ETSI GS NFV-SOL 013 V2.6.1 (2019-03)

Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; Specification of common aspects 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
DGS/NFV-SOL013ed261

Keywords
API, 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 5
Foreword 5
Modal verbs terminology 5
1 Scope 6
2 References 6
2.1 Normative references 6
2.2 Informative references 7
3 Definition of terms, symbols and abbreviations 8
3.1 Terms 8
3.2 Symbols 8
3.3 Abbreviations 8
4 HTTP usage 8
4.1 URI structure and supported content formats 8
4.2 Usage of HTTP header fields 9
4.2.1 Introduction 9
4.2.2 Request header fields 9
4.2.3 Response header fields 10
5 Result set control 11
5.1 Introduction 11
5.2 Attribute-based filtering 11
5.2.1 Overview and example (informative) 11
5.2.2 Specification 12
5.3 Attribute selectors 14
5.3.1 Overview and example (informative) 14
5.3.2 Specification 14
5.3.2.1 GET request 14
5.3.2.2 GET response 15
5.4 Handling of large query results 15
5.4.1 Overview 15
5.4.2 Specification 16
5.4.2.1 Alternatives 16
5.4.2.2 Error response 16
5.4.2.3 Paged response 16
6 Error reporting 17
6.1 Introduction 17
6.2 General mechanism 17
6.3 Type: ProblemDetails 17
6.4 Common error situations 18
7 Common data types 20
7.1 Structured data types 20
7.1.1 Introduction 20
7.1.2 Type: Object 20
7.1.3 Type: Link 20
7.1.4 Type: NotificationLink 20
7.1.5 Type: KeyValuePairs 20
7.1.6 Type: ApiVersionInformation 21
7.2 Simple data types and enumerations 21
7.2.1 Introduction 21
7.2.2 Simple data types 21
7.2.3 Enumerations 22
8 Authorization of API requests and notifications 22
8.1 Introduction 22

8.2 Flows (informative) 23
8.2.1 General 23
8.2.2 Authorization of API requests using OAuth 2.0 access tokens 23
8.2.3 Authorization of API requests using TLS certificates 25
8.2.4 Authorization of notifications using the HTTP Basic authentication scheme 26
8.2.5 Authorization of notifications using OAuth 2.0 access tokens 27
8.2.6 Authorization of notifications using TLS certificates 29
8.3 Specification 31
8.3.1 Introduction 31
8.3.2 General mechanism 31
8.3.3 Authorizing API requests 31
8.3.4 Authorizing the sending of notifications 32
8.3.5 Client roles 33
8.3.6 Negotiation of the authorization method 34
8.3.6.1 Authorization of API requests 34
8.3.6.2 Authorization of notification requests 36

9 Version management 37
9.1 Version identifiers and parameters 37
9.1.1 Version identifiers 37
9.1.2 Version parameters 37
9.2 Rules for incrementing version identifier fields 37
9.2.1 General 37
9.2.2 Examples of backward and non-backward compatible changes 38
9.3 Version information retrieval 39
9.3.1 General 39
9.3.2 Resource structure and methods 39
9.3.3 Resource: API versions 40
9.3.3.1 Description 40
9.3.3.2 Resource definition 40
9.3.3.3 Resource methods 40
9.3.3.3.1 POST 40
9.3.3.3.2 GET 40
9.3.3.3.3 PUT 41
9.3.3.3.4 PATCH 41
9.3.3.3.5 DELETE 41
9.4 Version signaling 41

Annex A (informative): Change History 42

History 43

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 specifies common aspects of RESTful protocols and data models for ETSI NFV management and orchestration (MANO) interfaces.

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] IETF RFC 2818: "HTTP Over TLS".

NOTE: Available at https://tools.ietf.org/html/rfc2818.

[2] IETF RFC 3339: "Date and Time on the Internet: Timestamps".

NOTE: Available at https://tools.ietf.org/html/rfc3339.

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

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

[4] IETF RFC 4918: "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)".

NOTE: Available at https://tools.ietf.org/html/rfc4918.

[5] IETF RFC 5246: "The Transport Layer Security (TLS) Protocol Version 1.2".

NOTE: Available at https://tools.ietf.org/html/rfc5246.

[6] IETF RFC 6585: "Additional HTTP Status Codes".

NOTE: Available at https://tools.ietf.org/html/rfc6585.

[7] IETF RFC 6749: "The OAuth 2.0 Authorization Framework".

NOTE: Available from https://tools.ietf.org/html/rfc6749.

[8] IETF RFC 6750: "The OAuth 2.0 Authorization Framework: Bearer Token Usage".

NOTE: Available from https://tools.ietf.org/html/rfc6750.

[9] IETF RFC 8259: "The JavaScript Object Notation (JSON) Data Interchange Format".

NOTE: Available at https://tools.ietf.org/html/rfc8259.

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

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

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

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

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

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

[13] IETF RFC 7235: "Hypertext Transfer Protocol (HTTP/1.1): Authentication".

NOTE: Available at https://tools.ietf.org/html/rfc7235.

[14] IETF RFC 7617: "The 'Basic' HTTP Authentication Scheme".

NOTE: Available from https://tools.ietf.org/html/rfc7617.

[15] IETF RFC 7807: "Problem Details for HTTP APIs".

NOTE: Available at https://tools.ietf.org/html/rfc7807.

[16] IETF RFC 6901: "JavaScript Object Notation (JSON) Pointer".

NOTE: Available at https://tools.ietf.org/html/rfc6901.

[17] IETF RFC 8288: "Web Linking".

NOTE: Available at https://tools.ietf.org/html/rfc8288.

[18] Semantic Versioning 2.0.0.

NOTE: Available at https://semver.org/.

[19] IETF RFC 4229: "HTTP Header Field Registrations".

NOTE: Available at https://tools.ietf.org/html/rfc4229.

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] ETSI TS 133 310: "Universal Mobile Telecommunications System (UMTS); LTE; Network Domain Security (NDS); Authentication Framework (AF) (3GPP TS 33.310)".

[i.3] Hypertext Transfer Protocol (HTTP) Status Code Registry at IANA.

NOTE: Available at http://www.iana.org/assignments/http-status-codes.

[i.4] ETSI NFV OpenAPI repository.

NOTE: Available at 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
EM	Element Manager
ETSI	European Telecommunications Standards Institute
GMT	Greenwich Mean Time
GS	Group Specification
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
MAC	Medium Access Control
MANO	Management and Orchestration
MIME	Multipurpose Internet Mail Extensions
NFV	Network Functions Virtualisation
NFVO	NFV Orchestrator
REST	Representational State Transfer
RFC	Request For Comments
TLS	Transport Layer Security
URI	Uniform Resource Identifier
VIM	Virtualised Infrastructure Manager
VNF	Virtualised Network Function
VNFM	VNF Manager

4 HTTP usage

4.1 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, except the "API versions" resource which shall follow the rules specified in clause 9.3:

{apiRoot}/{apiName}/{apiMajorVersion}/

where:

{apiRoot}	indicates the scheme ("http" or "https"), the host name and optional port, and an optional sequence of path segments that together represent a prefix path.

EXAMPLE: http://orchestrator.example.com/nfv_apis/abc

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

{apiMajorVersion} indicates the current major version (see clause 9.1) 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 8259 [9]) 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 [1]). TLS version 1.2 as defined by IETF RFC 5246 [5] shall be supported.

NOTE 1: The HTTP protocol elements mentioned in the present document originate from the HTTP specification; HTTPS runs the HTTP protocol in a TLS layer. The present document therefore uses the statement above to mention "HTTP request", "HTTP header", etc., without explicitly calling out whether or not these are run over TLS.

NOTE 2: There are a number of best practices and guidelines how to configure and implement TLS 1.2 in a secure manner, as security threats evolve. A detailed specification of those is beyond the scope of the present document; the reader is referred to external documentation such as annex E of ETSI TS 133 310 [i.2].

All resource URIs of the API shall comply with the URI syntax as defined in IETF RFC 3986 [3]. An implementation that dynamically generates resource URI parts (individual path segments, sequences of path segments that are separated by "/", query parameter values) shall ensure that these parts only use the character set that is allowed by IETF RFC 3986 [3] for these parts.

NOTE 3: This means that characters not part of this allowed set are escaped using percent-encoding as defined by IETF RFC 3986 [3].

Unless otherwise specified explicitly, all request URI parameters that are part of the path of the resource URI shall be individual path segments, i.e. shall not contain the "/" character.

NOTE 4: A request URI parameter is denoted by a string in curly brackets, e.g. {subscriptionId}.

4.2 Usage of HTTP header fields

4.2.1 Introduction

HTTP headers are components of the header section of the HTTP request and response messages. They contain the information about the server/client and metadata of the transaction. The use of HTTP header fields shall comply with the provisions defined for those header fields in the specifications referenced from tables 4.2.2-1 and 4.2.3-1. The following clauses describe the HTTP header fields that are explicitly mentioned in the present document.

4.2.2 Request header fields

This clause describes the usage of HTTP header fields of the request messages applicable to the APIs defined in the present document. The HTTP header fields used in the request messages are specified in table 4.2.2-1.

Table 4.2.2-1: Header fields supported in the request message

Header field name	Reference	Example	Descriptions
Accept	IETF RFC 7231 [10]	application/json	Content-Types that are acceptable for the response. This header field shall be present if the response is expected to have a non-empty message body.
Content-Type	IETF RFC 7231 [10]	application/json	The MIME type of the body of the request. This header field shall be present if the request has a non-empty message body.
Authorization	IETF RFC 7235 [13]	Bearer mF_9.B5f-4.1JqM	The authorization token for the request. Details are specified in clause 8.3.
Range	IETF RFC 7233 [12]	1 000-2 000	Requested range of bytes from a file.
Version	IETF RFC 4229 [19]	1.2.0 or 1.2.0-impl:example.com:myProduct:4	Version of the API requested to use when responding to this request.

4.2.3 Response header fields

This clause describes the usage of HTTP header fields of the response messages applicable to the APIs defined in the present document. The HTTP header fields used in the response messages are specified in table 4.2.3-1.

Table 4.2.3-1: Header fields supported in the response message

Header field name	Reference	Example	Descriptions
Content-Type	IETF RFC 7231 [10]	application/json	The MIME type of the body of the response. This header field shall be present if the response has a non-empty message body.
Location	IETF RFC 7231 [10]	http://www.example.com/vnflcm/v1/vnf_instances/123	Used in redirection, or when a new resource has been created. This header field shall be present if the response status code is 201 or 3xx. In the present document this header field is also used if the response status code is 202 and a new resource was created.
WWW-Authenticate	IETF RFC 7235 [13]	Bearer realm="example"	Challenge if the corresponding HTTP request has not provided authorization, or error details if the corresponding HTTP request has provided an invalid authorization token.
Accept-Ranges	IETF RFC 7233 [12]	bytes	Used by the server to signal whether or not it supports ranges for certain resources.
Content-Range	IETF RFC 7233 [12]	bytes 21 010 - 47 021/47 022	Signals the byte range that is contained in the response, and the total length of the file.
Retry-After	IETF RFC 7231 [10]	Fri, 31 Dec 1999 23:59:59 GMT or 120	Used to indicate how long the user agent ought to wait before making a follow-up request. It can be used with 503 responses. The value of this field can be an HTTP-date or a number of seconds to delay after the response is received.

