ETSI EG 203 647 V1.1.1 (2020-11)

ETSI GUIDE

Methods for Testing and Specification (MTS); Methodology for RESTful APIs specifications and testing

Reference

DEG/MTS-203647

Keywords

API, methodology, testing

ETSI

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

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

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

Contents

Intellectual Property Rights

Foreword

This ETSI Guide (EG) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Modal verbs terminology

In the present document "**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.

Executive Summary

The present document offers a report of standardization activities for telecommunication interfaces and application programming interfaces based on the REpresentational State Transfer paradigm (RESTful APIs).

The guide collects conventions, methodology and design patterns from ETSI groups and from the industry and proposes consolidated guidelines to serve the complete lifecycle of standardization, from design to validation.

Introduction

More and more telecommunication and digital interfaces are being implemented as software-based solutions.
A well-known and largely adopted design methodology is taking place across several standardization activities: using the REpresentational State Transfer (REST) paradigm and resource-oriented protocols (e.g. HTTP(S), CoAP) or other possibly applicable protocols (MQTT, AMQP).

This phenomenon is becoming common practice in ETSI Technical Bodies (TBs) and Industry Specification Groups (ISGs) as well as in ETSI's Partnership Projects standardization activities, across several technologies, often quite different in scope and user community.

As adoption of standardizing RESTful APIs rises, it is becoming clear that specification of "RESTful APIs" needs to be:

- **Fast**, as the interfaces are simpler than other approaches and tend to have a shorter lifespan.

- **Automatable**, given the high number of conventions in the design of an API, parts of the specification, implementation and testing process are well suited to be automated.
- **Developer friendly**, since developers need support in the discovery and implementation of the interfaces by using tools and methodologies more closely aligned with software development.

In this regard, the present Guide for RESTful API specification and testing intends to support:

- **Consolidation of efforts** among different standardization groups and activities, who would be able to leverage from others' experience.
- **Delivery time** of specifications to be spent on the design of the application level features, more than re-assessing the principles and details at a transport protocol level.
- **Standards quality** to meet the excellence expected in the whole lifecycle of standardization, such as design, specification, testability and interoperability validation.

Several TBs and ISGs have already specified RESTful APIs and documented their conventions and processes in group specific guidelines. Further initiatives will be carried out during the upcoming years by the same groups as well as new ones, therefore it is strategic to align and consolidate the standardization efforts among ETSI membership.

The present document is structured as follows:

- clause 4 introduces the main concepts and terminology for the RESTful approach, then presents recommendations for RESTful API specifications development, with the introduction of a *code-first* approach and discussion of the foreseen benefits of its application;
- clause 5 presents recommendation and methodology for development of test specifications for RESTful APIs;
- clause 6 collects best practices and references to the available tools to manipulate and present the code needed artefacts;
- clause 7 contains a collection of examples on the expected outcomes of the different parts of the presented methodology;
- clause 8 reports on the outcomes from the analysis of the base documents - from ETSI groups or from other organizations - for the preparation of the present work and the results of a survey conducted among ETSI delegates on REST APIs adoption.

1 Scope

The scope of the present document is to present a methodology for specification and testing of RESTful APIs, i.e. telecommunication interfaces based on the Representational State Transfer paradigm, suitable for application in the standardization context.

In particular, the present guide is meant to serve ETSI membership and groups in the effort to unify and consolidate the approaches and practices in current and future standardization activities at ETSI and its Partnership Projects.

The Guide collects the best practices from standardization, industry and research in order to provide a modern and future-proofed approach to the subject.

The intended audience is primarily standardization groups at ETSI, but the guide may also serve as reference for users and vendors in industry, with a special focus in Open Source communities.

The Guide recommendations on conventions, methodologies, design-patterns and architectural choices to be used in standardization of RESTful APIs, specification and execution of standardized conformance and interoperability test suites.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

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] "A Guide to Writing World Class Standards".

NOTE: Available at https://portal.etsi.org/Portals/0/TBpages/edithelp/Docs/AGuideToWritingWorldClassStandards.pdf?ver=2014-05-19-124137-453.

[i.2] Recommendation ITU-T I.130: "Method for the characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN".

[i.3] IETF RFC 5023: "The Atom Publishing Protocol".

[i.4] ETSI EG 203 130: "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Methodology for standardized test specification development".

