ETSI GS MEC 009 V3.1.1 (2021-06)

Multi-access Edge Computing (MEC); General principles, patterns and common aspects of MEC Service APIs

Disclaimer

The present document has been produced and approved by the Multi-access Edge Computing (MEC) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
RGS/MEC-0009v311ApiPrinciples

Keywords
API, MEC

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 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are 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 Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, 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.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Multi-access Edge Computing (MEC).

Modal verbs terminology

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

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

1 Scope

The present document defines design principles for RESTful MEC service APIs, provides guidelines and templates for the documentation of these, and defines patterns of how MEC service APIs use RESTful principles.

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 7231: "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content".

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

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

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

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

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

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

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

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

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

[6] IETF RFC 6902: "JavaScript Object Notation (JSON) Patch".

NOTE: Available at https://tools.ietf.org/html/rfc6902.

[7] IETF RFC 5261: "An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors".

NOTE: Available at https://tools.ietf.org/html/rfc5261.

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

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

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

NOTE: Available at https://tools.ietf.org/html/rfc63986.

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

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

[11] W3C Recommendation 16 August 2006: "Extensible Markup Language (XML) 1.1" (Second Edition).

NOTE: Available at https://www.w3.org/TR/2006/REC-xml11-20060816/.

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

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

[13] Void.

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

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

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

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

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

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

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

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

[18] IETF RFC 7540: "Hypertext Transfer Protocol Version 2 (HTTP/2)".

NOTE: Available at https://tools.ietf.org/html/rfc7540.

[19] Void.

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

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

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

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

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

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

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

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

[24] IETF RFC 8446: "The Transport Layer Security (TLS) Protocol Version 1.3".

NOTE: Available at https://tools.ietf.org/html/rfc8446.

[25] IETF RFC 6455: "The WebSocket Protocol".

NOTE: Available at https://tools.ietf.org/html/rfc6455.

[26] ETSI TS 129 122: "Universal Mobile Telecommunications System (UMTS); LTE; 5G; T8 reference point for Northbound APIs" (3GPP TS 29.122).

[27] ETSI TS 133 210: "Universal Mobile Telecommunications System (UMTS); LTE; 3G Security; Network Domain Security (NDS); IP network layer security (3GPP TS 33.210)".

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 MEC 001: "Multi-access Edge Computing (MEC); Terminology".

[i.2] William Durand: "Please. Don't Patch Like An Idiot".

NOTE: Available at http://williamdurand.fr/2014/02/14/please-do-not-patch-like-an-idiot/.

[i.3] Martin Fowler: "Richardson Maturity Model: steps toward the glory of REST".

NOTE: Available at http://martinfowler.com/articles/richardsonMaturityModel.html.

[i.4] JSON Schema, Draft Specification 2020-12, December 8, 2020.

NOTE: Referenced version available as Internet Draft (work in progress) at https://tools.ietf.org/html/draft-bhutton-json-schema-00. All versions are available at http://json-schema.org/specification.html.

[i.5] W3C® Recommendation: "XML Schema Part 0: Primer Second Edition".

NOTE: Available at https://www.w3.org/TR/xmlschema-0/.

[i.6] ETSI GS MEC 011: "Multi-access Edge Computing (MEC); Edge Platform Application Enablement".

[i.7] ETSI GS MEC 012: "Multi-access Edge Computing (MEC); Radio Network Information API".

[i.8] IANA: "Hypertext Transfer Protocol (HTTP) Status Code Registry".

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

[i.9] MQTT Version 3.1.1 Plus Errata 01, OASIS™ Standard, 10 December 2015.

NOTE: Available at http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[i.10] Apache Kafka®.

NOTE: Available at https://kafka.apache.org/.

[i.11] gRPC®.

NOTE: Available at http://www.grpc.io/.

[i.12] Protocol buffers.

NOTE: Available at https://developers.google.com/protocol-buffers/.

[i.13] IETF RFC 7519: "JSON Web Token (JWT)".

NOTE: Available at https://tools.ietf.org/html/rfc7519.

[i.14] OpenAPI™ Specification.

NOTE: Available at https://github.com/OAI/OpenAPI-Specification.

[i.15] ETSI TS 129 222 (V16.8.0): "Universal Mobile Telecommunications System (UMTS); LTE; 5G; T8 reference point for Northbound APIs (3GPP TS 29.122 version 16.8.0 Release 16)".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI GS MEC 001 [i.1] and the following apply:

resource: object with a type, associated data, a set of methods that operate on it, and, if applicable, relationships to other resources

NOTE: A resource is a fundamental concept in a RESTful API. Resources are acted upon by the RESTful API using the Methods (e.g. POST, GET, PUT, DELETE, etc.). Operations on Resources affect the state of the corresponding managed entities.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI GS MEC 001 [i.1] and the following apply:

AA	Authentication and Authorization
API	Application Programming Interface
BYOT	Bring Your Own Transport
CRUD	Create, Read, Update, Delete
DDoS	Distributed Denial of Service
DN	Distinguished Name
GS	Group Specification
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
ISG	Industry Specification Group
JSON	JavaScript Object Notation
JWT	JSON Web Token
MEC	Multi-access Edge Computing
PKI	Public Key Infrastructure
POX	Plain Old XML
REST	Representational State Transfer
RFC	Request For Comments
RPC	Remote Procedure Call
SCS/AS	Services Capability Server/Application Server
SCEF	Service Capability Exposure Function
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UE	User Equipment
URI	Uniform Resource Indicator
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

4 Design principles for developing RESTful MEC service APIs

4.1 REST implementation levels

The Richardson Maturity Model as defined in [i.3] breaks down the principal elements of a REST approach into three steps.

All RESTful MEC service APIs shall implement at least Level 2 of the Richardson Maturity Model explained in annex C.

It is recommended to implement Level 3 when applicable.

4.2 General principles

RESTful MEC service APIs are not technology implementation dependent.

RESTful MEC service APIs embrace all aspects of HTTP/1.1 (IETF RFC 7231 [1]) including its request methods, response codes, and HTTP headers. Support for PATCH (IETF RFC 5789 [3]) is optional.

For each RESTful MEC service API specification, the following information should be included:

- Purpose of the API.
- URIs of resources including version number.
- HTTP methods (IETF RFC 7231 [1]) supported.
- Representations supported: JSON and, if applicable, XML.
- Response schema(s).
- Request schema(s) when PUT, POST, or PATCH are supported.
- Links supported (Optional in Level 2 APIs).
- Response status codes supported.

Since the release of HTTP/1.1, major revisions have been introduced through HTTP/2 (IETF RFC 7540 [18]) including binary serialization in place of textual, single TCP connection, full multiplexing, header compression and server push. MEC system deployments may utilize HTTP/2. However, this is transparent to the RESTful MEC service APIs, as the main semantic of HTTP has been retained in HTTP/2 thereby providing backwards compatibility. If HTTP/2 (IETF RFC 7540 [18]) is supported, its use shall be negotiated as specified in section 3 of IETF RFC 7540 [18].

4.3 Entry point of a RESTful MEC service API

Entry point for a RESTful MEC service API:

- Needs to have one and exactly one entry point. The URI of the entry point needs to be communicated to API clients so that they can find the API.
- It is common for the entry point to contain some or all of the following information:
  - Information on API version, supported features, etc.
  - A list of top-level collections.
  - A list of singleton resources.

  - Any other information that the API designer deemed useful, for example a small summary of operating status, statistics, etc.

- It can be made known via different means:
  - Client discovers automatically the entry point and meaning of the API.
  - Client developer knows about the API at time of client development.

4.4 API security and privacy considerations

To allow proactive protection of the APIs against the known security and privacy issues, e.g. DDoS, frequency attack, unintended or accidental information disclosure, etc. the design for a secure API should consider at least the following aspects:

- Ability to control the frequency of the API calls (calls/min), e.g. by supporting the definition of a validity time or expiration time for a service response.
- Anonymization of the real identities.
- Authorization of the applications based on the sensitivity of the information exposed through the API.

5 Documenting RESTful MEC service APIs

5.1 RESTful MEC service API template

Annex D defines a template for the documentation of RESTful MEC service APIs. Examples how to use this template can for instance be found in ETSI GS MEC 011 [i.6] and ETSI GS MEC 012 [i.7].

5.2 Conventions for names

5.2.1 Case conventions

The following case conventions for names and strings are used in the RESTful MEC service APIs:

1) UPPER_WITH_UNDERSCORE

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

EXAMPLE 1:

   a) ETSI_MEC_MANAGEMENT;

   b) MULTI_ACCESS_EDGE_COMPUTING.

2) lower_with_underscore

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

EXAMPLE 2:

   a) etsi_mec_management;

   b) multi_access_edge_computing.

3) UpperCamel

A string is formed by concatenating words. Each word starts with an uppercase letter (this implies that the string starts with an uppercase letter). All other letters are lowercase letters. Digits are allowed but not at the first position. No other characters are allowed. Abbreviations follow the same scheme (i.e. first letter uppercase, all other letters lowercase).

EXAMPLE 3:

a) EtsiMecManagement;

b) MultiAccessEdgeComputing.

4) lowerCamel

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

EXAMPLE 4:

a) etsiMecManagement;

b) multiAccessEdgeComputing.

5.2.2 Conventions for URI parts

5.2.2.1 Introduction

Based on IETF RFC 3986 [9], the parts of the URI syntax that are relevant in the context of the RESTful MEC service APIs are as follows:

- *Path*, consisting of *segments*, separated by "/" (e.g. segment1/segment2/segment3).
- *Query*, consisting of pairs of parameter name and value (e.g. ?org=etsi&isg=mec, where two pairs are presented).

5.2.2.2 Path segment naming conventions

a) Each path segment of a resource URI which represents a constant string shall use lower_with_underscore. Only letters, digits and underscore "_" are allowed.

EXAMPLE 1: etsi_mec_management

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

EXAMPLE 2: …/prefix/api/v1/user**s**

c) If a resource is not a task resource and the last path segment of that resource's URI is a string constant, the last path segment should be a (composite) noun.

EXAMPLE 3: …/prefix/api/v1/users

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

EXAMPLE 4:

…/app_instances/{appInstanceId}/instantiate

…/app_instances/{appInstanceId}/do_something_else

e) A name that represents a URI path segment or multiple URI path segments in the API documentation but serves as a placeholder for an actual value created at runtime (URI path variable) shall use lowerCamel, and shall be surrounded by curly brackets.

EXAMPLE 5: {appInstanceId}

f) Once a variable is replaced at runtime by an actual string, the string shall follow the rules for a path segment or sequence of path segments (depending on whether the variable represents a single path segment or a sequence thereof) defined in IETF RFC 3986 [9]. IETF RFC 3986 [9] disallows certain characters from use in a path segment. Each actual RESTful MEC service API specification shall define this restriction to be followed when generating values for path segment variables, or propose a suitable encoding (such as percent-encoding according to IETF RFC 3986 [9]), to escape such characters if they can appear in input strings intended to be substituted for a path segment variable.

5.2.2.3 Query naming conventions

a) Parameter names in queries shall use lower_with_underscore.

EXAMPLE 1: ?isg_name=MEC

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

EXAMPLE 2: ?isg_name={chooseAName}

c) Once a variable is replaced at runtime by an actual string, the convention defined in clause 5.2.2.2 item f) applies to that string.

5.2.3 Conventions for names in data structures

The following syntax conventions apply when defining the names for attributes and parameters in the RESTful MEC service API data structures:

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

EXAMPLE 1: appName.

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

EXAMPLE 2: users, mecApps.

c) The identifier of a data structure via which this data structure can be referenced externally should be named "id".

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

EXAMPLE 3: NOT_INSTATIATED.

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

EXAMPLE 4: ResourceHandle, AppInstance.

5.3 Provision of an OpenAPI definition

An ETSI ISG MEC GS defining a RESTful MEC service API should provide a supplementary description file (or supplementary description files) compliant to the OpenAPI specification [i.14], which inherently include(s) a definition of the data structures of the API in JSON schema or YAML format. A description file is machine readable facilitating content validation and autocreation of stubs for both the service client and server. A link to the specific repository containing the file(s) shall be provided. All API repositories can be accessed from https://forge.etsi.org. The file (or files) shall be informative. In case of a discrepancy between supplementary description file(s) and the underlying specification, the underlying specification shall take precedence.

5.4 Documentation of the API data model

5.4.1 Overview

Clause 5.4 and its clauses specify provisions for API data model documentation for ETSI ISG MEC GSs defining RESTful MEC service APIs. Clause 5 in annex D provides a related data model template.

The data model shall be defined using a tabular format as described in the following clauses. The name of the data type shall be documented appropriately in the heading of the clause and in the caption of the table, preferably as defined in clause 5.2.2 and in annex D.

5.4.2 Structured data types

Structured data types shall be documented in tabular format, as in table 5.4.2-1 (one table per named data type).

Table 5.4.2-1: Template for a table defining a named structured data type

Attribute name	Data type	Cardinality	Description

The following provisions apply to the content of the table:

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

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

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

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

   c) An indication of the definition of an **inlined enumeration type**. In case of inlining an enumeration type, the "Data type" column shall contain the string "Enum (inlined)", and the "Description" column shall contain the allowed values and (optionally) their meanings. There are two possible ways to define enums (see clause 5.4.4): just to define the valid enum values or to define the valid values and their mapping to integers. It is good practice not to mix the two patterns in the same data structure.

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

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

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

Table 5.4.2-2 provides an example.

Table 5.4.2-2: Example of a structured data type definition

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

5.4.3 Simple data types

Simple data types shall be documented in tabular format, as in table 5.4.3-1 (one table row per simple data type).

Table 5.4.3-1: Simple data types

Type name	Description

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

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

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

Table 5.4.3-2 provides an example.

Table 5.4.3-2: Example of a simple data type definition

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

5.4.4 Enumerations

Enumerations can be specified just as a set of values, or as a set of values mapped to integers.

Enumeration types shall be documented in tabular format, as in table 5.4.4-1 or table 5.4.4-2 (one table row per enumeration value, one table per enumeration type).

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

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

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

If the intent is to map the enum values to strings (often used in JSON), or only to define the valid values (with the intent to define their mappings to integers in a different place, specific to certain serializers), the format defined in table 5.4.4-1 shall be used.

Table 5.4.4-1: Enumeration type

Enumeration value	Description
A_VALUE	The description of this enum value
ANOTHER_VALUE	The description of this other enum value

If the intent is to map the enum values to integers independent of the serializer, the format in defined in table 5.4.4-2 shall be used. "<int>" in the table below shall be replaced by an actual integer value.

Table 5.4.4-2: Enumeration type with mapping to integer values

Enumeration value	Description
<int> = A_VALUE	The description of this enum value
<int> = ANOTHER_VALUE	The description of this other enum value

5.4.5 Serialization

This clause only applies to the serialization of the top-level named data types that define the structure of a resource representation in the payload body of an HTTP request or response, but not of named data types that are referenced from other named data types. In the template in annex D, such data types represent resources ("resource data types"), subscriptions ("subscription data types") or notifications ("notification data types").

When the data model is serialized as XML, the name of the applicable named top-level data type shall be converted to lowerCamel (see clause 5.2.1) and used as the name of the root element.

When the data model is serialized as JSON, no root element shall be synthesized, but all attributes of the applicable resource, subscription or notification data type shall appear at the root level (under consideration of their cardinality). Individual APIs may deviate from this rule, for example when re-using pre-existing data models. Such deviation shall be documented in the API specification.

The following examples illustrate this convention. Assume the resource data type "Person" is defined as in table 5.4.5-1. The XML serialization is illustrated in example 1 and the JSON serialization is illustrated in example 2.

Table 5.4.5-1: Example: Definition of the "PersonData" data type

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

EXAMPLE 1: XML serialization:

```
<personData>
    <lastName>Doe</lastName>
    <firstName>John</firstName>
    <address>
        <street>Route des Lucioles</street>
        <number>650</number>
        <city>Sophia Antipolis</city>
    </address>
</personData>
```

EXAMPLE 2: JSON serialization:

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

6 Patterns of RESTful MEC service APIs

6.1 Introduction

This clause describes patterns to be used to model common operations and data types in the RESTful MEC service APIs. The defined patterns are used consistently throughout different RESTful MEC service APIs as defined by ETSI ISG MEC.

For RESTful APIs exposed by MEC services designed by third parties, it is recommended to use these patterns if and where applicable.

6.2 Void

6.3 Pattern: Resource identification

6.3.1 Description

Every resource is identified by at least one resource URI. A resource URI identifies at most one resource.

6.3.2 Resource definition(s) and HTTP methods

The syntax of each resource URI shall follow IETF RFC 3986 [9]. In the RESTful MEC service APIs, the resource URI structure shall be as follows:

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

"apiRoot" consists of the scheme ("https"), host and optional port, and an optional prefix string. "apiName" defines the name of the API. The "apiVersion" represents the version of the API. "apiSpecificSuffixes" define the tree of resource URIs in a particular API. The combination of "apiRoot", "apiName" and "apiVersion" is called the root URI. "apiRoot" is under control of the deployment, whereas the remaining parts of the URI are under control of the API specification.