Header field name	Reference	Example	Descriptions
Link	IETF RFC 8288 [17]	<http://example.com/resources?nextpage_opaque_marker=abc123>; rel="next"	Reference to other resources. Used for paging in the present document, see clause 5.4.2.1.
Version	IETF RFC 4229 [19]	1.2.0 or 1.2.0-impl:example.com:myProduct:4	Version of the API requested to use when responding to this request.

5 Result set control

5.1 Introduction

This clause specifies procedures that allow to control the size of the result set of GET requests w.r.t. the number of entries in a response list (using attribute-based filtering) or w.r.t. the number of attributes returned in a response (using attribute selection).

5.2 Attribute-based filtering

5.2.1 Overview and example (informative)

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":4, "color":"blue"}]}
```

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":4, "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?filter=(eq,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 "(eq,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 two expressions in the filter "(eq,parts/color,green);(eq,parts/id,3)" would be evaluated together for each entry in the array "parts". As the result, obj2 matches the filter.

5.2.2 Specification

An attribute-based filter shall be represented by a URI query parameter named "filter". The value of this parameter shall consist of one or more strings formatted according to "simpleFilterExpr", concatenated using the ";" character:

```
simpleFilterExprOne     := <opOne>","<attrName>["/"<attrName>]*","<value>
simpleFilterExprMulti   := <opMulti>","<attrName>["/"<attrName>]*","<value>[","<value>]*
simpleFilterExpr        := "("<simpleFilterExprOne>")" | "("<simpleFilterExprMulti>")"
filterExpr              := <simpleFilterExpr>[";"<simpleFilterExpr>]*
filter                  := "filter"=<filterExpr>
opOne                   := "eq" | "neq" | "gt" | "lt" | "gte" | "lte"
opMulti                 := "in" | "nin" | "cont" | "ncont"
attrName                := string
value                   := string
```

where:

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

"AttrName" is the name of one attribute in the data type that defines the representation of the resource. The slash ("/") character in "simpleFilterExprOne" and " simpleFilterExprMulti" allows concatenation of <attrName> entries to filter by attributes deeper in the hierarchy of a structured document. The elements "opOne" and "opMulti" stand for the comparison operators (accepting one comparison value or a list of such values). If the expression has concatenated <attrName> entries, it means that the operator 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, denoted by ";".

In a concatenation of <attrName> entries in a <simpleFilterExprOne> or <simpleFilterExprMulti>, 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 <simpleFilterExprOne> or <simpleFilterExprMulti> shall not be structured but shall be of a simple (scalar) type such as String, Number, Boolean or DateTime, or shall be an array of simple (scalar) values. 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 listed in table 5.2.2-1 shall be supported.

Table 5.2.2-1: Operators for attribute-based filtering

Operator with parameters	Meaning
eq,<attrName>,<value>	Attribute **equal** to <value>
neq,<attrName>,<value>	Attribute **not equal** to <value>
in,<attrName>,<value>[,<value>]*	Attribute equal to one of the values in the list ("**in set**" relationship)
nin,<attrName>,<value>[,<value>]*	Attribute not equal to any of the values in the list ("**not in set**" relationship)
gt,<attrName>,<value>	Attribute **greater than** <value>
gte,<attrName>,<value>	Attribute **greater than or equal** to <value>
lt,<attrName>,<value>	Attribute **less than** <value>
lte,<attrName>,<value>	Attribute **less than or equal** to <value>
cont,<attrName>,<value>[,<value>]*	String attribute **contains** (at least) one of the values in the list
ncont,<attrName>,<value>[,<value>]*	String attribute **does not contain** any of the values in the list

Table 5.2.2-2: Applicability of the operators to data types

Operator	String	Number	DateTime	Enumeration	Boolean
eq	x	x	-	x	x
neq	x	x	-	x	x
in	x	x	-	x	-
nin	x	x	-	x	-
gt	x	x	x	-	-
gte	x	x	x	-	-
lt	x	x	x	-	-
lte	x	x	x	-	-
cont	x	-	-	-	-
ncont	x	-	-	-	-

Table 5.2.2-2 defines which operators are applicable for which data types. All combinations marked with a "x" shall be supported.

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 of type Number, String, Boolean, Enum or DateTime. The content of a <value> entry shall be formatted the same way as the representation of the related attribute in the resource representation: The syntax of DateTime <value> entries shall follow the "date-time" production of IETF RFC 3339 [2]. The syntax of Boolean and Number <value> entries shall follow IETF RFC 8259 [9].

A <value> entry of type String shall be enclosed in single quotes (') if it contains any of the characters ")", "'" or ",", and may be enclosed in single quotes otherwise. Any single quote (') character contained in a <value> entry shall be represented as a sequence of two single quote characters.

The "/" and "~" characters in <attrName> shall be escaped according to the rules defined in section 3 of IETF RFC 6901 [16]. The "," character in <attrName> shall be escaped by replacing it with "~a".

In the resulting <filterExpr>, percent-encoding as defined in IETF RFC 3986 [3] shall be applied to the characters that are not allowed in a URI query part according to Appendix A of IETF RFC 3986 [3], and to the ampersand "&" character.

NOTE: In addition to the statement on percent-encoding above, it is reminded that the percent "%" character is always percent-encoded when used in parts of a URI, according to IETF RFC 3986 [3].

Attribute-based filters are supported for certain resources. Details are defined in the clauses specifying the actual resources.

5.3 Attribute selectors

5.3.1 Overview and example (informative)

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.

5.3.2 Specification

5.3.2.1 GET request

The URI query parameters for attribute selection are defined in table 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 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 the "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.

The "/" and "~" characters in attribute names in an attribute selector shall be escaped according to the rules defined in section 3 of IETF RFC 6901 [16]. The "," character in attribute names in an attribute selector shall be escaped by replacing it with "~a". Further, percent-encoding as defined in IETF RFC 3986 [3] shall be applied to the characters that are not allowed in a URI query part according to Appendix A of IETF RFC 3986 [3], and to the ampersand "&" character.

5.3.2.2 GET response

Table 5.3.2.2-1 defines the valid parameter combinations in a GET request and their effect on the GET response.

Table 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 document 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 document 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 API producer 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"}
}
```

5.4 Handling of large query results

5.4.1 Overview

If the response to a query to a container resource (i.e. a resource that contains child resources whose representations will be returned when responding to a GET request) will become so large that the response will adversely affect the performance of the server, the server either rejects the request with a 400 Bad Request response, or the server provides a paged response, i.e. it returns only a subset of the query result in the response, and also provides information how to obtain the remainder of the query result.

When returning a paged response, depending on the underlying storage organization, it might be problematic for the server to determine the actual size of the result; however, it is usually possible to determine whether there will be additional results returned when knowing, for the last entry in the returned page, the position in the overall query result or some other property that has ordering semantics. For example, the time of creation of a resource has such an ordering property. When using such an (implementation-specific) property, the API producer can correctly handle deletions of child resources that happen between sending the first page of the query result, and sending the next page. It cannot be guaranteed that child resources inserted between returning subsequent pages can be considered in the query result, however, it shall be guaranteed that this does not lead to skipping of entries that have existed prior to insertion.

At minimum, a paged response needs to contain information telling the API consumer that the response is paged, and how to obtain the next page of information. For that purpose, a link to obtain the next page is returned in an HTTP header, containing a parameter that is opaque to the API consumer, but that allows the API producer to determine the start of the next page.

NOTE: In the present document, this functionality is designed for overload protection only. Additional functionality, such as configuring the page size by the API consumer, determining the size of the overall query result or the number of pages, and determining the previous page, is left outside the scope of the present document.

5.4.2 Specification

5.4.2.1 Alternatives

For each container resource (i.e. a resource that contains child resources whose representations will be returned when responding to a GET request), the API producer shall support one of the following two behaviours specified below to handle the case that a response to a query (GET request) will become so large that the response will adversely affect performance:

1) Return an error response, as defined in clause 5.4.2.2.

2) Return the result in a paged manner, as defined in clause 5.4.2.3.

5.4.2.2 Error response

In this alternative, the server shall reject the request with a 400 Bad Request response, shall include the "ProblemDetails" payload body, and shall provide in the "detail" attribute more information about the error.

This error code indicates to the API consumer that with the given attribute-based filtering query (or absence thereof), the response would have been so big that performance is adversely affected. The client can obtain a query result by specifying a (more restrictive) attribute-based filtering query (see clause 5.2).

5.4.2.3 Paged response

In this alternative, the API producer shall provide a response that contains a first page (subset) of the results to the query, and shall include a LINK HTTP header (see IETF RFC 8288 [17]) with the "rel" attribute set to "next", which communicates a URI that allows to obtain the next page of results to the original query.

The API consumer can send a GET request to the URI communicated in the LINK header to obtain the next page of results. The response which returns that next page shall contain the LINK header to point to the next page, as specified above, unless there are no further pages available in which case the LINK header shall be omitted.

To allow the API producer to determine the start of the next page, the LINK header shall contain the URI query parameter "nextpage_opaque_marker" whose value is chosen by the API producer. This parameter has no meaning for the API consumer, but is echoed back by the API consumer to the API producer when requesting the next page. The URI in the link header may include further parameters, such as those passed in the original request.

The size of each page may be chosen by the API provider, and may vary from page to page. The maximum page size is determined by means outside the scope of the present document.

The response need not contain entries that correspond to child resources which were created after the original query was issued.

6 Error reporting

6.1 Introduction

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

6.2 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 [10], IETF RFC 7232 [11], IETF RFC 7233 [12] and IETF RFC 7235 [13], as well as by IETF RFC 6585 [6]). In addition, the response body should contain a JSON representation of a "ProblemDetails" data structure according to IETF RFC 7807 [15] that provides additional details of the error. In that case, as defined by IETF RFC 7807 [15], the "Content-Type" HTTP header shall be set to "application/problem+json".

6.3 Type: ProblemDetails

The definition of the general "ProblemDetails" data structure from IETF RFC 7807 [15] is reproduced in table 6.3-1. Compared to the general framework defined in IETF RFC 7807 [15], the "status" and "detail" attributes are mandated to be included by the present document, to ensure that the response contains additional textual information about an error. IETF RFC 7807 [15] 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 [15].

Table 6.3-1: Definition of the ProblemDetails data type

Attribute name	Data type	Cardinality	Description
type	URI	0..1	A URI reference according to IETF RFC 3986 [3] 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 [15].			