[i.5] IETF RFC 7231: "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content".

[i.6] IETF RFC 8259: "The JavaScript Object Notation (JSON) Data Interchange Format".

[i.7] "The State of API 2019 survey", SmartBear.

[i.8] ETSI EG 201 015: "Methods for Testing and Specification (MTS); Standards engineering process; A Handbook of validation methods".

[i.9] ETSI EG 201 058 (V1.2.4): "Methods for Testing and Specification (MTS); Implementation Conformance Statement (ICS); proforma style guide".

[i.10] Repository of attachments for ETSI EG 203 647 on ETSI Forge.

NOTE: Available at https://forge.etsi.org/rep/mts/eg-203647-restful-api-guide.

[i.11] ETSI Forge Platform.

NOTE: Available at https://forge.etsi.org.

[i.12] ETSI Labs Platform.

NOTE: Available at https://labs.etsi.org.

[i.13] TDL Open Source Project.

NOTE: Available at https://top.etsi.org.

[i.14] "YAML Ain't Markup Language (YAML™) Version 1.2" specification.

NOTE: Available at https://yaml.org/spec/1.2/spec.html.

[i.15] OpenAPI™ specification, Version 3.0.3.

NOTE: Available at https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md.

[i.16] JSON Schema definition.

NOTE: Available at https://json-schema.org/.

[i.17] IETF RFC 3986: "Uniform Resource Identifier (URI): Generic Syntax".

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

[i.18] IETF RFC 2388: "Returning Values from Forms: multipart/form-data".

NOTE: Available at https://tools.ietf.org/html/rfc2388.

[i.19] IETF RFC 1738: "Uniform Resource Locators (URL)".

NOTE: Available at https://tools.ietf.org/html/rfc1738.

[i.20] ETSI Drafting Rules.

NOTE: Available at https://portal.etsi.org/Services/editHelp/How-to-start/ETSI-Drafting-Rules.

[i.21] ETSI TS 129 501 (V15.3.0): "5G; 5G System; Principles and Guidelines for Services Definition; Stage 3 (3GPP TS 29.501 version 15.3.0 Release 15)".

[i.22] ETSI GS NFV-SOL 015 (V1.1.1): "Network Functions Virtualisation (NFV); Protocols and Data Models; Specification of Patterns and Conventions for RESTful NFV-MANO APIs".

[i.23] ETSI GS MEC 009 (V2.1.1): "Multi-access Edge Computing (MEC); General principles for MEC Service APIs".

[i.24] ETSI GR MEC-DEC 025 (V2.1.1): "Multi-access Edge Computing (MEC); MEC Testing Framework".

[i.25] ETSI GS NFV-TST 002: "Network Functions Virtualisation (NFV); Testing Methodology; Report on NFV Interoperability Testing Methodology".

[i.26] ETSI EG 202 568: "Methods for Testing and Specification (MTS); Internet Protocol Testing (IPT); Testing: Methodology and Framework".

[i.27] ETSI ES 203 119-4: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)".

[i.28] ETSI ES 203 119-1: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics".

[i.29] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[i.30] ETSI ES 203 119-2: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax".

[i.31] ETSI ES 203 119-6: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3".

[i.32] ETSI ES 201 873-11: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 11: Using JSON with TTCN-3".

[i.33] ETSI GR NFV-TST 007: "Network Functions Virtualisation (NFV) Release 2; Testing; Guidelines on Interoperability Testing for MANO".

[i.34] ETSI TS 118 115 (V2.0.0): "oneM2M; Testing Framework (oneM2M TS-0015 version 2.0.0 Release 2)".

[i.35] oneM2M TS-0018 (V3.2.0): "Test Suite Structure & Test Purposes".

[i.36] OpenAPI™ Tools website.

NOTE: Available at https://openapi.tools/.

[i.37] 3GPP validation tools repository.

NOTE: Available at https://forge.3gpp.org/rep/tools/3gpp-scripts.

[i.38] MEC API validation script example.

NOTE: Available at https://forge.etsi.org/rep/mec/gs011-app-enablement-api/blob/v2.1.1/.jenkins.sh.

[i.39] NFV API validation script example.

NOTE: Available at https://forge.etsi.org/rep/nfv/SOL002-SOL003/blob/v2.7.1/.jenkins.sh.

