			

4.4.3 Simple data types

Simple data types shall be documented in tabular format, as in table 4.4.3-1 (one table row per simple data type).

Table 4.4.3-1: Simple data types

Type name	Description

The following provisions shall be applied to the content of the table:

1) "Type name" provides the name of the simple data type.

2) "Description" describes the meaning and use of the data type and may contain normative statements.

Table 4.4.3-2 provides an example.

Table 4.4.3-2: Example of a simple data type definition

Type name	Description
DozenInt	An integral number with a minimum value of 1 and a maximum value of 12.

4.4.4 Enumerations

Enumerations specify a set of valid values.

Enumeration types shall be documented in tabular format, as in table 4.4.4-1 (one table row per enumeration value, one table per enumeration type).

Table 4.4.4-1: Enumeration type

Enumeration value	Description

The following provisions shall be applied to the content of the table:

1) "Enumeration value" provides a mnemonic identifier of the enum value, optionally with an integer to which the value is mapped.

2) "Description" describes the meaning of the value and may contain normative statements.

Table 4.4.4-2 provides an example.

Table 4.4.4-2: Example of an enumeration type

Enumeration value	Description
ENABLED	The service is enabled.
DISABLED	The service is disabled.

4.4.5 JSON representation of the data model

This clause only applies to the JSON representation of the top-level named data types that define the structure of a resource representation or notification in the payload body of an HTTP request or response. In the template in Annex A, such data types are defined in the clause "Resource and notification data types".

When the data model is represented as JSON, all top-level attributes of the applicable resource or notification data type shall be mapped into the root level of the JSON object. Individual APIs may deviate from this rule, for example when re-using pre-existing data models. Such deviation shall be documented in the API specification.

The following example illustrates this convention. Assume the resource data type "Person" is defined as in table 4.4.5-1. The JSON representation is illustrated in the example.

Table 4.4.5-1: Example: Definition of the "PersonData" data type

Attribute name	Data type	Cardinality	Description
lastName	String	1	The surname of the person
firstName	String	1	The first name of the person.
address	Structure (inlined)	0..1	The address of the person, if known.
>street	String	1	The street
>number	Integer	1	The number of the house or apartment
>city	String	1	The city

EXAMPLE: JSON representation

```
{
    "lastName": "Doe",
    "firstName": "John",
    "address": {
        "street": "Route des Lucioles",
        "number": 650,
        "city": "Sophia Antipolis"
    }
}
```

5 Patterns

5.1 Pattern: Creating a resource (POST)

5.1.1 Description

This clause describes the "resource creation by POST" pattern, where the API consumer requests the API producer to create a new resource under a parent resource, which means that the URI identifying the created resource is under control of the API producer. This pattern shall be used for resource creation if the resource identifiers under the parent resource are managed by the API producer (see clause 5.2 for an alternative).

New resources are created on the origin server (API producer) as children of a parent resource. In order to request resource creation, the API consumer sends a POST request to the parent resource and includes a representation of the resource to be created. The API producer generates an identifier for the new resource that is unique for all child resources in the scope of the parent resource, and concatenates this with the resource URI of the parent resource to form the resource URI of the child resource. The API producer creates the new resource, and returns in a "201 Created" response a representation of the created resource along with a "Location" HTTP header that contains the resource URI of this resource.

Figure 5.1.1-1 illustrates creating a resource using POST.

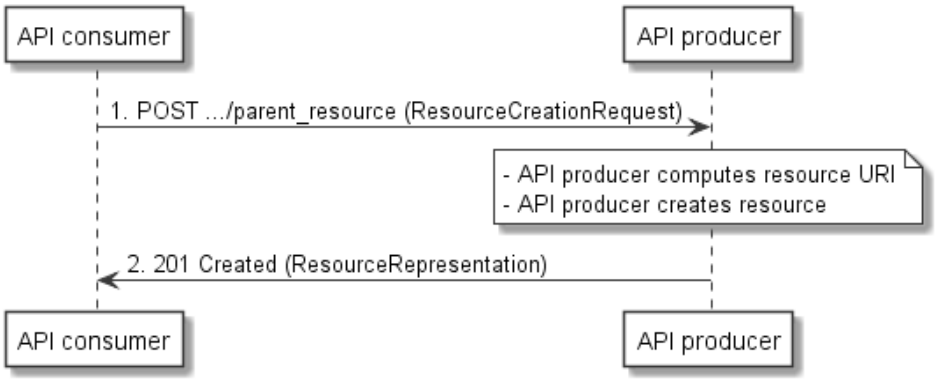

Figure 5.1.1-1: Resource creation flow by POST

5.1.2 Resource definition(s) and HTTP methods

The following resources are involved:

1) Parent resource: A container that can hold zero or more child resources.

2) Created resource: A child resource of a container resource that is created as part of the operation. The resource URI of the child resource is a concatenation of the resource URI of the parent resource with a string that is chosen by the API producer, and that is unique in the scope of the parent resource URI.

The HTTP method shall be POST.

5.1.3 Resource representation(s)

The entity body of the request shall contain a representation of the resource to be created. The entity body of the response shall contain a representation of the created resource.

NOTE: Compared to the entity body passed in the request, the entity body in the response may be different, as the resource creation process may have modified the information that has been passed as input, or generated additional attributes.

5.1.4 HTTP Headers

On success, the "Location" HTTP header shall be returned, and shall contain the URI of the newly created resource.

5.1.5 Response codes and error handling

On success, "201 Created" shall be returned. On failure, the appropriate error code shall be returned.

Resource creation can also be asynchronous in which case "202 Accepted" shall be returned instead of "201 Created". See clauses 5.11 and 5.12 for more details about asynchronous operations.

5.2 Pattern: Creating a resource (PUT)

5.2.1 Description

This clause describes the "resource creation by PUT" pattern, where the API consumer requests the API producer to create a new resource by providing the resource URI under which it expects the resource to be created, which means that the URI identifying the created resource is under control of the API consumer.

NOTE: The parent resource in this mode is implicit, i.e. it can be derived from the resource URI of the resource to be created by omitting the last path segment but is not provided explicitly in the request.

This pattern shall be used for resource creation if the resource identifiers under the parent resource are managed by the API consumer (see clause 5.1 for an alternative where the resource URI space is managed by the API producer). Typically, that alternative is safer as the API producer can prevent collisions. However, there are also valid use cases for creation by PUT, for instance when an API consumer manages different versions of a resource which are kept inside a "store" container, and the set of "store" containers is managed by the API producer i.e. "store" containers are created by POST.

In order to request resource creation, the API consumer sends a PUT request specifying the resource URI of the resource to be created and includes a representation of the resource to be created. The origin server (API producer) creates the new resource and returns in a "201 Created" response a representation of the created resource along with a "Location" HTTP header field that contains the resource URI of this resource.

Figure 5.2.1-1 illustrates creating a resource by PUT.

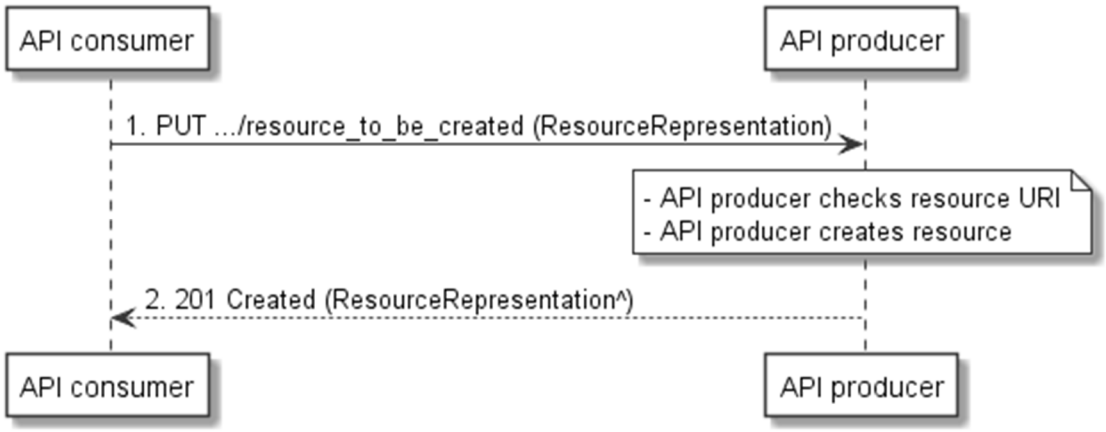

Figure 5.2.1-1: Resource creation by PUT

5.2.2 Resource definition(s) and HTTP methods

The following resource is involved:

1) Created resource: A resource that is created as part of the operation. The resource URI of that resource is passed by the API consumer in the PUT request.

The HTTP method shall be PUT.

5.2.3 Resource representation(s)

The entity body of the request shall contain a representation of the resource to be created. The entity body of the response shall contain a representation of the created resource.

NOTE: Compared to the entity body passed in the request (ResourceRepresentation in figure 5.2.1-1), the entity body in the response (ResourceRepresentation^ in figure 5.2.1-1) may be different, as the resource creation process may have modified the information that has been passed as input.

5.2.4 HTTP Headers

On success, the "Location" HTTP header shall be returned, and shall contain the URI of the newly created resource.

5.2.5 Response codes and error handling

The API producer shall check whether the resource URI of the resource to be created does not conflict with the resource URIs of existing resources (i.e. whether or not the resource requested to be created already exists).

In case the resource does not yet exist:

- Upon successful resource creation, "201 Created" shall be returned. Upon failure, the appropriate error code shall be returned.
- Resource creation can also be asynchronous in which case "202 Accepted" shall be returned instead of "201 Created". See clauses 5.11 and 5.12 for more details about asynchronous operations.

In case the resource already exists:

- If the "Update by PUT" operation is not supported for the resource, the request shall be rejected with "403 Forbidden", and a "ProblemDetails" payload should be included to provide more information about the error.
- If the "Update by PUT" operation is supported for the resource, interpret the request as an update request, i.e. the request shall be processed as defined in clause 5.6.

5.3 Pattern: Reading a resource (GET)

5.3.1 Description

This pattern obtains a representation of the resource, i.e. reads a resource, by using the HTTP GET method. For most resources, the GET method should be supported. An exception is task resources (see clause 5.8); these cannot be read.

See clause 5.3 for the related "query" pattern which allows to obtain a representation of a resource that has child resources and allows to influence the content of the representation using query parameters.

Figure 5.3.1-1 illustrates reading a resource.

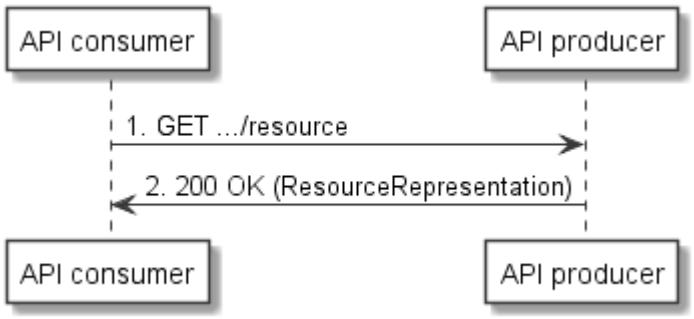

Figure 5.3.1-1: Flow of reading a resource

5.3.2 Resource definition(s) and HTTP methods

This pattern is applicable to any resource that can be read. The HTTP method shall be GET.

5.3.3 Resource representation(s)

The entity body of the request shall be empty; the entity body of the response shall contain a representation of the resource that was read, if successful.

5.3.4 HTTP Headers

There are no specific provisions for HTTP headers for this pattern.

5.3.5 Response codes and error handling

On success, "200 OK" shall be returned. On failure, the appropriate error code shall be returned.

5.4 Pattern: Querying a resource with filtering/selection (GET)

5.4.1 Description

This pattern influences the response of the GET method by passing resource URI parameters in the query part of the resource URI. Typically, this pattern is applied to container resources whose representation is a list of the child resources.

Typically, query parameters are used for controlling the content of the returned resource representation, including:

- restricting a set of objects to a subset, based on filtering criteria;
- reducing the content of the result (such as suppressing optional attributes).

Figure 5.4.1-1 illustrates querying a resource. The query parameters are just examples based on a subset of the parameters defined in ETSI GS NFV-SOL 013 [1].