6.4 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. The full definition of each error code can be obtained from the referenced specification.

In general, error response codes used for application errors should be mapped to the most similar HTTP error status code. 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 listed in this clause, as long as they are valid HTTP response codes; and should include a ProblemDetails structure in the payload body as defined in clause 6.3. A list of all valid HTTP response codes and their specification documents can be obtained from the HTTP status code registry [i.3].

NOTE 1: The error handling defined in this clause only applies to REST resources defined in the present document. For the token endpoint defined in IETF RFC 6749 [7] and re-used in the present document as defined in clause 8.3, the error handling provisions are defined in clause 8.3.

400 Bad Request: If the request is malformed or syntactically incorrect (e.g. if the request URI contains incorrect query parameters or the payload body contains a syntactically incorrect data structure), the API producer shall respond with this response code. More details are defined in IETF RFC 7231 [10]. 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 response to a GET request which queries a container resource would be so big that the performance of the API producer is adversely affected, and the API producer does not support paging for the affected resource, it 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 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 2: It is by design to represent these application error situations with the same HTTP error response code 400.

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 [8]. The ProblemDetails structure may be provided.

NOTE 3: The use of this HTTP error response code described above is applicable to the use of the OAuth 2.0 for the authorization of API requests and notifications, as defined in clauses 8.3.3 and 8.3.4.

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 [8] and IETF RFC 7235 [13]. 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. More details are defined in IETF RFC 7231 [10]. 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. A typical reason for this error can e.g. be that resource URI variables were set wrongly. More details are defined in IETF RFC 7231 [10]. 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.

NOTE 4: This response code is not appropriate in case the resource addressed by the URI is a container resource which is designed to contain child resources, but does not contain any child resource at the time the request is received. For a GET request to an existing empty container resource, a typical response contains a 200 OK response code and a payload body with an empty array.

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.

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.

413 Payload Too Large: If the payload body of a request is larger than the amount of data the API producer is willing or able to process, it shall respond with this response code, following the provisions in IETF RFC 7231 [10] for the use of the "Retry-After" HTTP header and for closing the connection. The "ProblemDetails" structure may be omitted.

414 URI Too Long: If the request URI of a request is longer than the API producer is willing or able to process, it shall respond with this response code. This condition can e.g. be caused by passing long queries in the request URI of a GET request. More details are defined in IETF RFC 7231 [10]. The "ProblemDetails" structure may be omitted.

422 Unprocessable Entity: If the content type of the payload body is supported and 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. More details are defined in IETF RFC 4918 [4]. The "ProblemDetails" structure shall be provided, and should include in the "detail" attribute more information about the source of the problem.

NOTE 5: This error response code is only applicable for methods that have a request body.

429 Too Many Requests: If the API consumer has sent too many requests in a defined period of time and the API producer is able to detect that condition ("rate limiting"), the API producer shall respond with this response code, following the provisions in IETF RFC 6585 [6] for the use of the "Retry-After" HTTP header. The "ProblemDetails" structure shall be provided, and shall include in the "detail" attribute more information about the source of the problem.

NOTE 6: The period of time and allowed number of requests are configured within the API producer by means outside the scope of the present document.

500 Internal Server Error: If the Server is unable to process the request, and retrying the same request later might eventually succeed, the server shall respond with this response code. Further, 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. More details are defined in IETF RFC 7231 [10]. 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 [10] for the use of the "Retry-After" HTTP header and for the alternative to refuse the connection. The "ProblemDetails" structure may be omitted.

504 Gateway Timeout: If the API producer encounters a timeout while waiting for a response from an upstream server (i.e. a server that the API producer communicates with when fulfilling a request), it should respond with this response code. More details are defined in IETF RFC 7231 [10]. The "ProblemDetails" structure may be omitted.

7 Common data types

7.1 Structured data types

7.1.1 Introduction

This clause defines data structures that are referenced from data structures in multiple interfaces.

7.1.2 Type: Object

An object contains structured data, and shall comply with the provisions of clause 4 of IETF RFC 8259 [9].

7.1.3 Type: Link

This type represents a link to a resource using an absolute URI. It shall comply with the provisions defined in table 7.1.3-1.

Table 7.1.3-1: Definition of the Link data type

Attribute name	Data type	Cardinality	Description
href	Uri	1	URI of another resource referenced from a resource. Shall be an absolute URI (i.e. a URI that contains {apiRoot}).

7.1.4 Type: NotificationLink

This type represents a link to a resource in a notification, using an absolute or relative URI. It shall comply with the provisions defined in table 7.1.4-1.

Table 7.1.4-1: Definition of the NotificationLink data type

Attribute name	Data type	Cardinality	Description
href	Uri	1	URI of a resource referenced from a notification. Should be an absolute URI (i.e. a URI that contains {apiRoot}), however, may be a relative URI (i.e. a URI where the {apiRoot} part is omitted) if the {apiRoot} information is not available.

7.1.5 Type: KeyValuePairs

This type represents a list of key-value pairs. The order of the pairs in the list is not significant. In JSON, a set of key-value pairs is represented as an object. It shall comply with the provisions defined in clause 4 of IETF RFC 8259 [9]. In the following example, a list of key-value pairs with four keys ("aString", "aNumber", "anArray" and "anObject") is provided to illustrate that the values associated with different keys can be of different type.

EXAMPLE:

```
{
    "aString" : "ETSI NFV SOL",
    "aNumber" : 0.03,
    "anArray" : [1,2,3],
    "anObject" : {"organization" : "ETSI", "isg" : "NFV", workingGroup" : "SOL"}
}
```

7.1.6 Type: ApiVersionInformation

This type represents API version information. It shall comply with the provisions defined in table 7.1.6-1.

Table 7.1.6-1: ApiVersionInformation data type

Attribute name	Data type	Cardinality	Description
uriPrefix	String	1	Specifies the URI prefix for the API, in the following form {apiRoot}/{apiName}/{apiMajorVersion}/.
apiVersions	Structure (inlined)	1..N	Version(s) supported for the API signaled by the uriPrefix attribute.
>version	String	1	Identifies a supported version. The value of the version attribute shall be a version identifier as specified in clause 9.1.
>isDeprecated	Boolean	0..1	If such information is available, this attribute indicates whether use of the version signaled by the version attribute is deprecated (true) or not (false). See note.
>retirementDate	DateTime	0..1	The date and time after which the API version will no longer be supported. This attribute may be included if the value of the isDeprecated attribute is set to true and shall be absent otherwise.
NOTE: A deprecated version is still supported by the API producer but is recommended not to be used any longer. When a version is no longer supported, it does not appear in the response body.			

7.2 Simple data types and enumerations

7.2.1 Introduction

This clause defines simple data types and enumerations that can be referenced from data structures defined in multiple interfaces.

7.2.2 Simple data types

Table 7.2.2-1 lists the simple data types that are defined in the present document.

Table 7.2.2-1: Simple data types

Type name	Description
Identifier	An identifier with the intention of being globally unique. Representation: string of variable length. See note.
DateTime	Date-time stamp. Representation: String formatted as defined by the date-time production in IETF RFC 3339 [2].
Uri	String formatted according to IETF RFC 3986 [3].
Boolean	The Boolean is a data type having two values (true and false).
MacAddress	A MAC address. Representation: string that consists of groups of two hexadecimal digits, separated by hyphens or colons.
IpAddress	An IPV4 or IPV6 address. Representation: In case of an IPV4 address, string that consists of four decimal integers separated by dots, each integer ranging from 0 to 255. In case of an IPV6 address, string that consists of groups of zero to four hexadecimal digits, separated by colons.
Version	A Version. Representation: string of variable length.
String	A string as defined in IETF RFC 8259 [9].
Number	A number as defined in IETF RFC 8259 [9].
NOTE: Individual API specifications are assumed to define types for additional identifiers with dedicated scope (e.g. identifiers scoped by the VIM).	

7.2.3 Enumerations

This clause is empty in the present document.

8 Authorization of API requests and notifications

8.1 Introduction

The ETSI NFV MANO APIs are only allowed to be accessed by authorized consumers. Handling of authorization differs between making an API call and sending a notification. In the former case, OAuth 2.0 is used. In the latter case, OAuth 2.0 or HTTP Basic authentication is used, and the flows differ from those used in the former case. Alternatively, a solution based on public/private key pairs as authentication alternative to client identifier/password is also allowed.

The following terms (set in italics below) are used as defined by IETF RFC 6749 [7]:

- *client,*
- *resource server,*
- *authorization server,*
- *token endpoint,*
- *access token.*

The description below is based on the "client credentials" grant type as defined by IETF RFC 6749 [7].

For API calls, the producer functional block of an API in NFV terms corresponds to the "*resource server*", and the consumer functional block of an API corresponds to the "*client*" as defined by IETF RFC 6749 [7]. For sending a notification, these roles are reversed: The producer (notification sender) corresponds to the "*client*", and the consumer (notification receiver) corresponds to the "*resource server*".

Before invoking an HTTP method on a REST resource provided by a *resource server*, a functional block (referred to as "*client*" from now on) first obtains authorization from another functional block fulfilling the role of the "*authorization server*". The present document makes no assumption about which functional block in the architecture plays the role of the *authorization server*. It is however assumed that the address of the *token endpoint* exposed by the *authorization server* and further specified in the clauses below is provisioned to the *client* together with additional authorization-related configuration parameters, such as valid client credentials. The *client* requests an *access token* from the *token endpoint*. As part of the request, it authenticates towards the *authorization server* by presenting its client credentials, consisting of client identifier and client password. The *authorization server* responds with an *access token* which the *client* will present to the *resource server* with every HTTP method invocation. An *access token* represents a particular access right (defining the particular set of protected resources to access in a particular manner) with a defined duration. The token is opaque to the *client*, and can typically be used by the *authorization server* and the *resource server* as an identifier to retrieve authorization information, such as information that identifies the client, its role and access rights. An *access token* expires after a certain time, or can be revoked. If that happens, the *client* can try to obtain a new *access token* from the *authorization server*.

In order to ensure that no third party can eavesdrop on sensitive information such as client credentials or access tokens, HTTP over TLS is used to protect the transport. If mutual authentication using TLS protocol is used, then the producer/server is authenticated to the consumer/client, but also the consumer/client is authenticated by the producer/server at the same time. To facilitate this mutual authentication, the server shall request a client certificate. This can be done as described in IETF RFC 5246 [5], including the optional CertificateRequest from server to client.

HTTP over TLS enables authorization based on TLS certificates as an alternative to a token-based approach.

8.2 Flows (informative)

8.2.1 General

This clause outlines several alternative methods for authentication and authorization. Clause 8.2.2 presents an approach for authorizing API requests using OAuth 2.0 access tokens. Clause 8.2.3 describes an alternative method for authorization of API requests using TLS certificates. Clauses 8.2.4 and 8.2.5 outline a method to authorize notifications using basic authentication and OAuth2.0based approaches respectively. Finally, authorization of notifications using TLS certificates is presented in clause 8.2.6.