[i.40] RapiPDF project.

NOTE: Available at https://mrin9.github.io/RapiPdf/.

[i.41] REST API Design Rulebook by Mark Massé.

[i.42] ETSI GS NFV-SOL 013 (V2.7.1): "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; Specification of common aspects for RESTful NFV MANO APIs".

[i.43] IETF RFC 7807: "Problem Details for HTTP APIs".

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

[i.44] IETF RFC 5246: "The Transport Layer Security (TLS) Protocol Version 1.2".

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

[i.45] IETF RFC 7519: "JSON Web Token (JWT)".

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

[i.46] ETSI TErms and Definitions Database Interactive (TEDDI).

NOTE: Available at https://webapp.etsi.org/Teddi/.

[i.47] IETF RFC 7396: "JSON Merge Patch".

[i.48] IETF RFC 6902: "JavaScript Object Notation (JSON) Patch".

[i.49] IETF RFC 5261: "An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors".

[i.50] IETF RFC 8288: "Web Linking".

[i.51] IETF RFC 7232: "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests".

[i.52] Git website.

NOTE: Available at https://git-scm.com.

[i.53] ETSI GS MEC-DEC 032-1: "Multi-access Edge Computing (MEC); API Conformance Test Specification; Part 1: Test Requirements and Implementation Conformance Statement (ICS)".

[i.54] ETSI GS CIM 009 (V1.2.1): "Context Information Management (CIM); NGSI-LD API".

[i.55] ETSI GS QKD 014 (V1.1.1): "Quantum Key Distribution (QKD); Protocol and data format of REST-based key delivery API".

[i.56] ETSI GS NFV-TST 010 (V2.4.1): "Network Function Virtualisation (NFV) Release 2; Testing; API Conformance Testing Specification".

[i.57] TMF630 API Design Guidelines 4.0.1.

NOTE: Available at https://www.tmforum.org/resources/how-to-guide/tmf630-api-design-guidelines-4-0/.

[i.58] ETSI GS NFV-SOL 002: "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Ve-Vnfm Reference Point".

[i.59] ETSI GS NFV-SOL 003: "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Or-Vnfm Reference Point".

[i.60] ETSI GS NFV-SOL 005: "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Os-Ma-nfvo Reference Point".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

Application Programming Interface (API): interface implemented by a software program to be able to interact with other software programs

ATOM: Atom Publishing Protocol

NOTE: For more information see IETF RFC 5023 [i.3].

collections: set of resources

Hypertext Transfer Protocol (HTTP): application level protocol, on layer 7 of the ISO/OSI model

OpenAPI™ Specification (OAS™): standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection

representation: concrete entity, which encodes a resource in e.g. HTML, JSON or XML

NOTE: A resource may be available in multiple representation, such as a JSON message and as an XML message.

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.

SDO: Standards Developing or Standards Setting Organization

Uniform Resource Identifier (URI): address of the resource and identification of the resource

3.2 Symbols

Void.

3.3 Abbreviations

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

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ATS	Abstract Test Suite
BWC	BackWard Compatible
CCI	Co-Channel Interference
CIM	Cross-cutting Information Management
CR	Change Request
CRUD	Create, Read, Update, Delete
CTC	Change Type Code
CTK	Conformance ToolKit
EDM	Entity Data Model
EHR	Electronic Health Record
EM	Element Manager
ETag	Entity Tag
ETSI	European Telecommunications Standards Institute
GS	Group Specification
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HTTP Secure
ICS	Implementation Conformance Statement
IETF	Internet Engineering Task Force
IFA	InterFaces and Architecture
IFS	Interoperable Function Statement

NOTE: The acronym IFS may also refer to Interoperable Feature Statement, Implementable Functions Statement and other similar terminology, all referring to the identification of a communication behaviour which has relevance for successful interoperability among communicating entities. The list of usages of IFS in different ETSI specification may be retrieved using the TEDDI tool at the ETSI Portal [i.46].