All RESTful MEC service APIs shall support HTTP over TLS (also known as HTTPS) using TLS version 1.2 as defined by IETF RFC 5246 [14]). TLS 1.3 (including the new specific requirements for TLS 1.2 implementations) defined by IETF RFC 8446 [24] should be supported. HTTP without TLS shall not be used. Versions of TLS earlier than 1.2 shall neither be supported nor used. If HTTP/2 (IETF RFC 7540 [18]) is supported, its use shall be negotiated as specified in section 3 of IETF RFC 7540 [18].

TLS implementations should meet or exceed the security algorithm, key length and strength requirements specified in clause 6.2.3 (if TLS version 1.2 as defined by IETF RFC 5246 [14] is used) or clause 6.2.2 (if TLS version 1.3 as defined by IETF RFC 8446 [24] is used) of ETSI TS 133 210 [27] (3GPP Release 16 or later).

NOTE: This means that for HTTP/2 over TLS connections, negotiation uses TLS with the application-layer protocol negotiation (ALPN) extension, whereas for unencrypted HTTP/2 connections, negotiation is based on the HTTP Upgrade mechanism, or HTTP/2 is used immediately based on prior knowledge.

With every HTTP method, exactly one resource URI is passed in the request to address one particular resource.

6.4 Pattern: Resource representations and content format negotiation

6.4.1 Description

Resource representations are an important concept in REST. Actually, a resource representation is a serialization of the resource state in a particular content format. A resource representation is included in the payload body of an HTTP request or response. It depends on the HTTP method whether a representation is required or not allowed in a request, as defined in IETF RFC 7231 [1] (see table 6.4.1-1). If no representation is provided in a response, this shall be signalled by the "204 No Content" response code.

Table 6.4.1-1: Payload bodies requirements in HTTP requests for the different HTTP methods

HTTP method	Payload body is…
GET	unspecified; not recommended
PUT	required
POST	required
PATCH	required
DELETE	unspecified; not recommended

HTTP (IETF RFC 7231 [1]) provides a mechanism to negotiate the content format of a representation. Each ETSI MEC API specification defines the content formats that are mandatory or optional by the server to support for that API; the client may use any of these. Examples of content types are JSON (IETF RFC 8259 [10]) and XML [11]. In HTTP requests and responses, the "Content-Type" HTTP header is used to signal the format of the actual representation included in the payload body. If the format of the representation in an HTTP request is not supported by the server, it responds with a "415 Unsupported Media Type" response code. The content formats that a client supports in a HTTP response are signalled by the "Accept" HTTP header of the HTTP request. If the server cannot provide any of the accepted formats, it returns the "406 Not Acceptable" response code.

6.4.2 Resource definition(s) and HTTP methods

This pattern is applicable to any resource and any HTTP method.

6.4.3 Resource representation(s)

This pattern is applicable to any resource representation.

6.4.4 HTTP headers

The client uses the "Accept" HTTP header to signal to the server the content formats it supports. It is also possible to provide priorities. The HTTP specification can be found in IETF RFC 7231 [1].

As defined in the HTTP specification, both client and server use the "Content-Type" HTTP header to signal the content format of the payload included in the payload body of the request or response, if a payload body is present.

For the RESTful MEC service APIs, the following applies: In the "Accept" and "Content-Type" HTTP headers, the string "application/json" shall be used to signal the use of the JSON format (IETF RFC 8259 [10]) and "application/xml" shall be used to signal the use of the XML format [11].

6.4.5 Response codes and error handling

Servers that do not support the content format of the representation received in the payload body of a request return the "415 Unsupported Media Type" response code.

A server returns "406 Not Acceptable" in a HTTP response if it cannot provide any of the formats signalled by the client in the "Accept" HTTP header of the associated HTTP request.

A server that wishes to omit the payload body in a successful response returns "204 No Content" instead of "200 OK". This can make sense for DELETE, PUT and PATCH, but makes no sense for GET, and makes rarely sense for POST.

6.5 Pattern: Creating a resource (POST)

6.5.1 Description

This clause describes the "resource creation by POST" mode, where the client requests the origin server to create a new resource under a parent resource, i.e. the URI that identifies the created resource is under control of the server. This pattern shall be used for resource creation if the resource identifiers under the parent resource are managed by the server (see clause 6.5a for an alternative).

In order to request resource creation, the client sends a POST request specifying the resource URI of the parent resource and includes a representation of the resource to be created. The server generates a name for the new resource that is unique for all child resources in the scope of the parent resource, and concatenates this name with the resource URI of the parent resource to form the resource URI of the child resource. The server creates the new resource, and returns in a "201 Created" response a representation of the created resource along with a "Location" HTTP header field that contains the resource URI of this resource.

Figure 6.5.1-1 illustrates creating a resource by POST.

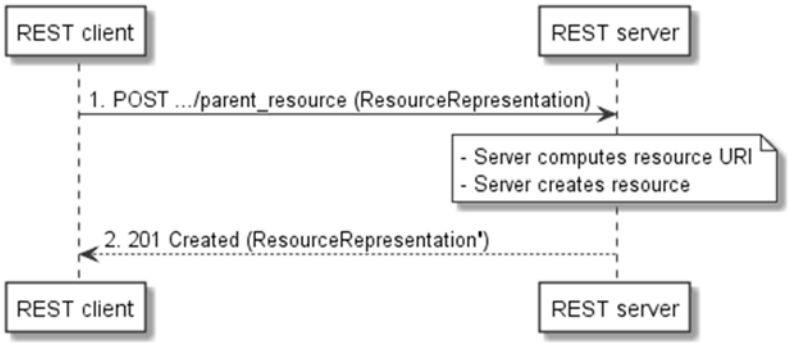


Figure 6.5.1-1: Resource creation flow (POST)

6.5.2 Resource definition(s) and HTTP methods

The following resources are involved:

1) parent resource: A container resource that can hold zero or more child resources;

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

The HTTP method shall be POST.

6.5.3 Resource representation(s)

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

NOTE: Compared to the payload body passed in the request (ResourceRepresentation in figure 6.5.1-1), the payload body in the response (ResourceRepresentation' in figure 6.5.1-1) may be different, as the resource creation process may have modified the information that has been passed as input.

6.5.4 HTTP headers

Upon successful resource creation, the "Location" HTTP header field in the response shall be populated with the URI of the newly created resource.

6.5.5 Response codes and error handling

Upon successful resource creation, "201 Created" shall be returned. On failure, the appropriate error code (see annex B) shall be returned.

Resource creation can also be asynchronous in which case "202 Accepted" shall be returned instead of "201 Created". See clause 6.13 for more details about asynchronous operations.

6.5a Pattern: Creating a resource (PUT)

6.5a.1 Description

This clause describes the "resource creation by PUT" mode, where the client requests the origin server to create a new resource, i.e. the URI that identifies the created resource is under control of the client.

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

This pattern shall be used for resource creation if the resource identifiers under the parent resource are managed by the client (see clause 6.5a for an alternative).

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

Figure 6.5a.1-1 illustrates creating a resource by PUT.

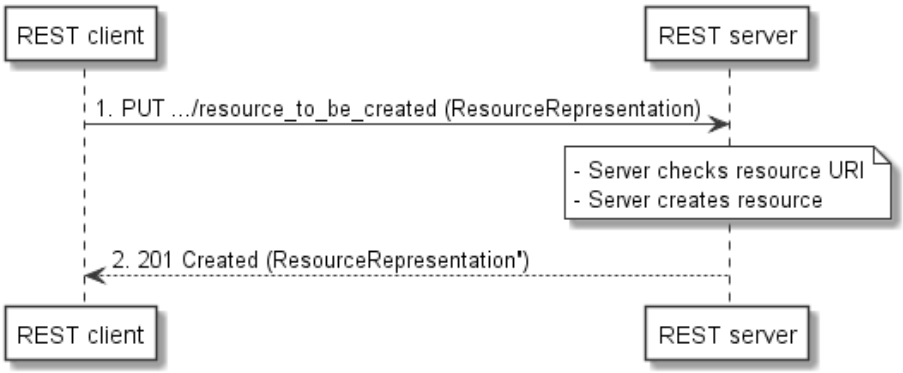


Figure 6.5a.1-1: Resource creation flow (PUT)

6.5a.2 Resource definition(s) and HTTP methods

The following resource is involved:

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

The HTTP method shall be PUT.

6.5a.3 Resource representation(s)

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

NOTE: Compared to the payload body passed in the request (ResourceRepresentation in figure 6.5a.1-1), the payload body in the response (ResourceRepresentation' in figure 6.5a.1-1) may be different, as the resource creation process may have modified the information that has been passed as input.

6.5a.4 HTTP headers

Upon successful resource creation, the "Location" HTTP header field in the response shall be populated with the URI of the newly created resource.

6.5a.5 Response codes and error handling

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

In case the resource does not yet exist:

- Upon successful resource creation, "201 Created" shall be returned. Upon failure, the appropriate error code (see annex B) shall be returned.
- Resource creation can also be asynchronous in which case "202 Accepted" shall be returned instead of "201 Created". See clause 6.13 for more details about asynchronous operations.

In case the resource already exists:

- If the "Update by PUT" operation is not supported for the resource, the request shall be rejected with "403 Forbidden", and a "ProblemDetails" payload should be included to provide more information about the error.
- If the "Update by PUT" operation is supported for the resource, interpret the request as an update request, i.e. the request shall be processed as defined in clause 6.8.

6.6 Pattern: Reading a resource

6.6.1 Description

This pattern obtains a representation of the resource, i.e. reads a resource, by using the HTTP GET method. For most resources, the GET method should be supported. An exception is task resources (see clause 6.11); these cannot be read.

Figure 6.6.1-1 illustrates reading a resource.

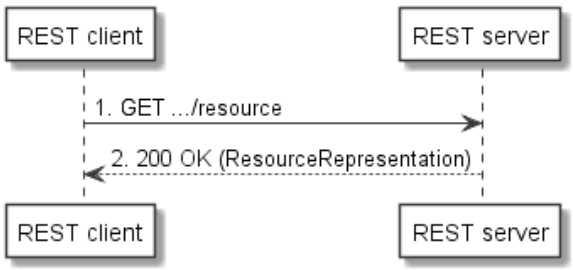


Figure 6.6.1-1: Reading a resource

6.6.2 Resource definition(s) and HTTP methods

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

6.6.3 Resource representation(s)

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

6.6.4 HTTP headers

No specific provisions for HTTP headers for this pattern.

6.6.5 Response codes and error handling

On success, "200 OK" shall be returned. On failure, the appropriate error code (see annex B) shall be returned.

6.7 Pattern: Queries on a resource

6.7.1 Description

This pattern influences the response of the GET method by passing resource URI parameters in the query part of the resource URI. The syntax of the query part is specified by IETF RFC 3986 [9].

Typically, query parameters are used for:

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

EXAMPLES:

GET …/foo_list?vendor=MEC&ue_ids=ab1,cd2

→ would return a foo_list representation that includes only those entries where vendor is "MEC" and the UE IDs are "ab1" or "cd2".

GET …/foo_list?group=group1

→ would return a foo_list representation that includes only those entries that belong to "group1".

GET …/foo_list/123?format=reduced_content

→ would return a representation of the resource …/foo_list/123 with content tailored according to the application-specific "reduced_content" scope.

GET …/foo_list?fields=name,address,key

→ would return a representation of the resource …/foo_list where the entries are reduced to the attributes "name", "address" and "key".

The list above provides just examples; the normative definition of individual simple queries and the related URI query parameters are left to the actual API specifications. For a comprehensive query mechanism, the actual API specifications can reference the mechanisms for attribute-based filtering and attribute selectors that are specified in clauses 6.18 and 6.19 of the present document.

Query values that are not compatible with URI syntax shall be escaped properly using percent encoding as defined in IETF RFC 3986 [9].

6.7.2 Resource definition(s) and HTTP methods

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

6.7.3 Resource representation(s)

The payload body of the request shall be empty; the payload body of the response shall contain a representation of the resource that was read, adjusted according to the parameters that were passed.

6.7.4 HTTP headers

No specific provisions for HTTP headers for this pattern.

6.7.5 Response codes and error handling

On success, "200 OK" shall be returned. On failure, the appropriate error code (see annex B) shall be returned.

6.8 Pattern: Updating a resource (PUT)

6.8.1 Description

When a resource is updated using the PUT HTTP method, this operation has "replace" semantics. That is, the new state of the resource is determined by the representation in the payload body of PUT, previous resource state is discarded by the REST server when executing the PUT request.

If the client intends to use the current state of the resource as the baseline for the modification, it is required to obtain a representation of the resource by reading it, to modify that representation, and to place that modified representation in the payload body of the PUT. If, on the other hand, the client intends to overwrite the resource without considering the existing state, the PUT can be executed with a resource representation that is created from scratch.

Figure 6.8.1-1 illustrates this flow.

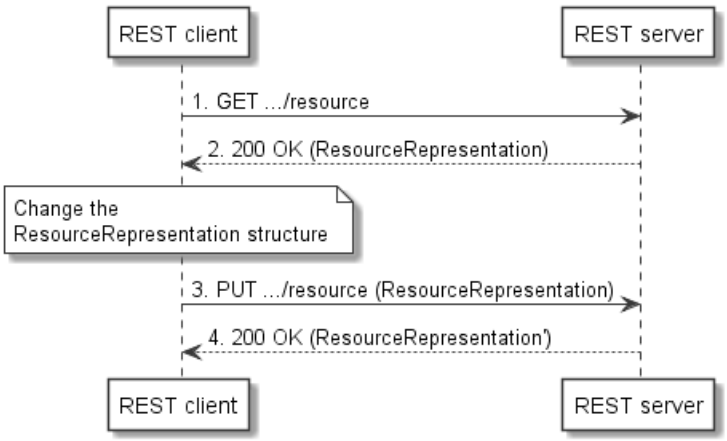


Figure 6.8.1-1: Basic resource update flow with PUT

The approach illustrated above can suffer from race conditions. If another client modifies the resource after receiving the response to the GET request and before sending the PUT request, that modification gets lost (which is known as the lost update phenomenon in concurrent systems). HTTP (see IETF RFC 7232 [2]) supports conditional requests to detect such a situation and to give the client the opportunity to deal with it. For that purpose, each version of a resource gets assigned an "entity-tag" (ETag) that is modified by the server each time the resource is changed. This information is delivered to the client in the "ETag" HTTP header in HTTP responses. If the client wishes that the server executes the PUT only if the ETag has not changed since the time the GET response was generated, the client adds to the PUT request the HTTP header "If-Match" with the ETag value obtained from the GET request. The server executes the PUT request only if the ETag in the "If-Match" HTTP header matches the current ETag of the resource, and responds with "412 Precondition Failed" otherwise. In that conflict case, the client needs to repeat the GET-PUT sequence. This is illustrated in figure 6.8.1-2.

Figure 6.8.1-2: Resource update flow with PUT, considering concurrent updates

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

6.8.2 Resource definition(s) and HTTP methods

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

6.8.3 Resource representation(s)

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

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

6.8.4 HTTP headers

If multiple clients can update the same resource, the client should pass in the "If-Match" HTTP header of the PUT request the value of the "ETag" HTTP header received in the response to the GET request.

NOTE: This prevents the "lost update" phenomenon.

6.8.5 Response codes and error handling

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

Resource update can also be asynchronous in which case "202 Accepted" shall be returned instead of "200 OK". See clause 6.13 for more details about asynchronous operations.

6.9 Pattern: Updating a resource (PATCH)

6.9.1 Description

The PATCH HTTP method (see IETF RFC 5789 [3]) is used to update a resource on top of the existing resource state with partial changes described by the client (unlike resource update using PUT which overwrites a resource (see clause 6.8)). The "Update by PATCH" pattern can be used in all places where "Update by PUT" can be used, but is typically more efficient for partially updating a large resource.

As opposed to PUT, PATCH does not carry a representation of the resource in the payload body, but a "deltas document" that instructs the server how to modify the resource representation. For JSON, JSON Patch (see IETF RFC 6902 [6]) and JSON Merge Patch (IETF RFC 7396 [5]) are defined for that purpose. Whereas JSON Patch declares commands that transform a JSON document, JSON Merge Patch defines fragments that are merged into the target JSON document. For XML, a patch framework is specified in IETF RFC 5261 [7] which defines operations to modify the target document.

Figure 6.9.1-1 illustrates updating a resource by PATCH.