8.2.2 Authorization of API requests using OAuth 2.0 access tokens

The flow below illustrates the authorization of API requests that the API consumer sends to the API producer.

NOTE 1: Typical choices for the implementation of the authorization server include the authorization server as a component of the API producer, or as an external component.

Preconditions:

- Certificates are enrolled in the communicating entities as shown in the figure 8.2.2-1.
- Authorization server is configured with the authorization policy and access rights against the client credentials.

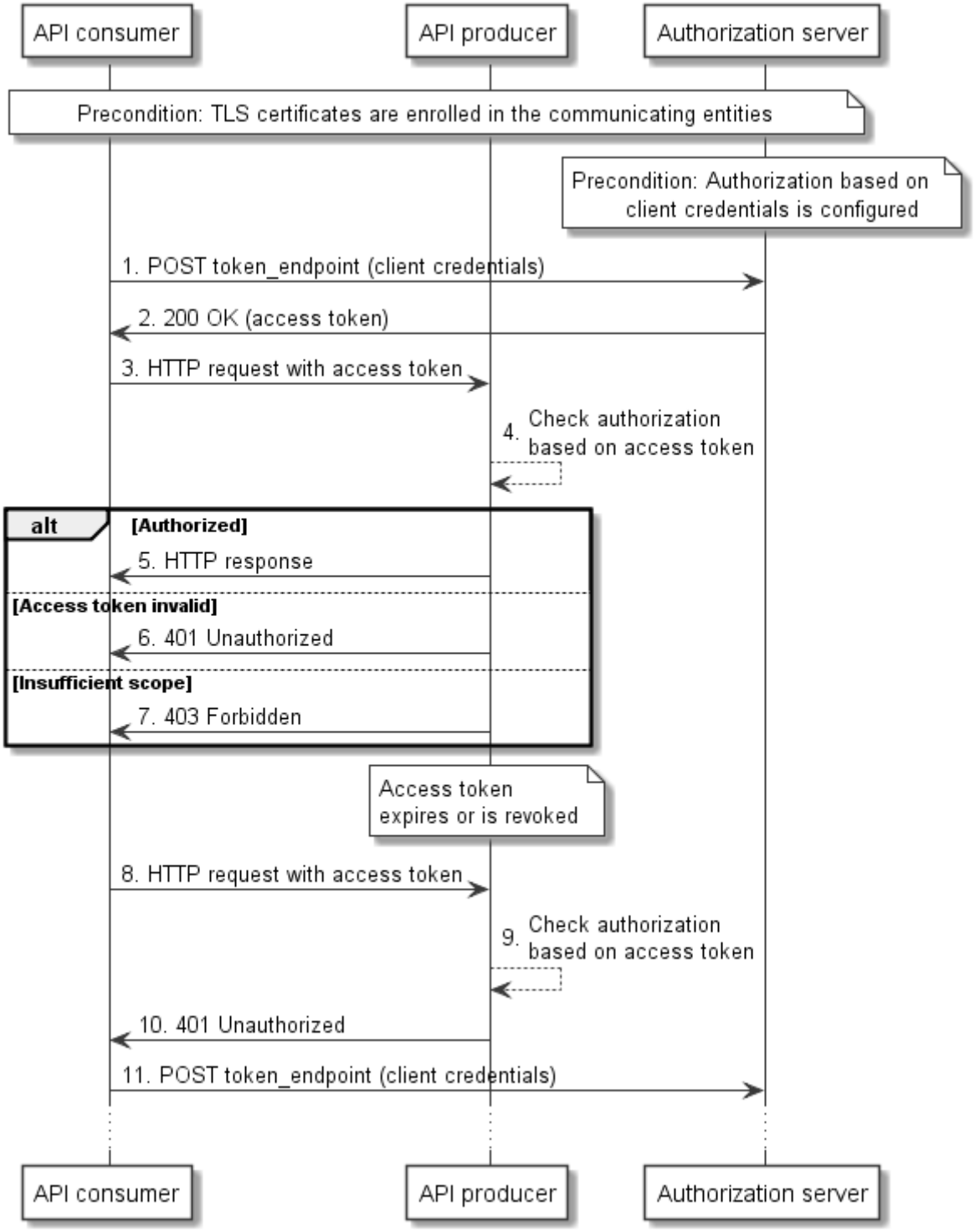

Figure 8.2.2-1: Authorization of API requests using OAuth 2.0 access tokens

The flow consists of the following steps:

1) To obtain an access token, the API consumer sends a POST request to the token endpoint of the authorization server and includes its client credentials.

2) The authorization server responds to the API consumer with an access token, and possibly additional information such as expiry time.

3) The API consumer sends an HTTP request to a resource provided by the API producer and includes the received access token.

4) The API producer checks the token for validity. This assumes that it has received information about the valid access tokens, and additional related information (e.g. time of validity, client identity, client access rights) from the authorization server. Such exchange is outside the scope of the present document, and assumed to be trivial if deployments choose to include the authorization server as a component into the API producer.

5) In case the token is valid and refers to access rights that allow accessing the actual resource with the actual request and its parameters, the API producer returns the HTTP response.

6) In case the token is invalid or expired, the API producer returns a "401 Unauthorized" response.

7) In case the access rights are insufficient to access the resource or to use the parameters, the API producer returns a "403 Forbidden" response.

8) The API consumer sends an HTTP request to the API producer and includes in the request the access token.

9) The API producer checks the token for validity, and establishes that it has expired, or has been revoked by the authorization server using means outside the scope of the present document.

10) The API producer responds with a "401 Unauthorized" response, indicating that the access token is invalid.

11) The API consumer attempts to obtain a new access token, as defined in step 3. This may eventually succeed or fail, depending on whether access is allowed for that API consumer any longer.

NOTE 2: All the communication presented in this flow diagram is done over encrypted tunnel using TLS as described in clause 4.1.

8.2.3 Authorization of API requests using TLS certificates

As an alternative to the authorization using OAuth 2.0 access tokens, authentication and authorization is defined herein based on TLS certificates, applying the IETF RFC 5246 [5]. To facilitate mutual authentication during TLS tunnel setup process, the server requests a client certificate as described in section 7.4.4 in IETF RFC 5246 [5].

Preconditions:

- Certificates are enrolled in the communicating entities as shown in the figure 8.2.3-1.
- Authorization server is configured with the authorization policy and access rights against the certificates.

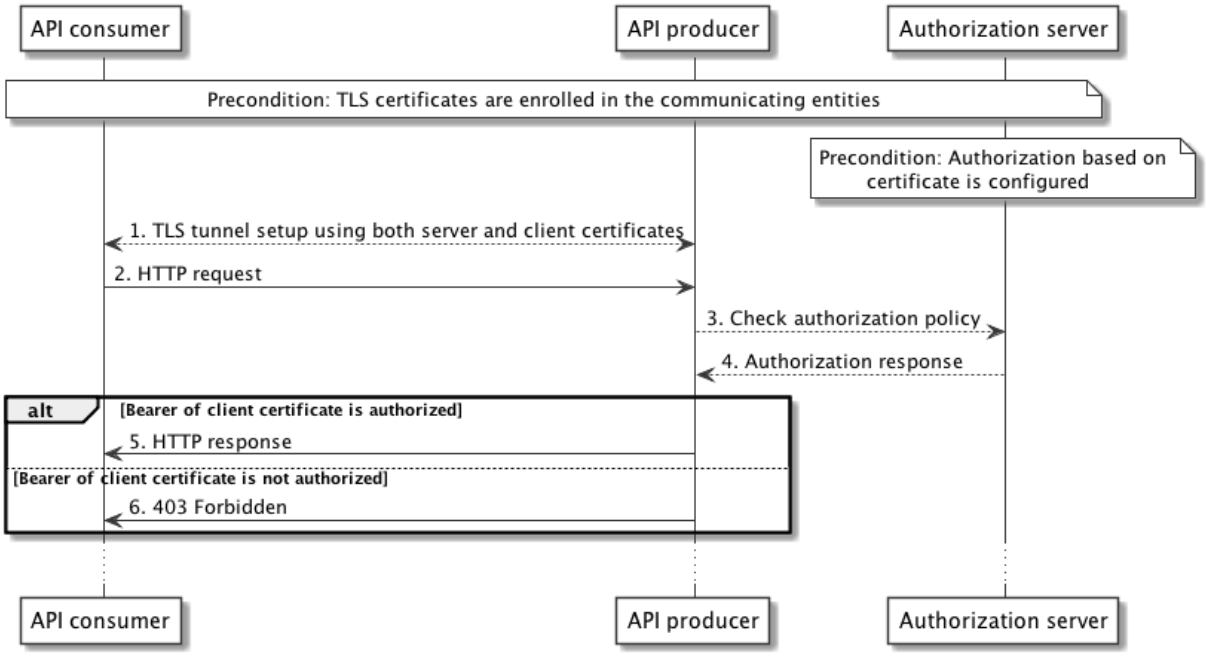

Figure 8.2.3-1: Authorization of API requests using TLS certificates

The flow consists of the following steps:

1) The API consumer initiates the TLS tunnel setup process with the API producer. During the tunnel setup process the API producer sends its certificate to API consumer and obtains the certificate from the API consumer by including the CertificateRequest message specified in IETF RFC 5246 [5]. This ensures the mutual authentication between the consumer and the producer.

2) API consumer further sends the HTTP request for a resource over the TLS tunnel.

3) API producer now checks for the authorization information from the authorization server based on the API consumer client certificate.

4) Authorization server checks its policy and sends the response to the API producer.

5) If the API consumer is authorized, then the API producer sends the response related to the requested resource.

6) If the API consumer is Unauthorized, then the API producer sends "403 Forbidden" response to the API consumer.

NOTE 1: Steps 3 and 4 are outside the scope of the present document. However, typical implementations can use the certificates in such a way that the API producer verifies the certificate of the API consumer and extracts the subject name from the certificate. This information will be sent to the authorization server in order to check the authorization. In a response, the authorization server will send the associated client profile that contains the access rights.

NOTE 2: All the communication presented in this flow diagram is done over encrypted tunnel using TLS as described in clause 4.1.

NOTE 3: Authorization based on TLS certificates assumes the existence of a trust relationship between the API producer and the authorization server. The authorization server has no direct communication with the API consumer and thus cannot authenticate it but relies on the API producer to perform this authentication.

8.2.4 Authorization of notifications using the HTTP Basic authentication scheme

Figure 8.2.4-1 illustrates the authorization of notifications that the API producer sends to the API consumer based on the HTTP Basic authentication scheme (see IETF RFC 7617 [14]). In this flow, no authorization server is needed.