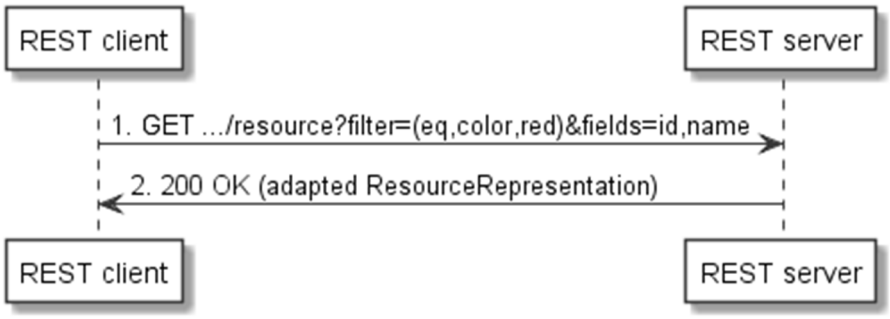

Figure 5.4.1-1: Flow of querying a resource

Two typical query mechanisms are *attribute-based filtering* and *attribute selectors*.

Attribute-based filtering allows to restrict the set of objects (i.e. the set of representations of child resources) in the returned representation to a subset that matches a set of filtering criteria applied to the values of the attributes of the objects. The mechanism is specified in clause 5.2 of ETSI GS NFV-SOL 013 [1].

Attribute selectors allow to restrict the set of attributes of the objects (i.e. the set of attributes in the representations of child resources) in the returned representation to a subset of the attributes as defined in the selector criteria. The mechanism is specified in clause 5.3 of ETSI GS NFV-SOL 013 [1].

5.4.2 Resource definition(s) and HTTP methods

This pattern is applicable to the GET request of any resource that has child resources. The query parameters are passed is URI query parameters. Passing the query parameters is optional. A default behaviour is defined in ETSI GS NFV-SOL 013 [1] for the case of the query parameters being absent.

5.4.3 Resource representation(s)

The entity body of the request shall be empty.

The entity body of the response shall be an array of representations of child resources, adapted to match the query parameters passed in the request.

5.4.4 HTTP Headers

There are no specific provisions for HTTP headers for this pattern.

5.4.5 Response codes and error handling

On success, "200 OK" shall be returned. If the query parameters are invalid, the "400 Bad Request" shall be returned. Otherwise, on failure, the appropriate error code shall be returned.

5.5 Pattern: Updating a resource (PATCH)

5.5.1 Description

The PATCH HTTP method (see IETF RFC 5789 [2]) is used to update a resource on top of the existing resource state with the changes described by the API consumer.

NOTE: There is an alternative to use PUT to update a resource which overwrites the resource completely with the representation passed in the payload body of the PUT request. PUT is not used for the update of structured data in RESTful NFV-MANO APIs but is used to upload raw files.

PATCH does not carry a representation of the resource in the entity body, but a document that instructs the API producer how to modify the resource representation. In the RESTful NFV-MANO API specifications, JSON Merge Patch (IETF RFC 7396 [3]) is used for that purpose, which defines fragments that are merged into the target JSON document.

Figure 5.5.1-1 illustrates updating a resource by PATCH.

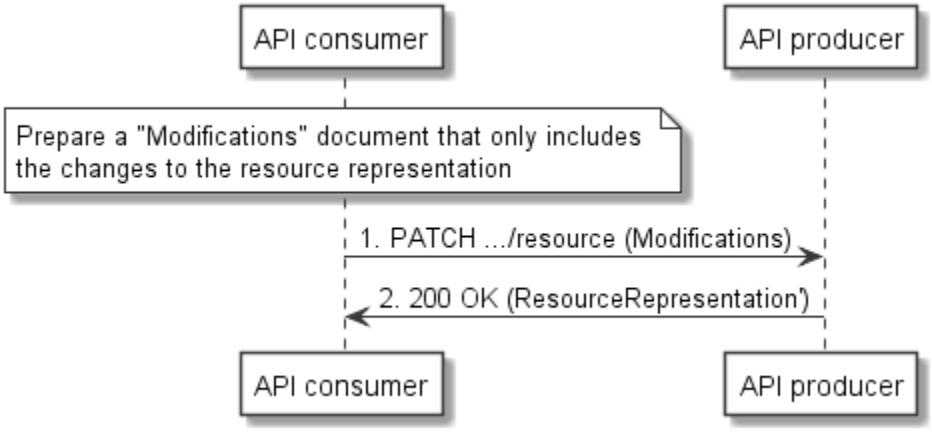

Figure 5.5.1-1: Basic resource update flow with PATCH

The approach illustrated above can suffer from the "lost update" phenomenon when concurrent changes are applied to the same resource. HTTP (see IETF RFC 7232 [4]) supports conditional requests to detect such a situation and to give the API consumer the opportunity to deal with it. For that purpose, each version of a resource gets assigned an "entity tag" (ETag) that is modified by the API producer each time the resource is changed. This information is delivered to the API consumer in the "ETag" (entity tag) HTTP header in HTTP responses. If the API consumer wishes that the API producer executes the PATCH only if the ETag has not changed since the time it has last read the resource (GET), the API consumer adds to the PATCH request the HTTP header "If-Match" with the ETag value obtained from the GET request. The API producer executes the PATCH request only if the ETag in the "If-Match" HTTP header matches the current ETag of the resource and responds with "412 Precondition Failed" otherwise. This is illustrated in figure 5.5.1-2.

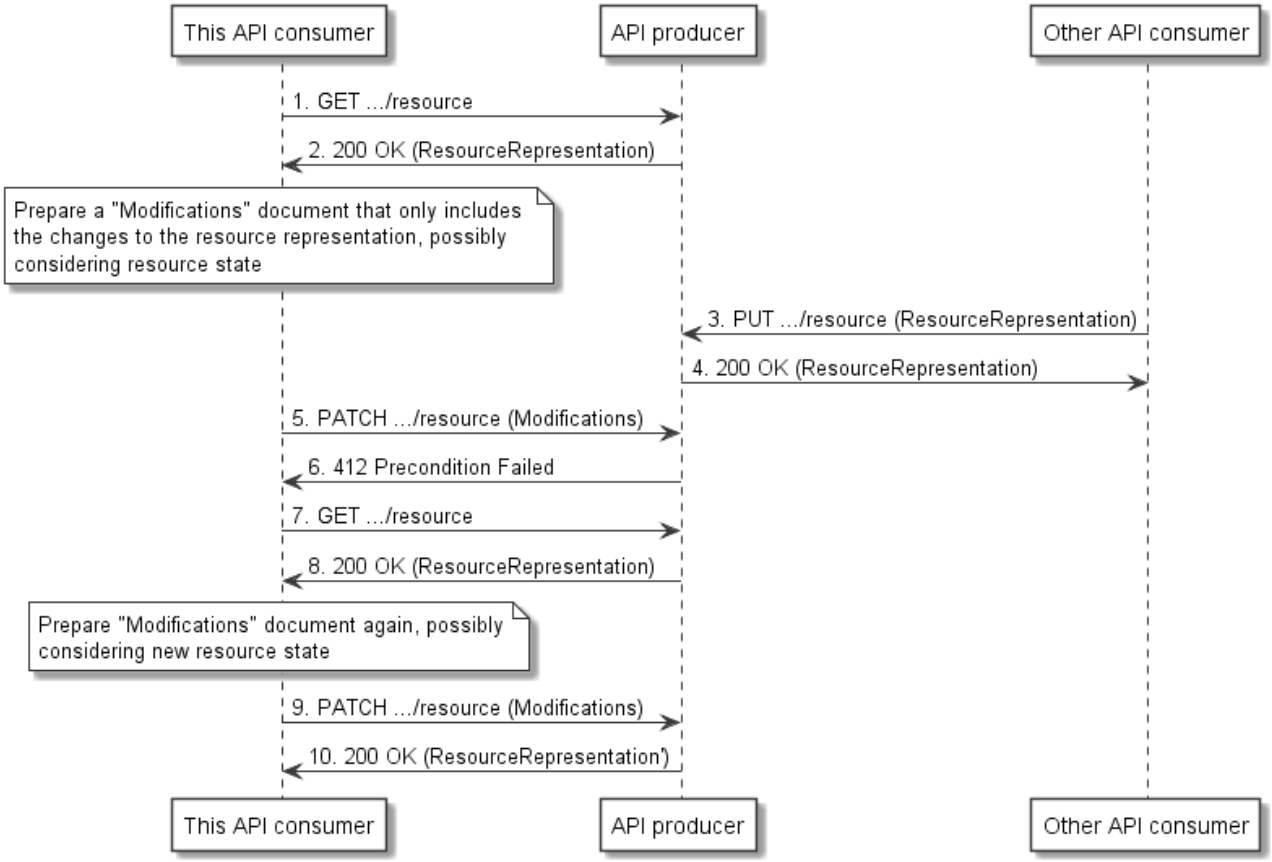

Figure 5.5.1-2: Resource update flow with PATCH, considering concurrent updates

5.5.2 Resource definition(s) and HTTP methods

This pattern is applicable to any resource that allows update by PATCH.

5.5.3 Resource representation(s)

The entity body of the PATCH request does not carry a representation of the resource, but a description of the changes.

Wherever applicable, it is recommended to use the JSON Merge Patch format defined by IETF RFC 7396 [3]. It needs to be pointed out that IETF RFC 7396 [3] does not define selective update of arrays, which means that the complete updated array always needs to be included.

However, in the RESTful NFV-MANO API specifications, many array entries are objects that contain an identifier attribute. In order to allow providing only deltas when modifying arrays of that type, the following custom PATCH payload format that allows to add and to update array entries is defined for use in the RESTful NFV-MANO APIs. The payload can be used instead of or in combination with JSON Merge Patch.

Assumptions:

1) "oldList" is the array to be modified (part of the resource representation) and "newList" is the array in the "Modifications" document (the payload body of the PATCH request) that contains the changes.

2) "oldEntry" is an entry in "oldList" and "newEntry" is an entry in "newList".

3) A "newEntry" has a "corresponding entry" if there exists an "oldEntry" that has the same content of an *identifier attribute* (typically named "id") as the "newEntry"; a "newEntry" has no corresponding entry if no such "oldEntry" exists.

4) In any array of "oldEntry" and "newEntry" structures, the content of the *identifier attribute* is unique (i.e. there are no two entries with the same content of the *identifier attribute*).

5) "DeleteIdList" is a parameter that can be supplied as part of the PATCH payload alongside "newList", containing values of the *identifier attribute* that represent entries of "oldList" to be removed from the list as part of the update.

Provisions:

1) For each "newEntry" in "newList" that has no corresponding entry in "oldList", the "oldList" array shall be modified by adding that "newEntry".

2) For each "newEntry" in "newList" that has a corresponding "oldEntry" in "oldList", the value of "oldEntry" shall be updated with the value of "newEntry" either by replacement or according to the rules of JSON Merge PATCH (see IETF RFC 7396 [3]).

3) For each entry in "deleteIdList", delete the entry in "oldList" that has the same content of the *identifier attribute* as the entry in "deleteIdList".

The entity body of the PATCH response may either be empty, may carry a representation of the updated resource, or may represent the performed modifications.

5.5.4 HTTP Headers

In the request, the "Content-type" HTTP header needs to be set to the content type registered for the format used to describe the changes, according to IETF RFC 7396 [3].

If conflicts and data inconsistencies are foreseen when multiple API consumers update the same resource, each API consumer should pass in the "If-Match" HTTP header of the PATCH request the value of the "ETag" HTTP header received in the response to the GET request.

5.5.5 Response codes and error handling

On success, either "200 OK" or "204 No Content" shall be returned. If the ETag value in the "If-Match" HTTP header of the PATCH request does not match the current ETag value of the resource, "412 Precondition Failed" shall be returned. Otherwise, on failure, the appropriate error code shall be returned.

Resource update can also be asynchronous in which case "202 Accepted" shall be returned instead of "200 OK". See clause 5.11 for more details about asynchronous operations.

When using the custom PATCH payload format defined in clause 5.5.3, it is an error that shall be rejected by "422 Unprocessable Entity" if a value of the *identifier attribute* in "deletedIdList" appears also in "newList".

5.6 Pattern: Updating a resource (PUT)

5.6.1 Description

The PUT HTTP method is used to update a resource with "replace" or "overwrite" semantics. That is, the new state of the resource is determined by the representation in the payload body of PUT; previous resource state is discarded by the API producer when executing the PUT request.

In the RESTful NFV-MANO APIs, PUT is typically used to upload bulk content; resources that have a structured representation (JSON) are updated with PATCH.