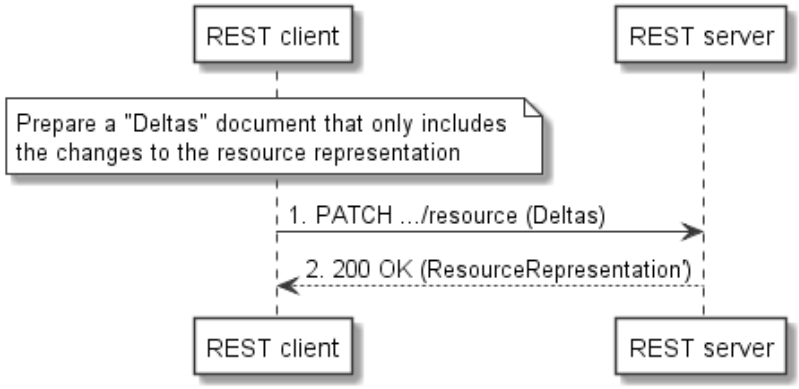


Figure 6.9.1-1: Basic resource update flow with PATCH

Careful design of the PATCH payload can make the method idempotent, i.e. the order in which particular PATCH operations are executed does not matter. If this can be achieved, the "lost update" phenomenon cannot occur. However, if conflicts are possible, the If-Match HTTP header should be used in the same way as with PUT, as illustrated by figure 6.9.1-2.

NOTE: Like in the PUT case, the ETag refers to the whole resource representation, not only to the portion modified by the PATCH.

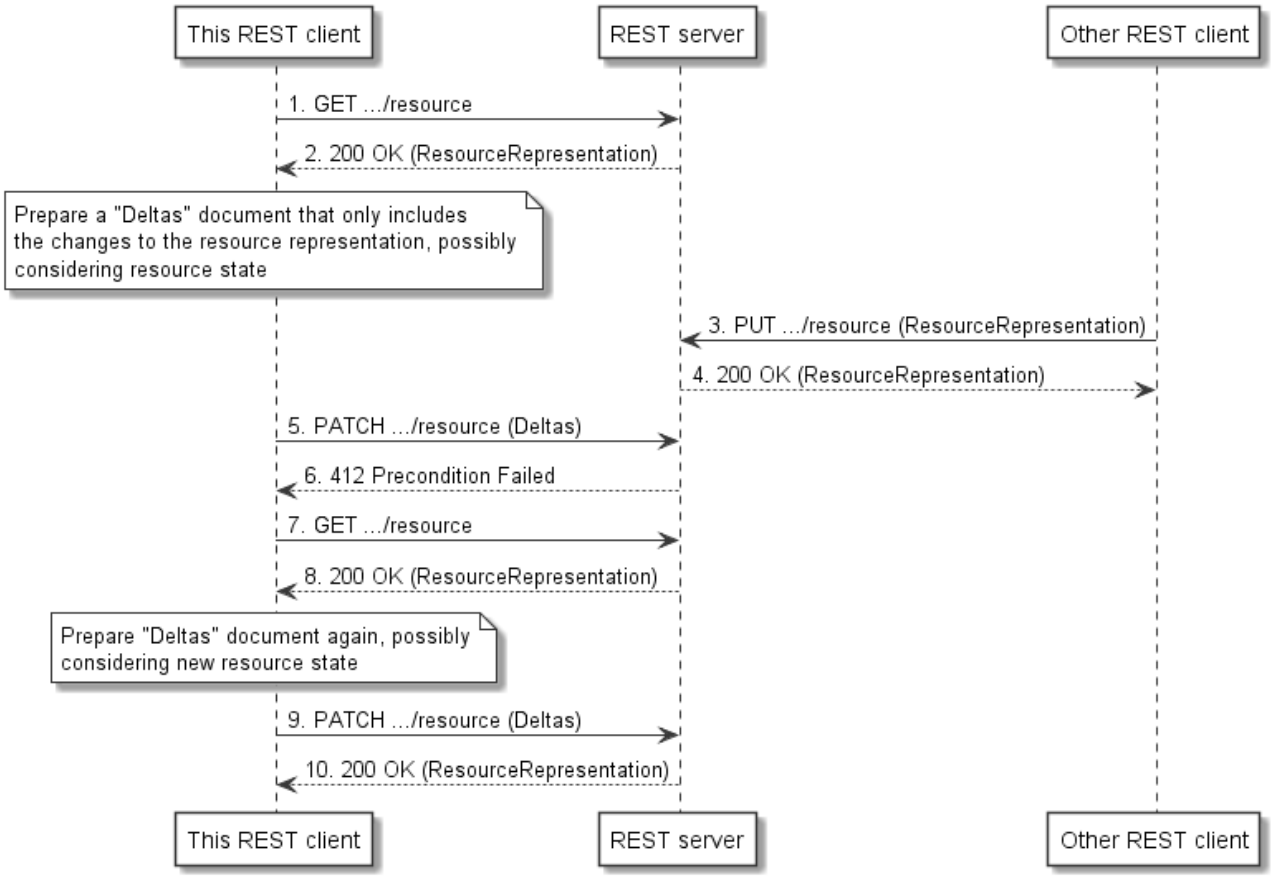


Figure 6.9.1-2: Resource update flow with PATCH, considering concurrent updates

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

6.9.2 Resource definition(s) and HTTP methods

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

6.9.3 Resource representation(s)

The payload body of the PATCH request does not carry a representation of the resource, but a description of the changes in one of the formats defined by IETF RFC 6902 [6], IETF RFC 7396 [5] and IETF RFC 5261 [7].

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

An API specification should suitably specify which parts of the representation of a resource (objects in JSON and elements in XML) are allowed to be modified by the PATCH operation.

When using PATCH with JSON either JSON Patch [6] or JSON Merge Patch [5] are applicable:

- **JSON Patch** addresses an object in the JSON representation of the resource to be changed using a JSON Pointer (IETF RFC 6901 [4]) expression, and provides for that object an operation with the necessary parameters to modify the object, delete the object, or insert a new object. The deltas document is a set of such operations. JSON Patch is capable of expressing modifications to individual array elements. When specifying the allowed modifications, a normative set of restrictions needs to be defined on the path expressions and the operations on these.

- **JSON Merge Patch** uses a subset of the representation of the resource as the deltas document, where this subset only carries the modified objects. The deltas document is a JSON file formatted the same way as the representation of the resource. Objects that are not to be modified are omitted from the deltas document. JSON Merge Patch is not capable of addressing individual array elements; which means that the deltas document needs to contain a complete representation of the array with the changes applied, in case an array needs to be modified. However, lists of objects that have an identifying attribute can be stored in a JSON map (list of key-value-pairs) instead of in an array. The restriction of JSON Merge Patch for arrays does not apply to maps. Therefore, using maps instead of arrays where applicable can make a data model design more JSON Merge Patch friendly. The allowed modifications can simply be specified using the same format (tables, OpenAPI) as for defining the data model for the resource representations.

The APIs defined as part of the ETSI MEC specifications will use IETF RFC 7396 [5] when using PATCH with JSON.

6.9.4 HTTP headers

In the request, the "Content-type" HTTP header needs to be set to the content type registered for the format used to describe the changes, according to IETF RFC 6902 [6], IETF RFC 7396 [5] or IETF RFC 5261 [7].

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

6.9.5 Response codes and error handling

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

Resource update can also be asynchronous in which case "202 Accepted" shall be returned instead of "200 OK". See clause 6.13 for more details about asynchronous operations.

6.10 Pattern: Deleting a resource

6.10.1 Description

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

The response of the DELETE request is typically empty, but it is also possible to return the final representation of the resource prior to deletion.

When a deleted resource is accessed subsequently by any HTTP method, typically the server responds with "404 Not Found", or, if the server maintains knowledge about the URIs of formerly-existing resources, "410 Gone".

Figure 6.10.1-1 illustrates deleting a resource.

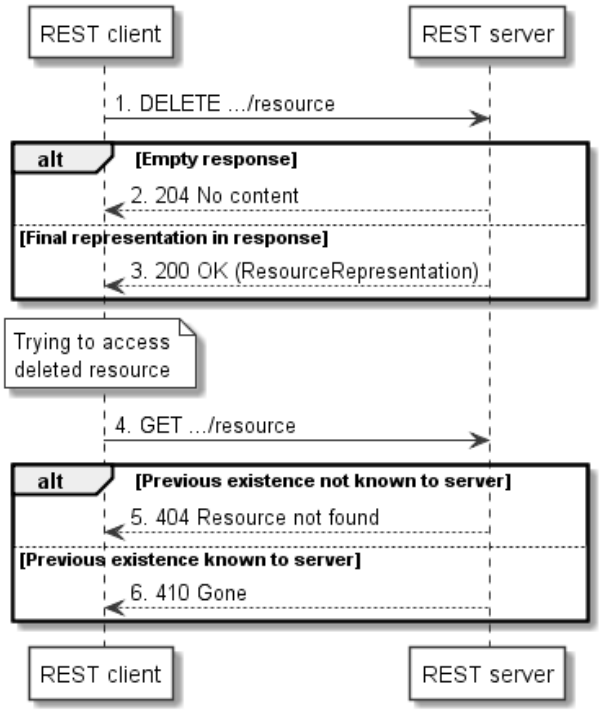


Figure 6.10.1-1: Resource deletion flow

6.10.2 Resource definition(s) and HTTP methods

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

6.10.3 Resource representation(s)

The payload body of the request shall be empty. The payload body of the response is typically empty, but may also include the final representation of the resource prior to deletion.

6.10.4 HTTP headers

No specific provisions for HTTP headers for this pattern.

6.10.5 Response codes and error handling

On success, "204 No Content" should be returned, unless it is the intent to provide the final representation of the resource, in which case "200 OK" should be returned. On failure, the appropriate error code (see annex B) shall be returned.

If a deleted resource is accessed subsequently by any HTTP method, the server shall respond with "410 Gone" in case it has information about the deleted resource available, or shall respond with "404 Not Found" in case it has no such information.

Resource deletion can also be asynchronous in which case "202 Accepted" shall be returned instead of "204 No Content" or "200 OK". See clause 6.13 for more details about asynchronous operations.

6.11 Pattern: Task resources

6.11.1 Description

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

NOTE: In strict REST terms, these URIs are not resources, but endpoints that are included in the resource tree that represent specific non-CRUD operations. Therefore, these special URIs are also sometimes called "custom operations".

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

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

6.11.2 Resource definition(s) and HTTP methods

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

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

The HTTP method shall be POST.

6.11.3 Resource representation(s)

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

6.11.4 HTTP headers

In case the task resource represents an operation that is asynchronous, the provisions in clause 6.13 shall apply.

In case the operation modifies a different resource than the primary resource and the response contains the "303 See Other" response code, the "Location" HTTP header shall point to the modified resource.

6.11.5 Response codes and error handling

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

- For long-running operations, "202 Accepted" is returned. See clause 6.13 for more details about asynchronous operations.
- If the operation modifies another resource, "303 See Other" is returned.
- If the operation returns a computation result, "200 OK" is returned.
- If the operation returns no result, "204 No Content" is returned.

On failure, the appropriate error code (see annex B) shall be returned.

6.12 Pattern: REST-based subscribe/notify

6.12.1 Description

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

NOTE: This clause describes REST-based subscribe/notify. Notifications can also be subscribed to and delivered by an alternative transport mechanism, such as a message bus. There is a separate pattern for this, see clause 7.

To manage subscriptions, the REST server needs to expose a container resource under which the REST client can request the creation/deletion of subscription resources. Those resources optionally define criteria of the subscription as part of the resource URI (such as the "{subscriptionType}" example in figure 6.12.1-1). See clauses 6.5 and 6.10 for the patterns of creating and deleting resources which apply to subscription resources as well.

To receive notifications, the client exposes one or more HTTP endpoints on which it can receive POST requests. When creating a subscription, the client shall inform the server of the endpoint to which the server will later deliver notifications related to that particular subscription. The structure of the URI of that endpoint (aka callback URI) is defined by the client, the string "evt_sink" in figure 6.12.1-1 is an example.

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

Figure 6.12.1-1 illustrates the creation of subscriptions and the delivery of a notification.

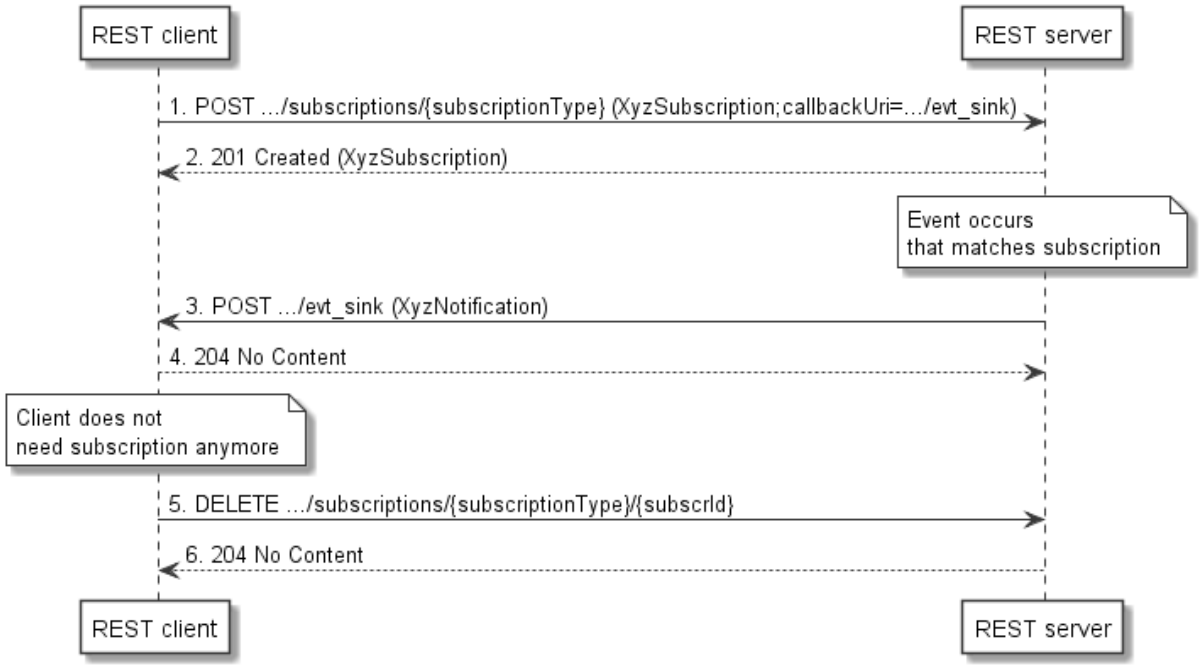


Figure 6.12.1-1: Creation of subscriptions and delivery of a notification

Beyond this very basic scheme described above, the server may also allow the client to update subscriptions, and subscriptions may carry an expiry deadline. Update shall be performed using PUT. In particular, when applying the update operation, the REST client can modify the expiry deadline to refresh a subscription. If the server expires a subscription, it sends an ExpiryNotification to the client's HTTP endpoint defined in the subscription. See clause 6.8 for the pattern of updating a resource using PUT, which applies to the update of subscription resources as well.

Once a subscription is expired, the subscription resource is not available anymore.

Figure 6.12.1-2 illustrates a realization with update and expiry of subscriptions.

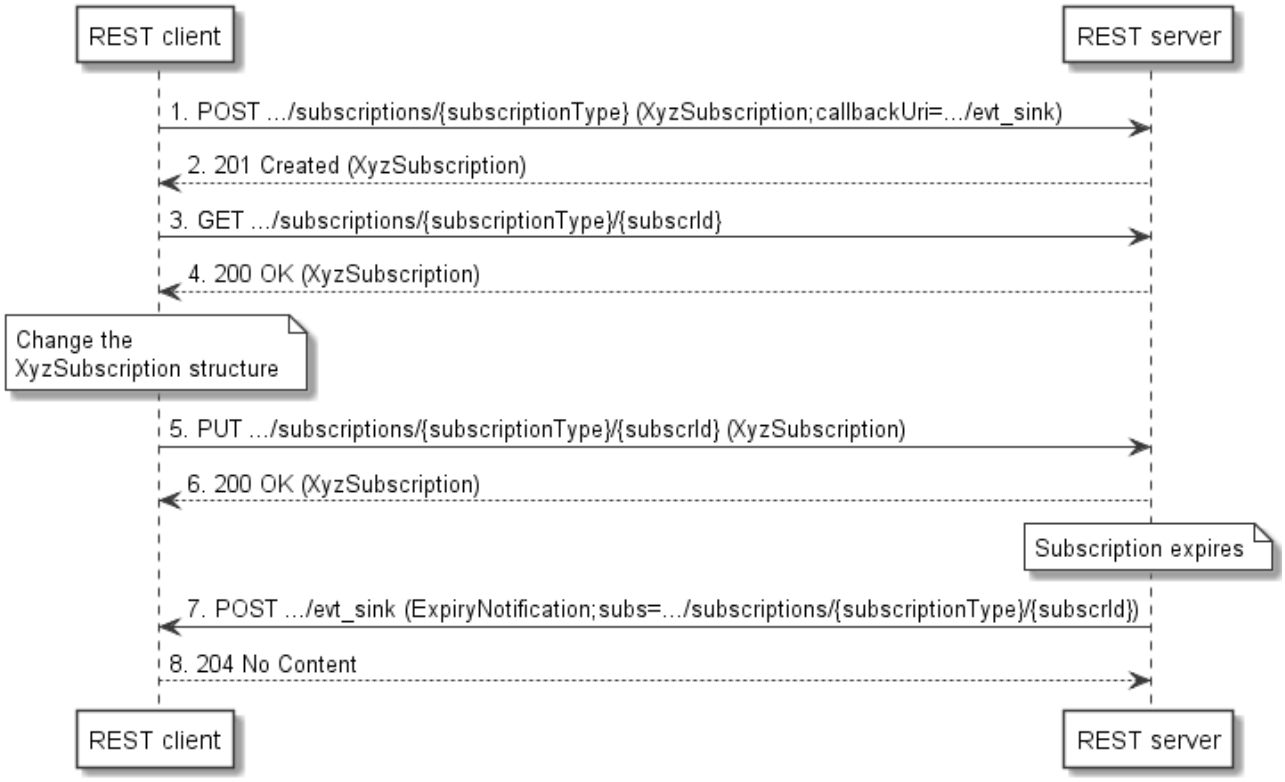


Figure 6.12.1-2: Management of subscriptions with expiry

6.12.2 Resource definition(s) and HTTP methods

The following resources are involved:

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

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

3) An HTTP endpoint that is exposed by the REST client to receive the notifications.

