Title*: ETSI NFV SOL REST API Conventions

from **Source***: SOL WG

Contact: Uwe Rauschenbach (as editor)

input for **Committee***: NFV SOL

Contribution **For***:		
	Decision	
	Discussion	
	Information	X

Submission date*: 2017-07-27

Meeting & Allocation: **NFVSOL#34**

Relevant WI(s), or deliverable(s):

ABSTRACT: *This document collects all agreed REST API conventions to be applied to the ETSI NFV SOL REST APIs (including SOL002, SOL003 and SOL005).*

Table of Contents

1 Scope

This living document collects the conventions for the ETSI NFV REST APIs as agreed by the ETSI NFV SOL working group. It is a document that is intended for ETSI NFV SOL to guide the development of the ETSI NFV SOL REST API specifications. An example for an API developed based on the conventions in this document can be found in [1].

2 References

[1] ETSI GS NFV-SOL 003: "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Or-Vnfm Reference Point". Available from: http://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/003/02.03.01_60/gs_NFV-SOL003v020301p.pdf

[2] PlantUML website, http://plantuml.com/

3 Process

This document is classified as "for information". It collects all REST API conventions that were agreed by ETSI NFV SOL. Each convention shall be proposed by a contribution to ETSI NFV SOL for decision, clearly indicating the additions / changes to the present document. Once Accepted by ETSI NFV SOL, the changes will be incorporated by the editor into a new revision of the conventions document.

Even though the present document is classified as "For Information", the conventions and patterns contained in it are binding for ETSI NFV SOL when developing the SOL REST APIs, since the document reflects content that was agreed by group decision. To deviate from the agreed conventions for particular contributions requires SOL WG consensus.

4 Conventions for names, strings and URIs

4.1 Case conventions

The following case conventions for names and strings are available for potential use in the Stage 3 specifications.

1. ALLUPPER

All letters of a string are capital letters. Digits are allowed but not at the first position. No other characters are allowed.

EXAMPLES:

a. MANAGEMENTINTERFACE
b. ETSINFVMANAGEMENT

2. alllower

All letters of a string are lowercase letters. Digits are allowed but not at the first position. No other characters are allowed.

EXAMPLES:

a. managementinterface
b. etsinfvmanagement

3. UPPER_WITH_UNDERSCORE

All letters of a string are capital letters. Digits are allowed but not at the first position. Word boundaries are represented by the underscore "_" character. No other characters are allowed.

EXAMPLES:

a. MANAGEMENT_INTERFACE
b. ETSI_NFV_MANAGEMENT

4. lower_with_underscore

All letters of a string are lowercase letters. Digits are allowed but not at the first position. Word boundaries are represented by the underscore "_" character. No other characters are allowed.

EXAMPLES:

a. management_interface
b. etsi_nfv_management.

5. UpperCamel

A string is formed by concatenating words. Each word starts with an uppercase letter (this implies that the string starts with an uppercase letter). All other letters are lowercase letters. Digits are

allowed but not at the first position. No other characters are allowed. Abbreviations follow the same scheme (i.e. first letter uppercase, all other letters lowercase).

EXAMPLES:

a. ManagementInterface
b. EtsiNfvManagement

6. **lowerCamel**

As UpperCamel, but with the following change: The first letter of a string shall be lowercase (i.e. the first word starts with a lowercase letter).

EXAMPLES:

a. managementInterface
b. etsiNfvManagement.

4.2 Conventions for URI parts

Based on IETF RFC 3986, the parts of the URI syntax that are relevant in the context of the ETSI NFV SOL REST API are 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)

Decision 1. Path Segment Naming Conventions.

a. The path segments of a resource URI which represent a constant string shall use lower_with_underscore.

   EXAMPLE: vnf_instances

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

   EXAMPLE: …/prefix/API/1_0/user**s**

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

   EXAMPLE: …/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 at least start with a verb.

   EXAMPLES:
   …/vnf_instances/{vnfInstanceId}/scale

…/vnf_instances/{vnfInstanceId}/scale_to_level

e. The path segments of a resource URI which represent a variable name shall use lowerCamel, and shall be surrounded by curly brackets.

   EXAMPLE: {vnfInstanceId}

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

Decision 2. Query Naming Conventions

a. Parameter names in queries shall use lower_with_underscore.

   EXAMPLE: ?working_group=SOL

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

   EXAMPLE: ?working_group={chooseAWorkingGroup}

c. Once a variable is replaced at runtime by an actual string, the convention defined in Decision1.f. applies 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 ETSI NFV SOL REST API data structures.

a. Names of attributes / parameters shall be represented using lowerCamel.

   EXAMPLE: vnfName

   NOTE: It is assumed that, deviating from this rule, the names of attributes/parameters in the VNFD in SOL001 use lower_with_underscore.

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

   EXAMPLES: users, extVirtualLinks

c. "True" identifiers defined in IFA007 and IFA008 using the name syntax "xyzStructureId" shall be represented using the name "id."

NOTE: In IFA007/8, the name of an attribute that is a "true" identifier of a data structure "XyzStructure" uses the name syntax "xyzStructureId". The term "true" identifier denotes an identifier embedded in "XyzStructure" for identifying and/or externally referencing an instance of that structure.

d. Each value of an enumeration types shall be represented using UPPER_WITH_UNDERSCORE.

EXAMPLE: NOT_INSTATIATED

e. The names of data types shall be represented using UpperCamel.

EXAMPLES: ResourceHandle, VnfInstance

4.4 Conventions for URI structure and supported content formats

This clause specifies the URI prefix and the supported formats applicable to the APIs defined in the present document.

All resource URIs of the APIs shall have the following prefix:

{apiRoot}/{apiName}/{apiVersion}/

where

{apiRoot} indicates the scheme ("http" or "https"), the host name and optional port, and an optional prefix path.

{apiName} indicates the interface name in an abbreviated form. The {apiName} of each interface is defined in the clause specifying the corresponding interface.

{apiVersion} indicates the current version of the API and is defined in the clause specifying the corresponding interface.

For HTTP requests and responses that have a body, the content format JSON (see IETF RFC 7159) shall be supported. The JSON format shall be signalled by the content type "application/json".

All APIs shall support and use HTTP over TLS (also known as HTTPS) (see IETF RFC 2818). TLS version 1.2 as defined by IETF RFC 5246 shall be supported.

All resource URIs of the API shall comply with the URI syntax as defined in IETF RFC 3986. An implementation that dynamically generates resource URI parts (path segments, query parameter values) shall ensure that these parts only use the character set that is allowed by IETF RFC 3986 for these parts.

NOTE: This means that characters which are not part of this allowed set need to be escaped using percent-encoding as defined by IETF RFC 3986.

5 Conventions for Message flows

5.1 Tool support

To obtain a unique appearance of all flow diagrams in the REST specifications, ETSI NFV SOL uses the free tool PlantUML [2].

The tool can be obtained here: https://sourceforge.net/projects/plantuml/files/plantuml.jar/download.

Documentation can be obtained here: http://plantuml.com/PlantUML_Language_Reference_Guide.pdf.

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

- Save the text below in a text file named "skin.inc" and pPut 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

5.2 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: client -> server: POST …/vnf_instances (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: server -> client: 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: server -> client: 201 Created (FooBarType:id=123)
```

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

```
      EXAMPLE: note over client, server
                  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 client or server.

```
      EXAMPLE: note over server
                  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 server or client side, with a dashed slim-headed arrow.

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

7) HTTP requests and responses shall be numbered, with an increment of one. The necessary definitions for automatic numbering are included in skin.inc.

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: server -->> client: <i>Send XyzNotification</i>
```

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

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

An overall example that illustrates the provisions above is given in the figures 5.2-1 and 5.2-2.

NOTE: This illustrates VNF instantiation but may differ from the actual technical content that will eventually be agreed for inclusion into the GS.