ISG	Industry Specification Group
IT	Information Technology
ITU-T	International Telecommunication Union - Telecommunication standardization sector
IUT	Implementation Under Test
JSON	JavaScript Object Notation
JWE	JSON Web Encryption
JWS	JSON Web Signature
JWT	JSON Web Token
MANO	Manager and Orchestrator
MEC	Multi-access Edge Computing
MQTT	Message Queuing Telemetry Transport
MSC	Message Sequence Chart

MTS	Methods for Testing and Specification
NBWC	Non-BackWard Compatible
NFV	Network Functions Virtualisation
OAS	OpenAPI™ Specification
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
PICS	Protocol Implementation Conformance Statement
QKD	Quantum Key Distribution
RAML	RESTful API Modelling Language
REST	REpresentational State Transfer
RFC	Request For Comments
RPC	Remote Procedure Call
SBI	South Bound Interface
SDO	Standard Development Organization
SOL	SOLutions
STF	Specialist Task Force
SUT	System Under Test
TC	Technical Committee
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol over the Internet Protocol
TD	Test Description
TDL	Test Description Language
TDL-TO	Test Description Language - Test Objective extension
TDL-GR	Test Description Language - Graphical notation
TLS	Transport Layer Security
TM	Tele Management
TOP	TDL Open Source Project
TP	Test Purpose
TSS	Test Suite Structure
TTCN	Test and Test Control Notation
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Unified Resource Locator
VCS	Version Control System
WADL	Web Application Description Language
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language
ZSM	Zero-touch Service Management

4 Specification methodologies for RESTful APIs

4.1 RESTful APIs specification in a staged standardization approach

Standardization best practices for telecommunications recommend that a staged approach is taken in the definition of communications systems. The methodology recommended in "A Guide to Writing World Class Standards" [i.1], page 31, builds upon the Recommendation ITU-T I.130 [i.2] and indicates three stages to design telecommunication standards:

- Stage 1: Specify objectives from the user perspective.
- Stage 2: Develop a functional model to meet those objectives.
- Stage 3: Develop a specification of the detailed implementation requirements.

These three steps may be refined as follows:

- Stage 1: Specification of high-level user requirements on the technology, i.e. the expectations for the communications system to meet.

- Stage 2: Develop architectures, identify atomic components and reference points interconnecting them and within the reference points identify the interfaces and information models required.

- Stage 3: Specify implementation level requirements for the interfaces identified, i.e. the protocols, data models and serialization techniques expected to be seen at a "wire" level.

The specification of RESTful APIs plays its role at stage 3: given an interface between two components in the system architecture, a RESTful API provides the implementation details for the communication between these two entities, which are then identified as the API Producer (or Server) and the API Consumer (or Client).

In this respect, the RESTful approach lets the designer of the standard focus on the entities and data exchanged or manipulated, while providing a framework where (typically) the underlying protocols and serialization techniques are already specified elsewhere.

As an example, the HTTP protocol over TCP/IP and XML serialization may be selected (among many others). Once these choices are made, the designer of the API needs only to focus on the entities manipulated by the interface.

A fourth stage in this process may be identified: the development of testing specifications, for interoperability or for conformance. As for any communication technology, sound test specifications are required to validate the standards and to certify the implementations. Given the specific characteristics of RESTful APIs, the generic ETSI test development methodology will be tailored and documented in subsequent clauses of the present document.

4.2 Introduction on RESTful interfaces

4.2.1 Introduction

In computer communications, the term REST, coined by Roy Fielding in the year 2000, indicates the Representational State Transfer architectural style, defining a set of constraints and agreements based on the concept of Resource Representations. The REST approach does not enforce rules for implementations at a lower level, rather, it draws high-level design guidelines for interactions among different communicating entities.

An Application Programming Interface (API) is a set of libraries or specifications which allows interaction with an external artefact or agent from a third party. APIs which comply with the REST constraints are said to be *RESTful* and refer to the description of a communication interface that allows interacting with a system based on the REST architecture style.

Before introducing a methodology for specification of RESTful APIs in standardization, in the next clause the main principles of REST are presented, to introduce the required terminology and set the background.

4.2.2 Main Principles of the RESTful paradigm

It is recommended for the implementation of APIs to be technology or protocol independent. RESTful APIs take all aspects of HTTP/1.1 [i.5] including its request methods, response codes, and HTTP headers. A RESTful API specification comprises of the following information:

- Purpose of the API;

- URIs of resources;

- HTTP methods for a given resource [i.5];

- Supported representations (e.g. JSON, see [i.6]);

- Request body schema(s) (where applicable);

- Response body schema(s) (where applicable);

- HTTP response status codes.