The HTTP method to create a subscription resource inside the subscription container resource shall be POST. The HTTP method to terminate a subscription by removing a subscription resource shall be DELETE. The HTTP method used by the server to deliver the notification shall be POST.

If update of subscriptions is supported, the HTTP method to perform the update shall be PUT.

If expiry of subscriptions is supported, the delivery of an ExpiryNotification to the subscribed clients, and the update of subscription resources should be supported to allow extension of the lifespan of a resource.

6.12.3 Resource representation(s)

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

- It shall contain a callback URI which addresses the HTTP endpoint that the REST client exposes to receive notifications. The callback URI shall be in the form of an absolute URI as defined in section 4.3 of IETF RFC 3986 [9] excluding any query component, any fragment component and any userinfo subcomponent.
- It should contain criteria that allow the server to determine the events about which the client wishes to be notified.
- If expiry of subscriptions is supported, it shall contain an expiry time after which the subscription is no longer valid, and no notifications will be generated for it.

If subscription expiry is supported, the following provisions are applicable to the representation of an ExpiryNotification:

- It shall contain a reference to the subscription that has expired.
- It may contain information about the reason of the expiry.

The following provisions are applicable to the representation of any other notification:

- It should contain a reference to the related subscription.
- It shall contain information about the event.

6.12.4 HTTP headers

No specific provisions are applicable here.

6.12.5 Response codes and error handling

The response codes for subscription creation, subscription deletion, subscription read and subscription update are the same as for the general resource creation, resource deletion, resource read and resource update.

On success of notification delivery, "204 No Content" shall be returned.

On failure, the appropriate error code (see annex B) shall be returned.

If expiry of subscriptions is supported: Once an expiry notification has been delivered to the client, any HTTP request to the expired subscription resource shall fail. For a timespan determined by policy or implementation, "410 Gone" is recommended to be used as the response code in that case, and "404 Not Found" shall be used afterwards.

NOTE: In order to be able to respond with "410 Gone", the server needs to keep information about the expired subscription.

6.12a Pattern: REST-based subscribe/notify with Websocket fallback

6.12a.1 Description

REST-based delivery of notifications as defined in clause 6.12 might not work in case middleboxes block the HTTP connection attempts to send the POST requests that deliver the notifications. An example where a NAT middlebox blocks the notification delivery over HTTP is illustrated in figure 6.12a.1-1.

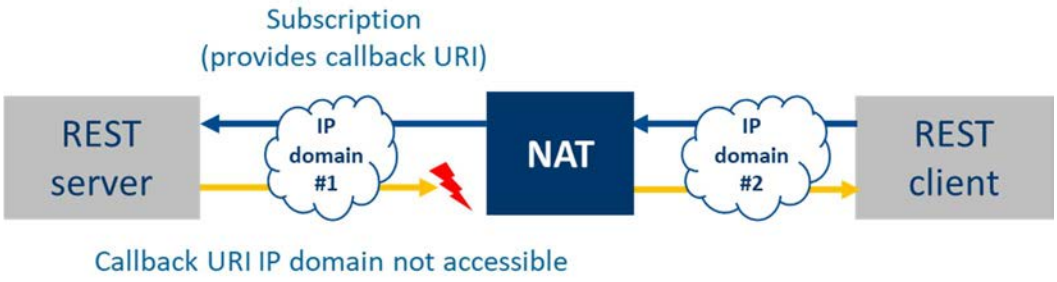


Figure 6.12a.1-1: NAT as an example of a middlebox blocking notification delivery over HTTP

In order to cope with such network topologies, the direction of setting up the TCP connection through which the notifications are sent needs to be reversed, and this needs to be done in an HTTP-proxy compatible way. Websockets (see IETF RFC 6455 [25]) provide a solution that fulfils these requirements.

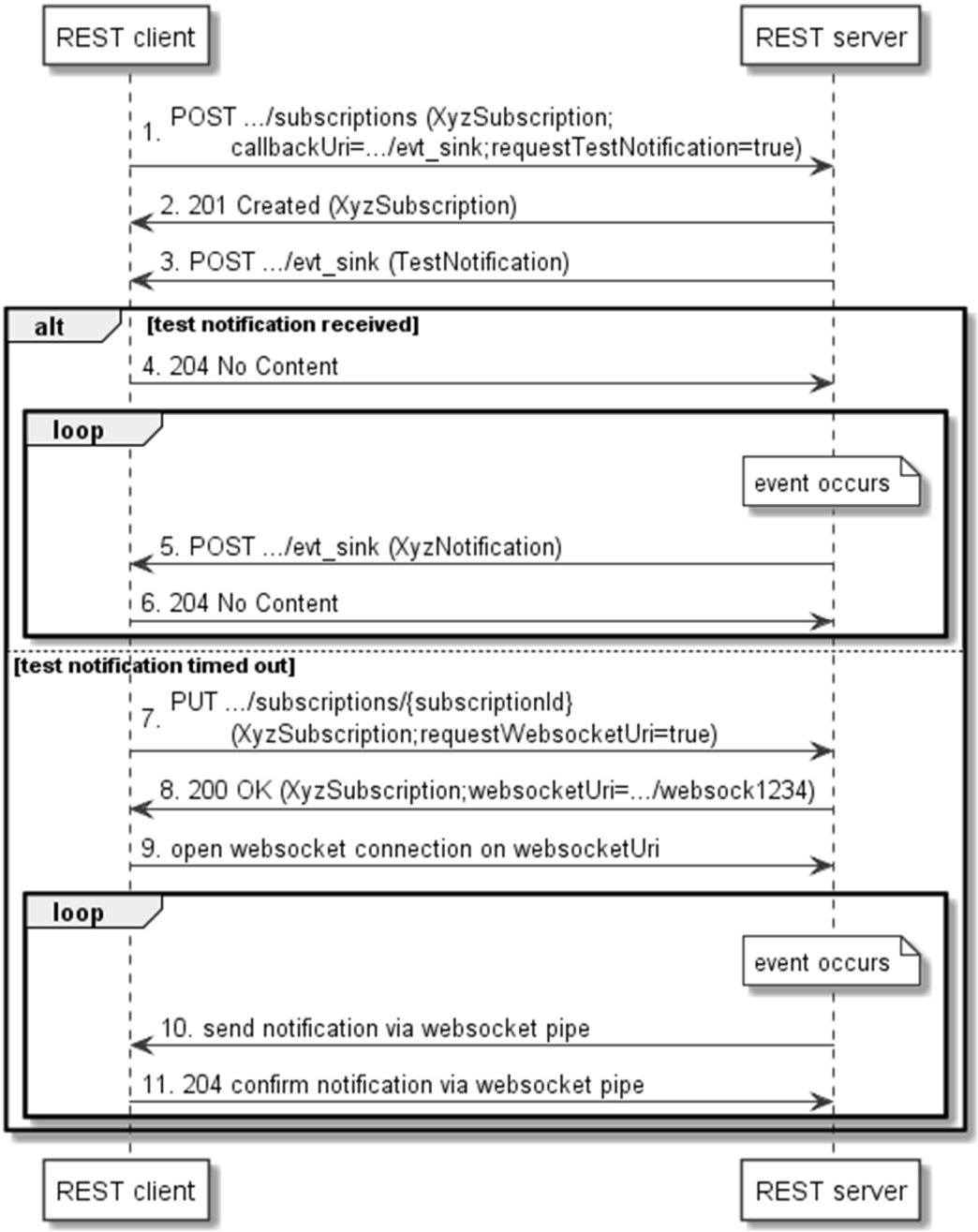


Figure 6.12a.1-2: Topology probing and notification delivery mechanism negotiation

Figure 6.12a.1-2 illustrates the steps of topology probing during subscription, and the following negotiation of the Websocket URI in case of HTTP transport of notifications is blocked. This mechanism is aligned with 3GPP T8 (see ETSI TS 129 122 [26] and re-used by CAPIF (see ETSI TS 129 222 [i.15]).

1) The client subscribes to notifications by sending a POST request to the container resource that holds the subscriptions, providing in the request body a callback URI (e.g. "…/evt_sink") and a flag to request a test notification.

2) The server creates a subscription resource and informs the client about the creation of that resource.

3) The server sends to the callback URI that was indicated in the subscription a POST request and includes in the payload body a test notification.

In case the test notification is received by the client, steps 4-6 are executed:

4) The client confirms with "204 No Content" that the test notification was received.

The following steps 5 and 6 are executed in a loop for each event that triggers a notification:

5) The server sends a POST request to the callback URI and includes in the payload body a notification structure related to the event.

6) The client confirms with "204 No Content" that the notification was received.

In case the test notification is not received by the client before a time-out, steps 7-11 are executed:

7) The client sends a PUT request to the subscription resource to update the notification delivery method to Websockets.

8) The server provides a Websocket endpoint to which the client will subsequently connect to set up a Websocket connection. Further, the server includes the URI of that endpoint in the representation of the subscription resource and returns a "200 OK" response.

9) The client opens to the provided Websocket endpoint a Websocket connection through which the server can subsequently push notifications.

The following steps 10 and 11 are executed in a loop for each event that triggers a notification:

10) The server sends to the client, via the Websocket connection, a notification structure related to the event with the appropriate framing.

11) The client confirms the delivery to the server, via the Websocket connection, with a "204" response code in the appropriate framing.

In case neither the 204 response (step 4) nor the subscription update (step 7) is received by the server within an implementation-specific time interval, the server should retry sending the test notification. The behaviour of the server in case of multiple subsequent failures is out of the scope of the present document.

6.12a.2 Resource definition(s) and HTTP methods

The resources defined in clause 6.12.2 and the following are involved:

1) A Websocket endpoint that is exposed by the REST server to establish a Websocket connection for notifications delivery.

The HTTP methods are as defined in clause 6.12.2, with the following addition: The PUT method to update the subscription shall be supported in order to enable updating the delivery mechanism.

In addition to the HTTP methods, the delivery of notifications from the server to the client through a Websocket connection shall be supported. Such delivery shall use the framing that is defined in clause 5.2.5.4 of ETSI TS 129 122 [26], where the REST client corresponds to the SCS/AS and the REST server corresponds to the SCEF. For that purpose, the establishment of a Websocket connection by the REST client towards a Websocket endpoint provided by the REST server shall be supported.

6.12a.3 Resource representation(s)

The provisions defined in clause 6.12.3 for the representation of a subscription resource apply.

In addition, the following shall be supported:

1) Specific attributes in the subscription data structure to negotiate and signal the use of Websockets

   The representation of a subscription resource shall include the attributes defined in table 6.12a.3-1.

Table 6.12a.3-1: Definition of subscription attributes for Websocket negotiation and signaling

Attribute name	Data type	Cardinality	Description
callbackUri	Uri	0..1	URI exposed by the client on which to receive notifications via HTTP. See note.
requestTestNotification	Boolean	0..1	Shall be set to TRUE in a request to create a subscription if the client intends to receive a test notification via HTTP on the callbackUri that it provides in the same subscription request. Default: FALSE.
websockNotifConfig	WebsockNotifConfig	0..1	Provides details to negotiate and signal the use of a Websocket connection, as defined in clause 5.2.1.2.10-1 of ETSI TS 129 122 [26]. The server may assign the same websocket URI to multiple subscriptions of the same client. See note.
…	…	…	(Additional attributes of the subscription data structure)
NOTE: At least one of callbackUri and websocketNotifConfig shall be provided in any subscription. If both are provided, it is up to the server to choose an alternative and return only that alternative in the response.			

2) A test notification payload

   The test notification shall include the attributes defined in clause 5.2.1.2.9 of ETSI TS 129 122 [26]. It may include additional attributes.

   It shall be sent via HTTP by the REST server to the callback URI provided by the REST client upon subscription, if the REST client has indicated in the subscription the flag "requestTestNotification=true", according to clause 5.2.5.3 of ETSI TS 129 122 [26].

6.12a.4 HTTP headers

No specific provisions are applicable here.

6.12a.5 Response codes and error handling

The same provisions as in clause 6.12.5 apply.

6.13 Pattern: Asynchronous operations

6.13.1 Description

Certain operations, which are invoked via a RESTful interface, trigger processing tasks in the underlying system that may take a long time, from minutes over hours to even days. In this case, it is inappropriate for the REST client to keep the HTTP connection open to wait for the result of the response - the connection will time out before a result is delivered. For these cases, asynchronous operations are used. The idea is that the operation immediately returns the provisional response "202 Accepted" to indicate that the request was understood, can be correctly marshalled in, and processing has started. The client can check the status of the operation by polling; additionally, or alternatively, the subscribe-notify mechanism (see clause 6.12) can be used to provide the result once available. The progress of the operation is reflected by a monitor resource.

Figure 6.13.1-1 illustrates asynchronous operations with polling. After receiving an HTTP request that is to be processed asynchronously, the server responds with "202 Accepted" and includes in the payload body or in a specific "Link" HTTP header a data structure that points to a monitor resource which represents the progress of the processing operation. The client can then poll the monitor resource by using GET requests, each returning a data structure with information about the operation, including the processing status such as "processing", "success" and "failure". Initially, the status is set to "processing". Eventually, when the processing is finished, the status is set to "success" (for successful completion of the operation) or "failure" (for completion with errors). Typically, the representation of a monitor resource will include additional information, such as information about an error if the operation was not successful.

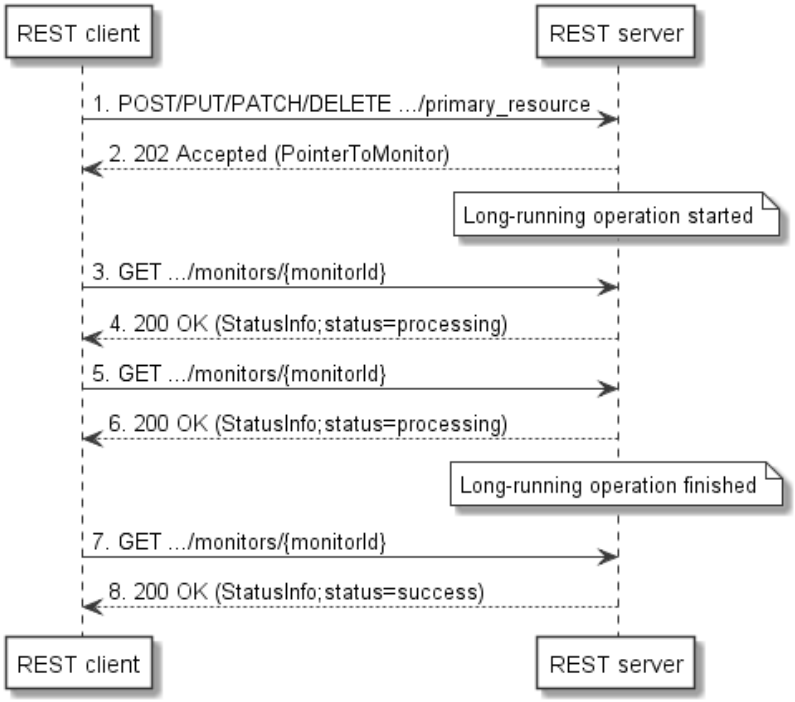


Figure 6.13.1-1: Asynchronous operation flow - with polling

Figure 6.13.1-2 illustrates asynchronous operations with subscribe/notify. Before a client issues any request that may be processed asynchronously, it subscribes for monitor change notifications. Later, after receiving an HTTP request that is to be processed asynchronously, the server responds with "202 Accepted" and includes in the payload body or in a specific "Link" HTTP header a data structure that points to a monitor resource which represents the progress of the processing operation. The client can now wait for receiving a notification about the operation finishing, which will change the status of the monitor. Once the operation is finished, the server will send to the client a notification with a structure in the payload body that typically includes the status of the operation (e.g. "success" or "failure"), a link to the actual monitor affected, and a link to the resource that is modified by the asynchronous operation, The client can then poll the monitor to obtain further information.