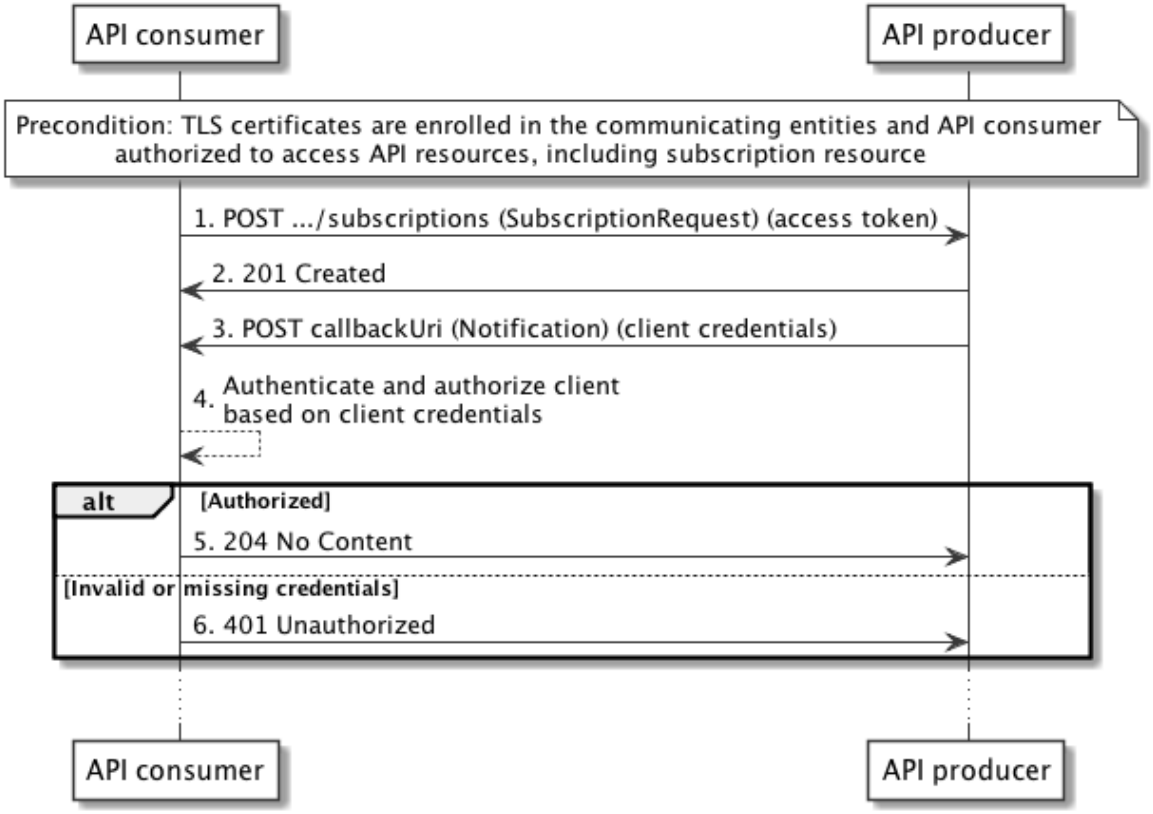

Figure 8.2.4-1: Authorization of notifications using the HTTP Basic authentication scheme

It is a precondition for this flow that the API consumer is authorized to access the "subscriptions" resource provided by the API producer, using the procedure illustrated in clause 8.2.2. Additionally, to ensure secure communication, it is a precondition that the TLS certificates are enrolled in the communicating entities.

The flow consists of the following steps:

1) The API consumer sends a request to create a new subscription resource to the API producer and includes in the request a valid access token to prove that it is authorized to access the API. Also, it includes in the subscription client credentials that the API producer can use to authenticate towards the API consumer when subsequently sending notifications. Note that these credentials are typically different from the client credentials used in the flow in clause 8.2.2.

2) The API producer creates the subscription resource and responds with "201 Created".

3) The API producer sends an HTTP POST request with a notification to the callback URI registered by the API consumer during subscription, and includes the client credential in the request to authenticate.

4) The API consumer checks the credentials against the information it has sent in step 1.

5) In case the credentials are valid, the API consumer returns a "204 No Content" HTTP response to indicate successful delivery of the notification.

6) In case the credentials are invalid, the API consumer returns a "401 Unauthorized" response.

NOTE: All the communication presented in this flow diagram is done over encrypted tunnel using TLS as described in clause 4.1.

8.2.5 Authorization of notifications using OAuth 2.0 access tokens

The flow below illustrates the authorization of notifications that the API producer sends to the API consumer using OAuth 2.0. In this flow, the authorization server can be a different entity than the authorization server in clause 8.2.2.

NOTE 1: Typical choices for the implementation of the authorization server include the authorization server as a component of the API consumer, or as an external component.

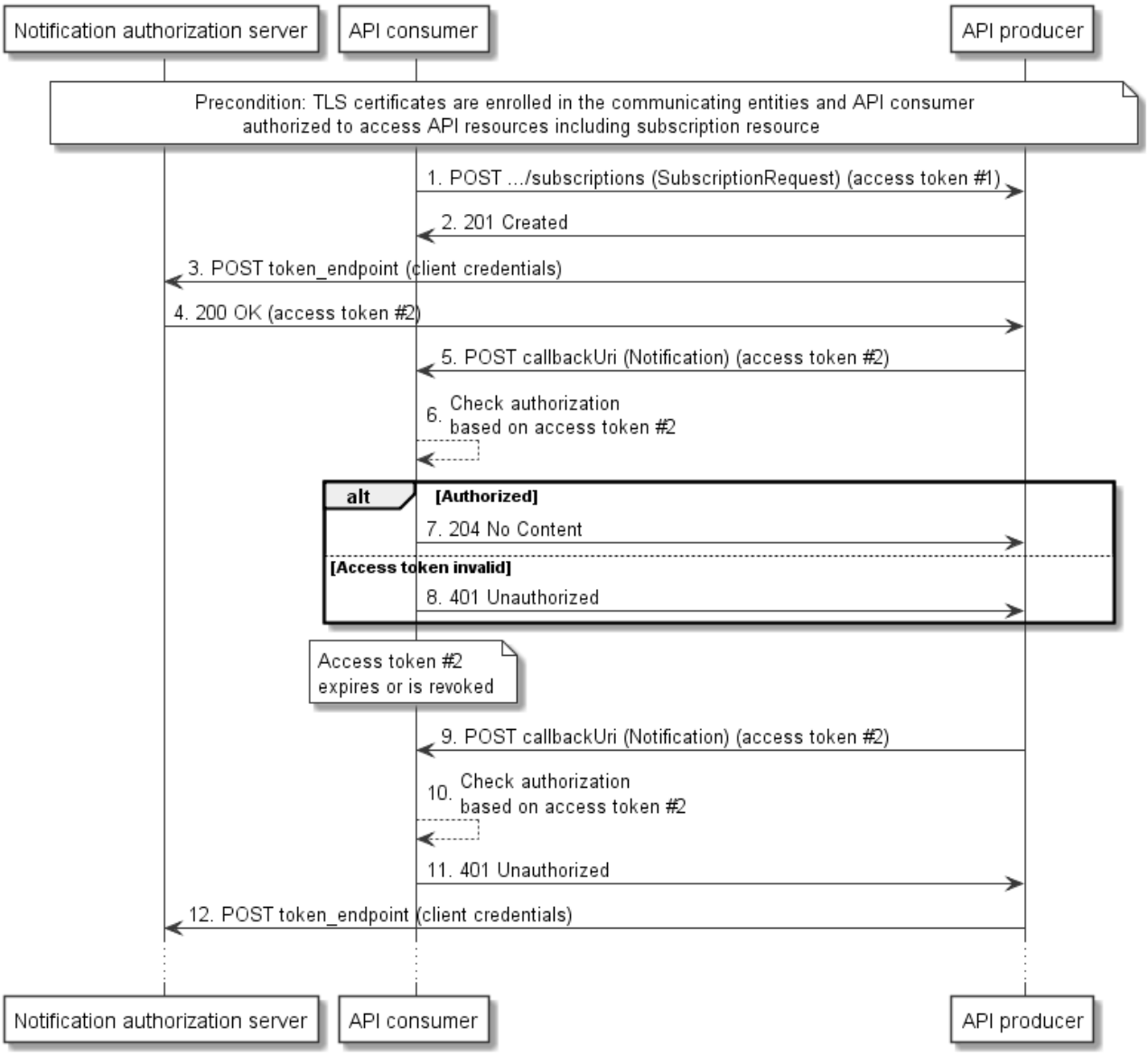

Figure 8.2.5-1: Authorization of notifications using OAuth 2.0

It is a precondition for this flow that the API consumer is authorized to access the "subscriptions" resource provided by the API producer, using the procedure illustrated in clause 8.2.2. Additionally, to ensure secure communication, it is a precondition that the TLS certificates are enrolled in the communicating entities.

The flow consists of the following steps:

1) The API consumer sends a request to create a new subscription resource to the API producer and includes in the request a valid access token #1 to prove that it is authorized to access the API. Also, it includes in the subscription request parameters that the API producer can use to obtain authorization to send notifications to the API consumer, such as client credentials and a token endpoint. Note that these are typically different from the credentials and token endpoint used in the flow in clause 8.2.2.

2) The API producer creates the subscription resource and responds with "201 Created".

3) Subsequently, and prior to sending any notification to the API consumer, the API producer obtains authorization to do so by requesting an access token from the authorization server, using the end point and notification client credentials that were sent in the subscription request, or provisioned otherwise.

4) The authorization server responds to the API producer with an access token, hereafter called access token #2, and possibly additional information such as expiry time.

5) The API producer sends an HTTP POST request with a notification to the callback URI registered by the API consumer during subscription, and includes the received access token #2.

6) The API consumer checks the token for validity. This assumes that it has received information about the valid access tokens, and additional related information (e.g. time of validity, client identity, client access rights) from the authorization server. Such exchange is outside the scope of the present document, and assumed to be trivial if deployments choose to include the authorization server as a component into the API consumer.

7) In case the token #2 is valid, the API consumer returns a "204 No Content" HTTP response to indicate successful delivery of the notification.

8) In case the token #2 is invalid or expired, the API consumer returns a "401 Unauthorized" response.

9) The API producer sends another notification in an HTTP POST request to the API consumer and includes in the request the access token #2.

10) The API consumer checks the token #2 for validity, and establishes that it has expired, or has been revoked by the authorization server using means outside the scope of the present document.

11) The API consumer responds with a "401 Unauthorized" response, indicating that the access token #2 is invalid.

12) The API producer attempts to obtain a new access token. This may eventually succeed or fail, depending on whether access is allowed for that API producer any longer.

NOTE 2: All the communication presented in this flow diagram is done over encrypted tunnel using TLS as described in clause 4.1.

8.2.6 Authorization of notifications using TLS certificates

The flow in figure 8.2.6-1 illustrates the authorization of notifications that the API producer sends to the API consumer using TLS certificates.