To abide by certain principles, the use of OpenAPI™ specifications is recommended to design the APIs first.

REST defines a number of constraints for the API design. Many of the REST constraints are actually HTTP constraints, and REST leverages these HTTP constraints for the design and specification of APIs. The REST style ensures that the APIs use HTTP correctly. These constraints limit the freedom of design. REST imposes the following constraints:

- design resources (nouns), not the methods or operations (verbs);
- use of uniform interface. All resources have the same, uniform interface, which can be used to perform operations on the resource. The uniform HTTP interface defines the CRUD operations for creating, reading, updating and deleting resources;
- stateless communication between client and server;
- use of loose coupling and independence of the requests;
- cacheable;
- use of media-types.

4.2.3 HTTP Methods and their Usage

4.2.3.1 Overview

In REST, all API operations are based on the HTTP methods. The most used HTTP methods in RESTful APIs are GET, POST, PUT/PATCH and DELETE.

4.2.3.2 POST

The HTTP POST method is typically used to create a new resource. It creates a new resource anonymously as part of the collection resource. Requests sent with the POST method cannot be resubmitted and cannot be cached, since they are neither safe nor idempotent. If a resource has been created on the origin server in response to the POST method, an HTTP response with response code *201 Created* is returned. The response also contains an entity which describes the status of the request and refers to the new resource. In case of failure, an appropriate error response is returned.

4.2.3.3 GET

The HTTP GET method is used to retrieve information for a given resource. The GET request does not contain a payload only response contains the payload. As the GET operation is idempotent, the information retrieved by the GET operation can be cached and resubmitted. For any given HTTP GET API, if the resource is found on the server, an HTTP response with response code *200 OK* is returned along with the response body. If the resource does not exist, an HTTP response with *404 Not Found* response code is returned.

4.2.3.4 PUT/PATCH

The HTTP PUT method is used to completely update a resource identified by its resource URI. The request typically contains a representation of the resource to be updated. If applicable, PUT can be used to create a new resource directly. If the provided resource in the URI already exists, a PUT method is interpreted as an update. If the resource does not exist, it is interpreted as a creation. It is recommended that the origin server informs the client via the HTTP response code *201 Created* if a new resource has been created by the PUT operation. For the successful completion of the request, either, the *200 OK* or the *204 No Content* response code is returned.

NOTE 1: It is not advisable to mix creation by PUT and creation by POST in the same API.

The PATCH method, if supported, is used to partially update a resource. If this method is not supported, the response code *405 Method Not Allowed* is returned. For the successful completion of the request, either the *200 OK* or the *204 No Content* response code is returned.

NOTE 2: The PATCH method needs to be used with care if it is intended to be idempotent. The data format is defined by IETF RFC 7396 [i.47], IETF RFC 6902 [i.48] for JSON and IETF RFC 5261 [i.49] for XML.

4.2.3.5 DELETE

The HTTP DELETE method is used to remove a resource at the specified URL. The DELETE method is idempotent i.e. if the DELETE method is called on the already deleted resource, the same response will be returned. The response code *200 OK* is expected with the representation of deleted resource. If the action has been queued or the action has been performed but the response does not include an entity, the response code *204 No Content* is expected. However, if the resource never existed and DELETE is requested, the expected response code is *404 Not Found*. The DELETE method is not supported on collections of resources. Calling it on a collection results in a *405 Method Not Allowed* response code is returned.

4.2.4 Error Reporting

4.2.4.1 Overview

For the interaction in a distributed system, it is essential that API Consumers and Producers agree, how to behave if an error occurs. To achieve this common agreement, conventions, such as status codes are very valuable for error handling. Among the standardized HTTP status codes, some are defined to indicate error conditions that have occurred and give indications on which party (the API Consumer or the Producer) is responsible for the generated error.

Together with status code, HTTP allows the possibility to add information on the error conditions in the Body part of a Response. A best practice for response bodies in error situations is to include a representation of a *ProblemDetails* data structure, as specified in IETF RFC 7807 [i.43] that provides additional details of the error in a standardized manner. When the data structure is serialized in JSON, IETF RFC 7807 [i.43] mandates the *Content-Type* HTTP header to be set to *application/problem+json*.