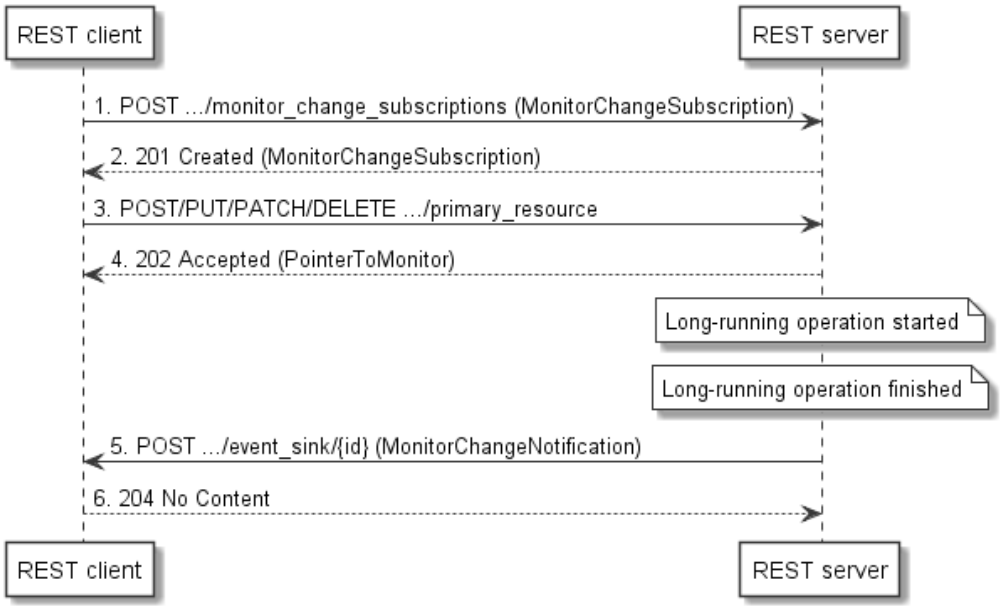


Figure 6.13.1-2: Asynchronous operation flow - with subscribe/notify

6.13.2 Resource definition(s) and HTTP methods

The following resources are involved:

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

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

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

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

If monitor change notifications and subscriptions to these are supported, the resources and methods described in clause 6.12 for the RESTful subscribe/notify pattern are applicable here too.

6.13.3 Resource representation(s)

If present, the structure included in the payload body of the response to the long-running operation request shall contain the resource URI of the monitor for the operation, and shall also contain the resource URI of the actual primary resource. See clause 6.14 for further information on links. If no payload body is present, the "Link" HTTP header shall be used to convey the link to the monitor.

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

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

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

6.13.4 HTTP headers

The link to the monitor should be provided in the "Link" HTTP header (see IETF RFC 8288 [12]), with the "rel" attribute set to "monitor". If the payload body of the message is not present, the "Link" as defined above shall be provided.

EXAMPLE: Link: <http://mecplat0815.example.com/.../monitors/mtr061070>; rel="monitor".

6.13.5 Response codes and error handling

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

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

If subscribe/notify is supported, the provisions in clause 6.12.5 apply in addition.

6.14 Pattern: Links (HATEOAS)

6.14.1 Description

The REST maturity level 3 requires the use of links between resources, allowing the REST client to traverse the resource space. ETSI MEC recommends using level 3. This is also known as "hypermedia controls" or "HATEOAS" (hyperlinks as the engine of application state). This clause describes a pattern for hyperlinks.

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

6.14.2 Resource definition(s) and HTTP methods

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

6.14.3 Resource representation(s)

Links are communicated in the resource representation. Links that occur at the same level in the representation shall be bundled in an object (JSON) or element containing complexContent (XML schema), named "_links" which should occur as the first object/element at a particular level.

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

As an example, the "_links" portion of a resource representation is shown that represents paged information.

For the case of using XML, figure 6.14.3-1 illustrates the XML schema and figure 6.14.3-2 illustrates the XML instance. The XML schema language is defined in [i.5].

```
<xsd:complexType name="LinkType">
    <xsd:attribute name="href" type="xsd:anyURI" use="required"/>
</complexType>


<xsd:element name="_links">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="self" type="LinkType"/>
            <xsd:element name="next" type="LinkType" minOccurs="0"/>
            <xsd:element name="prev" type="LinkType" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Figure 6.14.3-1: XML schema fragment for an example "_links" element

```
<_links>
    <self href="http://api.example.com/my_api/v1/pages/127"/>
    <next href="http://api.example.com/my_api/v1/pages/128"/>
    <prev href="http://api.example.com/my_api/v1/pages/126"/>
</_links>
```

Figure 6.14.3-2: XML instance fragment for an example "_links" element

For the case of using JSON, figure 6.14.3-3 illustrates the JSON schema and figure 6.14.3-4 illustrates the JSON object. The JSON schema language is defined in [i.4].

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

Figure 6.14.3-3: JSON schema fragment for an example "_links" object

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

Figure 6.14.3-4: JSON fragment for an example "_links" object

6.14.4 HTTP headers

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

NOTE: Specific links, such as a link to the monitor in a "202 Accepted" response, can be communicated in the "Link" HTTP header. See clause 6.13 for more details.

6.14.5 Response codes and error handling

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

6.15 Pattern: Error responses

6.15.1 Description

In RESTful interfaces, application errors are mapped to HTTP errors. Since HTTP error information is typically not enough to discover the root cause of the error, there is the need to deliver additional application specific error information.

When an error occurs that prevents the REST server from successfully fulfilling the request, the HTTP response includes 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 [1] and IETF RFC 6585 [8]). In addition, to provide additional application-related error information, the present document recommends the response body to contain a representation of a "ProblemDetails" data structure according to IETF RFC 7807 [15] that provides additional details of the error.

6.15.2 Resource definition(s) and HTTP methods

The pattern is applicable to the responses of all HTTP methods.

6.15.3 Resource representation(s)

If an HTTP response indicates non-successful completion (error codes 400..499 or 500..599), the response body should contain a "ProblemDetails" data structure as defined below, formatted using the same format as the expected response. The response body may be omitted if the HTTP error code itself provides enough information of the error, or if there are security concerns disclosing detailed error information.

The definition of the general "ProblemDetails" data structure from IETF RFC 7807 [15] is reproduced in table 6.15.3-1. Compared to the general framework in IETF RFC 7807 [15] where the "status" and "detail" attributes are recommended to be included, these attributes shall be included when this data structure is used in the context of the ETSI MEC REST APIs, 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 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 6.15.3-1: Definition of the ProblemDetails data type

Attribute name	Data type	Cardinality	Description
type	Uri	0..1	A URI reference according to IETF RFC 3986 [9] 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". See note 1.
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 (see note 2)	The HTTP status code for this occurrence of the problem. See note 3.
detail	String	1 (see note 2)	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 1: For the definition of specific "type" values as well as extension attributes by implementations, detailed guidance can be found in IETF RFC 7807 [15].
NOTE 2: In IETF RFC 7807 [15], the "status" and "detail" are recommended which would translate into a cardinality of 0..1, but the present document requires the presence of these attributes as the minimum set of information returned in ProblemDetails.
NOTE 3: IETF RFC 7807 [15] requires that this attribute duplicates the value of the status code in the HTTP response message. See section 5 of IETF RFC 7807 [15] for guidance related to the two values differing.

6.15.4 HTTP headers

As defined by IETF RFC 7807 [15]:

- In case of serializing the "ProblemDetails" structure using the JSON format, the "Content-Type" HTTP header shall be set to "application/problem+json".
- In case of serializing the "ProblemDetails" structure using the XML format, the "Content-Type" HTTP header shall be set to "application/problem+xml".

6.15.5 Response codes and error handling

In general, 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 any valid HTTP response code as error code in the HTTP response, but shall not use any code that is not a valid HTTP response code. A list of all valid HTTP response codes and their specification documents can be obtained from the HTTP status code registry [i.8]. Annex B lists a set of error codes that is frequently used in HTTP-based RESTful APIs. Annex E provides more detail on common error situations.

6.16 Pattern: Authorization of access to a RESTful MEC service API using OAuth 2.0

6.16.1 Description

This pattern defines the use of OAuth 2.0 to secure a RESTful MEC service API. It is used for the RESTful APIs that are defined by ETSI ISG MEC. Service-producing applications defined by third parties may use other mechanisms to secure their APIs, such as standalone use of JWT (see IETF RFC 7519 [i.13]).

The API security framework assumes an AA (authentication and authorization) entity to be available for both the REST client and the REST server. This AA entity performs the authentication for the credentials of the REST clients and the REST servers. The AA entity and the communication between the REST server and the AA entity are out of scope of the present document.

It is assumed that the AA entity is configured by a trusted Manager entity with the appropriate credentials and access rights information. This configuration information is exchanged between the AA entity and the REST server in an appropriate manner to allow the REST server to enforce the access rights. The trusted Manager and the actual way of performing the exchange of this information are out of scope.

The exchanges between REST client and REST server are in scope of the present document. The REST client has to authenticate towards the AA entity in order to obtain an access token. Subsequently, the client shall present the access token to the REST server with every request in order to assert that it is allowed to access the resource with the particular method it invokes. In the present version of the specification, the client credentials grant type of OAuth 2.0 (see IETF RFC 6749 [16]) shall be supported by the AA entity, and it shall be used by the REST client to obtain the access token. In any HTTP request to a resource, the access token shall be included as a bearer token according to IETF RFC 6750 [17].

Access rights are bound to access tokens, and typically configured at the granularity of methods applied to resources. This means, for any resource in the API, the use of every individual method can be allowed or disallowed. In APIs that define a REST-based subscribe-notify pattern, also the use of individual subscription types can be allowed or prohibited by access rights. Additional policies can be bound to access tokens too, such as the frequency of API calls. A token has a lifetime after which it is invalid. Depending on how the AA communicates with the REST server, it can also be possible to revoke a token before it expires.

Figure 6.16.1-1 illustrates the information flow between the three actors involved in securing the REST-based service API, the REST client, the AA entity and the REST server. Dotted lines indicate exchanges that are out of scope of the present document. It is assumed that information about the valid access tokens, such as expiry time, related client identity, client access rights, scope values, optional revocation information, need to be made available by the AA entity to the REST server by means outside the scope of the present document.

The AA entity exposes the "token endpoint" as defined by OAuth 2.0.

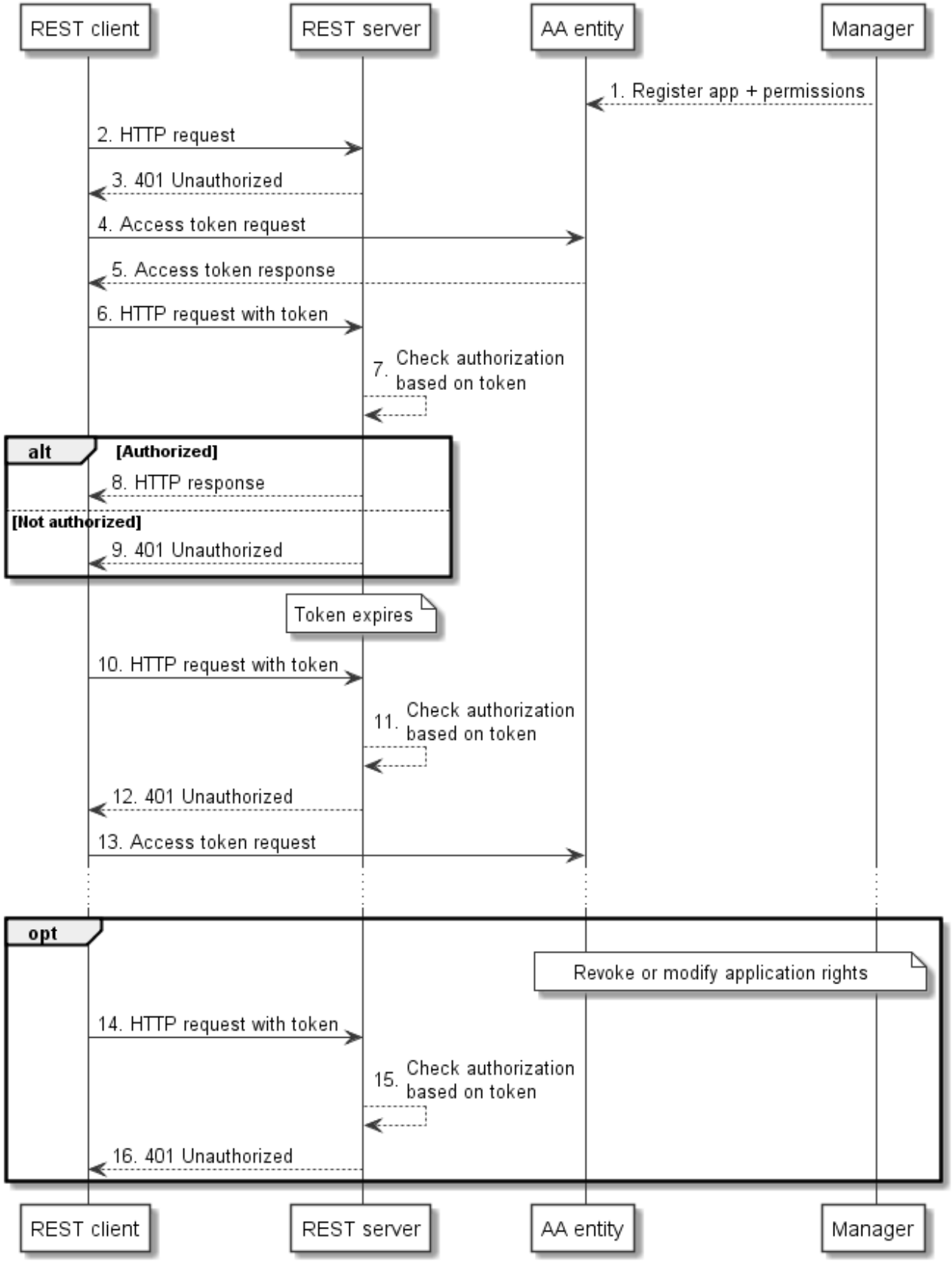


Figure 6.16.1-1: Securing a RESTful MEC service API with OAuth 2.0

The flow consists of the following steps:

1) The manager registers the REST client application with the AA entity and configures the permissions of the application. The method for this is out of scope of the present document.

2) The REST client sends an HTTP request to the REST server to access a resource.

3) The REST server responds with "401 Unauthorized" which indicates to the client that it has to obtain an access token for access to the resource.

4) The REST client sends an access token request to the token endpoint provided by the AA entity as specified by IETF RFC 6749 [16], and authenticates towards the AA entity with its client credentials.

5) The AA entity provides the token and additional configuration information to the REST client, as specified by IETF RFC 6749 [16].

6) The REST client repeats the request from step (2) with the access token included as a bearer token according to IETF RFC 6750 [17].

7) The REST server checks the token for validity, and determines whether the client is authorized to perform the request. This assumes that the REST server has received from the AA entity information about the valid access tokens, and additional related parameters (e.g. expiry time, client identity, client access rights, scope values). Exchange of such information is outside the scope of the present document, and is assumed to be trivial if deployments choose to include the AA entity as a component into the REST server.

8) In case the client is authorized, the REST server executes the HTTP request and returns an appropriate HTTP response rather than a "401 Unauthorized" error.

9) In case the client is not authorized, the REST server returns a "401 Unauthorized" error as defined in IETF RFC 6750 [17].

10) The REST client sends to the REST server an HTTP request with an expired token.

11) The REST server checks the token for validity, and establishes that it has expired. This assumes that the REST server has previously received information about the valid access tokens, and additional related information (in particular, the time of expiry) from the AA entity. Exchange of such information is outside the scope of the present document, and is assumed to be trivial if deployments choose to include the AA entity as a component into the REST server.

12) The REST server responds with "401 Unauthorized", and uses the format defined in IETF RFC 6750 [17] to communicate that the access token is expired.

13) The REST client sends a new access token request to the AA entity, as defined in step (4). Subsequently, steps (5) to (9) repeat.

Optionally:

14) The REST client sends to the REST server an HTTP request with a revoked token. For this optional sequence, it is assumed that the Manager has arranged to block an application from accessing a particular resource or set of resources, or has changed the application's access rights prior to that request. By means outside the scope of the present document, the Manager has further informed the AA entity about this change.

15) The REST server checks the token for validity, and establishes that it has been revoked. This assumes that the REST server has previously received information about the validity of the access token from the AA entity. Exchange of such information is outside the scope of the present document, and is assumed to be trivial if deployments choose to include the AA entity as a component into the REST server.

16) The REST server responds with "401 Unauthorized". Eventually, the REST client can succeed with another subsequent access token request if the revocation only affected a subset of the resources.

6.16.2 Resource definition(s) and HTTP methods

The HTTP methods follow the corresponding RESTful MEC service API definitions. Typically, when configuring the AA entity, access rights can be expressed separately for each resource and HTTP method. In case subscriptions are supported, separate access rights can also be defined per subscription data type.

6.16.3 Resource representation(s)

The representation of the information exchanged between the REST client and the Token endpoint of the AA entity shall follow the provisions defined in IETF RFC 6749 [16] for the client credentials grant type. The representation of information exchanged between the Manager and the AA entity, as well as between the AA entity and the REST server, are outside the scope of the present document.

6.16.4 HTTP headers

In this pattern, the access token is provided as defined by IETF RFC 6750 [17]. To protect the access token from wiretapping, HTTPS shall be used.

6.16.5 Response codes and error handling

The response codes on the API between the REST server and the REST client are defined in the corresponding RESTful MEC service API definitions, and shall include the provisions in IETF RFC 6750 [17]. The response codes on the token endpoint provided by the AA entity shall follow IETF RFC 6749 [16].

6.16.6 Discovery of the parameters needed for exchanges with the token endpoint

In order to be able to communicate with the token endpoint for a particular API, the REST client needs to discover its URI. The valid scope values (if supported) are part of the API documentation. The client further needs to know which set of client credentials to use to access the token endpoint. The token endpoint URI and the optional scope values will be provided as part of the security information during service discovery. The client credentials consist of the client identifier which is defined based on information in the application descriptor such as the values of the attributes "appProvider" and "appName", and the client password which is provisioned during application on-boarding, and configured into the client and the AA entity by means outside the scope of the present document.

6.16.7 Scope values

OAuth 2.0 (IETF RFC 6749 [16]) supports the concept of scope values to signal which actual access rights a token represents. The scope of the token can be requested by the client in the access token request by listing one or more scope values in the "scope" parameter. The AA entity can then potentially downscope the request, and respond with the actual scope(s) represented by an access token in the access token response in the "scope" parameter. The use of scopes is optional in OAuth 2.0. Per API, valid scopes can be defined in the API specification. Possible granularities are resources, combinations of resources and methods, or even combinations of resources and methods with actual parameter values, or values of attributes in the payload body. If no scope is defined, an access token always applies to all resources and methods of a particular API. For a REST API using OAuth 2.0, the "permission identifiers" as defined in clause 7.2 can be modelled as scope values, as illustrated in table 7.2-3. It is good practice to define one additional scope value per API that includes all individual access rights, for simplification of use.

6.17 Pattern: Representation of lists in JSON

6.17.1 Description

Lists of objects in JSON can be represented in two ways: arrays and maps.

6.17.2 Representation as arrays

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

When modifying an array with PATCH (see clause 6.9), modifications can be represented by passing only the changes when using the JSON Patch (IETF RFC 6902 [6]) format for the delta document, or by passing the whole modified array when using the JSON Merge Patch (IETF RFC 7396 [5]) format for the delta document.

Figure 6.17.2-1 provides an example of a list of objects represented as JSON array.

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

Figure 6.17.2-1: Example of an array of JSON objects

6.17.3 Representation as maps

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

When modifying a map with PATCH (see clause 6.9), modifications can be represented by passing only the changes when using either of the JSON Patch (IETF RFC 6902 [6]) format or the JSON Merge Patch (IETF RFC 7396 [5]) format for the delta document.

Figure 6.17.3-1 provides an example of a list of objects represented as JSON map, using the same data as in figure 6.17.2-1.

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

Figure 6.17.3-1: Example of a map of JSON objects

6.18 Pattern: Attribute selectors

6.18.1 Description

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 client application.

An attribute selector, which is typically part of a query, allows the client 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.

6.18.2 Resource definition(s) and HTTP methods

The pattern is applicable to GET methods on specific resources. The applicability of attribute selectors is specified in the API specifications per resource.

The attribute selector is represented using URI query parameters, as defined in table 6.18.2-1.

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

Table 6.18.2-1: 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.
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.
exclude_fields	This URI query parameter requests that the listed complex attributes are excluded from the response. For the format and eligible attributes, 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 API specification. 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. This parameter is a flag, i.e. it has no value.

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 [4]. 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 [9] shall be applied to the characters that are not allowed in a URI query part according to Appendix A of IETF RFC 3986 [9], and to the ampersand "&" character.

Support of the attribute selector parameters can be defined per API. Only resources that represent a list of items and that support a GET request are candidates for supporting attribute selectors. It can be decided in the API design if all such resources actually need to support attribute selectors. Typically, list resources with items that have many and/or complex attributes benefit from support of selectors, whereas for list resources with items that have only a few simple attributes, support of attribute selectors can be overhead with no benefit.

For each resource, it needs to be specified which attribute selector parameters are mandatory to support by the server, and which ones are optional to support by the server. Use of these parameters is typically optional for the client. Support for all_fields only makes sense if exclude_default is supported as well. There are two possible default values for attribute selectors: "all_fields" and "exclude_default".

The "all_fields" value is recommended to be used as default when the goal is to represent a list of items the same way as the individual items, i.e. including all attributes, and if the response lists typically are short. For long lists and many complex or array-type attributes, this default can result in large response data volumes.

The "exclude_default" value is recommended to be used as default when the goal is to create a list that is a digest of the individual items. This way, detailed information can be omitted, and the size of the response can be reduced. If the individual list items contain a "self" link (see clause 6.14.3), it is recommended that this link is included in the response list, as this facilitates easy drilldown on individual list items using subsequent GET requests.

6.18.3 Resource representation(s)

Table 6.18.3-1 defines the valid parameter combinations in a GET request and their effect on the response body.

Table 6.18.3-1: Effect of valid combinations of attribute selector parameters on the response body

Parameter combination	The GET response body 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 API specification for the particular resource.
exclude_default and fields=<list>	… all attributes except those complex attributes with a minimum cardinality of zero that are not conditionally mandatory and that are part of the "default exclude set" defined in the API specification for the particular resource, but that are not part of <list>.

6.18.4 HTTP headers

There are no specific HTTP headers for this pattern.

6.18.5 Response codes and error handling

In case of success, the response code 200 OK shall be returned.

In case of an invalid attribute selector, "400 Bad Request" shall be returned, and the response body shall contain a ProblemDetails structure, in which the "detail" attribute should convey more information about the error.

6.19 Pattern: Attribute-based filtering

6.19.1 Description

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 queries 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.

6.19.2 Resource definition(s) and HTTP methods

This pattern is used in GET operations on a resource that will return a list or collection of objects, such as a container resource with child resources.

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 special attribute name "@key" refers to the key of a map.

EXAMPLE 1: Referencing the key of a map