Preconditions:

- Certificates are enrolled in the communicating entities as shown in the figure 8.2.6-1.
- The API consumer is authorized to access the "subscriptions" resource provided by the API producer, using the procedure illustrated in clause 8.2.2 or 8.2.3.

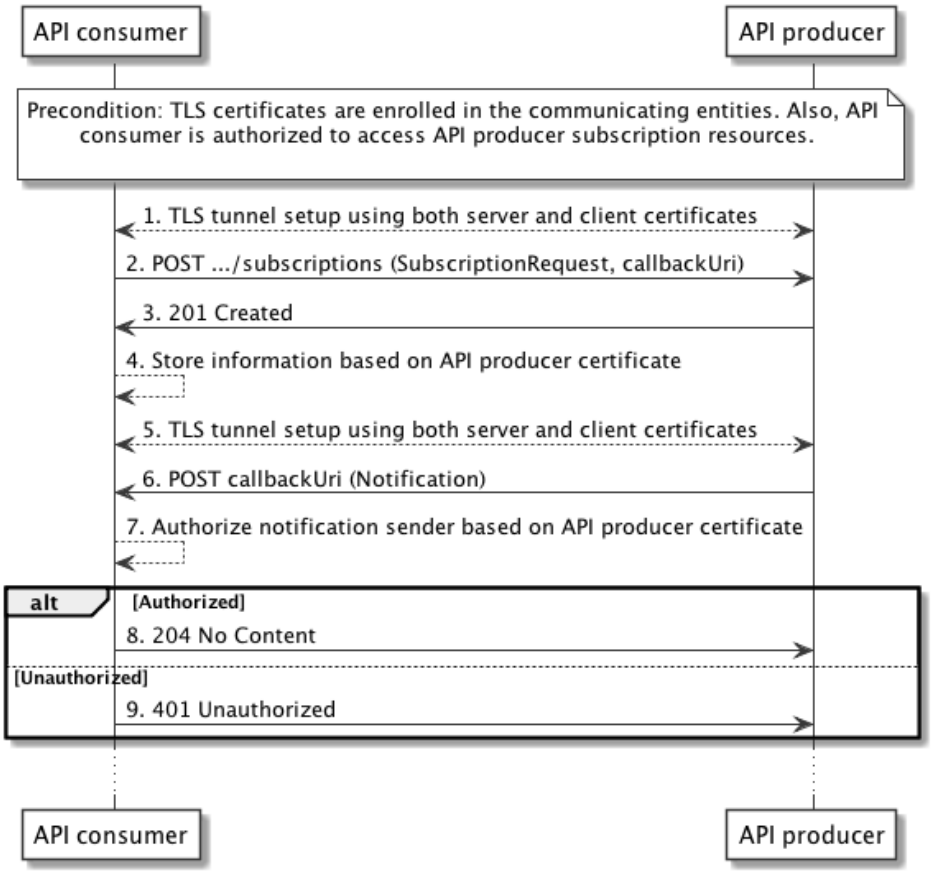

Figure 8.2.6-1: Authorization of notifications using TLS certificates

The flow consists of the following steps:

1) The API consumer initiates the TLS tunnel setup process with the API producer. During the tunnel setup process the API producer obtains the certificate from the API consumer. This ensures the mutual authentication between the consumer and the producer.

2) The API consumer sends a request to create a new subscription resource to the API producer. The API producer can authenticate and authorize this request based on the API consumer certificate as illustrated in clause 8.2.3. The request also includes the callbackURI where the notification will be sent in future.

3) The API producer creates the subscription resource and responds with "201 Created".

4) The API consumer now stores the relevant information of the API producer's certificate in association with the requested notification subscription.

5) The API producer initiates the TLS tunnel with the API consumer whenever there is a notification to send. During the tunnel setup process the API consumer sends its certificate to API producer and obtains the client certificate from the API producer. This ensures the mutual authentication between the consumer and the producer.

6) The API producer sends the notification over the established TLS tunnel.

7) API consumer can now verify whether this sender is allowed to send this notification by matching the sender's certificate information with the previously stored information at step 4.

8) In case is the API producer is authorized to send a notification, then the API consumer sends a "204 No Content" response to indicate successful delivery of the notification.

9) In case if the API producer is not authorized to send a notification, the API consumer returns a "403 Forbidden" response.

NOTE 1: Steps 4 and 7 are outside the scope of the present document. However, typical implementation can use the certificates in such a way that the API consumer verifies the certificate of the API producer and extract subject name from the certificate. This information is used in order to check the authorization at the API consumer.

NOTE 2: All the communication presented in this flow diagram is done over encrypted tunnel using TLS as described in clause 4.1.

NOTE 3: It is assumed that the API producer uses the same certificate for both the client and server role.

8.3 Specification

8.3.1 Introduction

OAuth 2.0 provides a framework for authorization of web applications that has multiple modes and options. This clause profiles the framework for use in the context of the NFV MANO reference points. Clause 8.3.2 specifies the general mechanism. Two different uses of the general mechanism, actually for API requests and for sending notifications, are defined in clauses 8.3.3 and 8.3.4.

8.3.2 General mechanism

For all requests to an API defined in the present document, and for all notifications sent via such an API, authorization as defined below shall be used. Requests and notifications without authorization credentials shall be rejected.

To allow the *client* to obtain an access token, the *authorization server* shall expose a *token endpoint* that shall comply with the provisions defined by the OAuth 2.0 specification for the *client credentials* grant type (see IETF RFC 6749 [7]). A *client* shall use the access token request and response according to this grant type, as defined by IETF RFC 6749 [7], to obtain an *access token* for access to the REST resources defined by the present document. The content of the *access token* is out of the scope of the present document; however, it shall not be possible for an attacker to easily guess it. The *access token* shall be a string. The set of allowed characters is defined in IETF RFC 6749 [7].

A *client* that invokes an HTTP request towards a resource defined by one of the APIs of the present document shall include the *access token* as a bearer token in every HTTP method in the "Authorization" HTTP header, as defined by IETF RFC 6750 [8]. A *resource server* that receives an HTTP request with an invalid *access token*, or without an *access token*, shall reject the request, and shall signal the error in the HTTP response according to the provisions for the error codes and the "WWW-Authenticate" response HTTP header as defined by IETF RFC 6750 [8].

A *client* that receives a rejection of an *access token* may obtain a new *access token* from the *token endpoint* of the *authorization server*, and retry the request.

As an alternative to OAuth 2.0 access tokens, certificates, as defined by TLS 1.2 in IETF RFC 5246 [5], can be used to facilitate the authentication and authorization between client and the server.

8.3.3 Authorizing API requests

A consumer of an API that wishes to issue HTTP requests towards resources provided by that API shall act as a *client* according to clause 8.3.2 to obtain an access token, and shall include this access token in every HTTP request, as defined in clause 8.3.2. The respective API producer shall act as a *resource server* as defined in clause 8.3.2.

Alternatively, API requests can be authorized based on TLS certificates.

These two different alternatives are listed in the following:

1) API consumer passes access token when accessing a resource provided by API producer. API producer checks authorization based on access token. Access token can be obtained from the authorization server based on client ID and password.

2) API consumer accesses a resource provided by API producer using TLS tunnel where both server and client certificates are used to establish the secure tunnel. API producer checks authorization based on client's TLS certificate. The client's TLS certificate is obtained during the TLS handshake.

8.3.4 Authorizing the sending of notifications

The procedure defined in clause 8.2 allows an API consumer to obtain authorization to perform API requests towards the API producer, including subscription requests. For sending the actual notifications matching a subscription, the API producer needs to obtain separate authorization to actually *send* the notification to the API consumer.

If an API consumer requires the API producer to authorize for sending notifications to that API consumer, it shall include in the subscription request a data structure that defines the authorization requirements, as defined in table 8.3.4-1.

Table 8.3.4-1: Definition of the SubscriptionAuthentication data type

Attribute name	Data type	Cardinality	Description
authType	Enum (inlined)	1..N	Defines the types of Authentication/Authorization which the API consumer is willing to accept when receiving a notification. Permitted values: - BASIC: In every HTTP request to the notification endpoint, use HTTP Basic authentication with the client credentials. - OAUTH2_CLIENT_CREDENTIALS: In every HTTP request to the notification endpoint, use an OAuth 2.0 bearer token, obtained using the client credentials grant type. - TLS_CERT: Every HTTP request to the notification endpoint is sent over a mutually authenticated TLS session, i.e. not only the server is authenticated, but also the client is authenticated during the TLS tunnel setup.
paramsBasic	Structure (inlined)	0..1	Parameters for authentication/authorization using BASIC. Shall be present if authType is "BASIC" and the contained information has not been provisioned out of band. Shall be absent otherwise.
>userName	String	0..1	Username to be used in HTTP Basic authentication. Shall be present if it has not been provisioned out of band.
>password	String	0..1	Password to be used in HTTP Basic authentication. Shall be present if it has not been provisioned out of band.
paramsOauth2ClientCredentials	Structure (inlined)	0..1	Parameters for authentication/authorization using OAUTH2_CLIENT_CREDENTIALS. Shall be present if authType is "OAUTH2_CLIENT_CREDENTIALS" and the contained information has not been provisioned out of band. Shall be absent otherwise.
>clientId	String	0..1	Client identifier to be used in the access token request of the OAuth 2.0 client credentials grant type. Shall be present if it has not been provisioned out of band. See note.
>clientPassword	String	0..1	Client password to be used in the access token request of the OAuth 2.0 client credentials grant type. Shall be present if it has not been provisioned out of band. See note.
>tokenEndpoint	Uri	0..1	The *token endpoint* from which the access token can be obtained. Shall be present if it has not been provisioned out of band.
NOTE: The clientId and clientPassword passed in a subscription shall not be the same as the clientId and clientPassword that are used to obtain authorization for API requests. Client credentials may differ between subscriptions. The value of clientPassword should be generated by a random process.			

The authType attribute is used to propose supported authorization methods of the API consumer for the authorization of notifications. If multiple methods are supported, the API producer shall choose a method as defined in clause 8.3.6.2. The expected behaviour of the authorization methods that can be signalled in the "authType" attribute is defined as follows:

"OAUTH2_CLIENT_CREDENTIALS":

- The API producer shall, prior to sending any notification, obtain an *access token* from the *token endpoint* using the OAuth 2.0 client credentials grant type as defined in IETF RFC 6749 [7]. The API consumer should include expiry information with the token response.
- The API producer shall include that *access token* as a bearer token in every POST request that sends a notification (according to IETF RFC 6750 [8]).
- If the *access token* is expired, the API consumer shall reject the notification. In that case, the API producer shall obtain a new *access token*, and repeat sending the notification.
- If the token expiry time is known to the API producer, it may obtain proactively a new access token.

"BASIC":

- The API producer shall pass its client credentials in every POST request that sends a notification, as defined in IETF RFC 7617 [14].