Figure 5.6.1-1 illustrates a typical flow.

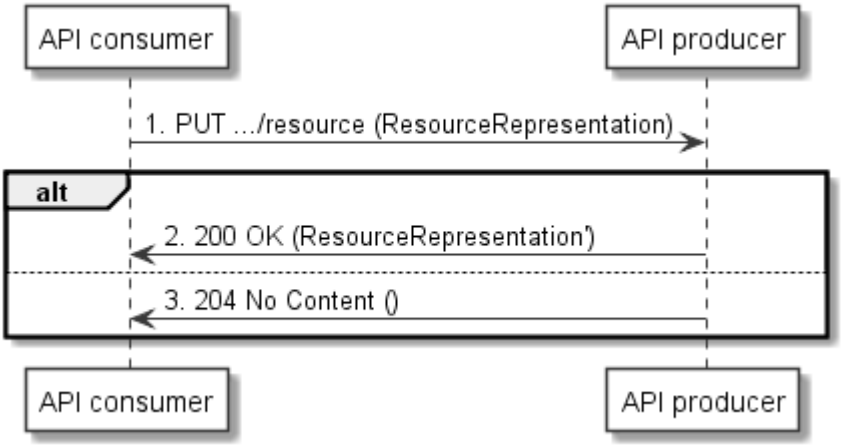

Figure 5.6.1-1: Basic resource update flow with PUT

5.6.2 Resource definition(s) and HTTP methods

This pattern is applicable to any resource that allows update by PUT.

5.6.3 Resource representation(s)

This pattern has no specific provisions for resource representations, other than the following note.

NOTE: Compared to the payload body passed in the request, the payload body in the response can be different, as the resource update process can have modified the information that has been passed as input.

5.6.4 HTTP headers

In the request, the "Content-type" HTTP header needs to be set to the content type of the payload provided in the request body.

5.6.5 Response codes and error handling

On success, either "200 OK" or "204 No Content" shall be returned. The "204 No Content" is recommended to be used in case of large resource representations.

Otherwise, on failure, the appropriate error code shall be returned.

Resource update can also be asynchronous in which case "202 Accepted" shall be returned instead of "200 OK". See clause 5.11 for more details about asynchronous operations.

5.7 Pattern: Deleting a resource (DELETE)

5.7.1 Description

The Delete pattern deletes a resource by invoking the HTTP DELETE method on that resource. After successful completion, the API consumer shall not assume that the resource is available any longer.

The response of the DELETE request is typically empty.

When a deleted resource is accessed subsequently by any HTTP method, typically the API producer responds with "404 Resource Not Found".

Figure 5.7.1-1 illustrates deleting a resource.

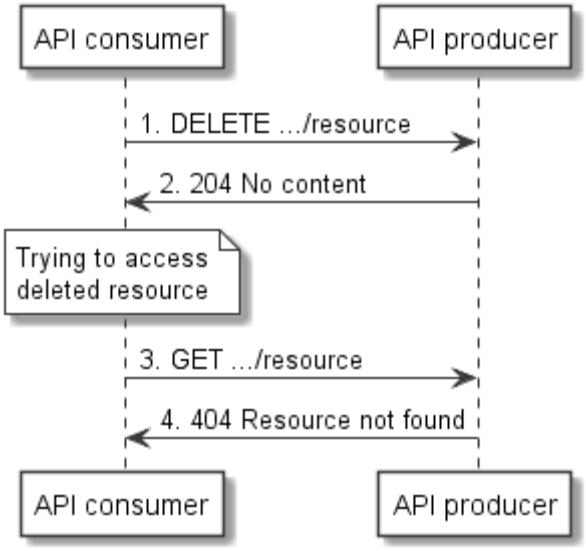

Figure 5.7.1-1: Resource deletion flow

5.7.2 Resource definition(s) and HTTP methods

This pattern is applicable to any resource that can be deleted. The HTTP method shall be DELETE.

5.7.3 Resource representation(s)

The entity body of the request shall be empty.

The entity body of the response is typically empty. Alternatively, the response can include the final representation of the resource prior to deletion.

5.7.4 HTTP Headers

No specific provisions for HTTP headers for this pattern.

5.7.5 Response codes and error handling

On success, the following applies for the response codes:

- In the typical case of returning an empty response body, "204 No Content" shall be returned.
- Alternatively, if returning in the response body the final representation of the resource prior to deletion, "200 OK" shall be returned.

On failure, the appropriate error code shall be returned.

If a deleted resource is accessed subsequently by any HTTP method, the API producer shall respond with "404 Resource Not Found".

Resource deletion can also be asynchronous in which case "202 Accepted" shall be returned instead of "204 No Content". See clause 5.11 for more details about asynchronous operations.

5.8 Pattern: Task resources

5.8.1 Description

In REST interfaces, the goal is to use only four operations on resources: Create, Read, Update, Delete (the so-called CRUD principle). However, in a number of cases, actual operations needed in a system design are difficult to model as CRUD operations, be it because they involve multiple resources, or that they are processes that modify a resource and that take a number of input parameters that do not appear in the resource representation. Such operations are modelled as "task resources".

A task resource is a child resource of a primary resource which is intended as an endpoint for the purpose of invoking a non-CRUD operation. That non-CRUD operation executes a procedure that modifies the state of that actual resource in a specific way or performs a computation and returns the result. Task resources are an escape means that allows to incorporate aspects of a service-oriented architecture or RPC endpoints into a RESTful interface.

The only HTTP method that is supported for a task resource is POST, with an entity body that provides input parameters to the process which is triggered by the request. Different responses to a POST request to a task resource are possible, such as "202 Accepted" (for asynchronous invocation), "200 OK" (to provide a result of a computation based on the state of the resource and additional parameters), "204 No Content" (to signal success but not return a result), or "303 See Other" (to indicate that a different resource was modified). The actual code used depends greatly on the actual system design.

5.8.2 Resource definition(s) and HTTP methods

A task resource that models an operation on a particular primary resource is often defined as a child resource of that primary resource. The name of the resource should be a verb that indicates which operation is executed when sending a POST request to the resource.

EXAMPLE: …/call_sessions/{sessionId}/call_participants/{participantId}/transfer.

The HTTP method shall be POST.

5.8.3 Resource representation(s)

The entity body of the POST request does not carry a resource representation but contains input parameters to the process that is triggered by the POST request.

5.8.4 HTTP Headers

In case the task resource represents an operation that is asynchronous, the provisions in clause 5.11 apply.

In case the operation modifies a primary resource and the response contains the "303 See Other" response code, the "Location" HTTP header points to the primary resource.

5.8.5 Response codes and error handling

The response code returned depends greatly on the actual operation that is represented as a task resource, and may include the following:

- For long-running operations, "202 Accepted" is returned. See clause 5.11 for more details about asynchronous operations.
- If the operation modifies another resource, "303 See Other" is returned.
- If the operation returns a computation result, "200 OK" is returned.
- If the operation returns no result, "204 No Content" is returned.

On failure, the appropriate error code is returned.

5.9 Pattern: Subscribe-Notify

5.9.1 Description

A common task in distributed systems is to keep all involved components informed of changes that appear in a particular component at a particular time. A common approach to spread information about a change is to distribute notifications about the change to those components that have indicated interest earlier on. Such pattern is known as Subscribe/Notify. In REST which is request-response by design, meaning that every request is initiated by the API consumer, specific mechanisms need to be put into place to support the delivery of notifications initiated by the API producer. The basic principle is that the API consumer exposes a (lightweight) HTTP server towards the API producer. The (lightweight) HTTP server only needs to support a small subset of the HTTP functionality - namely the POST and GET methods, the "204 No Content" success response code plus the relevant error response codes, and, if applicable, authentication/authorization. The API consumer exposes the (lightweight) HTTP server in a way that it is reachable via TCP by the API producer.

To manage subscriptions, the API producer needs to expose a container resource under which the API consumer can request the creation/deletion of individual subscription resources. Those resources typically define criteria of the subscription. Subscription resources can also be read using GET. Termination of a subscription is done by a DELETE request.

To receive notifications, the API consumer exposes one or more notification endpoints (callback URIs) on which it can receive POST and GET requests. When creating a subscription, the API consumer informs the API producer about the notification endpoint (callback URI) to which the API producer will later deliver notifications related to that particular subscription.

To deliver notifications, the API producer includes the actual notification payload in the entity body of a POST request and sends that request to the notification endpoint(s) (callback URI(s)) it knows from the subscription(s). The API consumer acknowledges the receipt of the notification with "204 No Content".

Figure 5.9.1-1 illustrates the management of subscriptions. Figure 5.9.1-2 illustrates the delivery of a notification.

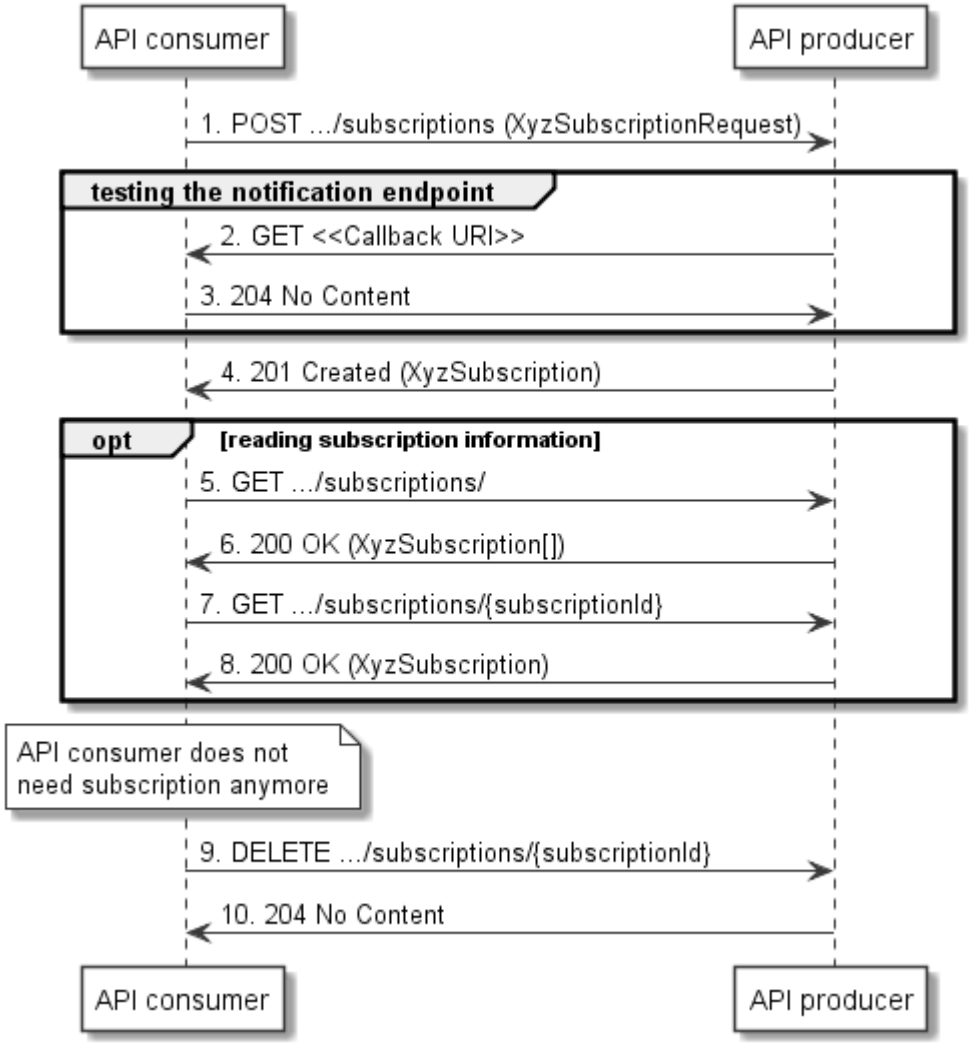

Figure 5.9.1-1: Management of subscriptions

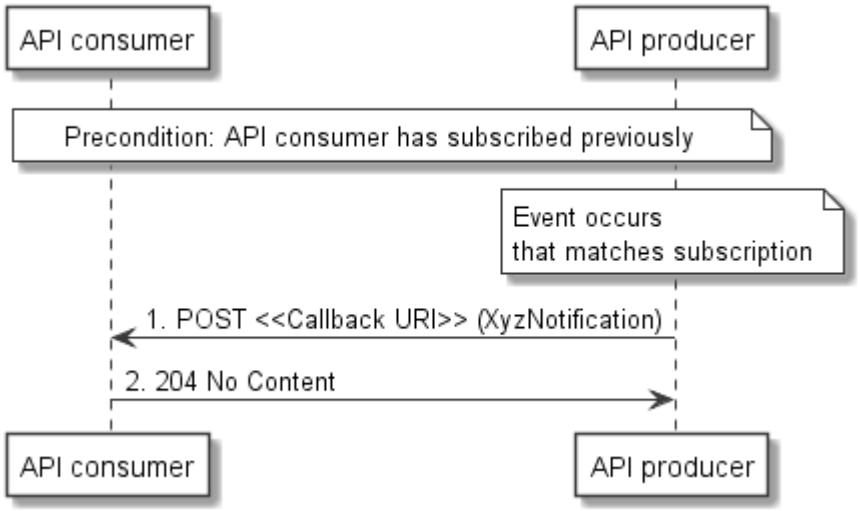

Figure 5.9.1-2: Delivery of notifications

5.9.2 Resource definition(s) and HTTP methods

The following resources are involved:

1) Subscriptions resource: A resource that can hold zero or more subscription resources as child resources.