```
GET …/resource?filter=(eq,mymap/@key,abc123)
```

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 6.19.2-1 shall be supported.

Table 6.19.2-1: 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 6.19.2-2: 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 6.19.2-2 defines which operators are applicable for which data types. All combinations marked with a "x" shall be supported.

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 [20].
- The syntax of Boolean and Number <value> entries shall follow IETF RFC 8259 [10].

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 [4]. If the "," character appears in <attrName> it shall be escaped by replacing it with "~a". If the "@" character appears in <attrName> other than in the keyword "@key", it shall be escaped by replacing it with "~b".

In the resulting <filterExpr>, percent-encoding as defined in IETF RFC 3986 [9] shall be applied to the characters that are not allowed in a URI query part according to Appendix A of IETF RFC 3986 [9], 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 [9].

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

6.19.3 Resource representation(s)

The resource representation in the response body shall only contain those objects that match the filter.

6.19.4 HTTP headers

There are no specific HTTP headers for this pattern.

6.19.5 Response codes and error handling

In case of success, the response code 200 OK shall be returned.

In case of an invalid attribute filtering query, "400 Bad Request" shall be returned, and the response body shall contain a ProblemDetails structure, in which the "detail" attribute should convey more information about the error.

6.20 Pattern: Handling of too large responses

6.20.1 Description

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 server 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 client 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 client, but that allows the server to determine the start of the next page.

For each container resource (i.e. a resource that contains child resources whose representations will be returned when responding to a GET request), the server 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) Option 1 (error response): Return an error response if the result set gets too large.

2) Option 2 (paging): Return the result in a paged manner if the result set gets too large.

Clauses 6.20.2, 6.20.3 and 6.20.4 specify these two options.

6.20.2 Resource definition(s) and HTTP methods

This pattern is applicable to any container resource and the GET HTTP method.

For resources that support option 2 (paging), the "nextpage_opaque_marker" URI query parameter shall be supported in GET requests.

6.20.3 Resource representation(s)

If option 2 (paging) is supported and the response result is too big to be returned in a single response, the resource representation in the response body shall contain a single page (subset of the complete query result).

6.20.4 HTTP headers

If option 2 (paging) is supported, the server shall include in a paged response a LINK HTTP header (see IETF RFC 8288 [12]) with the "rel" attribute set to "next", which communicates a URI that allows to obtain the next page of results to the original query, unless there are no further pages available after the current page in which case the LINK header shall be omitted.

The client 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 subsequent next page, as specified above.

To allow the server 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 server. This parameter has no meaning for the client but is echoed back by the client to the server 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.

6.20.5 Response codes and error handling

If option 1 (error response) is supported, 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 client 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 6.19).

7 Alternative transport mechanisms

7.1 Description

A MEC service needs a transport to be delivered to a MEC application. The default transport fully specified by ETSI MEC for MEC service APIs is HTTP-REST.

An alternative transport can also be specified for certain services that require higher throughput and lower latency than a REST-based mechanism can provide. Possible alternative transports at the time of writing are topic-based message buses (e.g. MQTT [i.9] or Apache Kafka® [i.10]) and Remote Procedure Call frameworks (e.g. gRPC® [i.11]). Note that not all aspects of such alternative transport mechanisms can be fully standardized, but some are left to implementation.

A transport can either be part of the MEC platform, or can be made available by the MEC application that provides the service (BYOT - Bring Your Own Transport). REST and gRPC® are always BYOT as the endpoint is the piece of software that provides the service.

Service registration consists of two phases:

1) Transport discovery (only for non-BYOT)

2) Service registration including transport binding

Step 1 is performed using the "transport discovery" procedure on Mp1 (see ETSI GS MEC 011 [i.6]), to obtain a list of available transports, and only needed for a non-BYOT service. Step 2 is the "service registration" procedure on Mp1 which allows to bind a provided service to a transport. This means, a non-BYOT service registers the identifier of the platform-provided transport it intends to use, and a BYOT service registers the information of its own transport.

Transport information includes a definition of the access to the transport (e.g. URI, network address, or implementation-specific), the type of the transport (e.g. HTTP-REST, message bus, RPC, etc.), security information, metadata such as identifier, name and description, and a container for implementation-specific information. It is further specified in ETSI GS MEC 011 [i.6].

Sending data on a transport requires serialization. There are different serialization formats, for example JSON [10], XML [11] or Protocol Buffers [i.12]. Binding a service to a transport therefore also requires choosing a serialization format to be used. The data structures defined for the service can be bound to different serialization formats. The definition of the binding has to be done as part of the service definition. The JSON binding is typically fully specified for a RESTful API. Bindings to additional serializers can be provided, either in the documents defined by ETSI MEC, or in documents provided to the developer community by the MEC application vendors.

A further aspect of alternative transports is the mechanism how to secure a transport pipe (TLS, typically) and how a transport or the service that uses it enforces authorization. Enforcing authorization means that the endpoint that provides the service, or the transport, provides mechanisms to withhold information from unauthorized parties. REST and RPC transports can work with tokens (e.g. OAuth 2.0, see IETF RFC 6749 [16]) for authorization; here, the service endpoint is responsible for the enforcement. Message bus transports typically work by using the TLS certificates to enforce authorization; enforcement can be built into the transport mechanism. In order to realize this in an interoperable way, a service can define a list of topics to be used with transports that are topic-based, and to use these topics to scope the access of MEC applications to the actual information.

7.2 Relationship of topics, subscriptions and access rights

In the RESTful MEC service APIs defined as part of ETSI MEC, a client registers interest in particular changes by defining a subscription structure that typically contains at least one criterion against which a notification needs to match in order to be sent to the subscriber for that particular subscription. Multiple criteria can be defined, in which case all criteria need to match. Each criterion defines a particular value, or a set of values.

EXAMPLE 1: Table 7.2-1 provides a sample of the criteria part of a data type that represents subscriptions to notifications about cell changes.

Table 7.2-1: A sample of the criteria part of a data type that represents subscriptions to notifications about cell changes

Attribute name	Data type	Cardinality	Description
filterCriteria	Structure (inlined)	1	List of filtering criteria for the subscription. Any filtering criteria from below, which is included in the request, shall also be included in the response.
>appInsId	String	0..1	Unique identifier for the MEC application instance.
>associateId	array(Structure (inlined))	0..N	
>>type	Enum	1	Numeric value (0 - 255) corresponding to specified type of identifier as following: 0 = reserved 1= UE IPv4 Address 2 = UE IPv6 Address 3 = NATed IP address 4= GTP TEID.
>>value	String	1	Value for the identifier.

In topic-based message buses, subscription is done against *topics*. Each topic is a string that defines the actual event about which the client wishes to be notified. Typically, topics are organized in a hierarchical structure. Also, in such structure, often wildcards are allowed that enable to abbreviate the subscription to a complete topic sub-tree.

EXAMPLE 2: Criteria from example 1 formulated as topic, prefixed by the service name and notification type

rnis/cell_change/{applnsId}/{associateId.type}/{associateId.value}

EXAMPLE 3: Criteria from example 1 formulated as topic, with wildcard

rnis/cell_change/{applnsId}/*

If a particular MEC service foresees binding to a topic based message bus as an alternative transport, it is encouraged to define the list or hierarchy of topics in the specification, in order to improve interoperability. If that MEC service also provides REST-based subscribe-notify, it is encouraged to also define the mapping between the subscription data structures used in the RESTful API and the topic list/topic hierarchy.

In MEC, an important feature is authorization of applications. Authorization also needs to apply to subscriptions, to enable the MEC platform operator to restrict access of MEC applications to privileged information. Each separate access right is expressed by a "*permission identifier*" which identifies this right. Permission identifiers need to be unique within the scope of a particular MEC service. For each access right, the service specification needs to define a string to name that particular right. These strings can then be used throughout the system to identify that particular access right.

For REST-based subscriptions, it is suggested that the set of subscriptions is structured such that the subscription type can be used to scope the authorization (i.e. clients can be authorized for each individual subscription type separately, and one permission identifier maps to one subscription type). If finer or coarser granularity is required, this needs to be expressed in the particular specification by suitably defining the meaning of each permission identifier. For Topic-based subscriptions, each permission identifier is suggested to map to a particular topic, or a whole sub-tree of the topics structure.

The following items are proposed to define permissions:

Permission identifier: A string that identifies the item to which access is granted or denied. It is unique within the scope of a particular MEC service specification.

Display name: A short human-readable string to describe the permission when represented towards human users.

Specification: A specification that defines what the actual permission means. Can be as short as just naming the resource, subscription type or topic, or can also express a condition to define authorization at finer granularity than subscription type. If multiple alternative transports are supported, can contain specifications for more than one transport.

EXAMPLE 4: Tables 7.2-2, 7.2-3 and 7.2-4 provide an example definition of permissions for two transports: REST-based and topic-based message bus. Queries only apply to the REST-based transport.

Table 7.2-2: Definition of permissions

Permission identifier	Display name	Remarks
queries	Queries	REST-based only
bearer_changes	Bearer changes	Subscribe-notify
priv_bearer_changes	Privileged bearer changes	Subscribe-notify

Table 7.2-3: Permission identifiers mapping for transport "REST"

Permission identifier	Specification
queries	Resource: …/rnis/v1/queries
bearer_changes	Resource: …/rnis/v1/subscriptions Subscription type: BearerChangeSubscription with "privileged" flag not set
priv_bearer_changes	Resource: …/rnis/v1/subscriptions Subscription type: BearerChangeSubscription with "privileged" flag set
all	All of the permissions identified by "queries", "bearer_changes" and "priv_bearer_changes".

If OAuth 2.0 is used to authorize access to a REST-based transport, the permission identifiers can be represented as OAuth 2.0 scope values.

Table 7.2-4: Permission identifiers mapping for transport "Topic-based message bus"

Permission identifier	Specification
queries	Not supported
bearer_changes	Topic: /rnis/bearer_changes/nonprivileged/*
priv_bearer_changes	Topic: /rnis/bearer_changes/privileged/*

To define the access rights that an application requests, the permission identifiers which represent the requested access rights are defined in the application descriptor.

7.3 Serializers

As indicated in clause 7.1, different serializers can be used with alternative transports for a particular service. The reason for allowing this choice is that certain serializers make more sense in combination with particular transports than others. For example, RESTful APIs nowadays typically use HTTP/1.1 for transport and JSON for the data payload in textural form. A large number of development tools support this combination. When using message buses or gRPC®, typically high throughput and low latency is a main requirement, which can be better met using a serializer into a binary format. The gRPC framework, for example, uses HTTP/2 for transport and by default Protocol Buffers for binary data serialization (although other data formats such as JSON in binary form can be used for serialization). Specifications of a MEC service can define in annexes the serializer(s) that are intended to be used for the data types defined for the service. The serializer to be used with a transport needs to be signalled over Mp1 when registering a service. More details can be found in ETSI GS MEC 011 [i.6].

7.4 Authorization of access to a service over alternative transports using TLS credentials

A method to authorize access to RESTful MEC service APIs using OAuth 2.0 has been defined in clause 6.16. For alternative transports, as defined in clause 7, using of OAuth might not be possible or supported in all the cases, e.g. for topic-based message buses. For these cases, other mechanisms are used to authorize access to the service. Several alternative transport mechanisms already require using TLS 1.2 [14] or TLS 1.3 [24] to protect the communication channels. TLS credentials can be used to authenticate the endpoints of the protected connection, and to authorize them to access the MEC services delivered using an alternative transport that is secured with TLS.