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

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

cli -> srv: POST …/vnf_instances/123/instantiate (InstantiateParams)
srv -> cli: 202 Accepted ()

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

```
opt
      note over cli
            Client 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

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

@enduml
```

Figure 5.2-1: PlantUML source to illustrate the conventions defined above

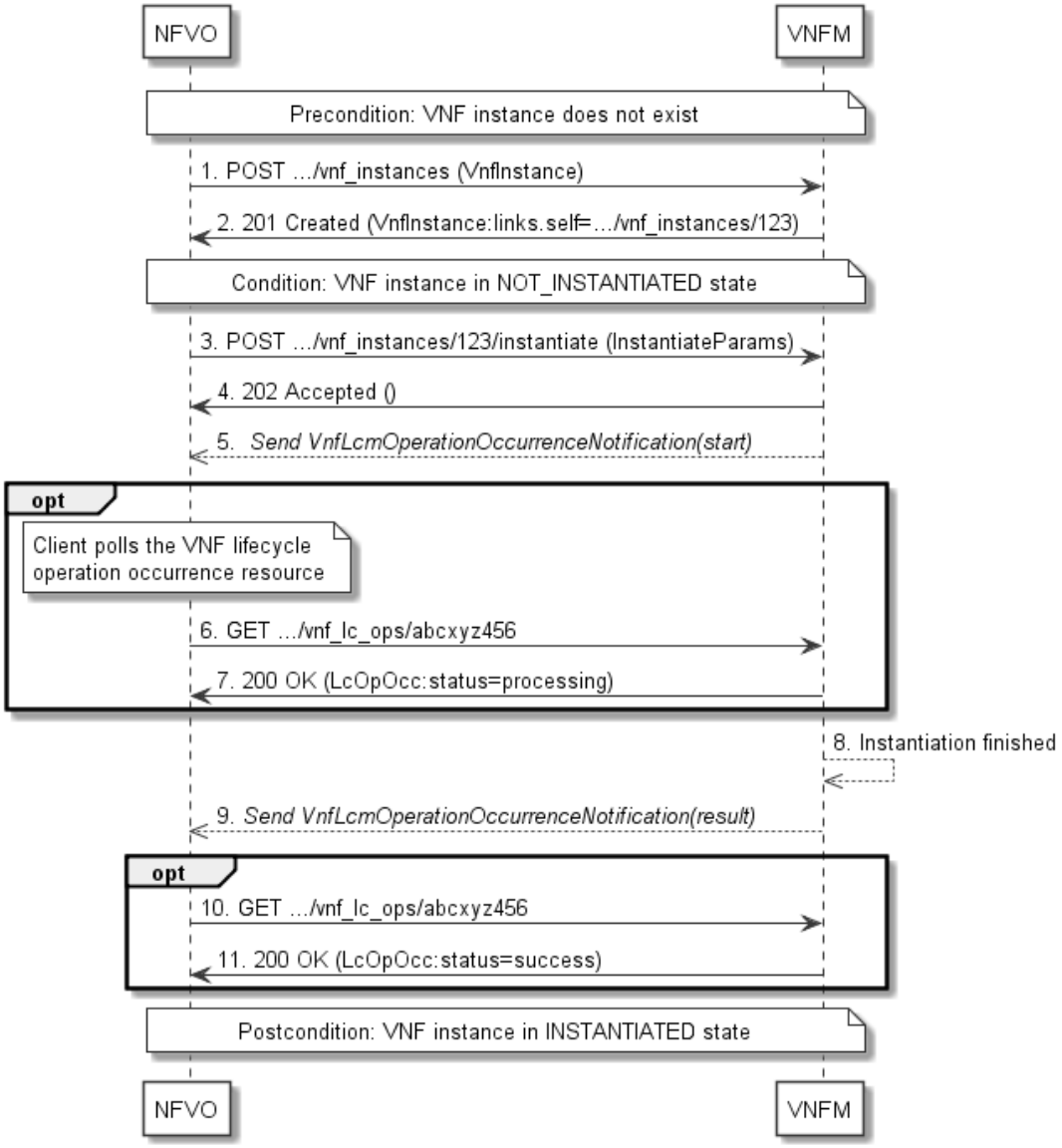


Figure 5.2-2: Flow diagram resulting from the PlantUML source in figure 5.2-1

6 API patterns

6.1 Pattern: Subscribe-Notify

6.1.1 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 client, specific mechanisms need to be put into place to support the server-initiated delivery of notifications. The basic principle is that the REST client exposes a lightweight HTTP server towards the REST server. The lightweight HTTP server only needs to support a small subset of the HTTP functionality – namely the POST method, the 204 success response code plus the relevant error response codes, and, if applicable, authentication/authorization. The REST client exposes the lightweight HTTP server in a way that it is reachable via TCP by the REST server.

To manage subscriptions, the REST server needs to expose a container resource under which the REST client 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 client exposes one or more HTTP endpoints on which it can receive POST requests. When creating a subscription, the client informs the server of the endpoint (callbackUri) to which the server will later deliver notifications related to that particular subscription.

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

Figure 6.1.1-1 illustrates the management of subscriptions. Figure 6.1.1-2 illustrates the delivery of a notification.

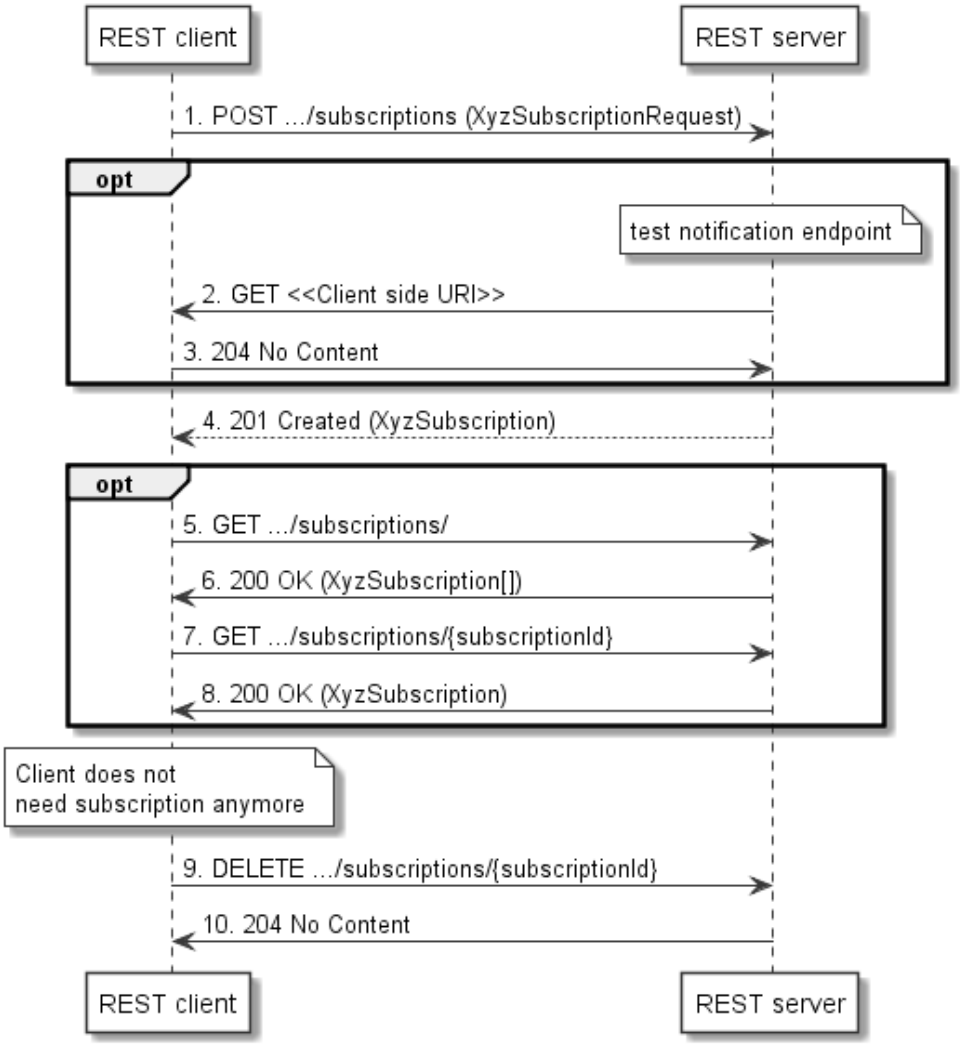


Figure 6.1.1-1: Management of subscriptions

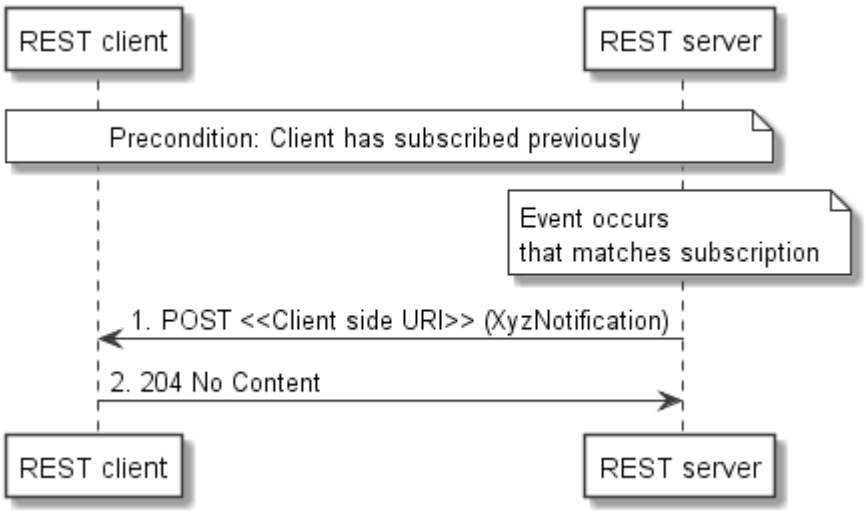


Figure 6.1.1-2: Delivery of notifications

6.1.2 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) An HTTP endpoint that is exposed by the REST client 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 server to deliver notifications shall be POST.