2) Individual subscription resource: A resource that represents a subscription.

3) A notification endpoint (callback URI) that is exposed by the REST API consumer to receive the notifications.

The HTTP method to create a new individual subscription resource inside the subscriptions resource shall be POST.

The HTTP method to terminate a subscription by removing an individual subscription resource shall be DELETE.

The HTTP method to read the subscriptions resource and to read individual subscription resources shall be GET.

The HTTP method used by the API producer to deliver notifications shall be POST.

The HTTP method used by the API producer to test the notification endpoint shall be GET.

5.9.3 Resource representation(s)

The following provisions are applicable to the representation of an individual subscription resource:

- It shall contain the URI of a notification endpoint (callback URI) that the API consumer exposes to receive notifications. That URI shall be provided by the API consumer on subscription.
- It should contain criteria that allow the API producer to determine the events about which the API consumer wishes to be notified. APIs may deviate from this recommendation for instance when there are just one or two event types, or when it is essential that the API consumer is informed about all events.

The following provisions are applicable to the representation of a notification:

- It shall contain a reference to the related subscription, using the "_links" attribute (see pattern for links, clause 5.10).
- It shall contain information about the event.

5.9.4 HTTP Headers

No specific provisions are applicable here.

5.9.5 Response codes and error handling

On successful subscription creation, "201 Created" shall be returned.

On successful subscription deletion, "204 No Content" shall be returned.

On successfully reading an "individual subscription" resource or querying the "subscriptions" resource, "200 OK" shall be returned.

On successful notification delivery or notification endpoint test, "204 No Content" shall be returned by the API consumer.

On failure, the resource shall not be created and the appropriate error code shall be returned. For the error condition that the test of the notification endpoint has failed, "422 Unprocessable Entity" shall be returned.

5.10 Pattern: Links

5.10.1 Description

It is commonly seen as good adherence to RESTful principles that resources link to other resources, allowing the API consumer to traverse the resource space. Such principle is also known as "hypermedia controls" or HATEOAS (Hypermedia As The Engine Of Application State). This clause describes a pattern for hyperlinks.

Hyperlinks to other resources should be embedded into the representation of resources where applicable. For each hyperlink, the target URI of the link and information about the meaning of the link shall be provided. Knowing the meaning of the link (typically conveyed by the name of the object that defines the link, or by an attribute such as "rel") allows the API consumer to automatically traverse the links to access resources related to the actual resource, in order to perform operations on them.

5.10.2 Resource definition(s) and HTTP methods

Links can be applicable to any resource and any HTTP method.

5.10.3 Resource representation(s)

Links are communicated in the resource representation. Links that occur at the same level in the representation shall be bundled in a JSON object, named "_links" which should occur as the first object at a particular level.

Links shall be embedded in that JSON object as contained objects. The name of each contained object defines the semantics of the particular link. The content of each link object shall be an object named "href" of type string, which defines the target URI the link points to. The link to the actual resource shall be named "self" and shall be present in every resource representation if links are used in that API.

As an example, the "_links" portion of a resource representation is shown that represents paged information. Figure 5.10.3-1 illustrates the JSON schema and figure 5.10.3-2 illustrates the JSON object.

```
"properties": {
    "_links": {
        "required": ["self"],
        "type": "object",
        "description": "Link relations",
        "properties": {
            "self": {
                "$ref": "#/definitions/Link"
            },
            "prev": {
                "$ref": "#/definitions/Link"
            },
           "next": {
                "$ref": "#/definitions/Link"
            }
        }
    }
},
"definitions": {
    "Link" : {
        "type": "object",
        "properties": {
            "href": {"type": "string"}
        },
       "required": ["href"]
    }
}
```

Figure 5.10.3-1: JSON schema fragment for an example "_links" element

```
{
    "_links": {
        "self": { "href": "http://api.example.com/my_api/v1/pages/127" },
        "next": { "href": "http://api.example.com/my_api/v1/pages/128" },
        "prev": { "href": "http://api.example.com/my_api/v1/pages/126" }
    }
}
```

Figure 5.10.3-2: JSON fragment for an example "_links" element

5.10.4 HTTP Headers

There are no specific provisions with respect to HTTP headers for this pattern.

5.10.5 Response codes and error handling

There are no specific provisions with respect to response codes and error handling for this pattern.

5.11 Pattern: Asynchronous invocation with monitor

5.11.1 Description

Certain operations, which are invoked via a RESTful interface, trigger processing tasks in the underlying system that may take a long time, from minutes over hours to even days. In this case, it is inappropriate for the API consumer to keep the HTTP connection open to wait for the result of the response - the connection will time out before a result is delivered. For these cases, asynchronous operations are used. The idea is that the operation immediately returns the provisional response "202 Accepted" to indicate that the request was understood, can be correctly marshalled in, and processing has started. The API consumer can check the status of the operation by polling; additionally or alternatively, the subscribe-notify mechanism (see clause 5.9) can be used to provide the result once available. The progress of the operation is reflected by a monitor resource.

Figure 5.11.1-1 illustrates asynchronous operations with polling. After receiving an HTTP request that is to be processed asynchronously, the API producer responds with "202 Accepted" and includes in a specific "Location" HTTP header a data structure that points to a monitor resource which represents the progress of the processing operation. The API consumer can then poll the monitor resource by using GET requests, each returning a data structure with information about the operation, including the (application-specific) processing status such as "processing", "success" and "failure". In the example, the initial status is set to "processing". Eventually, when the processing is finished, the status is set to "success" (for successful completion of the operation) or "failure" (for completion with errors). Typically, the representation of a monitor resource will include additional information, such as information about the error cause if the operation was not successful.

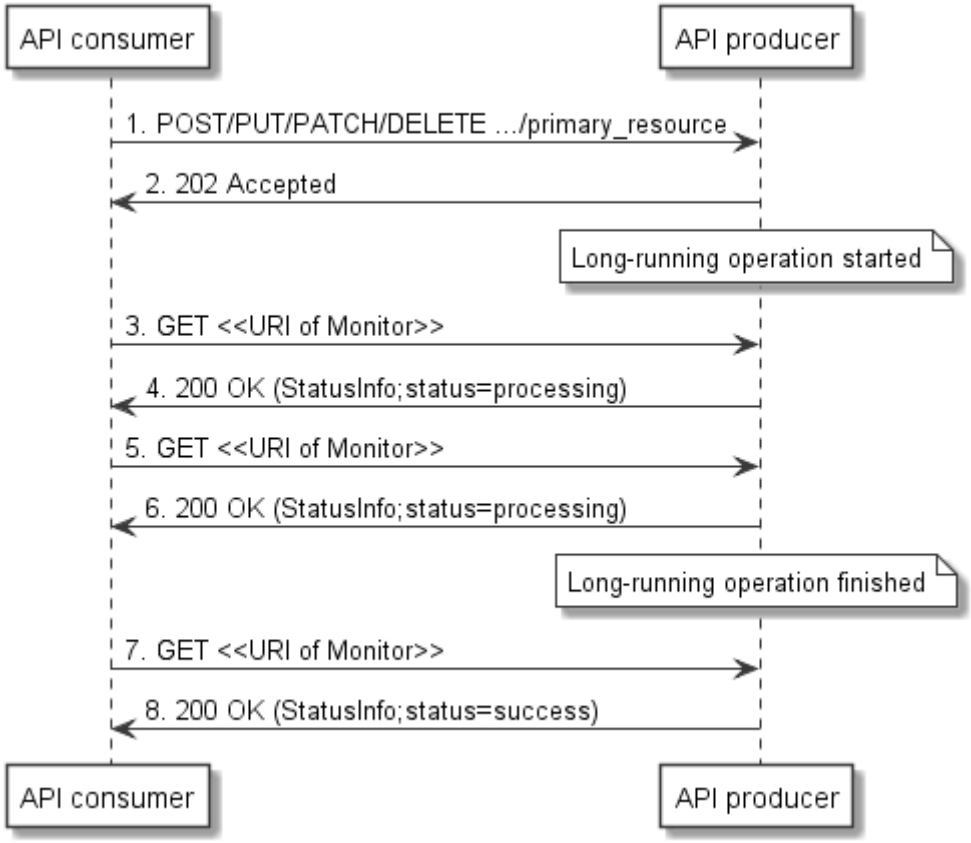

Figure 5.11.1-1: Asynchronous operation flow - with polling

Figure 5.11.1-2 illustrates asynchronous operations with subscribe/notify. Before an API consumer issues any request that might be processed asynchronously, it subscribes for monitor change notifications. Later, after receiving an HTTP request that is to be processed asynchronously, the API producer responds with "202 Accepted" and includes in the "Location" HTTP header a data structure that points to a monitor resource which represents the progress of the processing operation. The API consumer can now wait for receiving a notification about the operation finishing, which will change the status of the monitor. Once the operation is finished, the API producer will send to the API consumer a notification with a structure in the entity body that typically includes the status of the operation (e.g. "success" or "failure"), a link to the actual monitor affected, a link to the resource that is modified by the asynchronous operation and application-specific further information. The API consumer can then read the monitor resource to obtain additional information.

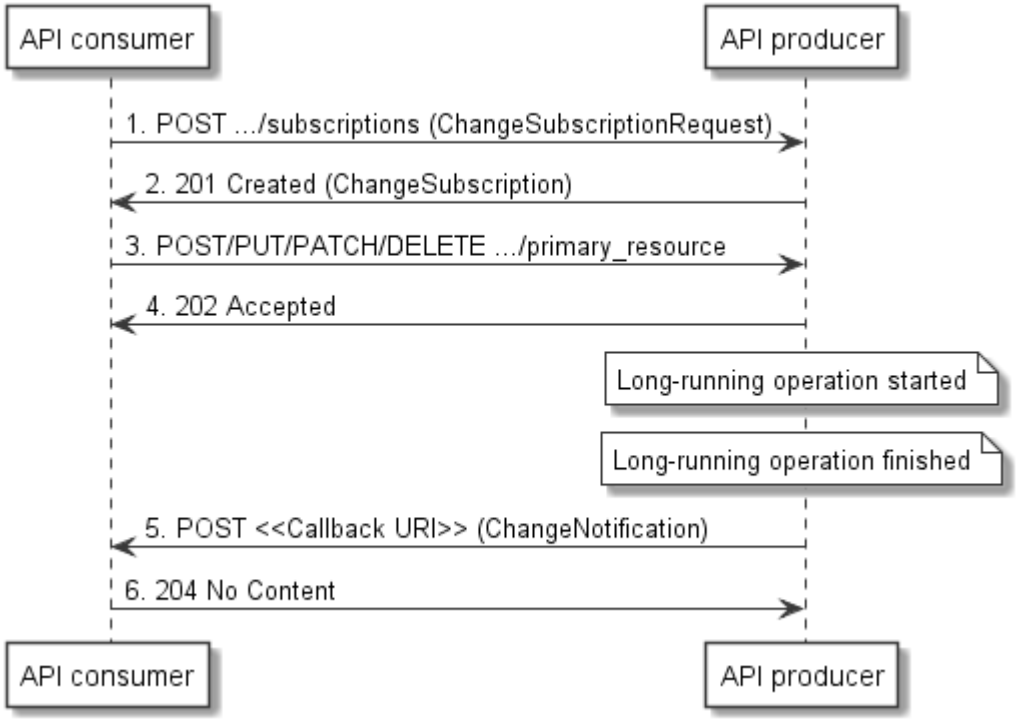

Figure 5.11.1-2: Asynchronous operation flow - with subscribe/notify

5.11.2 Resource definition(s) and HTTP methods

The following resources are involved:

1) Primary resource: The resource that is about to be created/modified/deleted by the long-running operation.