"TLS_CERT":

- The API producer (client) shall use its TLS certificate to create a mutually authenticated TLS session with the API consumer (server) and further the API consumer will do the authorization based on the API producer's certificate.

8.3.5 Client roles

An *access token* allows the API producer to identify information about the *client* that has obtained the access token, such as client identity, client role or client access rights. By having this property, *access tokens* can be used as a means to distinguish between different roles (and consequently different access rights) to the same set of resources.

The mechanism for this works as follows: By means out of scope of the present document, the role of the client identified by a particular client identifier is provisioned to the authorization server. When that client obtains an access token, it sends its client identifier and client password to the authorization server. The authorization sever can obtain the role of the client by evaluating the data that were provisioned for the client identifier, and associate that information to the access token. By means out of scope of the present document, that association is shared with the API producer. This enables the API producer to detect the role based on the access token.

In ETSI NFV, certain interfaces are exposed on multiple different reference points, i.e. the same interface is exposed to different consumer functional blocks. Depending on the consumer block that originates an HTTP request, not all resources/HTTP methods/request and parameters might be available. From the point of view of the producer functional block, this can be seen as consumers acting in different roles when accessing a particular interface.

Implementations may use the OAuth *access token* to differentiate between these cases, assuming that an *access token* can determine in which role (e.g. VNFM, NFVO, EM, VNF) a consumer functional block acts when accessing a particular interface. This assumes that the role of the consumer functional block is bound to its client credentials. The means of creating this binding is out of scope of the present document (e.g. a configuration step or policy).

As an alternative mechanism, the client role can be bound to its certificate. The mechanism for this works as follows: By means out of scope of the present document, the client is identified by a particular client subject name that is extracted from its certificate. This subject name is then provided to the authorization server in order to get the associated role of that particular client. By means out of scope of the present document, the authorization server is preconfigured to have this association between the client subject name and the role.

8.3.6 Negotiation of the authorization method

8.3.6.1 Authorization of API requests

The following provisions apply to the support of the authorization methods defined in the present document for the authorization of API requests:

- The API producer shall support checking the authorization of API requests it receives based on an OAuth 2.0 access token, and should support checking the authorization of API requests it receives based on TLS certificates, as defined in clause 8.3.3.
- The API consumer shall support the authorization of API requests it sends by including an OAuth 2.0 bearer token in the request, and should support the authorization of API requests it sends by providing its client certificate to the API producer during TLS tunnel setup as defined in clause 8.3.3.

When performing and authorizing an API request, API consumer and API producer shall use the following procedure, illustrated in figure 8.3.6.1-1, to negotiate the authorization method to use if the API consumer supports both the authorization based on OAuth 2.0 and the authorization based on TLS certificates, and the API consumer leaves the choice of OAuth or TLS to the API producer.

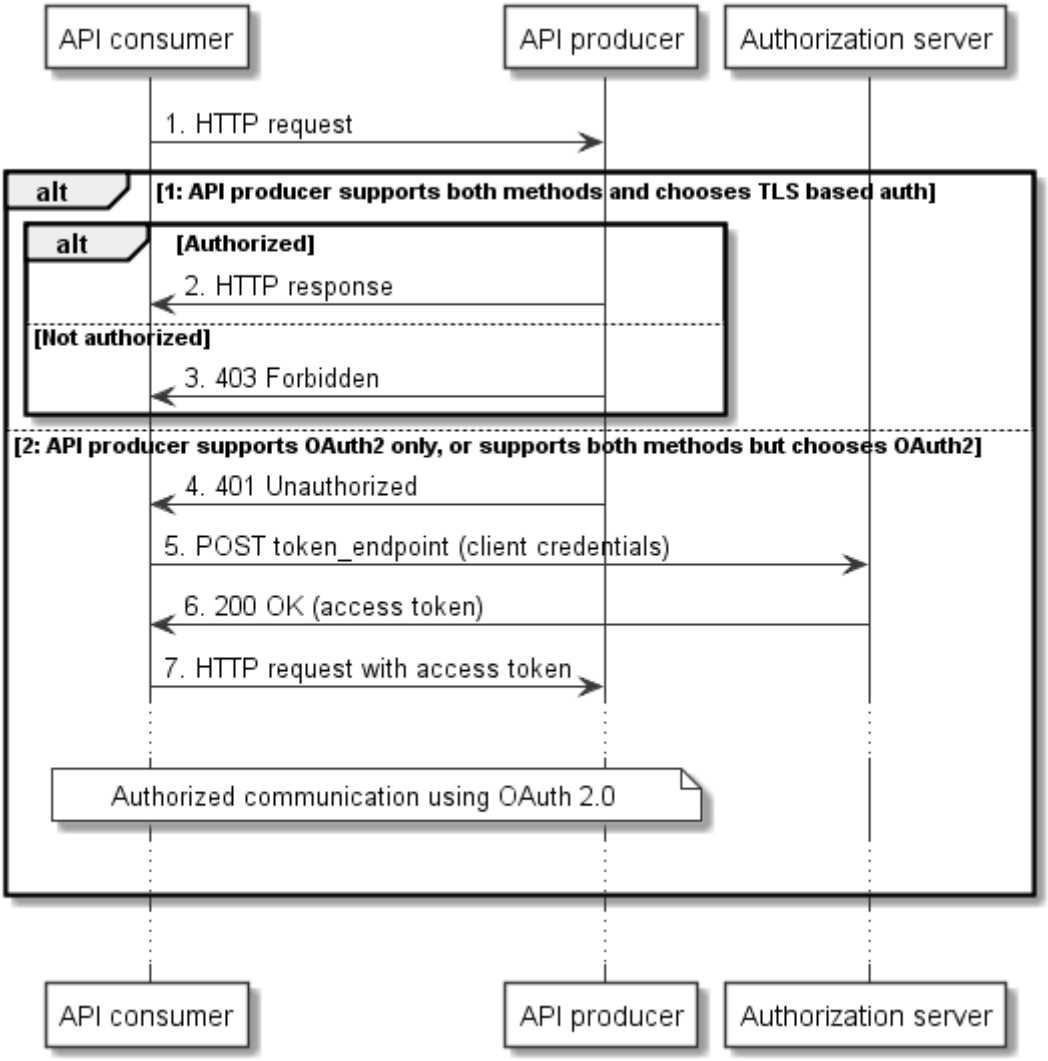

Figure 8.3.6.1-1: Negotiation of the authorization method to use for API requests

1) The API consumer shall send an HTTP request to the API producer without an access token.

2) If the API producer supports both authorization methods, chooses to use the method based on TLS certificates and the API consumer is authorized, it shall return the HTTP response to fulfil the request. Subsequent communication between API consumer and API producer shall use the authorization based on TLS credentials.

3) If the API producer supports both authorization methods, chooses to use the method based on TLS certificates and the API consumer is not authorized, it shall return a 403 Forbidden response.

4) If the API producer does not support the authorization based on TLS certificates, or chooses to use OAuth 2.0 for authorization, it shall return a 401 Unauthorized response to challenge the API consumer to use OAuth 2.0.

5) Once it has received the 401 Unauthorized response, the API consumer shall subsequently request an access token from the authorization server, according to clause 8.3.2.

6) The authorization server shall respond with an access token according to clause 8.3.2.

7) The API consumer shall subsequently retry the HTTP request with the access token included as a bearer token according to clause 8.3.2.

Subsequent authorized communication between API consumer and API producer shall take place as defined in clause 8.3.2 (see also the flow in clause 8.2.2, starting at step 4).

When performing and authorizing an API request and the API consumer does not support the method based on TLS certificates, or supports both methods but decides to use OAuth 2.0, no negotiation takes place, and the method defined in clause 8.3.2 shall be used (see also the flow in clause 8.2.2).

Table 8.3.6.1-1 illustrates the alternatives.

Table 8.3.6.1-1: Illustration of the alternatives

Consumer supports	Producer supports	Consumer request	Producer reaction
OAuth2	OAuth2	Consumer sends an access token in the 1st HTTP request.	Producer detects that OAuth2 is requested, and sends a success HTTP response if consumer is authorized (see note 1).
OAuth2+TLS	OAuth2	If consumer intends to use OAuth2, it sends an access token in the 1st HTTP request.	Producer detects that OAuth2 is requested, and sends a success HTTP response if consumer is authorized (see note 1).
		Otherwise, consumer sends the 1st HTTP request without access token.	Producer sends a 401 challenge to initiate use of OAuth2 (see note 2).
OAuth2	OAuth2+TLS	Consumer sends an access token in the 1st HTTP request.	Producer detects that OAuth2 is requested, and sends a success HTTP response if consumer is authorized (see note 1).
OAuth2+TLS	OAuth2+TLS	If consumer intends to use OAuth2, it sends an access token in the 1st HTTP request.	Producer detects that OAuth2 is requested, and sends a success HTTP response if consumer is authorized (see note 1).
		Otherwise, consumer sends the 1st HTTP request without access token.	If producer chooses OAuth2, it sends a 401 challenge (see note 2).
			Otherwise, if producer chooses TLS, it sends a success HTTP response if consumer is authorized (see note 3).
NOTE 1: This flow (OAuth2 method chosen by API consumer) is illustrated in figure 8.2.2-1. NOTE 2: This flow (OAuth2 method chosen by API producer) is illustrated in figure 8.3.6.1-1 as alternative 2. NOTE 3: This flow (TLS method chosen by API producer) is illustrated in figure 8.3.6.1-1 as alternative 1.			

8.3.6.2 Authorization of notification requests

The following provisions apply to the support of the authorization methods defined in the present document for the authorization of notification requests:

- The API consumer shall support checking the authorization of notification requests it receives based on an OAuth 2.0 access token as defined in clause 8.3.4. Further, the API consumer should support checking the authorization of notification requests it receives based on HTTP Basic authentication, and based on TLS certificates as defined in clause 8.3.4.

- The API producer shall support the authorization of notification requests it sends by including an OAuth 2.0 bearer token in the request as defined in clause 8.3.4. Further, the API producer should support the authorization of notification requests it sends by providing credentials based on HTTP Basic authentication, and by providing its client certificate to the API producer during TLS tunnel setup as defined in clause 8.3.4.

When performing and authorizing a notification request, API consumer and API producer shall use the following procedure to negotiate the authorization method to use:

1) The API consumer shall signal in the subscription the authorization methods it accepts for notifications related to that particular subscription.

2) If none of the methods signalled is supported by the API producer, the API producer shall reject the subscription with "422 Unprocessable Entity", and shall include in the payload body a ProblemDetails structure which shall provide the reason for the rejection in the "details" attribute.

3) Otherwise, the API producer shall select one of the authorization methods that was signalled in the subscription, and shall use that method for the authorization of notifications it sends based on that subscription.

9 Version management

9.1 Version identifiers and parameters

9.1.1 Version identifiers

API version identifiers shall consist of 3 numerical fields, following a MAJOR.MINOR.PATCH pattern and the rules for Semantic Versioning [18] with the additional clarifications defined in clause 9.2. The fields in an API version identifier shall be separated by dots ".". The last field may be followed by one or more version parameters.