6.1.3 Resource representation(s)

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

- It shall contain the URI of an HTTP endpoint that the REST client exposes to receive notifications. That URI shall be provided by the client on subscription.
- It should contain criteria that allow the server to determine the events about which the client 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 client 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 6.2)
- It shall contain information about the event.

6.1.4 HTTP Headers

No specific provisions are applicable here.

6.1.5 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, “204 No Content” shall be returned.

On failure, the appropriate error code shall be returned.

6.2 Pattern: Links

6.2.1 Description

It is commonly seen as good adherence to RESTful principles that resources link to other resources, allowing the client 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 client to automatically traverse the links to access resources related to the actual resource, in order to perform operations on them.

6.2.2 Resource definition(s) and HTTP methods

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

6.2.3 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 6.2.3-3 illustrates the JSON schema and figure 6.2.3-4 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 6.2.3-3: 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 6.2.3-4: JSON fragment for an example "_links" element

6.2.4 HTTP Headers

There are no specific provisions w.r.t. HTTP headers for this pattern.

6.2.5 Response codes and error handling

There are no specific provisions w.r.t. response codes and error handling for this pattern.

6.3 Pattern: Resource creation (POST)

6.3.1 Description

New resources are created on the origin server as children of a parent resource. In order to request resource creation, the client sends a POST request to the parent resource and includes a representation of the resource to be created. The server 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 server 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 6.3.1-1 illustrates creating a resource.

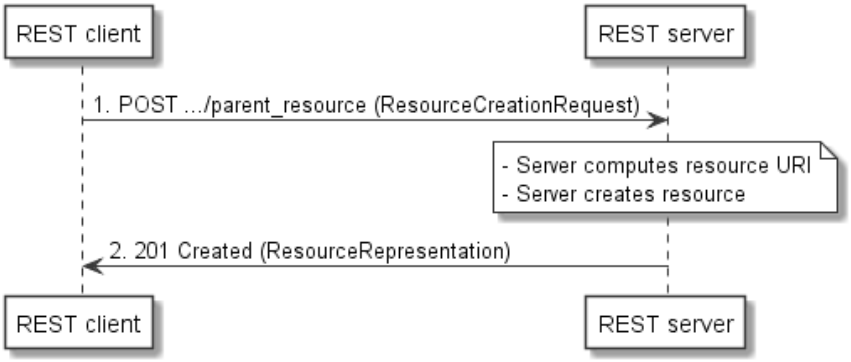


Figure 6.3.1-1: Resource creation flow

6.3.2 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 server, and that is unique in the scope of the parent resource URI.

The HTTP method shall be POST.

6.3.3 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.

6.3.4 HTTP Headers

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

6.3.5 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 6.8 and 6.9 for more details about asynchronous operations.

6.4 Pattern: Reading a resource (GET)

6.4.1 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 6.10); these cannot be read.

Figure 6.4.1-1 illustrates reading a resource.

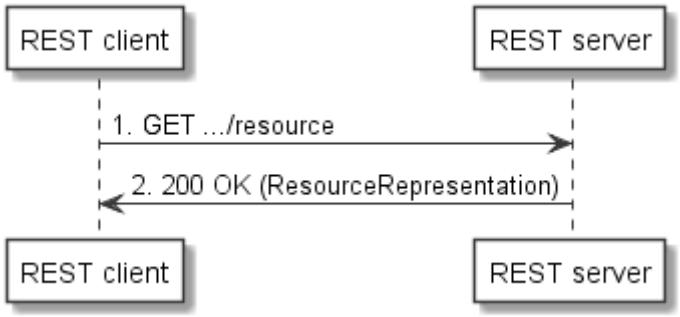


Figure 6.4.1-1: Reading a resource

6.4.2 Resource definition(s) and HTTP methods

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

6.4.3 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.

6.4.4 HTTP Headers

No specific provisions for HTTP headers for this pattern.

6.4.5 Response codes and error handling

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

6.5 Pattern: Resource query with filtering/selection (GET)

6.5.1 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

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

6.5.2 Attribute-based filtering

6.5.2.1 Overview and example

Attribute-based filtering allows to reduce the number of objects returned by a query operation. Typically, attribute-based filtering is applied to a GET request that reads a resource which represents a list of objects (e.g. child resources). Only those objects that match the filter are returned as part of the resource representation in the payload body of the GET response.

Attribute-based filtering can test a simple (scalar) attribute of the resource representation against a constant value, for instance for equality, inequality, greater or smaller than, etc. Attribute-based filtering is requested by adding a set of URI query parameters, the "attribute-based filtering parameters" or "filter" for short, to a resource URI.

The following example illustrates the principle. Assume a resource "container" with the following objects:

EXAMPLE 1: objects

```
obj1: {id:123, weight:100, parts:[{id:1, color:red}, {id:2, color:green}]}
obj2: {id:456, weight:500, parts:[{id:3, color:green}, {id:2, color:green}]}
```

A GET request on the "container" resource would deliver the following response:

EXAMPLE 2: Unfiltered GET

Request:

```
GET …/container
```

Response:

```
[
    {id:123, weight:100, parts:[{id:1, color:red}, {id:2, color:green}]},
    {id:456, weight:500, parts:[{id:3, color:green}, {id:2, color:blue}]}
]
```

A GET request with a filter on the "container" resource would deliver the following response:

EXAMPLE 3: GET with filter

Request:

```
GET …/container?weight.eq=100 or GET …/container?weight=100
```

Response:

```
[
    {id:123, weight:100, parts:[{id:1, color:red}, {id:2, color:green}]}
]
```

For hierarchically-structured data, filters can also be applied to attributes deeper in the hierarchy. In case of arrays, a filter matches if any of the elements of the array matches. In other words, when applying the filter "parts.color.eq=green" (or "parts.color=green") to the objects in Example 1, the filter matches obj1 when evaluating the second entry in the "parts" array of obj1, and matches obj2 already when evaluating the first entry in the "parts" array of obj2. As the result, both obj1 and obj2 match the filter.

If a filter contains multiple sub-parts that only differ in the leaf attribute (i.e. they share the same attribute prefix), they are evaluated together per array entry when traversing an array. As an example, the filter "parts.color.eq=green"&"parts.id.eq=3" (or "parts.color=green&parts.id=3") would be evaluated together for each entry in the array "parts". As the result, obj2 matches the filter.

6.5.2.2 Specification

A set of filter parameters shall be represented as a part of the URI query string. This means, it shall consist of one or more strings formatted according to "simpleFilterExpr", concatenated using the "&" character:

```
simpleFilterExpr := <attrName>["."<attrName>]*["."<op>]?"="<value>[","<value>]*
filterExpr       := <simpleFilterExpr>["&"<simpleFilterExpr>]*
op               := "eq" | "neq" | "gt" | "lt" | "gte" | "lte" | "cont" |
                    "ncont"
attrName         := string
value            := scalar value
```

where:

```
*     zero or more occurrences
?     zero or one occurrence
[]    grouping of expressions to be used with ? and *
""    quotation marks for marking string constants
<>    name separator
```

"AttrName" is the name of one attribute in the data type that defines the representation of the resource. The dot (".") character in "simpleFilterExpr" allows concatenation of <attrName> entries to filter by attributes deeper in the hierarchy of a structured document. "Op" stands for the comparison operator. If the expression has concatenated <attrName> entries, it means that the operator "op" is applied to the attribute addressed by the last <attrName> entry included in the concatenation. All simple filter expressions are combined by the "AND" logical operator.

In a concatenation of <attrName> entries in a <simpleFilterExpr>, the rightmost <attrName> entry is called "leaf attribute". The concatenation of all "attrName" entries except the leaf attribute is called the "attribute prefix". If an attribute referenced in an expression is an array, an object that contains a corresponding array shall be considered to match the expression if any of the elements in the array matches all expressions that have the same attribute prefix.

The leaf attribute of a <simpleFilterExpr> shall not be structured, but shall be of a simple (scalar) type such as String, Number or DateTime, or shall be an array of simple (scalar) attributes. Attempting to apply a filter with a structured leaf attribute shall be rejected with "400 Bad request". A <filterExpr> shall not contain any invalid <simpleFilterExpr> entry.

The operators "op" listed in table 6.5.2.2-1 shall be supported.

Table 6.5.2.2-1: Operators for attribute-based filtering

Operator <op>	Meaning
<attrName>.eq=<value>[,<value>]*	Attribute **equal** to one of the values in the list
<attrName>=<value>[,<value>]*	Alternative representation of equality. See note.
<attrName>.neq=<value>[,<value>]*	Attribute **not equal** to any of the values in the list
<attrName>.gt=<value>	Attribute **greater than** <value>
<attrName>.gte=<value>	Attribute **greater than or equal** to <value>
<attrName>.lt=<value>	Attribute **less than** <value>
<attrName>.lte=<value>	Attribute **less than or equal** to <value>
<attrName>.cont=<value>[,<value>]*	Attribute **contains** (at least) one of the values in the list
<attrName>.ncont=<value>[,<value>]*	Attribute **does not contain** any of the values in the list
NOTE: This representation shall not be used for attributes whose name is equal to the name of a defined URI query parameter.	