2) Monitor resource: The resource that provides information about the long-running operation.

The HTTP method applied to the primary resource can be any of POST/PUT/PATCH/DELETE.

The HTTP method applicable to read the monitor resource shall be GET.

If monitor change notifications and subscriptions to these are supported, the resources and methods described in clause 5.9 for the RESTful subscribe/notify pattern are applicable here too.

5.11.3 Resource representation(s)

The 202 response shall have an empty body.

The representation of the monitor resource shall contain at least the following information:

- Resource URI of the primary resource.
- Status of the operation (e.g. "processing", "success", "failure").
- Additional information about the result or the error(s) occurred, if applicable.
- Information about the operation (e.g. type, parameters, HTTP method used).

If subscribe/notify is supported, the monitor change notification shall include the status of the operation and the resource URI of the monitor, and shall include the resource URI of the affected primary resource.

5.11.4 HTTP Headers

The link to the monitor shall be provided in the "Location" HTTP header.

5.11.5 Response codes and error handling

On success, "202 Accepted" shall be returned as the response to the request that triggers the long-running operation. On failure, the appropriate error code shall be returned.

The GET request to the monitor resource shall use "200 OK" as the response code if the monitor could be read successfully, or the appropriate error code otherwise.

If subscribe/notify is supported, the provisions in clause 5.9.5 apply in addition.

5.12 Pattern: Asynchronous resource creation without monitor

5.12.1 Description

If only resource creation is asynchronous, there is a simplified pattern available that neither requires a monitor nor a subscription. The progress of the operation is tracked by the response code of a GET request to the resource that is being created. The POST request to create the resource returns "202 Accepted" and includes the URI of the resource to be created in the "Location" HTTP header, and the same response code is returned by the GET method on the to-be-created resource as long as the resource creation is ongoing. Once the resource has been created, the GET method returns "200 OK".

Figure 5.12.1-1 illustrates this pattern.

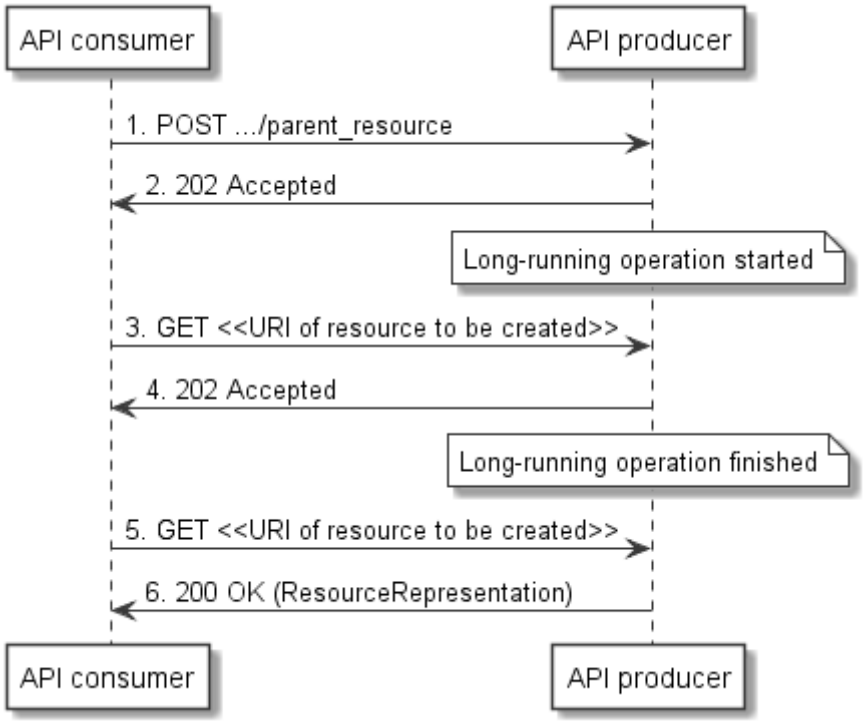

Figure 5.12.1-1: Asynchronous resource creation flow without monitor

5.12.2 Resource definition(s) and HTTP methods

The following resources are involved:

1) Parent resource: The resource under which a new child resource will be created by the long-running operation.

The HTTP method applied to the parent resource shall be POST.

The HTTP method applicable to read the created resource shall be GET.

5.12.3 Resource representation(s)

The provisions for the resource representation of the POST request to create a resource and the GET response to read a resource apply. The body of the 202 response shall be empty.

5.12.4 HTTP Headers

The URI of the resource to be created shall be provided in the "Location" HTTP header in the 202 response to the POST request.

5.12.5 Response codes and error handling

On success, "202 Accepted" shall be returned in the response to the POST request that triggers the long-running resource creation operation.

The response to the GET request to the resource that is being created shall use the "202 Accepted" response code as long as resource creation is ongoing.

The response to the GET request to the resource that has been successfully created shall use the "200 OK" response code.

On failure, the appropriate error code shall be returned by the POST as well as the GET request.

5.13 Pattern: Range requests (partial GET)

5.13.1 Description

This pattern obtains a partial representation of the resource when reading a resource using the HTTP GET method. This pattern is an extension of the simple pattern to read a resource as defined in clause 5.3. It provides advantages when reading resources with large representations over slow or unreliable network connections.

Using range requests, a resource can be read in pieces, or an interrupted download can be resumed by obtaining the missing part of the resource representation. Range requests are specified as an optional HTTP feature in IETF RFC 7233 [6]. This clause can only provide general coverage of the pattern; [6] is the authoritative source of information.

Figure 5.13.1-1 illustrates obtaining a partial representation of a resource. The requested range is signalled in a header of the GET request. The response contains a header that signals the included byte range(s), and a partial resource representation.

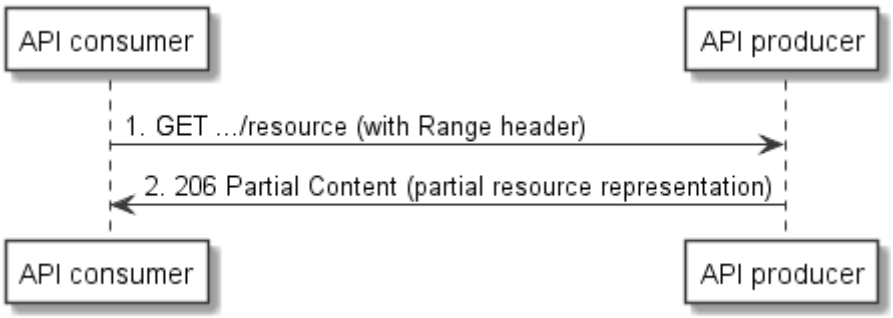

Figure 5.13.1-1: Flow of obtaining a partial representation of a resource

5.13.2 Resource definition(s) and HTTP methods

This pattern is applicable to resources that can be read and that have large, typically binary, representations. The HTTP method shall be GET.

5.13.3 Resource representation(s)

The entity body of the request shall be empty.

The entity body of the response shall contain the part of the representation of the resource that was read, controlled by the header information passed in the request. If a resource does not support range requests, the full representation of the resource shall be returned instead.

5.13.4 HTTP Headers

The "Range" request header shall be used to signal the requested byte range, as defined in IETF RFC 7233 [6].

The "Content-Range" response header shall be used to signal the byte range of the representation enclosed in the payload and the complete length of the representation as defined in IETF RFC 7233 [6].

5.13.5 Response codes and error handling

On success, "206 Partial Content" shall be returned.

If range requests are not supported by a resource, "200 OK" shall be returned together with a full representation of the resource.

If the information passed in the "Range" header is invalid or does not overlap with the range that can be provided as resource representation, the "416 Range Not Satisfiable" error response shall be returned as defined by IETF RFC 7233 [6].

On other errors, the appropriate error code shall be returned.

5.14 Pattern: Representation of lists in JSON

5.14.1 Description

Lists of objects in JSON can be represented in two ways: arrays and maps. For the purpose of illustration in this clause, an example of a structured type is introduced in table 5.14.1-1.

Table 5.14.1-1: Structured example data type "Person"

Attribute name	Data type	Cardinality	Description
name	String	1	Name of the person
age	Number	1	Age of the person

Further, in table 5.14.1-2, an "id" attribute is introduced that identifies the person, i.e. its value is unique among all instances of the data type.

Table 5.14.1-2: Structured example data type "PersonWithId"

Attribute name	Data type	Cardinality	Description
id	String	1	Identifier of the person
name	String	1	Name of the person
age	Number	1	Age of the person

5.14.2 Representation as array

A JSON array represents a list of objects as an ordered sequence of JSON objects. The order is significant; each object in the array can be addressed by its position, i.e. its index.

When modifying an array with PATCH (see clause 5.5), modifications can be represented by passing the whole modified array using the JSON Merge Patch (IETF RFC 7396 [3]) format, or using an NFV-specific delta format (see clause 5.5.3).

Figure 5.14.2-1 provides an example of a list of objects of type "PersonWithId" represented as JSON array. In this example, the "id" attribute identifies the entries as "identifying property". If one needs to refer to this property, one refers to "the 'id' attribute of the (entry in the) 'persons' array/list/attribute"'.

NOTE: Not all arrays have an identifying property.

```
{
    "persons": [
        {"id": "123", "name": "Alice", "age": 30},
        {"id": "abc", "name": "Bob", "age": 40}
    ]
}
```

Figure 5.14.2-1: Example of an array of JSON objects of type "PersonWithId"

In the data model notation (see Annex A), an array of elements of a particular type is denoted by the type name, followed by a cardinality with the upper bound larger than 1 (e.g. 0..N or 1..N). Optionally, the data type may be prefixed with the word "array" and put in parentheses.

Table 5.14.2-1 illustrates the two ways to define an array.

Table 5.14.2-1: Denoting an array in the API data model

Attribute name	Data type	Cardinality	Description
firstArray	PersonWithId	1..N	List of persons, implicitly defined as array by the multiplicity in the cardinality (optional "array" keyword omitted).
secondArray	array(PersonWithId)	0..N	List of persons, explicitly defined as array by the keyword "array".

5.14.3 Representation as map

A JSON map represents a list of objects as an associative set of key-value pairs (where each value is a JSON object and the key is the identifying property). The order of the entries in the map is not significant; each object in the map can be addressed by its unique key. Representation as map requires that the objects in the list have an identifying property, i.e. an attribute with unique values, such as an identifier. That attribute is used as key in the map.

When modifying a map with PATCH (see clause 5.5), modifications can be represented naturally by passing only the changes using the JSON Merge Patch (IETF RFC 7396 [3]) format for the delta document. There is no need for specific signalling.

Figure 5.14.3-1 provides an example of a list of objects of type "Person" represented as a JSON map, using the same data as in figure 5.14.2-1. The "id" attribute is not needed here; its value is used as the name of the key. If one needs to refer to the value of the identifying property, one refers to "the key of the 'persons' map".

NOTE: All maps have an identifying property.

```
{
    "persons": {
        "123": {"name": "Alice", "age": 30},
        "abc": {"name": "Bob", "age": 40}
    }
}
```

Figure 5.14.3-1: Example of a map of JSON objects of type "Person"

In the data model notation (see Annex A), a map of elements of a particular type is denoted by the keyword "map, followed by the type name in parentheses, followed by a cardinality with the upper bound larger than 1 (e.g. 0..N or 1..N). Table 5.14.3-1 illustrates how to define a map.

Table 5.14.3-1: Denoting a map in the API data model

Attribute name	Data type	Cardinality	Description
firstMap	map(Person)	1..N	Map of persons

6 Specifying API and GS versions in the OpenAPI files

6.1 General

The concepts and mechanisms applicable to versioning of the RESTful ETSI NFV-MANO APIs are specified in clause 9 of ETSI GS NFV-SOL 013 [1]. ETSI ISG NFV publishes OpenAPI files [i.5] related to the APIs. This clause defines how to document in the OpenAPI files the relationship to the corresponding GS version and the applicable API version.