TLS is designed to provide three essential services: encryption, authentication, and data integrity:

- For encryption, a secure data channel is established between the peers. To set up this channel, information about the cipher suite and encryption keys is exchanged between the peers during the TLS handshake.
- As part of the TLS handshake, the procedure also allows the peers to mutually authenticate themselves based on certificates and chain of trust enabled by Certificate Authorities. In the present document, client access rights are bound to the TLS credentials related to a client identifier, which allows to authorize an authenticated client to access particular MEC services or parts of those.
- Besides this, integrity of the data exchanged can be ensured with the Message Authentication Code algorithm supported by the TLS protocol.

Figure 7.4-1 shows an example how TLS can be used, in case of a topic-based message bus as alternative transport, to both secure the communications between the peers, as well as to authorize the consumption of a MEC service by a MEC application.

The MEC application is identified by a client identifier, which may be derived from attributes such as Distinguished Name (DN) used in the client certificate, or the application name and application provider defined in the application package. In a system that is based on a topic-based message bus as alternative transport, the MEC service is structured into one or more topics to which the consuming application can subscribe. Permissions can be given to subscribe to individual topics. By binding these permissions to a client identifier, the MEC application that is identified by this client identifier can be authorized to consume the corresponding parts of the service. Likewise, a service-producing MEC application can be authorized to send messages to the message bus for certain topics defined for the service.

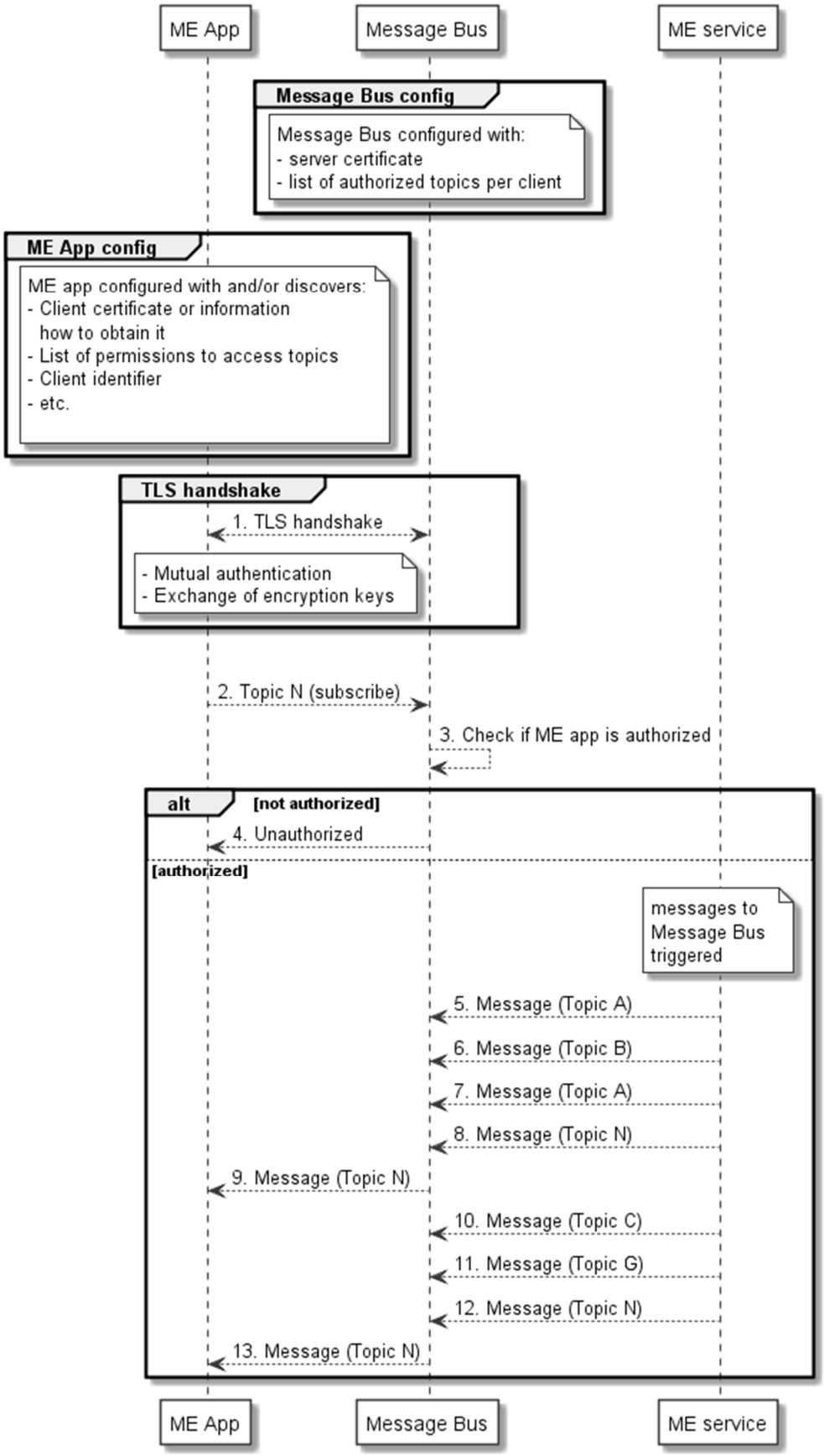


Figure 7.4-1: Using TLS for authorizing subscription to topics when using a topic-based message bus

As depicted in figure 7.4-1, there are preconditions and related procedures to provision the X.509 certificates for the message bus and for the MEC application. These procedures are based on the use of a Public Key Infrastructure (PKI) and they are out of scope of the present document.

The preconditions for MEC applications assume that authorization-related and security-related parameters are configured as part of the runtime configuration data of the application, and/or are discovered by the MEC application over Mp1. These include e.g. TLS version, list of permissions and topics that can be accessed, client identifier such as Distinguished Name or application provider and application name provided in the application package, and the instructions how to obtain the client certificate.

After having obtained the valid certificate to access the specific service offered over the message bus, the MEC application performs the TLS handshake and subscribes to topics offered, as follows:

1) The MEC application and the message bus perform the TLS handshake as defined in the TLS protocol according to the TLS version ([14] for TLS 1.2 or [24] for TLS 1.3), including mutual authentication and encryption key exchange. As a result, the MEC application is authenticated towards the message bus.

2) The MEC application subscribes to topic N with the message bus.

3) The message bus checks whether the authenticated MEC application is authorized to subscribe to topic N, by checking the list of authorized topics that were configured for this MEC application.

4) In case the MEC application is not authorized to subscribe to the topic, an "Unauthorized" response is returned.

Otherwise, in steps 5 through 7 the MEC service sends messages on topics A & B to the message bus. Since the MEC application has not subscribed to those topics, those messages are not forwarded to this application.

8) Message on topic N sent to the message bus by the MEC service.

9) The message bus forwards the message to the subscribed MEC application.

In steps 10 and 11 the MEC service sends messages on topics C & G to the message bus. Since the MEC application has not subscribed to those topics, those messages are not forwarded to this application.

12) Message on topic N is sent to the message bus by the MEC service.

13) The message bus forwards the message to the subscribed MEC application.

Annex A (informative): REST methods

All API operations are based on the HTTP Methods. GET and POST are not allowed to be used to tunnel other operations.

Table A-1 lists basic operations on entities and their mapping to HTTP methods.

Table A-1: Operations and HTTP methods

Operation on entities	Uniform API operation	Description
Read/Query Entity	GET Resource	GET is used to retrieve a representation of a resource.
Create Entity	POST Resource	POST is used to create a new resource as child of a collection resource (see note 1).
Create Entity	PUT Resource	If applicable, PUT can be used to create a new resource directly (see note 1).
Partial Update of an Entity	PATCH Resource	PATCH, if supported, is used to partially update a resource (see note 2).
Complete Update of an Entity	PUT Resource	PUT is used to completely update a resource identified by its resource URI.
Remove an Entity	DELETE Resource	DELETE is used to remove a resource
Execute an Action on an Entity	POST on TASK Resource	POST on a task resource is used to execute a specific task not related to Create/Read/Update/Delete (see note 3).
NOTE 1: It is not advised to mix creation by PUT and creation by POST in the same API. NOTE 2: PATCH needs to be used with care if it is intended to be idempotent. See [i.2] for general principles. The data format is defined by IETF RFC 7396 [5]/IETF RFC 6902 [6] for JSON and IETF RFC 5261 [7] for XML. NOTE 3: A task resource a resource that represents a specific operation that cannot be mapped to a combination of Create/Read/Update/Delete. Task resources are advised to be used with careful consideration.		

Annex B (normative): HTTP response status codes

The tables in this clause list HTTP response codes typically used in the ETSI MEC REST APIs. In addition to the codes listed below, clients need to be prepared to receive any other valid HTTP error response code. A list of all valid HTTP response codes and their specification documents can be obtained from the HTTP status code registry [i.8]. For each status code, an indicative description and a reference to the related specification are given. The use of each status code shall follow the provisions in the referenced specification.

Table B-1 lists the success codes which indicate that the client's request was accepted successfully.

Table B-1: 2xx - Success codes

Status Code	Reference	Description
200	IETF RFC 7231 [1]	OK - used to indicate nonspecific success. The response body usually contains a representation of the resource. If this code is returned, it is not allowed to communicate errors in the response body.
201	IETF RFC 7231 [1]	Created - used to indicate successful resource creation. The return message usually contains a resource representation and always contains a "Location" HTTP header with the created resource's URI.
202	IETF RFC 7231 [1]	Accepted - used to indicate successful start of an asynchronous action.
204	IETF RFC 7231 [1]	No Content - used to indicate success when the response body is intentionally empty.

Table B-2 lists the redirection codes which indicate that the client has to take some additional action in order to complete its request.

Table B-2: 3xx - Redirection codes

Status Code	Reference	Description
301	IETF RFC 7231 [1]	Moved Permanently - used to relocate resources.
302	IETF RFC 7231 [1]	Found - not used.
303	IETF RFC 7231 [1]	See Other - used to refer the client to a different URI.
304	IETF RFC 7231 [1]	Not Modified - used to preserve bandwidth.
307	IETF RFC 7231 [1]	Temporary Redirect - used to tell clients to resubmit the request to another URI.

Table B-3 lists the client error codes which indicate an error related to the client's request. Further provisions on the use of the "ProblemDetails" structure in the payload body of common error responses are defined in annex E.

Table B-3: 4xx - Client error codes

Status Code	Reference	Description
400	IETF RFC 7231 [1]	Bad Request - used to indicate a syntactically incorrect request message. Also used to indicate nonspecific failure caused by the input data, including "catch-all" errors.
401	IETF RFC 7231 [1]	Unauthorized - used when the client did not submit credentials.
403	IETF RFC 7231 [1]	Forbidden - used to forbid access regardless of authorization state.
404	IETF RFC 7231 [1]	Not Found - used when a client provided a URI that cannot be mapped to a valid resource URI.
405	IETF RFC 7231 [1]	Method Not Allowed - used when the HTTP method is not supported for that particular resource. Typically, the response includes a list of supported methods.
406	IETF RFC 7231 [1]	Not Acceptable - used to indicate that the server cannot provide the any of the content formats supported by the client.
409	IETF RFC 7231 [1]	Conflict - used when attempting to create a resource that already exists.
410	IETF RFC 7231 [1]	Gone - used when a resource is accessed that has existed previously, but does not exist any longer (if that information is available).
412	IETF RFC 7232 [2]	Precondition failed - used when a condition has failed during conditional requests, e.g. when using ETags to avoid write conflicts when using PUT.
413	IETF RFC 7231 [1]	Payload Too Large - The server is refusing to process a request because the request payload is larger than the server is willing or able to process.
414	IETF RFC 7231 [1]	URI Too Long - The server is refusing to process a request because the request URI is longer than the server is willing or able to process.
415	IETF RFC 7231 [1]	Unsupported Media Type - used to indicate that the server or the client does not support the content type of the payload body.
422	IETF RFC 4918 [21]	Unprocessable Entity - used to indicate that 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).
429	IETF RFC 6585 [8]	Too many requests - used when a rate limiter has triggered.

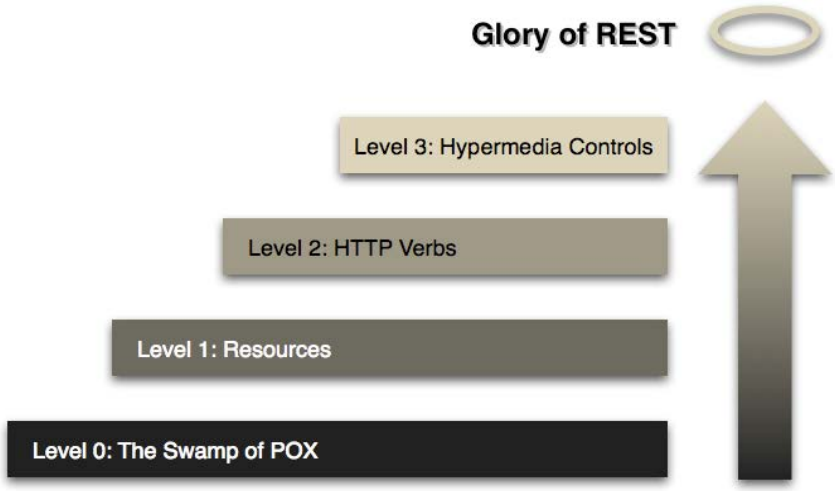
Table B-4 lists the server error codes which indicate that the server is aware of an error caused at the server side. Further provisions on the use of the "ProblemDetails" structure in the payload body of common error responses are defined in annex E.

Table B-4: 5xx - Server error codes

Status Code	Reference	Description
500	IETF RFC 7231 [1]	Internal Server Error - Server is unable to process the request. Retrying the same request later might eventually succeed. Also used to indicate nonspecific failure caused by server processes, including "catch-all" errors.
503	IETF RFC 7231 [1]	Service Unavailable - The server is unable to process the request due to internal overload. Retrying the same request later might eventually succeed.
504	IETF RFC 7231 [1]	Gateway Timeout - The server did not receive a timely response from an upstream server it needed to access in order to complete the request. Retrying the same request later might eventually succeed.

Annex C (informative): Richardson maturity model of REST APIs

The Richardson maturity model [i.3] breaks down the principal elements of a REST approach into three levels above the non-REST level 0.



NOTE: The figure is © by Martin Fowler and has been reproduced with permission from [i.3].

Figure C-1: Step towards REST

Level 0 - the swamp of POX: it is the starting point, using HTTP as a transport system for remote interactions, but without using any web mechanisms. Essentially it is to use HTTP as a tunnelling mechanism for remote interaction.

Level 1 - resources: the first step is to introduce resources. Instead of sending all requests to a singular service endpoint, they are now addressed to individual resources.

Level 2 - HTTP methods: HTTP methods (e.g. POST, GET) may be used for interactions in level 0 and 1, but as tunnelling mechanisms only. Level 2 moves away from this, using the HTTP methods as closely as possible to how they are used in HTTP itself.

Level 3 - hypermedia controls: this is often referred to HATEOAS (Hypermedia As The Engine Of Application State). It addresses the question of how to get from a list of resources to knowing what to do.

There are several advantages by adopting hypermedia controls:

- it allows the server to change its URI scheme without breaking clients;
- it helps client developers explore the protocol.

The links give client developers a hint as to what may be possible next, e.g. a starting point as to what to think about for more information and to look for a similar URI in the protocol documentation:

- it allows the server to advertise new capabilities by putting new links in the responses.

If the client developers are implementing handling for unknown links, these links can be a trigger for further exploration.

Annex D (informative): RESTful MEC service API template

This annex is a template that provides text blocks for normative specification text to be copied into other specifications. Therefore, even though this annex is informative, some of the text blocks contain modal verbs that have special meaning according to clause 3.2 of the ETSI Drafting Rules. A recommendation for the structuring an API definitions into main clauses is also provided.

<N> Sequence diagrams (informative)

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

<N>.1 Introduction

This clause …

<N>.2 <Procedure 1>

<Template note: Add introductory text>

This clause …

<Template note: Add flow diagram using PlantUML tool (see MEC(16)000115r1-Plantuml tutorial). Do not forget caption.)

<Template note: Start of Example>

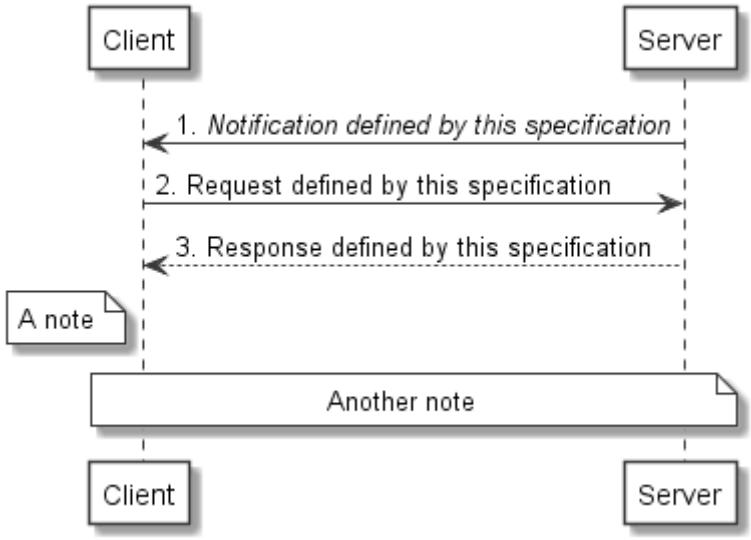


Figure <N>.2-1: Flow of <Procedure 1>

<Template note: End of Example>

<Template note: Add description of the steps>

<Procedure 1>, as illustrated in figure <N>.2-1, consists of the following steps:

<Template note: Start of Example>

1. *The server sends a notification to the client.*
2. *The client sends a request to the server.*
3. *The server returns a response.*

<Template note: End of Example>

<N+1> Data model (normative)

<N+1>.1 Introduction

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

<N+1>.2 Resource data types

<N+1>.2.1 Introduction

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

<N+1>.2.2 Type: <TypeName1>

<Template note: TypeName1 in UpperCamel>

This type represents…

<Template note: Provisions how to document a structured data type and an example are provided in clause <N+1>.4.2 of ETSI GS MEC 009>

Table <N>.2.2-1: Definition of type <TypeName1>

Attribute name	Data type	Cardinality	Description

<N+1>.3 Subscription data types

<N+1>.3.1 Introduction

This clause defines data structures for subscriptions.

<N+1>.3.2 Type: <TypeName2>

<Template note: Same structure as in clause <N+1>.2.2>

<N+1>.4 Notification data types

<N+1>.4.1 Introduction

This clause defines data structures for notifications.

<N+1>.4.2 Type: <TypeName3>