All objects that match the filter shall be returned as response to a GET request that contains a filter.

A <value> entry shall contain a scalar value, such as a number or string. The content of a <value> entry shall be formatted the same way as the representation of the related attribute in the resource representation, for instance as String, Integer, Boolean or DateTime.Attribute-based filters are supported for certain resources. Details are defined in the clauses specifying the actual resources.

6.5.3 Attribute selectors

6.5.3.1 Overview and example

Certain resource representations can become quite big, in particular, if the resource is a container for multiple sub-resources, or if the resource representation itself contains a deeply-nested structure. In these cases, it can be desired to reduce the amount of data exchanged over the interface and processed by the API consumer application. On the other hand, it can also be desirable that a "drill-deep" for selected parts of the omitted data can be initiated quickly.

An attribute selector allows the API consumer to choose which attributes it wants to be contained in the response. Only attributes that are not required to be present, i.e. those with a lower bound of zero on their cardinality (e.g. 0..1, 0..N) and that are not conditionally mandatory, are allowed to be omitted as part of the selection process. Attributes can be marked for inclusion or exclusion.

If an attribute is omitted, a link to a resource may be added where the information of that attribute can be fetched. Such approach is known as HATEOAS which is a common pattern in REST, and enables drilling down on selected issues without having to repeat a request that may create a potentially big response.

6.5.3.2 Specification

6.5.3.2.1 GET request

The URI query parameters for attribute selection are defined in table 6.5.3.2.1-1.

In the provisions below, "complex attributes" are assumed to be those attributes that are structured, or that are arrays.

Table 6.5.3.2.1-1: Attribute selector parameters

Parameter	Definition
all_fields	This URI query parameter requests that all complex attributes are included in the response, including those suppressed by exclude_default. It is inverse to te "exclude_default" parameter. The API producer shall support this parameter for certain resources. Details are defined in the clauses specifying the actual resources.
fields	This URI query parameter requests that only the listed complex attributes are included in the response. The parameter shall be formatted as a list of attribute names. An attribute name shall either be the name of an attribute, or a path consisting of the names of multiple attributes with parent-child relationship, separated by ".". Attribute names in the list shall be separated by comma (","). Valid attribute names for a particular GET request are the names of all complex attributes in the expected response that have a lower cardinality bound of 0 and that are not conditionally mandatory. The API producer should support this parameter for certain resources. Details are defined in the clauses specifying the actual resources.
exclude_fields	This URI query parameter requests that the listed complex attributes are excluded from the response. For the format, eligible attributes and support by the API producer, the provisions defined for the "fields" parameter shall apply.
exclude_default	Presence of this URI query parameter requests that a default set of complex attributes shall be excluded from the response. The default set is defined per resource in the present document. Not every resource will necessarily have such a default set. Only complex attributes with a lower cardinality bound of zero that are not conditionally mandatory can be included in the set. The API producer shall support this parameter for certain resources. Details are defined in the clauses specifying the actual resources. This parameter is a flag, i.e. it has no value. If a resource supports attribute selectors and none of the attribute selector parameters is specified in a GET request, the "exclude_default" parameter shall be assumed as the default.

6.5.3.2.2 GET response

Table 6.5.3.2.2-1 defines the valid parameter combinations in a GET request and their effect on the GET response.

Table 6.5.3.2.2-1: Valid combinations of attribute selector parameters

Parameter combination	The GET response shall include…
(none)	… same as "exclude_default"
all_fields	… all attributes.
fields=<list>	… all attributes except all complex attributes with minimum cardinality of zero that are not conditionally mandatory, and that are not provided in <list>.
exclude_fields=<list>	… all attributes except those complex attributes with a minimum cardinality of zero that are not conditionally mandatory, and that are provided in <list>.
exclude_default	… all attributes except those complex attributes with a minimum cardinality of zero that are not conditionally mandatory, and that are part of the "default exclude set" defined in the present specification for the particular resource
exclude_default and fields=<list>	… all attributes except those complex attributes with a minimum cardinality of zero that are not conditionally mandatory and that are part of the "default exclude set" defined in the present specification for the particular resource, but that are not part of <list>

If complex attributes were omitted in a GET response, the response may contain a number of links that allow to obtain directly the content of the omitted attributes. Such links shall be embedded into a structure named "_links" at the same level as the omitted attribute. That structure shall contain one entry for each link, named as the omitted attribute, and containing an "href" attribute that contains the URI of a resource that can be read with GET to obtain the content of the omitted attribute. A link shall not be present if the attribute is not present in the underlying resource representation. The

resource URI structure of such links is not standardized, but may be chosen by the VNFM implementation. Performing a GET request on such a link shall return a representation that contains the content of the omitted attribute.

EXAMPLE:

```
"_links" : {
       "vnfcs" : {"href" : ".../vnflcm/v1/vnf_instances/1234/vnfcs"},
       "extVirtualLinks" : {"href" : ".../vnflcm/v1/_dynamic/7d6bef4e-d86b-4abc-97ed-9dc9b951f206"}
}
```

6.6 Pattern: Resource update (PATCH)

6.6.1 Description

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

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 ETSI NFV SOL, but may be used to upload raw files.

PATCH does not carry a representation of the resource in the entity body, but a document that instructs the server how to modify the resource representation. In ETSI NFV SOL specifications, JSON Merge Patch (IETF RFC 7396) is used for that purpose, which defines fragments that are merged into the target JSON document.

Figure 6.6.1-1 illustrates updating a resource by PATCH.

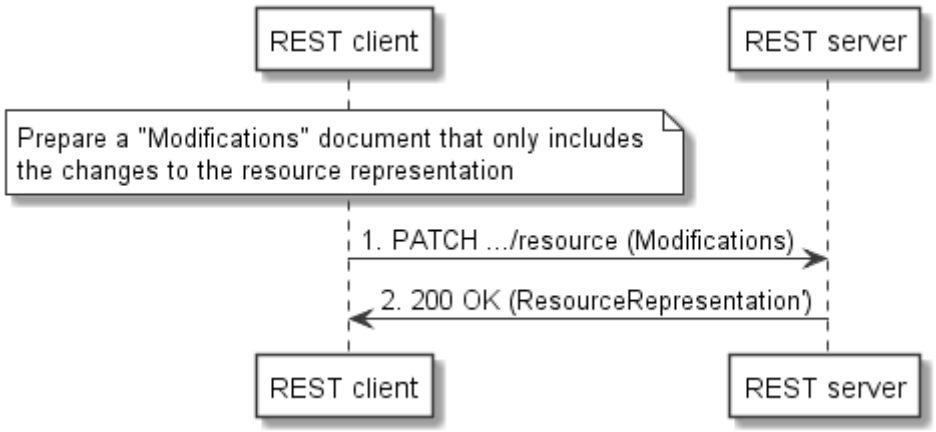


Figure 6.6.1-1: Basic resource update flow with PATCH

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) supports conditional requests to detect such a situation and to give the client 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 server each time the resource is changed. This information is delivered to the client in the "ETag" HTTP header in HTTP responses. If the client wishes that the server executes the PATCH only if the ETag has not changed since the time it has last read the resource (GET), the client adds to the PATCH request the HTTP header "If-Match" with the ETag value obtained from the GET request. The server 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 6.6.1-2.