6.2 Visibility of the API version identifier fields in the OpenAPI specifications

The full API version identifier (as defined in clause 9 of ETSI GS NFV-SOL 013 [1]) shall be visible in the OpenAPI specifications, in the "version" subfield of the "info" field, as illustrated in the example below. The "impl" parameter shall be used to signal the version of the OpenAPI representation of the API, using the following structure:

etsi.org:ETSI_NFV_OpenAPI:<impl_version>, where <impl_version> is a number.

In case of the OpenAPI files provided by ETSI, the "vendor" field in the "impl" version parameter shall be set to "etsi.org" and the "product" field in that parameter shall be set to "ETSI_NFV_OpenAPI".

EXAMPLE:

```
swagger: "2.0"
info:
  version: "1.2.1-impl:etsi.org:ETSI_NFV_OpenAPI:1"
  title: SOL003 – VNF LCM interface
  license:
    name: "ETSI Forge copyright notice"
    url: https://forge.etsi.org/etsi-forge-copyright-notice.txt
...
```

6.3 Relation between the API version identifiers of an OpenAPI specifications and the base GS

There is no one-to-one mapping between an API version identifier and the version identifier of its published base GS.

A GS specifies multiple APIs, which may have different versions.

A change in the 3rd field of a published GS version identifier (i.e. an editorial change) does not lead to a change in the version identifiers of the APIs specified in the GS.

Only the version identification of published OpenAPI representations based on published GS documents is within the scope of the present document. Non-published previews of OpenAPI representations during development shall be clearly identifiable and distinguishable from published OpenAPI representations.

A change in the 1st or 2nd fields of the published GS version identifier (i.e. a technical change) is likely to lead to at least a change in the minor version number of one of the APIs specified in the GS, which is documented in the definition of the applicable API version in the GS.

For example, if published version 2.4.1 of a base GS contains version 1.1.1 of API A, B and C, published version 3.1.1 of this base GS can contain version 1.2.1 of API A, version 2.1.1 of API B and version 1.1.1 of API C (if no changes were made to API C).

Each OpenAPI specification shall provide in an "externalDoc" field the reference of the published base GS, including the version identifier, as illustrated below.

EXAMPLE:

```
swagger: "2.0"
info:
  version: "1.2.1-impl:etsi.org:ETSI_NFV_OpenAPI:1"
  title: SOL003 - VNF LCM interface
  description: The VNF LCM API provide access to VNF lifecycle management services
  license:
    name: "ETSI Forge copyright notice"
    url: https://forge.etsi.org/etsi-forge-copyright-notice.txt
externalDocs:
  description: ETSI GS NFV-SOL 003 version 2.3.1, 2.4.1
  url : http://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/003/02.04.01_60/gs_nfv-sol003v020401p.pdf
...
```

Multiple published versions of the same GS can contain the same API (e.g. SOL 003 VNF LCM API) without any modification. In this case, all published GS versions supporting the same API should be listed in the "description" subfield under "externalDocs", and the "url" subfield should refer to the latest version of the published base GS.

Annex A (normative): REST API template for interface clauses

X <Long API name> interface

<Template note: One main clause per interface (e.g. VNF Lifecycle Management interface)>

X.1 Description

<Template note: Provides a description of the interface>

X.2 API version

For the *<Long API name>* interface as specified in the present document, the MAJOR version field shall be *<major>*, the MINOR version field shall be *<minor>* and the PATCH version field shall be *<patch>* (see clause 9.1 of ETSI GS NFV-SOL 013 *[<ref>]* for a definition of the version fields). Consequently, the {apiMajorVersion} URI variable shall be set to "v*<major>*".

X.3 Resource structure and methods

All resource URIs of this API shall use the base URI specification defined in clause 4.1 of ETSI GS NFV-SOL 013 *[<ref>]*. The string "*<putApiNameHere>*" shall be used to represent {apiName}. The {apiMajorVersion} shall be set to "v1" for the present document. All resource URIs in the clauses below are defined relative to the above base URI.

<Template note: the content formats to be supported are defined globally. If for a particular API there is deviation from the global definition this needs to be defined here>

Figure X.3-1 shows the overall resource URI structure defined for the *<long API name>* API. Table X.3-1 lists the individual resources defined, and the applicable HTTP methods.

<Template note: a node with a box represents a path segment that has at least one supported HTTP method associated. A solid box indicates a resource strictly following the REST paradigms (subset of CRUD). A dashed box indicates a task resource. A node without a box represents a path segment that has no supported HTTP method associated, i.e. that merely serves for structuring the resource tree. All node names are examples only>

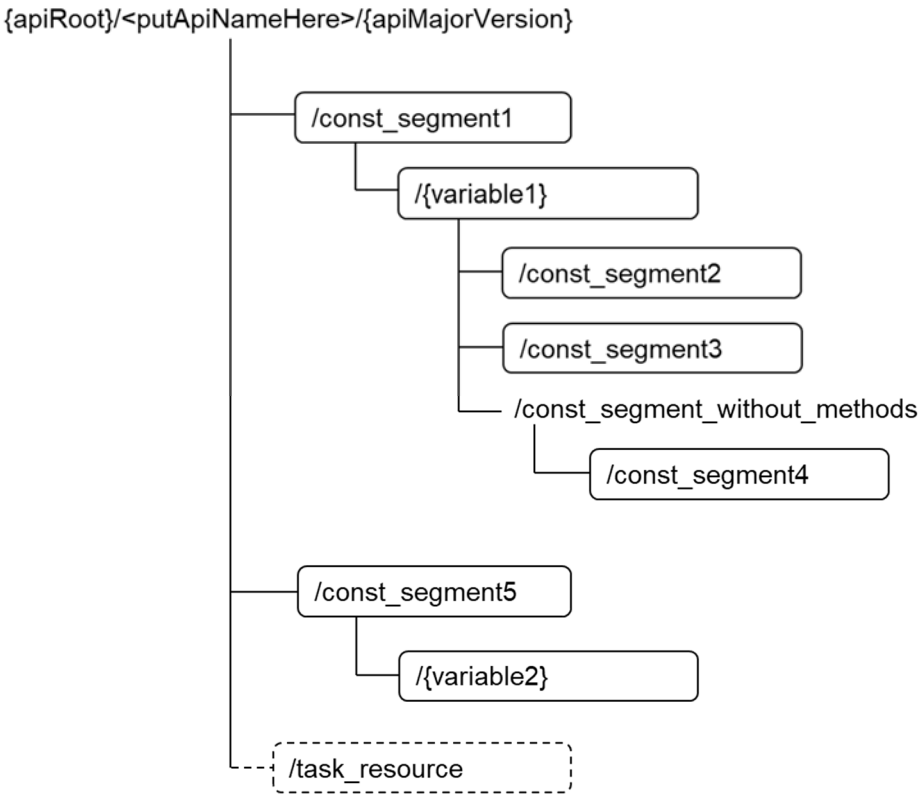

Figure X.3-1: Resource URI structure of the *<long API name>* interface

<Template note: A PPT template for the graph above is available in the following zip file: gs_nfv-sol015v010201p0.zip and attached to the present document>

<Template note: Overview table of resources and operations>

Table X.3-1 lists the individual resources defined, and the applicable HTTP methods.

<Template note: In the text below, parts that are not applicable (e.g. if there are no "C" or "O" resources) can be removed>

The *<API producer>* shall support responding to requests for all HTTP methods on the resources in table X.3-1 that are marked as "M" (mandatory) in the "Cat" column and may support responding to requests for those marked as "O" (optional). Conditions for support of responding to requests for those resources and methods marked as "C" (conditional) in the "Cat" column by the *<API producer>* are defined by notes in table X.3-1.

The *<API producer>* shall also support the "API versions" resources as specified in clause 9.3.3 of ETSI GS NFV-SOL 013 *[<ref>]*.

Table X.3-1: Resources and methods overview of the *<long API name>* interface

Resource name	Resource URI	HTTP METHOD	Cat	Meaning
<Resource Meaning>	<relative URI below root>	POST	M/C/O	<short description>
		GET	M/C/O	<short description>
		PUT	M/C/O	<short description>
		PATCH	M/C/O	<short description>
		DELETE	M/C/O	<short description>
…				
Subscriptions	<relative URI below root>	POST	M/C/O	<short description>
		[…]	M/C/O	<short description>
…				
Notification endpoint	(provided by API consumer)	POST	(*)	<short description>. (*) See note.
		GET	(*)	<short description>. (*) See note.
NOTE: The *<API producer>* shall support invoking the HTTP methods defined for the "Notification endpoint" resource exposed by the *<API consumer>*. If the *<API consumer>* supports invoking the POST method on the "Subscriptions" resource towards the *<API producer>*, it shall also support responding to the HTTP requests defined for the "Notification endpoint" resource.				

<Template note: In the table above, only include sub-rows for those HTTP methods that are applicable to the resource. Only include rows defining the "Notification endpoint" and "Subscriptions" resources and the related notes if subscribe-notify is supported in that API>

<Template note: Start of example>

Table X.3-2: Resources and methods overview of the Foo bar interface

Resource name	Resource URI	HTTP METHOD	Cat	Meaning
Foo instances	/foo_instances	GET	M	Query multiple Foo instances.
		POST	M	Create a Foo instance resource.
Individual Foo instance	/foo_instances/{instanceId}	GET	M	Query single Foo instance.
		PATCH	O	Modify Foo instance information.
		DELETE	M	Delete Foo instance resource.
Enlarge Foo task	/foo_instances/{instanceId} /enlarge	POST	C	Enlarge a Foo instance. See note 1.
Subscriptions	/subscriptions	POST	M	Create a new subscription.
Notification endpoint	(provided by API consumer)	POST	(*)	Notify about foo change events. (*) See note 2.
		GET	(*)	Test the notification endpoint. (*) See note 2.
NOTE 1: Support of this task resource by the Foo Manager depends on the capability level of the Foo manager. Foo Managers of capability level "super power" shall support this task resource; all others need not support this task resource. NOTE 2: The Foo Manager shall support invoking the HTTP methods defined for the "Notification endpoint" resource exposed by the Foo Orchestration Primary. If the Foo Orchestration Primary supports invoking the POST method on the "Subscription" resource towards the Foo Manager, it shall also support responding to the HTTP requests defined for the "Notification endpoint" resource.				

<Template note: End of Example>

X.4 Sequence diagrams

<Template note: this clause typically contains informative content, such as the message flows themselves. Normative keywords may be used in this clause, such as for pre/post-conditions>

<Template note: This clause will be included if needed to illustrate non-trivial message flows>

X.4.1 *<Procedure 1>*

<Template note: Add introductory text>

This clause …

<Template note: Add flow diagram. Do not forget caption. Conventions and examples are documented in annex B. It is recommended to use the PlantUML tool, see annex X, to create the diagram>

<placeholder for graphics, centred>

Figure X.4.1-1: Flow of *<Procedure 1>*

<Template note: Add precondition if applicable>

Precondition: Text text text

<Template note: Add description of the steps>

<Template note: Conventions and example for the description of a flow documented in clause 5>

<Procedure 1>, as illustrated in figure X.4.1-1, consists of the following steps:

<Template note: Add error handling if applicable>

Error handling: Text text text

X.4.2 *<Procedure 2>*

<Template note: same as clause X.4.1>

X.5 Resources

<Template note: this clause is normative>

X.5.1 Introduction

This clause defines the resources and methods of the *<long API name>* API.

<Template note: Repeat the following as often as needed, per resource>

X.5.2 Resource: API versions

The "API versions" resources as defined in clause 9.3.3 of ETSI GS NFV-SOL 013 *[<ref>]* are part of the *<long API name>* interface.

X.5.3 Resource: *<ResourceName>*

X.5.3.1 Description

This resource represents *<something>*. The API consumer can use this resource to *<do something>*.

<Template note: or similar text as applicable>

<Template note: Start of example

This resource represents VNF instances. The API consumer can use this resource to create individual VNF instance resources, and to query VNF instances.

<Template note: End of example

X.5.3.2 Resource definition

The resource URI is:

{apiRoot}/*<putApiNameHere>*/{apiMajorVersion}/*<foo_bar>*

This resource shall support the resource URI variables defined in table X.5.3.2-1.