The MAJOR, MINOR and PATCH fields are defined in [18] for Semantic Versioning.

The {apiMajorVersion} segment of the URIs used by an API shall be set to the character "v" followed by value of the MAJOR field of the API version identifier.

EXAMPLE: ".../vnflcm/v1/

The full version identifier (including parameters) is used in ApiVersionInformation (see clause 7.1.6) and in version signalling (see clause 9.4). Furthermore, it also appears in the corresponding OpenAPI file that ETSI publishes as collateral material for each RESTful NFV MANO API specification [i.4].

9.1.2 Version parameters

Version parameters are separated from the version identifier by a dash "-". Version parameters are separated by semicolons ";".

The present document defines the following version parameters:

- impl

The optional "impl" parameter identifies an implementation and a version of this implementation (e.g. implementation delivered by an open source community or a vendor). The OpenAPI specification that ETSI publishes as collateral material for each RESTful NFV MANO API specification [i.4] is also considered an implementation under this scheme. The "impl" parameter shall have the following structure: "impl:" <vendor>":"<product>":"<impl_version>, where:

- the <vendor> field shall be a string that contains either an IANA Enterprise Number assigned to that vendor, or an Internet domain name owned by that vendor;
- the <product> field shall contain a string identifying the product, chosen by the vendor;
- the <impl_version> field shall contain a number that defines the version of the implementation. Version numbers of subsequent implementations shall be monotonically increasing.

In case of the OpenAPI files provided by ETSI as collateral material for the RESTful NFV MANO API specifications [i.4], <vendor> shall be set to "etsi.org" and "product" shall be set to "ETSI_NFV_OpenAPI".

9.2 Rules for incrementing version identifier fields

9.2.1 General

In a REST API, versioning applies to the resources structure (URI structure, URI query parameters, and supported HTTP methods) and the payload body. Different criteria are applied to increment MAJOR, MINOR, and PATCH version fields for changes that affect the URI compared to changes that affect the payload body.

The fields of an API version identifier are incremented from a previous version to the current version according to the following rules:

- 1st field (MAJOR): This field is always incremented when one or more changes made to the resources structure defined in the present document break backward compatibility. This field is also incremented if one or more changes to at least one payload body defined in the present document break backward compatibility, unless that change is correcting an error.

NOTE 1: 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 not considered a non-backward compatible change, irrespective of the type of change. Indeed, compatibility between a new version and a previous version can only be assessed for a feature that is properly supported in the previous version.

NOTE 2: Void.

- 2nd field (MINOR): This field is incremented if one or more technical changes (at least one of which is not an error correction) are made to the API specification in the present document API but none of them (apart from error corrections to the payload body) breaks backward compatibility. It is reset to zero if the MAJOR version identifier is changed.

- 3rd field (PATCH): This field is incremented if one or more error corrections that are visible in communication between API producer and API consumer are made on the API specification in the present document but none of them (apart from error corrections to the payload body) breaks backward compatibility. It is reset to zero if the MINOR version identifier is changed.

NOTE 3: All the aforementioned types of changes affect the corresponding OpenAPI specification that ETSI publishes as collateral material for each RESTful NFV MANO API specification [i.4].

9.2.2 Examples of backward and non-backward compatible changes

Examples of backward compatible 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
- Responding with a new status code of an error class
- Certain cardinality changes (see note 2)

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).

Examples of non-backward compatible 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 non-backward compatible 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)

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 client, changing the meaning of the default behavior associated to the absence of an attribute of cardinality 0..N, etc.

9.3 Version information retrieval

9.3.1 General

The API producer shall support the following dedicated URIs to enable API consumers to retrieve information about API versions supported by an API producer:

```
1. {apiRoot}/{apiName}/api_versions
2. {apiRoot}/{apiName}/{apiMajorVersion}/api_versions
```

To obtain information about the supported API versions, the API consumer shall send a GET request to a URI of one of above forms. The information contained in the GET response depends on the form of URI used in the GET request, as follows:

- If the first form is used, the GET response shall provide the list of supported versions for the API corresponding to the apiName indicated in the GET Request URI.
- If the second form is used, the GET response shall provide the list of supported versions for the API corresponding to the {apiName} and the {apiMajorVersion} indicated in the GET Request URI.

If the API producer receives a GET request:

- In case of success, the API producer shall return in the body of a 200 OK response a value of the ApiVersionInformation data type specified in clause 7.1.6.
- In case URI query parameters are provided, the API producer shall return a "400 Bad request" response as defined in clause 6.4.
- In other cases of failure, the API producer shall return appropriate error information as defined in clause 6.4.

9.3.2 Resource structure and methods

Table 9.3.2-1 lists the individual resources defined for supporting API version information retrieval, and theapplicable HTTP method. The API producer shall support responding to GET requests on the resources in table 9.3.2-1.

Table 9.3.2-1: Resources and methods overview for API version information retrieval

Resource name	Resource URI	HTTP Method	Meaning
API versions	/{apiName}/api_versions	GET	Version information associated to an API
API versions	/{apiName}/{apiMajorVersion}/api_versions	GET	Version information associated to a major version of an API

Figure 9.3.2-1 shows the "API versions" resources in the overall resource URI structure defined for all APIs.

The "API versions" resources, as defined in the present clause, are part of the overall resource URI structure of each API defined in the present document.

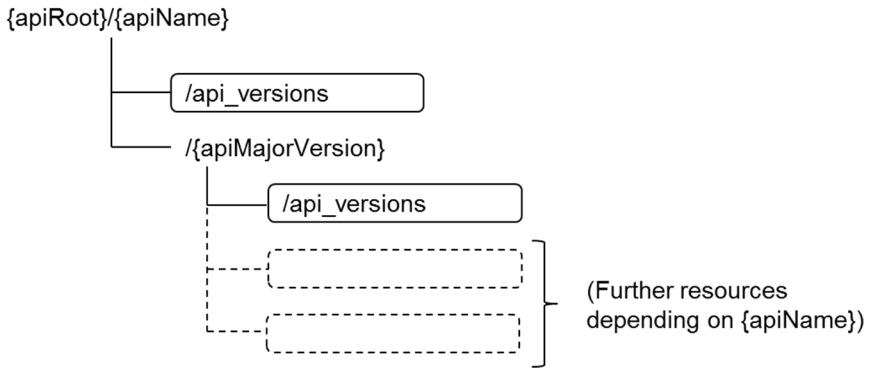

Figure 9.3.2-1: "API versions" resources

9.3.3 Resource: API versions

9.3.3.1 Description

There are two "API versions" resources defined for each API. The client can use these resources to obtain API version information.

9.3.3.2 Resource definition

The resource URI of each of the two "API versions" resources shall be of one of the following forms:

1. `{apiRoot}/{apiName}/api_versions`
2. `{apiRoot}/{apiName}/{apiMajorVersion}/api_versions`

These resources shall support the resource URI variables defined in table 9.3.3.2-1.

Table 9.3.3.2-1: Resource URI variables for these resources

Name	Definition
apiRoot	See clause 4.1
apiName	See clause 4.1
apiMajorVersion	See clause 4.1

9.3.3.3 Resource methods

9.3.3.3.1 POST

This method is not supported. When this method is requested on this resource, the API producer shall return a "405 Method Not Allowed" response as defined in clause 6.4.

9.3.3.3.2 GET

The GET method reads API version information. This method shall follow the provisions specified in table 9.3.3.3.2-1 for request and response data structures, and response codes. URI query parameters are not supported.

Table 9.3.3.3.2-1: Details of the GET request/response on this resource

Request body	Data type	Cardinality	Description	
	n/a			
Response body	**Data type**	**Cardinality**	**Response Codes**	**Description**
	ApiVersionInformation	1	200 OK	API version information was read successfully. The response body shall contain API version information, as defined in clause 7.1.6.
	ProblemDetails	See clause 6.4	4xx/5xx	In addition to the response codes defined above, any common error response code as defined in clause 6.4 may be returned.

9.3.3.3.3 PUT

This method is not supported. When this method is requested on this resource, the API producer shall return a "405 Method Not Allowed" response as defined in clause 6.4.

9.3.3.3.4 PATCH

This method is not supported. When this method is requested on this resource, the API producer shall return a "405 Method Not Allowed" response as defined in clause 6.4.

9.3.3.3.5 DELETE

This method is not supported. When this method is requested on this resource, the API producer shall return a "405 Method Not Allowed" response as defined in clause 6.4.

9.4 Version signaling

The API consumer shall include the "Version" HTTP header (see IETF RFC 4229 [19]) in each HTTP request. The "Version" header shall contain the three version identifier fields (MAJOR.MINOR.PATCH) indicating the API version the API consumer intends to use. The "impl" version parameter may be provided, indicating the version of the API producer implementation that the API consumer intends to use.

The API producer shall support receiving and interpreting the "Version" HTTP header. The API producer shall include in the response the "Version" HTTP header signaling the used API version, including the "impl" version parameter if available. If the "impl" version parameter has been omitted in the request, the API producer shall use the combination of MAJOR, MINOR and PATCH as requested and the highest supported value for the "impl_version" field of the "impl" version parameter for that combination, if available.

NOTE: In case multiple versions and/or implementation versions are supported by an API producer, this allows the API consumer to request a particular version.

API consumers conforming to versions of the ETSI NFV-SOL API specifications previous to version 2.5.1 omit this header. If the API producer receives a request without this header:

- If it supports the provisions defined in version 2.4.1 of the applicable ETSI NFV-SOL API specification document, it shall behave as defined in that document, and should indicate this by using MAJOR=1 and MINOR=1 and PATCH=0 in the "Version" HTTP header in the response.
- Otherwise, it shall respond with a 400 Bad Request response and shall include in the response payload body a ProblemDetails structure providing more information on the cause of the error in the "detail" attribute.

If the API version signaled in the "Version" request header is not supported by the API producer, the API producer shall respond with a "406 Not Acceptable" error and shall include in the response payload body a ProblemDetails structure providing more information on the cause of the error in the "detail" attribute.

Annex A (informative): Change History

Date	Version	Information about changes
Nov 2018	2.5.2	Initial version based on: - NFVSOL(18)000583r1 Contributions incorporated: - NFVSOL(18)000582r3_SOL013_Content_moved_from_SOL003
Feb 2019	2.5.3	Contributions incorporated - NFVSOL(18)000677r1_SOL013ed261_Addressing_EN_on_error_codes - NFVSOL(19)000003r1_SOL013_modifications_to_authorization_description - NFVSOL(19)000104r1_SOL013ed261_Version_related_update_for_publication Editorials: - Changed year - Fixed bullet numbering in clause 8 - Fixed mis-formatting and missing table reference in clause 7.2.2.

History

Document history		
V2.6.1	March 2019	Publication