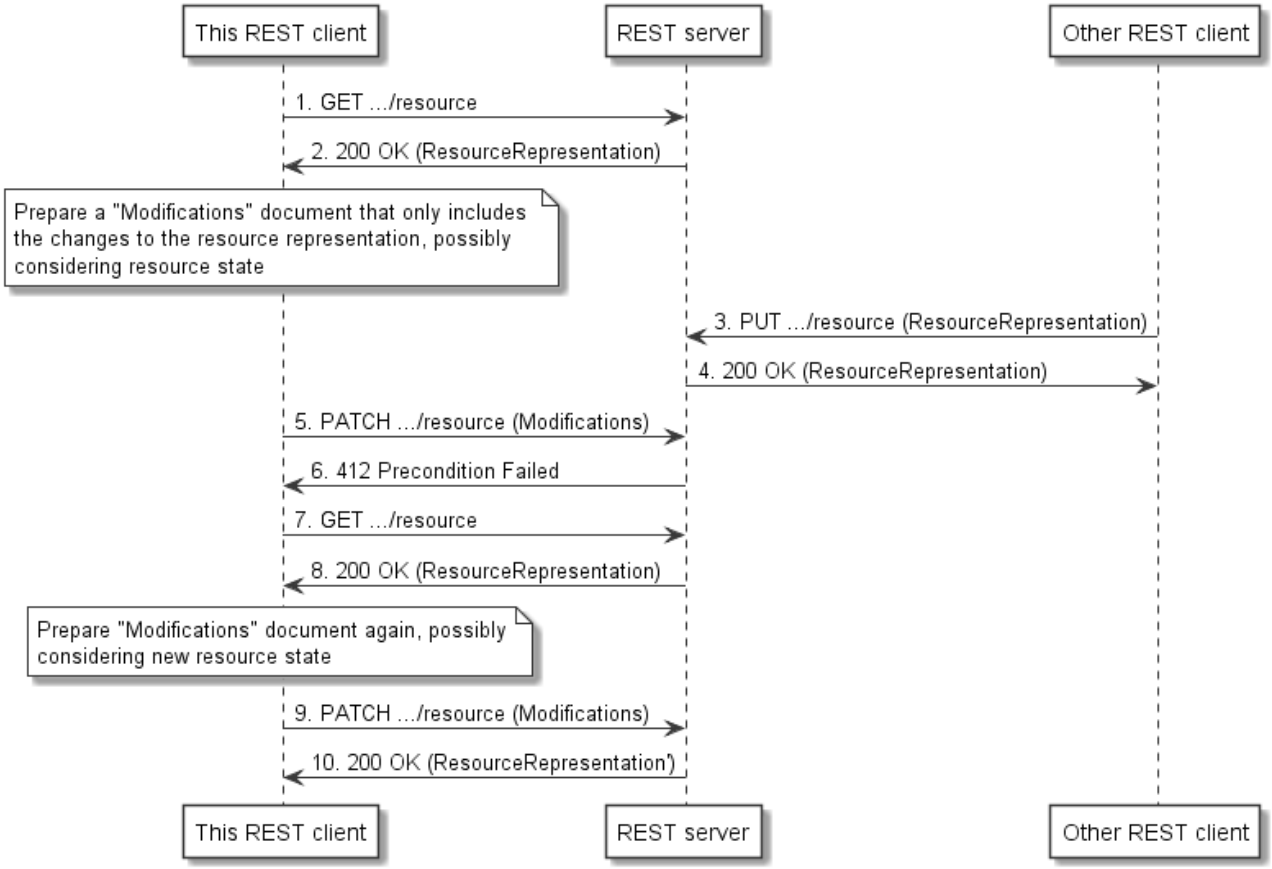


Figure 6.6.1-2: Resource update flow with PATCH, considering concurrent updates

In a particular API, it is recommended to stick to one update pattern - either PUT or PATCH.

6.6.2 Resource definition(s) and HTTP methods

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

6.6.3 Resource representation(s)

The entity body of the PATCH request does not carry a representation of the resource, but a description of the changes in one of the formats defined by IETF RFC 7396. IETF RFC 7396 does no selective update of arrays. However, in ETSI NFV SOL specifications, many array elements are objects that contain an identifier attribute. In order to modify arrays of that type, ETSI NFV SOL has defined an extension of IETF RFC 7396 that allows to add and to update elements, as follows:

Assumptions:

1) "oldList" is the array to be modified (part of the resource representation) and "newList" is the array in the "Modifications" document (part of the PATCH payload) 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 "id" attribute 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 "id" attribute is unique (i.e. there are no two entries with the same content of "id")

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" according to the rules of JSON Merge PATCH (see IETF RFC 7396).

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

6.6.4 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.

If conflicts and data inconsistencies are foreseen when multiple clients update the same resource, the client should pass in the "If-Match" HTTP header of the PUT request the value of the "ETag" HTTP header received in the response to the GET request.

6.6.5 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 (see annex B) shall be returned.

Resource update can also be asynchronous in which case "202 Accepted" shall be returned instead of "200 OK". See clause 6.8 for more details about asynchronous operations.

6.7 Pattern: Resource deletion (DELETE)

6.7.1 Description

The Delete pattern deletes a resource by invoking the HTTP DELETE method on that resource. After successful completion, the client 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 server responds with "404 Resource Not Found".

Figure 6.7.1-1 illustrates deleting a resource.

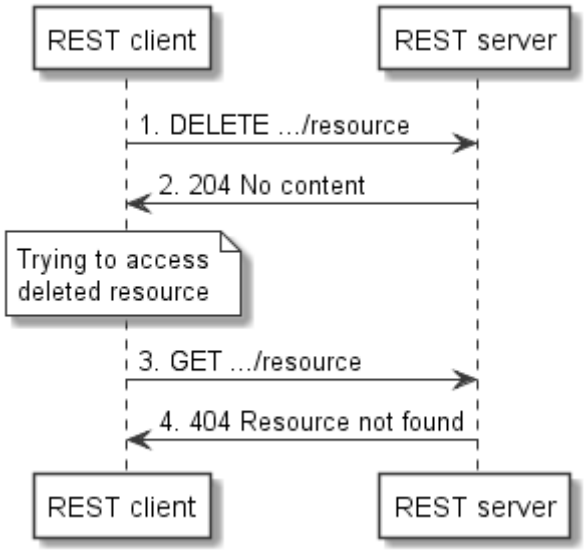


Figure 6.7.1-1: Resource deletion flow

6.7.2 Resource definition(s) and HTTP methods

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

6.7.3 Resource representation(s)

The entity body of the request shall be empty. The entity body of the response is typically empty.

NOTE: Alternatively, the response can include the final representation of the resource prior to deletion.

6.7.4 HTTP Headers

No specific provisions for HTTP headers for this pattern.

6.7.5 Response codes and error handling

On success, "204 No content" shall be returned. On failure, the appropriate error code shall be returned.

If a deleted resource is accessed subsequently by any HTTP method, the server 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 6.8 for more details about asynchronous operations.

6.8 Pattern: Asynchronous invocation with monitor

6.8.1 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 REST client 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 client can check the status of the operation by polling; additionally or alternatively, the subscribe-notify mechanism (see clause 6.1) can be used to provide the result once available. The progress of the operation is reflected by a monitor resource.

Figure 6.8.1-1 illustrates asynchronous operations with polling. After receiving an HTTP request that is to be processed asynchronously, the server 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 client 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.

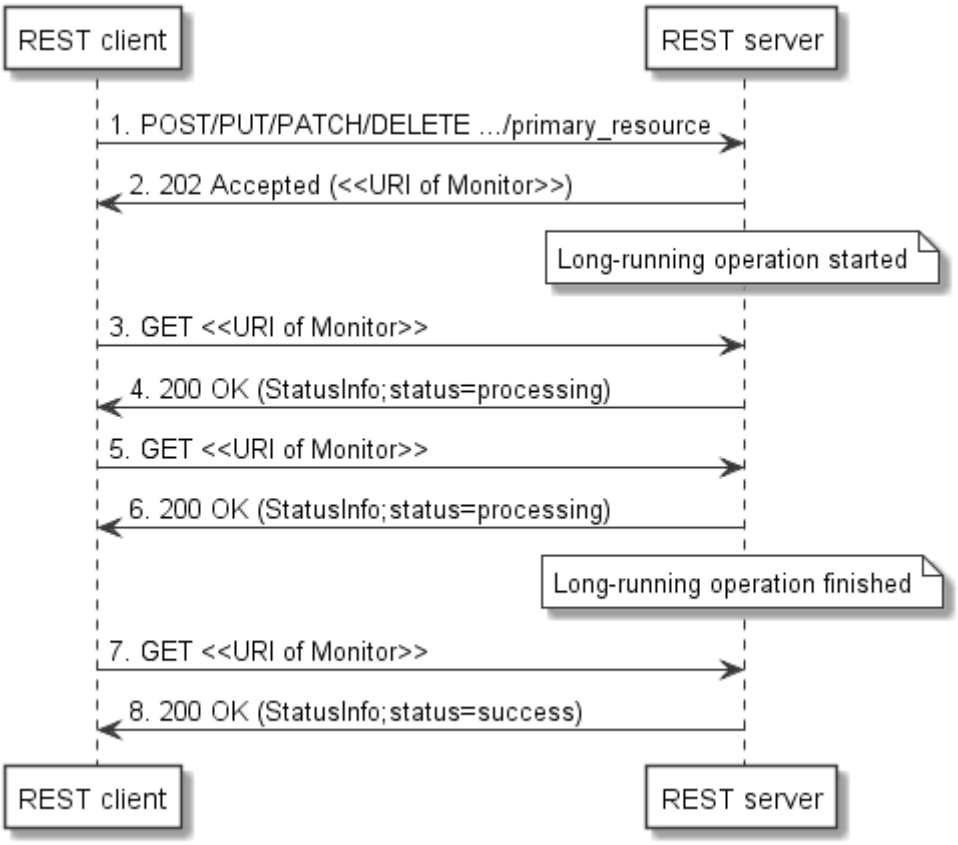


Figure 6.8.1-1: Asynchronous operation flow - with polling

Figure 6.8.1-2 illustrates asynchronous operations with subscribe/notify. Before a client 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 server 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 client 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 server will send to the client 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, and a link to the resource that is modified by the asynchronous operation, and application-specific further information, The client can then read the monitor resource to obtain further information.