<Template note: For API resources>

Table X.5.3.2-1: Resource URI variables for this resource

Name	Definition
apiRoot	See clause 4.1 of ETSI GS NFV-SOL 013 *[<ref>]*.
apiMajorVersion	See clause X.2.
<name>	*<definition>*

<Template note: Alternative to be used for the notification endpoint that typically has no standardized resource URI variables>

Table X.5.3.2-1: Resource URI variables for this resource

Name	Definition
none supported	

X.5.3.3 Resource Methods

X.5.3.3.1 POST

<Template note: Alternative 2>

The POST method … *<Meaning(s) of the operation in API space. Add specific normative statements about what the API producer block is expected to do when receiving this request. Also pay attention to normatively define post-conditions, such as the fact that a new resource shall exist (after a request that creates a resource)>*

<Template note: Alternative 2>

This method is not supported. When this method is requested, the *<API producer>* shall return a "405 Method Not Allowed" response as defined in clause 6.4 of ETSI GS NFV-SOL 013 *[<ref>]*.

<Template note: End of alternatives>

<Template note: Start Example>

The POST method creates a foo bar object and the associated resource.

As the result of successfully executing this method, a new "FooBar" resource shall exist as defined in clause x.y.z, and the value of the "foo" attribute in the representation of that resource shall be "bar". A notification of type FooBarCreationNotification shall be triggered as part of successfully executing this method as defined in clause x.y.z.a.

<Template note: End Example>

This method shall follow the provisions specified in the tables X.5.3.3.1-1 and X.5.3.3.1-2 for URI query parameters, request and response data structures, and response codes.

Table X.5.3.3.1-1: URI query parameters supported by the *<POST>* method on this resource

Name	Cardinality	Remarks
<name> *or* none supported	0..1 *or* 1 *or* 0..N *or <leave empty>*	*<only if applicable>*

Table X.5.3.3.1-2: Details of the *<POST>* request/response on this resource

	Data type	Cardinality	Response Codes	Description
Request body	*<type> or* n/a	1 *<(i.e. object)> or* 0..N / 1..N / m..n *<i.e. array> or <leave empty>*		*<Description of the case in which this data type is sent. Shall be present if multiple alternatives exist and may be omitted in case of a single alternative>*
Response body	**Data type**	**Cardinality**	**Response Codes**	**Description**
	<type> or n/a	1 *<(i.e. object)> or* 0..N / 1..N / m..n *<i.e. array> or <leave empty>*	*<list applicable codes with name from IETF RFC 7231 [<ref>] etc.>*	*< **Normative** statement defining the case in which this response is returned (success or error)* *- Suggestion for success: Shall be returned when (success condition)* *- Suggestion for error: Shall be returned upon the following error: (error condition)* *>* *<Normative statement about the response body>* *<if specific headers are applicable>* The HTTP response shall/should/may *<choose one>* include a *<name>* HTTP header that… *<endif>* *<Further text if applicable>*
	(…)			
	ProblemDetails	See clause 6.4 of [<ref>].	4xx/5xx	In addition to the response codes defined above, any common error response code as defined in clause 6.4 of ETSI GS NFV-SOL 013 *[<ref>]* may be returned.

<Template note: Start of example>

Table X.5.3.3.1-3: URI query parameters supported by the POST method on this resource

Name	Cardinality	Remarks
foo_bar	0..1	The foo bar

Table X.5.3.3.1-4: Details of the POST request/response on this resource

Request body	Data type	Cardinality	Description	
	FooBarCreateRequest	1	Foo bar instance creation parameters	
Response body	**Data type**	**Cardinality**	**Response Codes**	**Description**
	FooBarInstance	1	201 Created	Shall be returned when the foo bar instance was created successfully. The response body shall contain a representation of the created foo bar instance resource. The HTTP response shall include a "Location" HTTP header that contains the URI of the newly-created resource.
	ProblemDetails	1	400 Bad Request	Shall be returned upon the following error: Incorrect parameters were passed to the request. In the returned ProblemDetails structure, the "detail" attribute should convey more information about the error.
	ProblemDetails	1	404 Not Found	Shall be returned upon the following error: The resource URI was incorrect. In the returned ProblemDetails structure, the "detail" attribute should convey more information about the error.
	ProblemDetails	See clause 6.4 of *[<ref>].*	4xx/5xx	In addition to the response codes defined above, any common error response code as defined in clause 6.4 of ETSI GS NFV-SOL 013 *[<ref>]* may be returned.

<Template note: End of Example>

<Template note: Main place to define error handling is the table above. If necessary, describe ADDITIONAL error handling in text below>

Error handling: text text text

X.5.3.3.2 GET

<same structure as for POST>

X.5.3.3.3 PUT

<same structure as for POST>

X.5.3.3.4 PATCH

<same structure as for POST>

X.5.3.3.5 DELETE

<same structure as for POST>

X.6 Data model

<Template note: this clause is normative>

X.6.1 Introduction

<Template note: To be written according to the individual specification>

X.6.2 Resource and notification data types

X.6.2.1 Introduction

This clause defines data structures to be used in resource representations and notifications.

X.6.2.2 Type: *<TypeName1>*

This type represents a <…>. It shall comply with the provisions defined in table X.6.2.2-1.

<Template note: Data type names in UpperCamel>

<Template note: Provisions how to document a structured data type and an example are provided in clause 5.4.2 of ETSI GS NFV-SOL 015>

-

Table X.6.2.2-1: Definition of the *<TypeName1>* data type

Attribute name	Data type	Cardinality	Description

X.6.3 Referenced structured data types

X.6.3.1 Introduction

This clause defines data structures that can be referenced from data structures defined in the previous clauses, but can neither be resource representations nor bound to any subscribe/notify mechanism.

X.6.3.2 Type: *<TypeName2>*

<Template note: Same structure as in clause X.6.2.2>

X.6.4 Referenced simple data types and enumerations

X.6.4.1 Introduction

This clause defines simple data types that can be referenced from data structures defined in the previous clauses.

<Template note: This covers simple types, including enumerations>

X.6.4.2 Simple data types

The simple data types defined in table X.6.4.2-1 shall be supported.

Table X.6.4.2-1: Simple data types

Type name	Description

X.6.4.3 Enumeration: *<TypeName3>*

The enumeration *<TypeName3>* represents *<something>*. It shall comply with the provisions defined in table X.6.4.3-1.

Table X.6.4.3-1: Enumeration *<TypeName3>*

Enumeration value	Description

Annex B (informative): Conventions for message flows

B.1 Tool support

The PlantUML tool [i.2] is used to obtain a unique appearance of all flow diagrams in the RESTful NFV-MANO API specifications. The tool can be obtained from [i.3] and documentation is available from [i.4].

The appearance of the diagrams is controlled by the "skin.inc" file which is included in every PlantUML source file, as follows:

- Save the text below in a text file named "skin.inc" and put it into the same directory as the PlantUML source file.

```
skinparam monochrome true
skinparam sequenceActorBackgroundColor #FFFFFF
skinparam sequenceParticipantBackgroundColor #FFFFFF
skinparam noteBackgroundColor #FFFFFF
autonumber "#'.'"
```

- Use these instructions at the beginning of the PlantUML source file to include the file.

```
@startuml
!include skin.inc
```

- When making a contribution, ensure to include the PlantUML source file in a ZIP archive with the contribution.

B.2 Graphical conventions

1) An HTTP request is represented by a solid arrow (->) with the method name, followed by the URI, followed by an indication of the type of the entity body, if applicable.

```
EXAMPLE: consumer -> producer: POST …/container_resources (FooBarType)
```

2) An HTTP response is represented by a solid arrow (->) with the response code, followed by the meaning of the response code, followed by an indication of the type of the entity body, if applicable.

```
EXAMPLE: producer -> consumer: 201 Created (FooBarType)
```

3) If it is necessary to call out a particular attribute in the entity body for later reference, use the syntax <type>;<attribute>=<value>.

```
EXAMPLE: producer -> consumer: 201 Created (FooBarType:id=123)
```

4) A condition, postcondition, or precondition, if needed, is expressed as a note over consumer and producer.

```
EXAMPLE: note over consumer, producer
     Precondition: Everything was prepared
end note
```

5) A processing step, if there is no need to number it, or a comment, is expressed as a note over consumer or producer.

```
EXAMPLE: note over producer
     Update internal database
end note
```

6) A processing step, if there is the need to automatically number it for reference from the text, is expressed as a "signal to self" at producer or consumer side, with a dashed slim-headed arrow.

```
EXAMPLE: producer -->> producer: Update internal database
```

7) HTTP requests and responses are numbered, usually with an increment of one. The necessary definitions for automatic numbering are included in skin.inc (see clause B.1).

8) A message or message exchange of which the details are defined elsewhere is represented with a dashed slim-headed arrow and set in italics.

```
EXAMPLE: producer -->> consumer: <i>Send XyzNotification</i>
```

9) A message or sequence of messages that is optional to execute, and the associated notes, are enclosed into an "opt" section.

```
EXAMPLE:
opt
     consumer -> producer: GET …/nice_to_have
     producer -> consumer: 200 OK (NiceToHaveType)
end
```

10) Alternatives are represented using an "alt" section.

```
EXAMPLE:
     alt query multiple instances
          consumer -> producer: GET .../instances?filter=abc
          producer -> consumer: 200 OK (InstanceType[])
     else read information about individual instance
          consumer -> producer: GET .../instances/{instanceId}
          producer -> consumer: 200 OK (InstanceType)
     end
```

11) For better structuring, long message sequences may be separated into sections using "==" separators.

```
EXAMPLE:
== Initialization ==
```

12) Further, for better structuring, multiple messages may be grouped together in a named group to show that they belong together and serve an overarching purpose.

```
EXAMPLE:
      group Creation and instantiation
     consumer -> producer: POST …/instances (InstanceType)
     producer -> consumer: 201 Created (InstanceType)
     consumer -> producer: POST …/instances/{instanceId} (InstantiateRequest)
     producer -> consumer: 202 Accepted ()
   end
```

An overall example that illustrates the provisions above is given in the figures B.2-1 and B.2-2.

EXAMPLE: This example illustrates a VNF instantiation. The example might differ from the actual technical content that has been, is or will eventually be agreed for inclusion into a GS.

```
@startuml
!include skin.inc
participant "NFVO" as cli
participant "VNFM" as srv

== Create and instantiate VNF ==
```

```
note over cli, srv
  Precondition: VNF instance does not exist
end note

cli -> srv: POST …/vnf_instances (VnfInstance)
srv -> cli: 201 Created (VnfInstance:links.self=…/vnf_instances/123)

note over cli, srv
  Condition: VNF instance in NOT_INSTANTIATED state
end note

group VNF instantiation sequence
  cli -> srv: POST …/vnf_instances/123/instantiate (InstantiateVnfRequest)
  srv -> cli: 202 Accepted ()

  srv -->> cli: <i> Send VnfLcmOperationOccurrenceNotification (start)</i>

  opt
    note over cli
      API consumer polls the VNF lifecycle
      operation occurrence resource
    end note
  end

  cli -> srv: GET …/vnf_lc_ops/abcxyz456
  srv -> cli: 200 OK (LcOpOcc:status=processing)

  srv -->> srv: Instantiation finished

  srv -->> cli: <i>Send VnfLcmOperationOccurrenceNotification (result)</i>

  opt
    cli -> srv: GET …/vnf_lc_ops/abcxyz456
    srv -> cli: 200 OK (LcOpOcc:status=success)
  end
end

note over cli, srv
  Postcondition: VNF instance in INSTANTIATED state
end note

== Query / read VNF instance information ==

alt query information about multiple VNF instances

  cli -> srv: GET .../vnf_instances

  srv -> cli: 200 OK (VnfInstance[])

else read information about individual VNF instance

  cli -> srv: GET .../vnf_instances/{vnfInstanceId}

  srv -> cli: 200 OK (VnfInstance)

end

@enduml
```