<Template note: Same structure as in clause <N+1>.2.2>

<N+1>.5 Referenced structured data types

<N+1>.5.1 Introduction

This clause defines data structures that can be referenced from data structures defined in the previous clauses, but can neither be resource representations nor notifications.

<N+1>.5.2 Type: <TypeName4>

<Template note: Same structure as in clause <N+1>.2.2>

<N+1>.6 Referenced simple data types and enumerations

<N+1>.6.1 Introduction

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

<N+1>.6.2 Simple data types

The simple data types defined in table <N+1>.6.2-1 shall be supported.

Table <N+1>.6.2-1: Simple data types

Type name	Description

<N+1>.6.3 Enumeration: <EnumType1>

<Template note: If the intent is to map the enum values to strings (often used in JSON), or only to define the valid values (with the intent to define their mappings to integers in a different place, specific to certain serializers), the format in this clause is suggested to be used.>

The enumeration <EnumType1> represents <something>. It shall comply with the provisions defined in table <N+1>.6.3-1.

Table <N+1>.6.3-1: Enumeration <EnumType1>

Enumeration value	Description
A_VALUE	The description of this enum value
ANOTHER_VALUE	The description of this other enum value

<N+1>.6.4 Enumeration: <EnumType2>

<Template note: If the intent is to map the enum values to integers independent of the serializer, the format in this clause is suggested to be used. "<int>" in the table below to be replaced by an actual integer value.>

The enumeration <EnumType2> represents <something>. It shall comply with the provisions defined in table <N+1>.6.4-1.

Table <N+1>.6.4-1: Enumeration <EnumType2>

Enumeration value	Description
<int> = A_VALUE	The description of this enum value
<int> = ANOTHER_VALUE	The description of this other enum value

<N+2> RESTful API definition (normative)

<N+2>.1 Introduction

This clause defines the resources and operations of the <Name of the API here> API.

<N+2>.2 Global definitions and resource structure

All resource URIs of this API shall have the following root:

- **{apiRoot}/{apiName}/{apiVersion}/**

"ApiRoot" and "apiName" are discovered using the service registry. It includes the scheme ("https"), host and optional port, and an optional prefix string.

The API shall support HTTP over TLS (also known as HTTPS) Using TLS version 1.2 (as defined by IETF RFC 5246 [14]). TLS 1.3 (including the new specific requirements for TLS 1.2 implementations) defined by IETF RFC 8446 [24] should be supported. HTTP without TLS shall not be used. Versions of TLS earlier than 1.2 shall neither be supported nor used.

TLS implementations should meet or exceed the security algorithm, key length and strength requirements specified in clause 6.2.3 (if TLS version 1.2 as defined by IETF RFC 5246 [14] is used) or clause 6.2.2 (if TLS version 1.3 as defined by IETF RFC 8446 [24] is used) of ETSI TS 133 210 [27] (3GPP Release 16 or later).

The "apiVersion" shall be set to "v1" for the current version of the specification. All resource URIs in the clauses below are defined relative to the above root URI.

<Template note: Adapt the sentence below to normatively state which data types are appropriate for the particular API>

The content formats XML and JSON shall be supported.

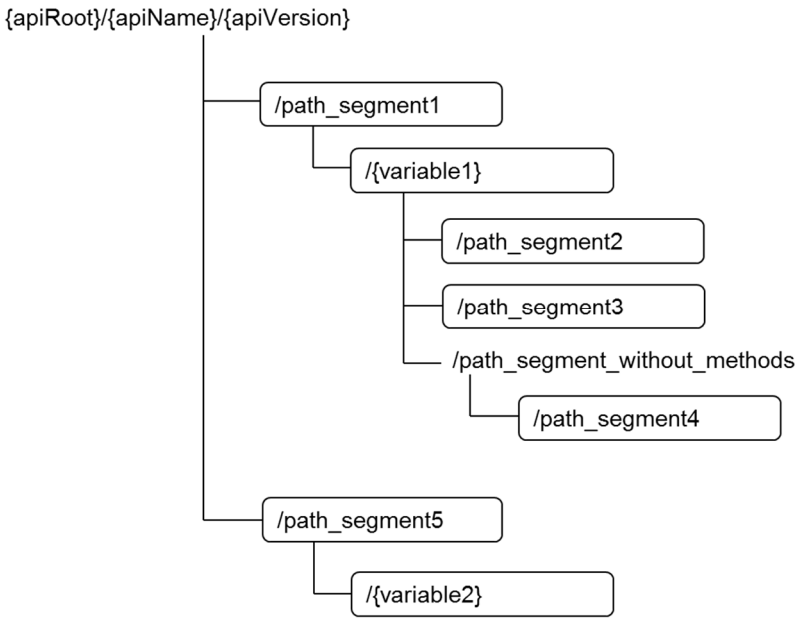
<Template note: Certain APIs may also want to introduce their own content type and register with IANA >

The JSON format is signalled by the content type "application/json" and the XML format is signalled by the content type "application/xml".

Figure <N+2>.2-1 illustrates the resource URI structure of this API. Table <N+2>.2-1 provides an overview of the resources defined by the present document, and the applicable HTTP methods.

<Template note: Start of Example>

<Template note: a filled node represents a sub-URI that has at least one supported operation associated. All node names are examples only>



<Template note: End of Example>

Figure <N+2>.2-1: Resource URI structure of the <xyz> API

<Template note: Overview table of resources and operations>

Table <N+2>.2-1: Resources and methods overview

Resource name	Resource URI	HTTP method	Meaning
<Resource name>	<relative URI below root>	GET	<Operation executed by GET>
		PUT	<Operation executed by PUT>
		PATCH	<Operation executed by PATCH>
		POST	<Operation executed by POST>
		DELETE	<Operation executed by DELETE>

<Template note: Start of Example>

Table <N+2>.2-2: Resources and methods overview

Resource name	Resource URI	HTTP method	Meaning
All foobar sessions	/{userId}/sessions	GET	Retrieve a list of foobar sessions
		POST	Create a new foobar session
Individual foobar session	/{userId}/sessions/{sessionId}	GET	Retrieve a foobar session
		DELETE	Terminate a foobar session

<Template note: End of Example>

<Template note: Repeat the following level 2 clause as often as needed, once per resource>

<N+2>.3 Resource: <Meaning>

<N+2>.3.1 Description

This resource is used to …/ This resource represents …/ <or similar text as applicable>

<N+2>.3.2 Resource definition

Resource URI: **{apiRoot}/{apiName}/v1/<name>**

This resource shall support the resource URI variables defined in table *<N+2>*.3.2-1.

Table <N+2>.3.2-1: Resource URI variables for resource "<Meaning>"

Name	Definition
apiRoot	See clause 6.2
apiName	See clause 6.2
<name>	<definition>

<N+2>.3.3 Resource methods

<N+2>.3.3.1 GET

The <GET> method … <Meaning(s) of the operation in API space> or <Not supported>

<Template note: Start Example>

The GET method retrieves information about a fooBar object.

<Template note: End Example>

This method shall support the URI query parameters, request and response data structures, and response codes, as specified in the tables *<N+2>*.3.3.1-1 and *<N+2>*.3.3.1-2.

Table <N+2>.3.3.1-1: URI query parameters supported by the <GET> method on this resource

Name	Data type	Cardinality	Remarks
<name> or n/a	<type> or <leave empty>	0..1 or 1 or 0..N <leave empty>	<only if applicable>

Table <N+2>.3.3.3.1-2: Data structures supported by the <GET> request/response on this resource

	Data type	Cardinality	Response codes	Remarks
Request body	<type> or n/a	<1 (i.e. object)> or <0..N, 1..N, m..n (i.e. array/map)> or <leave empty>	<only if applicable>	
Response body	<type> or n/a	<1 (i.e. object)> or <0..N, 1..N, m..n (i.e. array/map)> or <leave empty>	<list applicable codes with name from IETF RFC 7231 [1], etc.>	<Meaning of the success case> or <Meaning of the error case with additional statement regarding error handling>

<Template note: If a statement in the "Remarks" column repeats too often, it can be represented as a NOTE at the bottom of the table>

<Template note: Start of Example>

Table <N+2>.3.3.1-3: URI query parameters supported by the GET method on this resource

Name	Data type	Cardinality	Remarks
foo_bar	String	0..1	The foo bar

Table <N+2>.3.3.1-4: Data structures supported by the GET request/response on this resource

Request body	Data type	Cardinality	Remarks	
	n/a			
Response body	Data type	Cardinality	Response codes	Remarks
	FooBarInstance	1	201 Created	The foobar session was created successfully.
	ProblemDetails	0..1	400 Bad Request	Incorrect parameters were passed to the request. In the returned ProblemDetails structure, the "detail" attribute should convey more information about the error.
	ProblemDetails	0..1	404 Not Found	The resource URI was incorrect. In the returned ProblemDetails structure, the "detail" attribute should convey more information about the error.
	ProblemDetails	1	403 Forbidden	The operation is not allowed given the current status of the resource. More information shall be provided in the "detail" attribute of the "ProblemDetails" structure.

<Template note: Note that the cardinality of the "ProblemDetails" structure may differ between different error codes. Some may require the provision of application specific details, for some it may be merely optional or recommended. >

<Template note: End of Example>

<Template note: If applicable, add a statement on HTTP headers specific to the operation>

On success, the HTTP response shall/should/may include a <name> HTTP header that…

<Template note: Start of Example>

On success, the HTTP response shall include a "Location" HTTP header that contains the URI of the newly-created resource.

<Template note: End of Example>

<Template note: If necessary, describe error handling in text>

Error handling: text text text

<N+2>.3.3.2 PUT

<same structure as for GET>

<N+2>.3.3.3 PATCH

<same structure as for GET>

<Template note: The data type in the request body should be defined carefully in a particular format, refer to pattern 6.9 for detail>

<N+2>.3.3.4 POST

<same structure as for GET>

<N+2>.3.3.5 DELETE

<same structure as for GET>

Annex E (normative): Error reporting

E.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 ETSI MEC REST APIs.

E.2 General mechanism

When an error occurs that prevents the server 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 [1], IETF RFC 7232 [2], IETF RFC 7233 [22] and IETF RFC 7235 [23], as well as by IETF RFC 6585 [8] for normative specifications of the most commonly uses status codes, and refer to the HTTP status code registry [i.8] for a list of all valid HTTP response codes and their specification documents.

The response body of an error response should contain a representation of a "ProblemDetails" data structure according to IETF RFC 7807 [15] that provides additional details of the error. Provisions for the use of the "ProblemDetails" data type in the ETSI MEC REST APIs are defined in clause 6.15.3.

E.3 Common error situations

The following common error situations are applicable on all REST resources and related HTTP methods defined in the ETSI MEC REST API specifications 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 6.15.3. A list of all valid HTTP response codes and their specification documents can be obtained from the HTTP status code registry [i.8].

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 server shall respond with this response code. More details are defined in IETF RFC 7231 [1]. 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 server is adversely affected, and the server does not support paging for the affected resource, it shall respond with this response code. Clause 6.20.5 specifies provisions for the "ProblemDetails" structure provided in the response body.

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 server shall respond with this response code. The "ProblemDetails" structure shall be provided and shall include in the "detail" attribute more information about the source of the problem.

NOTE 1: It is by design to represent these multiple application error situations with the same HTTP error response code 400.

400 Bad Request: If the request contains a malformed access token, the server 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 [17]. The ProblemDetails structure may be provided.

NOTE 2: 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.

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 server 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 [17] and IETF RFC 7235 [23]. The ProblemDetails structure may be provided.

403 Forbidden: If the client is not allowed to perform a particular request to a particular resource, the server shall respond with this response code. More details are defined in IETF RFC 7231 [1]. 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 server 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 [1]. 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 3: 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 server shall respond with this response code. The "ProblemDetails" structure may be provided.

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

413 Payload Too Large: If the payload body of a request is larger than the amount of data the server is willing or able to process, it shall respond with this response code, following the provisions in IETF RFC 7231 [1] for the use of the "Retry-After" HTTP header and for closing the connection. The "ProblemDetails" structure may be provided.

414 URI Too Long: If the request URI of a request is longer than the server 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 [1]. The "ProblemDetails" structure may be provided.

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 server shall respond with this response code. More details are defined in IETF RFC 4918 [21]. The "ProblemDetails" structure shall be provided and should include in the "detail" attribute more information about the source of the problem.

NOTE 4: This error response code is only applicable for methods that have a request body.

429 Too Many Requests: If the client has sent too many requests in a defined period of time and the server is able to detect that condition ("rate limiting"), the server shall respond with this response code, following the provisions in IETF RFC 6585 [8] 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 5: The period of time and allowed number of requests are configured within the server 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 server shall respond with this response code. More details are defined in IETF RFC 7231 [1]. 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 server 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 [1] for the use of the "Retry-After" HTTP header and for the alternative to refuse the connection. The "ProblemDetails" structure may be provided.

504 Gateway Timeout: If the server encounters a timeout while waiting for a response from an upstream server (i.e. a server that the server communicates with when fulfilling a request), it should respond with this response code. More details are defined in IETF RFC 7231 [1]. The "ProblemDetails" structure may be provided.

Annex F (informative): Change History

Date	Version	Information about changes
December 2020	V2.2.2	Baseline draft created by rapporteur based on previous published version
January 2021	V2.2.3	Contribution implemented • MEC(21)000001r1_MEC009_notification_enhancements
February 2021	V2.2.4	Contribution implemented • MEC(21)000072r1_MEC009_websocket_notifications_addressing_EN_and_ comments
March 2021	V 2.2.5	Editorials • Clean-up done by ***editHelp!*** E-mail: edithelp@etsi.org
April 2021	V 2.2.6	Contributions implemented • MEC(21)000136_MEC009_fix_spelling_of_Uri_data_type • MEC(21)000102r3_MEC009_TLS_version_bugfix • MEC(21)000173r1_MEC009_schema_fixes
April 2021	V 3.0.0	At MEC#26 it was agreed that MEC009 is ready to go to final approval process, starting with the RC for review. MEC009 will be published as v3.1.1, final draft is uploaded as v3.0.0 and is similar to v2.2.6
May 2021	V 3.0.1	MEC(21)000218_MEC009_resolving_review_comment: • Align terminology and layout used in the figures in clause 6.12a with the rest of the document (REST server/REST client instead of MEC platform/MEC app; client on the left and server on the right)

History

Document history		
V1.1.1	July 2017	Publication
V2.1.1	January 2019	Publication
V2.2.1	October 2020	Publication
V3.1.1	June 2021	Publication