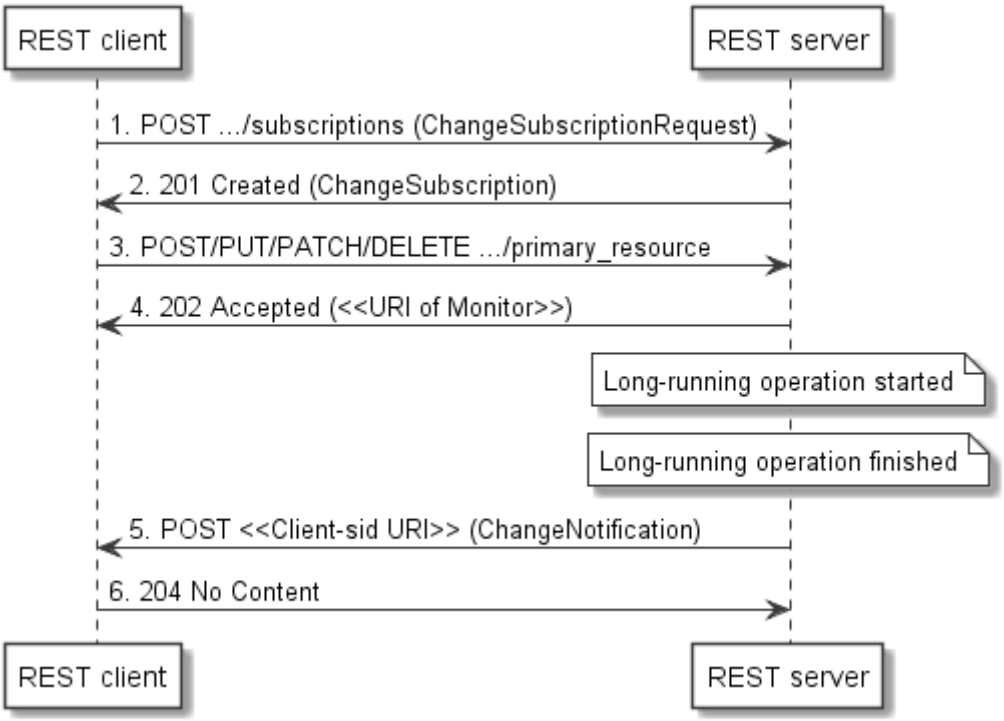


Figure 6.8.1-2: Asynchronous operation flow - with subscribe/notify

6.8.2 Resource definition(s) and HTTP methods

The following resources are involved:

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

4) 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 6.1 for the RESTful subscribe/notify pattern are applicable here too.

6.8.3 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.

6.8.4 HTTP Headers

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

6.8.5 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 6.1.5 apply in addition.

6.9 Pattern: Asynchronous resource creation without monitor

6.9.1 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 request on the to-be-created resource as long as the resource creation is ongoing.

Figure 6.9.1-1 illustrates this pattern.

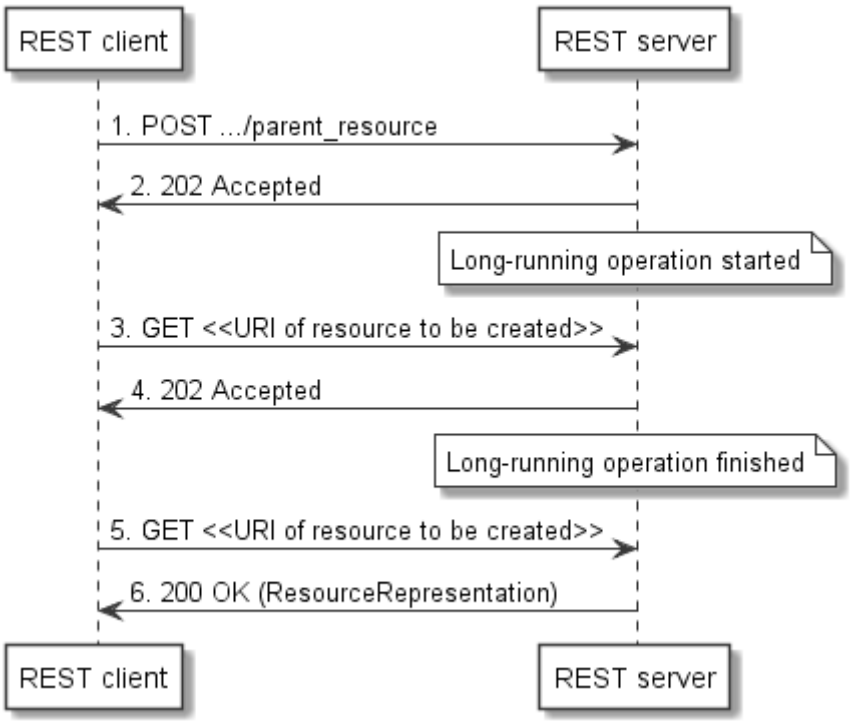


Figure 6.9.1-1: Asynchronous operation flow - with polling

6.9.2 Resource definition(s) and HTTP methods

The following resources are involved:

5) 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.

6.9.3 Resource representation(s)

The provisions for the resource representation of the POST request to create a resource and the GET response when reading a resource apply.

6.9.4 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.

6.9.5 Response codes and error handling

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

The GET request to the resource that is being created shall return the "202 Accepted" response code.

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

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

6.10 Task resources

6.10.1 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.

6.10.2 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.

6.10.3 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.

6.10.4 HTTP Headers

In case the task resource represents an operation that is asynchronous, the provisions in clause 6.8 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.

6.10.5 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 6.8 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.

6.11 Pattern: Authorization

ETSI NFV SOL uses OAuth 2.0 to authorize access to the APIs. Further details are provided in clause 4.5 of ETSI GS NFV-SOL003 V 1.1.1 [1].

6.12 Pattern: Error reporting

6.12.1 Introduction

In RESTful interfaces, application errors are mapped to HTTP errors. Since HTTP error information is typically not enough to discover the root cause of the error, there is the need to deliver additional application specific error information. The following clauses define such a mechanism to be used by the interfaces specified in the present document.

6.12.2 General mechanism

When an error occurs that prevents the API producer from successfully fulfilling the request, the HTTP response shall include in the response a status code in the range 400..499 (client error) or 500.599 (server error) as defined by the HTTP specification (see IETF RFC 7231, IETF RFC 7232, IETF RFC 7233 and IETF RFC 7235, as well as by IETF RFC 6585). In addition, the response body should contain a JSON representation of a "ProblemDetails" data structure according to IETF RFC 7807 that provides additional details of the error. In that case, as defined by IETF RFC 7807, the "Content-Type" HTTP header shall be set to "application/problem+json".

6.12.3 Type: ProblemDetails

The definition of the general "ProblemDetails" data structure from IETF RFC 7807 is reproduced in table 6.12.3-1. Compared to the general framework defined in IETF RFC 7807, the "status" and "detail" attributes are mandated to be included by the present specification, to ensure that the response contains additional textual information about an error. IETF RFC 7807 foresees extensibility of the "ProblemDetails" type. It is possible that particular APIs in the present document, or particular implementations, define extensions to define additional attributes that provide more information about the error.

The description column only provides some explanation of the meaning to facilitate understanding of the design. For a full description, see IETF RFC 7807.

Table 6.12.3-1: Definition of the ProblemDetails data type

Attribute name	Data type	Cardinality	Description
type	URI	0..1	A URI reference according to IETF RFC 3986 that identifies the problem type. It is encouraged that the URI provides human-readable documentation for the problem (e.g. using HTML) when dereferenced. When this member is not present, its value is assumed to be "about:blank".
title	String	0..1	A short, human-readable summary of the problem type. It should not change from occurrence to occurrence of the problem, except for purposes of localization. If type is given and other than "about:blank", this attribute shall also be provided.
status	Integer	1	The HTTP status code for this occurrence of the problem.
detail	String	1	A human-readable explanation specific to this occurrence of the problem.
instance	URI	0..1	A URI reference that identifies the specific occurrence of the problem. It may yield further information if dereferenced.
(additional attributes)	Not specified.	0..N	Any number of additional attributes, as defined in a specification or by an implementation.

NOTE: It is expected that the minimum set of information returned in ProblemDetails consists of "status" and "detail". For the definition of specific "type" values as well as extension attributes by implementations, guidance can be found in IETF RFC 7807.

6.12.4 Common error situations

The following common error situations are applicable on all REST resources and related HTTP methods specified in the present document, and shall be handled as defined in the present clause.