Figure B.2-1: PlantUML source to illustrate the conventions defined above

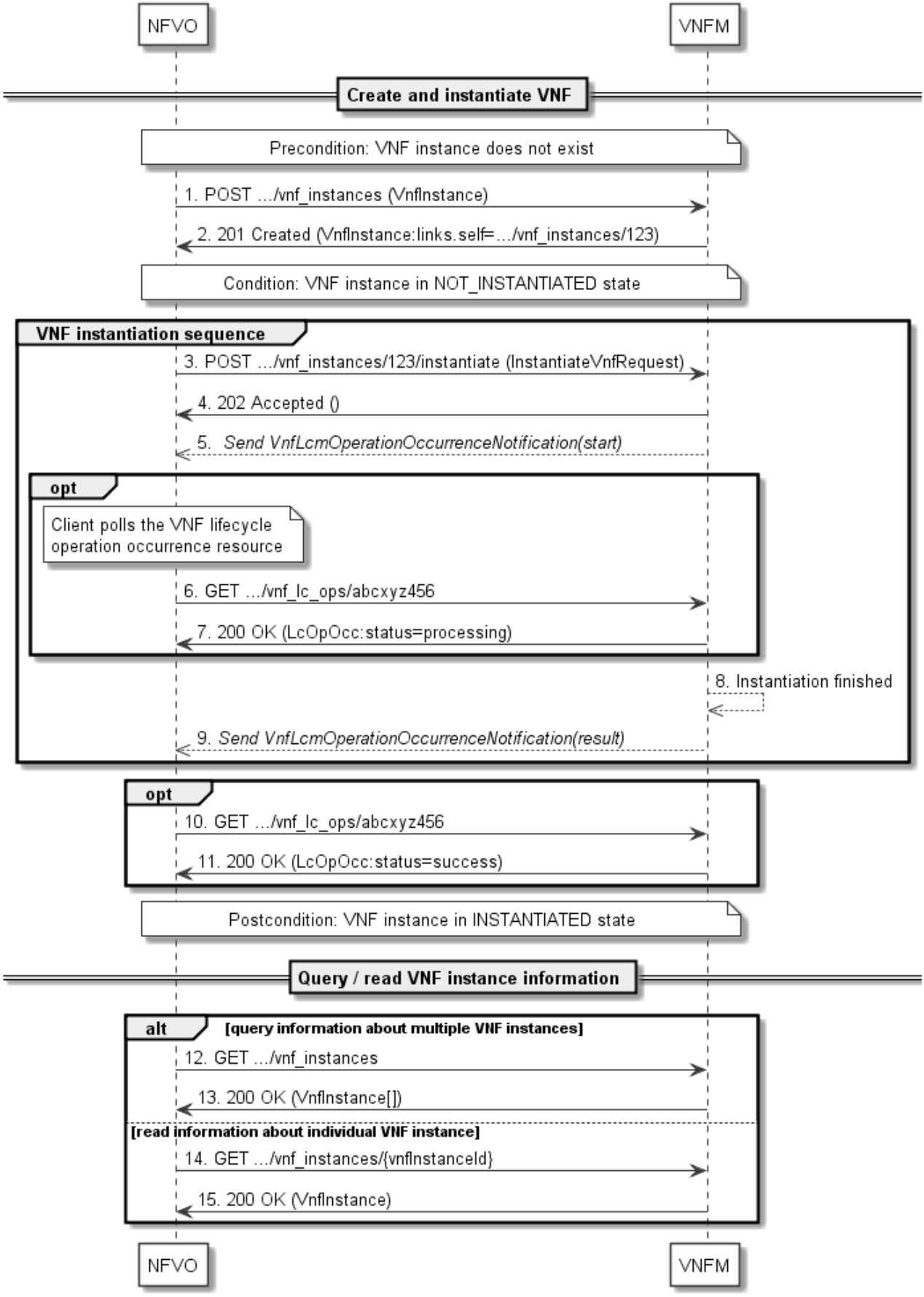

Figure B.2-2: Flow diagram resulting from the PlantUML source in figure B.2-1

Annex C (normative): Change requests classification

C.1 Introduction

This annex provides guidelines on how to fill the "Other comments" field of the Change Request (CR) template when submitting a CR on RESTful NFV-MANO API specifications. It also provides some examples of BackWard Compatible (BWC) and Non-BackWard Compatible (NBWC) changes to be specified in the CR template.

C.2 The Field "Other comments"

The field "Other comments" of the CR template shall contain at least the following two sub-fields:

- Change type.
- Change type code(s) (CTC).

The value of "Change type" and "Change type code(s)" sub-fields shall be set according to the following rules when submitting a CR:

1) "Change type" identifies whether the changes proposed in the CR are BackWard Compatible (BWC), Non-BackWard Compatible (NBWC) or whether backward compatibility is Not Applicable (N/A) to these changes. If at least one of the proposed changes in the CR is NBWC, then "Change type" shall be set to NBWC.

2) Valid values for the sub-field "Change type code(s)" are defined in the following clauses and are qualifiers of the sub-field "Change type". When "Change type" is set to BWC then the "Change type codes" are defined in clause C.3. In case "Change type" is set to NBWC then the "Change type codes" are defined in clause C.4.

3) When "Change type" is set to Not Applicable (N/A) then the "Change type code(s)" sub-field shall not be filled.

This approach is illustrated in figure C.2-1.

CHANGE REQUEST

	Version		**CR**		**rev**	-
CR Title:						
Source:						
Work Item Ref:				***Date:***		
Category:				***Release:***		
	Use one of the following categories: ***F*** *(correction)* ***A*** *(corresponds to a correction in an earlier release)* ***B*** *(addition of feature)* ***C*** *(functional modification of feature)* ***D*** *(editorial modification)*					
Reason for change:						

Summary of change:	
Clauses affected:	
Other deliverables affected:	
Other comments:	***Change type (BWC, NBWC or N/A):*** ***Change type code(s):***

Figure C.2-1: Other comments field with Change Type and Change Type Code(s)

C.3 Examples of BWC Changes

Examples of BWC changes include:

- Adding a new resource
- Adding a new URI
- Supporting a new HTTP method for an existing resource
- Adding new optional URI query parameters
- Adding new optional attributes to a resource representation in a request
- Adding new attributes to a resource representation in a response or to a notification message
- Adding a new notification message
- Responding with a new status code of an error class
- Changing the name of a named type or changing from an inlined structure to a named data type, or from a named data type to an inlined structure
- Certain cardinality changes (see note 1 in table C.3-1)
- Error corrections

Table C.3-1 defines the list of change type codes for BWC changes on the API.

Table C.3-1: Change type codes for BWC changes

BWC Change Type	Change Type Code
Adding a new resource	BWC_ADD_RESOURCE
Adding a new URI	BWC_ADD_URI
Supporting a new HTTP method for an existing resource	BWC_ADD_METHOD
Adding new optional URI query parameters	BWC_ADD_OPT_QUERY_PARAMS
Adding new optional attributes to a resource representation in a request	BWC_ADD_OPT_ATTR_REQST
Adding new attributes to a resource representation in a response or to a notification message	BWC_ADD_ATTR_RESP_OR_NOTIF
Adding a new notification message	BWC_ADD_NOTIFICATION
Responding with a new status code of an error class	BWC_NEW_STATUS_CODE
Renaming a named data type, or changing from a named data type to an inlined structure or vice versa	BWC_RENAME_DATATYPE See note 4.
Certain BWC cardinality changes	BWC_CARD_CHGS See note 1.
Error correction	BWC_ERROR_CORR See note 2.
Other	BWC_OTHER See note 3.
NOTE 1: Whether attribute cardinality changes are backward compatible depends on the type of change. An example of a backward-compatible cardinality change include making an attribute in a response required (e.g. changing cardinality from 0..1 to 1). NOTE 2: A change that corrects an error that would lead the API producer to always send an error response if a certain valid condition is met is considered a backward compatible change, irrespective of the type of change. NOTE 3: A BWC change type code equal to "OTHER_BWC" is defined to represent any BWC change besides those explicitly defined in this table. NOTE 4: Note that these changes do not show up in the message exchanges, so there is no need to change API producer and API consumer implementations.	

C.4 Examples of NBWC Changes

Examples of NBWC changes to the resources structure include:

- Removing a resource/URI
- Removing support for an HTTP method
- Changing a resource URI
- Adding new mandatory URI query parameters

Examples of NBWC changes to the payload body include:

- Renaming an attribute in a resource representation
- Adding new mandatory attributes to a resource representation in a request
- Changing the data type of an attribute
- Certain cardinality changes (see note 2 in table C.4-1)

Table C.4-1 defines the list of change type codes for NBWC changes on the API.

Table C.4-1: Change type codes for NBWC changes

NBWC Category	NBWC Change Type	Change Type Code
Resource Structure Changes	Removing a resource	NBWCR_REMOVE_RESOURCE
	Removing a URI	NBWCR_REMOVE_URI
	Removing support for an HTTP method	NBWCR_REMOVE_METHOD
	Changing a resource URI	NBWCR_CHG_URI
	Adding new mandatory URI query parameters	NBWCR_ADD_MAND_QUERY_PARAMS
	Other NBWC changes to the resources structure	NBWCR_OTHER See note 1.
Payload Body Changes	Renaming an attribute in a resource representation	NBWCP_RENAME_ATTR
	Adding new mandatory attributes to a resource representation in a request	NBWCP_ADD_MAND_ATTR_REQST
	Changing the data type of an attribute	NBWCP_CHG_DATA_TYPE See note 4.
	Certain NBWC cardinality changes	NBWCP_CARD_CHGS See note 2.
	Other NBWC changes to the payload body	NBWCP_OTHER See note 3.

NOTE 1: A NBWC change type code equal to "NBWCR_OTHER" is defined to represent any NBWC change to the resources structure besides those explicitly defined in this table.

NOTE 2: Whether attribute cardinality changes are backward compatible depends on the type of change. Examples of non-backward compatible cardinality changes include decreasing the upper bound of a cardinality range for attributes sent by the API consumer and changing the meaning of the default behaviour associated to the absence of an attribute of cardinality 0..N.

NOTE 3: A NBWC change type code equal to "NBWCP_OTHER" is defined to represent any NBWC change to the payload body besides those explicitly defined in this table.

NOTE 4: This change type relates to changes that modify the content of the attribute (e.g. change of primitive type) in the protocol messages. It does not relate to changes subsumed under change type "BWC_RENAME_DATATYPE".

Annex D (informative): Change History

Date	Version	Information about changes
March 2019	0.1.0	Initial skeleton based on - NFVSOL(19)000158r1 - NFVSOL(19)000159r1
June 2019	0.2.0	Contributions incorporated - NFVSOL(19)000313r3_SOL015_patterns - NFVSOL(19)000314r2_SOL015_conventions_for_names - NFVSOL(19)000315r2_SOL015_conventions_for_flows - NFVSOL(19)000316r3_SOL015_Annex_A_template - NFVSOL(19)000317r1_SOL015_Annex_B_CRs_classification Editorials - Use "NFV-MANO" across the board
August 2019	0.3.0	Contributions incorporated - NFVSOL(19)000329r3_SOL015_versioning - NFVSOL(19)000412r1_SOL015_clause_6 - NFVSOL(19)000470_SOL015_fixing_change_code Editorials - Use "NFV-MANO" across the board - Some placeholders in the template renamed to improve consistency
October 2019	0.4.0	Contributions incorporated - NFVSOL(19)000478_SOL015_Updating_PATCH_pattern - NFVSOL(19)000479_SOL015_Updating_simple_monitor_pattern - NFVSOL(19)000480_SOL015_add_more_PlantUML_primitives.docx - NFVSOL(19)000536r1_SOL015_use_of_PUT - NFVSOL(19)000537_SOL015_use_of_partial_GET - NFVSOL(19)000544_SOL015_small_fix Editorials - Making line styles of task resources in the resource tree consistent across GSs - Making line style of responses consistent with our graphical conventions - Using "(API)consumer" and "(API)producer" consistently (instead of client and server) - Collected abbreviations - Fixed clause 3 to align with latest template
November 2019	0.5.0	Contributions included - NFVSOL(19)000538_SOL015_Fixing_Query_via_GET - NFVSOL(19)000665_SOL015_add_none_supported - NFVSOL(19)000683_SOL015ed271_smart_mirror_of_679_adding_error_response_for_fa - NFVSOL(19)000666_SOL015_clarify_callbackURI__client_side_URI__notification_en - NFVSOL(19)000656_SOL015_Creation_by_PUT.docx - NFVSOL(19)000620r1_SOL015_resolve_ENs Editorials - Fixed the document structure after including 538. - Removed remaining rapporteur's note.
January 2020	1.1.1	Publication by ETSI
May 2020	1.1.2	Contributions included - NFVSOL(20)000207_SOL015ed211_adding_maps - NFVSOL(20)000319r2_SOL015ed211_BWC_NBWC_type_code_small_fixes
July 2020	1.1.3	Contributions included - NFVSOL(20)000609_SOL015ed121_Remove_optionality_of_notification_endpoint_test - NFVSOL(20)000608r1_SOL015ed121_Conventions_for_documenting_the_data_model Editorials: - Changed name of attachment to indicate the correct version number.

History

Document history		
V1.1.1	January 2020	Publication
V1.2.1	December 2020	Publication