400 Bad Request: If the request is malformed or syntactically incorrect (e.g. if the request URI contains incorrect query parameters or a syntactically incorrect payload body), the API producer shall respond with this response code. The "ProblemDetails" structure shall be provided, and should include in the "detail" attribute more information about the source of the problem.

400 Bad request: If the request contains a malformed access token, the API producer should respond with this response. The details of the error shall be returned in the WWW-Authenticate HTTP header, as defined in IETF RFC 6750 and IETF RFC 7235. The ProblemDetails structure may be provided.

400 Bad Request: If there is an application error related to the client's input that cannot be easily mapped to any other HTTP response code ("catch all error"), the API producer shall respond with this response code. The "ProblemDetails" structure shall be provided, and shall include in the "detail" attribute more information about the source of the problem.

NOTE 1: It is by design to represent this application error situation with the same HTTP error response code as the previous one.

401 Unauthorized: If the request contains no access token even though one is required, or if the request contains an authorization token that is invalid (e.g. expired or revoked), the API producer should respond with this response. The details of the error shall be returned in the WWW-Authenticate HTTP header, as defined in IETF RFC 6750 and IETF RFC 7235. The ProblemDetails structure may be provided.

403 Forbidden: If the API consumer is not allowed to perform a particular request to a particular resource, the API producer shall respond with this response code. The "ProblemDetails" structure shall be provided. It should include in the "detail" attribute information about the source of the problem, and may indicate how to solve it.

404 Not Found: If the API producer did not find a current representation for the resource addressed by the URI passed in the request, or is not willing to disclose that one exists, it shall respond with this response code. The "ProblemDetails" structure may be provided, including in the "detail" attribute information about the source of the problem, e.g. a wrong resource URI variable.

405 Method Not Allowed: If a particular HTTP method is not supported for a particular resource, the API producer shall respond with this response code. The "ProblemDetails" structure may be omitted in that case.

406 Not Acceptable: If the "Accept" HTTP header does not contain at least one name of a content type that is acceptable to the API producer, the API producer shall respond with this response code. The "ProblemDetails" structure may be omitted in that case.

422 Unprocessable Entity: If the payload body of a request contains syntactically correct data (e.g. well-formed JSON) but the data cannot be processed (e.g. because it fails validation against a schema), the API producer shall respond with this response code. The "ProblemDetails" structure shall be provided, and should include in the "detail" attribute more information about the source of the problem.

NOTE 2: This error response code is only applicable for methods that have a request body.

500 Internal Server Error: If there is an application error not related to the client's input that cannot be easily mapped to any other HTTP response code ("catch all error"), the API producer shall respond with this response code. The "ProblemDetails" structure shall be provided, and shall include in the "detail" attribute more information about the source of the problem.

503 Service Unavailable: If the API producer encounters an internal overload situation of itself or of a system it relies on, it should respond with this response code, following the provisions in IETF RFC 7231 for the use of the "Retry-After" HTTP header and for the alternative to refuse the connection. The "ProblemDetails" structure may be omitted.

NOTE 3: The error handling defined above only applies to REST resources defined in the present document. For the token endpoint defined in IETF RFC 6749, specific error handling applies as defined in IETF RFC 6749.

6.12.5 Specific HTTP error status codes

In general, error response codes used for specific application errors should be mapped to the most similar HTTP error status code and documented for each resource. If no such code is applicable, one of the codes 400 (Bad Request, for client errors) or 500 (Internal Server Error, for server errors) should be used.

Implementations may use additional error response codes on top of the ones defined in the specification, as long as they are valid HTTP response codes; and should include a ProblemDetails structure in the payload body as defined in clause 6.12.2. A list of all valid HTTP response codes and their specification documents can be obtained from the HTTP status code registry at IANA (https://www.iana.org/assignments/http-status-codes).

Annex A: 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.>

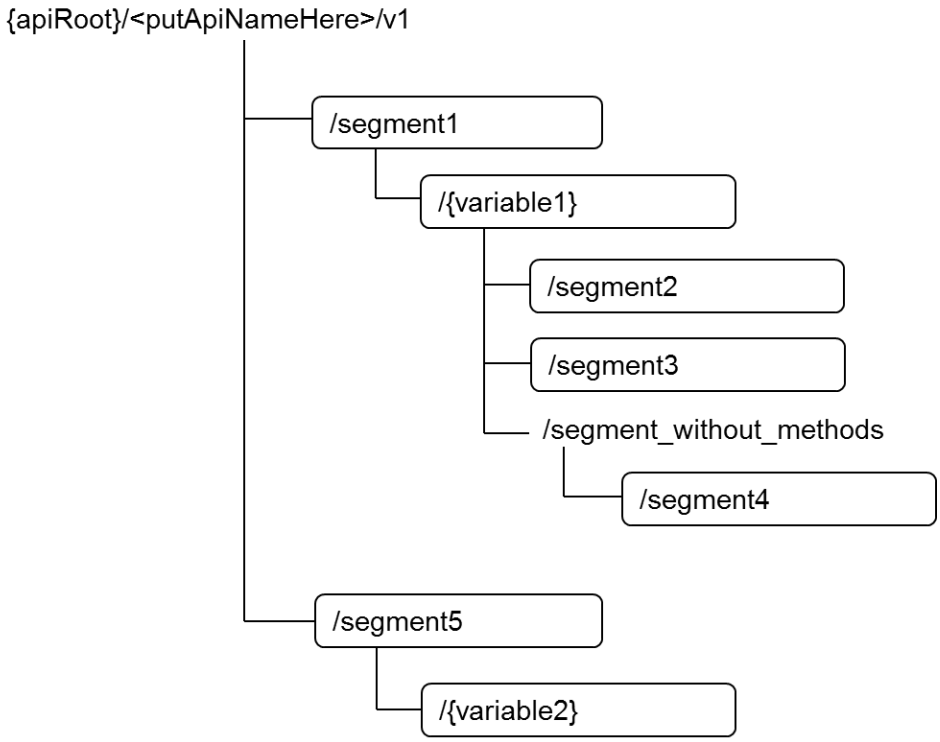
X.2 Resource structure and methods

All resource URIs of this API shall use the base URI specification defined in clause *<4.4>*. The string "*<putApiNameHere>*" shall be used to represent {apiName}. The {apiVersion} shall be set to “v1” for the present specification. All resource URIs in the sub-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.2-1 shows the overall resource URI structure defined for the *<long API name>* API. Table X.2-1 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 node without a box represents a path segment that has none. All node names are examples only>



<Template note: A PPT template for the graph above is available as part of the latest revision of this document>

Figure X.2-1 Resource URI structure of the *<long API name>* interface

<Template note: Overview table of resources and operations>

Table X.2-1: Resources and methods overview of the *<long API name>* interface

Resource name	Resource URI	HTTP METHOD	Meaning
<Resource Meaning>	<relative URI below root>	POST	<short description>
		GET	<short description>
		PUT	<short description>
		PATCH	<short description>
		DELETE	<short description>

<Template note: In the table above, only include sub-rows for those HTTP methods that are applicable to the resource>

<Template note: Start of Example>

Table X.2-2: Resources and methods overview of the Foobar interface

Resource name	Resource URI	HTTP METHOD	Meaning
VNF instances	/vnf_instances	GET	Query multiple VNF instances
		POST	Create a VNF instance resource
Individual VNF instance	/vnf_instances/{instanceId}	GET	Query single VNF instance
		PATCH	Modify VNF instance information
		DELETE	Delete VNF instance resource
Instantiate VNF task	/vnf_instances/{instanceId}/instantiate	POST	Instantiate a VNF

<Template note: End of Example>

X.3 Sequence diagrams

<Template note: this clause is Informative. No normative keywords in this clause>

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

X.3.1 *<Procedure 1>*

<Template note: Add introductory text>

This clause …

<Template note: Add flow diagram using PlantUML tool, see clause 5.1. Don't forget caption. Conventions and example documented in clause 5.2.>

<placeholder for graphics, centered>

Figure X.3.1-1 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.3.2-1, consists of the following steps:

<Template note: Add error handling if applicable>

Error handling: Text text text

X.3.2 *<Procedure 2>*

<Template note: same as clause X.3.1>

X.4 Resources

<Template note: this clause is Normative.

X.4.1 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.4.2 Resource: *<Meaning>*

X.4.2.1 Description

This resource represents <something>. The client 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 client can use this resource to create individual VNF instance resources, and to query VNF instances.

<Template note: End of Example

X.4.2.2 Resource definition

The resource URI is:

{apiRoot}/*<putApiNameHere>*/v1/*<foo_bar>*

This resource shall support the resource URI variables defined in table X.4.2.2-1.

Table X.4.2.2-1: Resource URI variables for this resource

Name	Definition
apiRoot	See clause 4.2
<name>	*<definition>*

X.4.2.3 Resource Methods

X.4.2.3.1 POST

<Template note: Alternative 2>

The POST method *... <Meaning(s) of the operation in API space>*.

<Template note: Alternative 2>

Not supported.

<Template note: End of alternatives>

<Template note: Start Example>

The POST method creates a fooBar object.

<Template note: End Example>

This method shall follow the provisions specified in the tables X.4.2.3.1-1 and X.4.2.3.1-2 for URI query parameters, request and response data structures, and response codes.

Table X.4.2.3.1-1: 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.4.2.3.1-2: 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 RFC7231 etc.>*	*<Statement defining the case in which this response is returned (success or error)>* *<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 clauses 6.12.4 / 6.12.5 <adapt clause numbers to your spec>	4xx/5xx	In addition to the response codes defined above, any common error response code as defined in clause 6.12.4, and any other valid HTTP error response as defined in 6.12.5, may be returned.

<Template note: Start of Example>

Table X.4.2.3.1-3: URI query parameters supported by the POST method on this resource

Name	Cardinality	Remarks
foo_bar	0..1	The foo bar

Table X.4.2.3.1-4: Details of the POST request/response on this resource

Request body	**Data type**	**Cardinality**	**Description**	
	FooBarCreateRequest	1	Foobar instance creation parameters	
Response body	**Data type**	**Cardinality**	**Response Codes**	**Description**
	FooBarInstance	1	201 Created	The foobar instance was created successfully. The response body shall contain a representation of the created foobar 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	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	The resource URI was incorrect. In the returned ProblemDetails structure, the "detail" attribute should convey more information about the error.
	ProblemDetails	See clauses 6.12.4 / 6.12.5	4xx/5xx	In addition to the response codes defined above, any common error response code as defined in clause 6.12.4, and any other valid HTTP error response as defined in 6.12.5, 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.4.2.3.2 GET

<same structure as for POST>

X.4.2.3.3 PUT

<same structure as for POST>

X.4.2.3.4 PATCH

<same structure as for POST>

X.4.2.3.5 DELETE

<same structure as for POST>

X.5 Data model

<Template note: this clause is normative.>

X.5.1 Introduction

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

X.5.2 Resource and notification data types

X.5.2.1 Introduction

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

X.5.2.2 Type: *<TypeName1>*

This type represents a <…>. It shall comply with the provisions defined in table X.5.2.2-1.

<Template note: Data type names in UpperCamel>

*<Template note: Short descriptive text of this data type, followed by a Table. Choices to be defined as follows: **"NOTE: One of "firstChoice" or at least one of "secondChoice" but not both shall be present."**>*

<Template note:

- *"Attribute name" provides the name of the attribute in lowerCamel*

- *"Data type" may provide the name of a **named data type** (structured, simple or enum) that is defined elsewhere in this document, or in a referenced document (e.g., a referenced type from another document.) In the latter case, a reference to the defining document shall be included in the "Description" column.*
- *"Data type" may also indicate 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 a number of closing angular brackets ">" that represent the level of nesting.*
- *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 structures are used if the same structure only appears in one or very few data types.*

  *"Data type" may also indicate 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.*

- *"Cardinality" defines the allowed number of occurrence*

- *"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 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".*

>

Table X.5.2.2-1: Definition of the *<TypeName1>* data type

Attribute name	Data type	Cardinality	Description

<Template note: Start of Example>

This type represents a foobar indicator. Typically, this corresponds to one distinct stream signalled by foobar. It shall comply with the provisions defined in table X.5.2.2-2.

Table X.5.2.2-2: Definition of the FooBarIndicator data type

Attribute name	Data type	Cardinality	Description
type	FooBarType	1	Indicates whether this is a foo, boo or hoo stream.
entryIdx	UnsignedInt	0..N	The index of the entry in the signaling table for correlation purposes, starting at 0.
terminationType	Enum (inlined)	1	Signals type of termination. Permitted values: - FORCEFUL: The VNFM will shut down the VNF and release the resources immediately after accepting the request. - GRACEFUL: The VNFM will first arrange to take the VNF out of service after accepting the request. Once the operation is successful or once the timer value specified in the "gracefulTerminationTimeout" attribute expires, the VNFM will shut down the VNF and release the resources.
firstChoice	MyChoiceOneType	0..1	First choice. See NOTE.
secondChoice	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	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.			

<Template note: End of Example>

X.5.3 Referenced structured data types

X.5.3.1 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.5.3.2 Type: *<TypeName3>*

<Template note: Same structure as in X.5.2.2>

X.5.4 Referenced simple data types and enumerations

X.5.4.1 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.5.4.2 Simple data types

The simple data types defined in table X.5.4.2-1 shall be supported.

Table X.5.4.2-1: Simple data types

Type name	Description

X.5.4.3 Enumeration: *<TypeName4>*

The enumeration *<TypeName4>* represents *<something>*. It shall comply with the provisions defined in table X.5.4.3-1.

Table X.5.4.3-1: Enumeration *<TypeName4>*

Enumeration value	Description

Annex B: History

Revision	Date	Changes
(16)00070	12 Sept 2016	Skeleton provided
(16)00070r1	22 Sept 2016	Included contributions agreed at the SOL#10 meeting in Sophia Antipolis - NFVSOL(16)000075r1_API_conventions_process - NFVSOL(16)000069r3_SOL_REST_API_conventions_for_names_in_URIs - NFVSOL(16)000074r5_Conventions_for_message_flows
(16)00070r2	13 Oct 2016	Included contributions agreed at the SOL#11 call - NFVSOL(16)000090_SOL_REST_API_conventions_for_names_in_data_structures - NFVSOL(16)000104r2_SOL002 SOL003 REST API document template_update
(16)00070r3	13 Oct 2016	Replaces r2 package file which was faulty.
(16)00070r4	29 Oct 2016	Included contributions agreed at the SOL#12 meeting, Bundang - NFVSOL(16)000123r2_Conventions_change_adding_inline_structures - NFVSOL(16)000125r2_Conventions_for_JSON_Schema (rapporteur's change: Style of JSON schema reference aligned with agreement in document 100r3 (JSON Annex for SOL003) - NFVSOL(16)000130r2_Conventions_Pattern_for_subscribe_notify - NFVSOL(16)000133r1_Conventions_-_Pattern_for_links (rapporteur's change: removed reference to JSON language from "5.2 Pattern for links" because the present document has no references clause) - NFVSOL(16)000135_Conventions_for_opt_group_in_flows - NFVSOL(16)000136_Conventions_change_of_definitition_of_data_schema_for_entity
(16)00070r5	12 Nov 2016	Included contributions agreed at the SOL#13 call - NFVSOL(16)000145r2_Conventions_simplifying_the_table
(16)00070r6	18 Nov 2016	Included contributions agreed at the SOL#15 call - NFVSOL(16)000142r2_Conventions_for_inline_enums with the following modification: EditHelp discourages the use of own paragraph style templates in GSs. Therefore, the following sentence has not been incorporated into the present document when implementing NFVSOL(16)000142r2: "Microsoft Word paragraph style shall be "SOLInlineEnum" (copy the example from the table below to inject the style into your document)".
(16)00070r7	15 Dec 2016	Included contributions agreed at the SOL#16 F2F in Shenzhen - NFVSOL(16)000199_Template_update - NFVSOL(16)000209_Conventions_nicer_tables
(16)00070r8	12 Jan 2017	Included contributions agreed in the SOL#17 call - NFVSOL(17)000002_Conventions_splitting_subscribe_and_notify
(17)00050	25 Jan 2017	Included contributions agreed at SOL#18 F2F - NFVSOL(16)000185r2 - NFVSOL(17)000030_Conventions_change_of_error_handling_in_resources_clause

(17)00050r1	30 Jan 2017	Included one more agreed contribution from SOL#18 - NFVSOL(17)000035r1_SOL003_LCM_flow_update
(17)00050r2	28 Feb 2017	Included contributions from SOL#19 call and SOL#20 F2F (Bilbao) - NFVSOL(17)000054_Conventions_Replace_URL_by_URI - NFVSOL(17)000084r1_Template_changes_for_error_handling - Removed "Data type" column from table X.3.3.3.1-1 to align with a decision made for the GS - NFVSOL(17)000106_Conventions_Document_NFVSOL_17_000050__Swagger_Representatio - NFVSOL(17)000111_SOL003_Conventions_move_Resource_structure_up_in_the_TOC - Aligned terminology LifecycleChangeNotificatzion → VnfLcmOperationOccurrenceNotification
(17)00050r3	27 Mar 2017	Included contributions from SOL#22 F2F Piscataway - NFVSOL(17)000121_Conventions_three_parts_of_remarks_column - NFVSOL(17)000187r1_SOL002_SOL003_Conventions_global_fix_for_normative_statement
(17)00050r4	27 Jul 2017	Included NFVSOL(17)000502r3 to align with patterns that were introduced during the development of SOL003, and to prepare the document for being shared on the ETSI NFV external wiki.