The definition of the general *ProblemDetails* data structure from IETF RFC 7807 [i.43] is reproduced in Table 4.2.4.1-1. For more details, the *status* and *detail* attributes are included to ensure that the response contains additional textual information about an error. It is possible that particular APIs or particular implementations define extensions to introduce additional attributes that provide more information about the error.

NOTE: 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.

ETSI GS NFV-SOL 013 [i.42] and ETSI GS MEC 009 [i.23] specified the *ProblemDetails* structure mentioned below. The description column in Table 4.2.4.1-1 only provides some explanation of the meaning to facilitate understanding of the design. For a full description, see IETF RFC 7807 [i.43].

Table 4.2.4.1-1: The ProblemDetails data structure

Attribute name	Data type	Cardinality	Description
type	URI	0..1	A URI reference according to IETF RFC 3986 [i.17] that identifies the problem type. It is encouraged that the URI provides human-readable documentation for the problem (e.g. using HTML) when dereferenced. When this member is not present, its value is assumed to be *about:blank*.
title	String	0..1	A short, human-readable summary of the problem type. It should not change from occurrence to occurrence of the problem, except for the purposes of localization. If a type is provided and it is other than *about:blank*, this attribute will also be provided.
status	Integer	1	The HTTP status code for this occurrence of the problem.
detail	String	1	A human-readable explanation specific to this occurrence of the problem.
instance	URI	0..1	A URI reference that identifies the specific occurrence of the problem. It may yield further information if dereferenced.
(additional attributes)	Not specified.	0..N	Any number of additional attributes, as defined in a specification or by an implementation.

The following common error situations are applicable to all REST resources and related HTTP methods specified in the present document.

4.2.4.2 Client Errors

The client errors are indicated by a *4xx* response codes. These errors can be fixed by the client. It is recommended that the API provides a suitable error message which enables the client to fix the error, rather than responding with an error code only.

It is a best practice to include a link to publicly accessible documentation of the error on a web page. The following client errors indicate the malformed request:

- **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).
- **404 Not Found:** If the API producer did not find a current representation for the resource addressed by the URI passed in the request or is not willing to disclose that one exists. For example, a non-existing resource-Id was specified as path parameter.
- **405 Method Not Allowed:** If a particular HTTP method is not supported for a particular resource, the API producer will respond with this response code. The *ProblemDetails* structure may be omitted.
- **406 Not Acceptable:** The API cannot produce a response in any of the media-types that client can accept (e.g. if the *Accept* HTTP header does not contain at least one name of a content type that is acceptable to the API producer).
- **422 Unprocessable Entity:** The input is in the appropriate content-type, is syntactically correct (e.g. well-formed JSON), but is semantically wrong (e.g. because it fails validation against a schema).
- **429 Too Many Requests:** The API consumer has sent too many requests per time window and the API producer is able to detect that condition ("rate limiting").

The following authentication and authorization status codes are typically used:

- **401 Unauthorized:** Credentials are incorrect or missing. The user is not authenticated in the first place.
- **403 Forbidden:** If the API consumer is not allowed to perform a particular request on a given resource.

4.2.4.3 Server Errors

A *5xx* response code is used to indicate an error in the API or Server. For server errors, good logging is essential to be able to find the root cause of the problem.

It is the best practice to inform the API consumers, when the server-side error will be fixed and when the consumer can retry to send the request. This can be done by including the HTTP header field *Retry-After* with the delay in seconds in the response.

- **500 Internal Server Error:** If the server is unable to process the request, and retrying the same request later might eventually succeed, the server will respond with this response code.
- **501 Not Implemented:** The functionality requested by the client is not implemented yet.
- **503 Service Unavailable:** If the API producer encounters an internal overload situation of itself or of a system it relies on, it should respond with this response code.
- **504 Gateway Timeout:** If the API producer encounters a timeout while waiting for a response from an upstream server (i.e. a server that the API producer communicates with when fulfilling a request), it should respond with this response code.

4.3 API specification process

4.3.1 RESTful interface description languages

Several languages have been developed for specifying interface contracts of RESTful APIs. The purpose of an API specification language is to facilitate the consumption of API specifications by both machines and humans for validation, implementation and testing of the interfaces.

An appropriate specification language for standardizing APIs is:

- agnostic to implementation language;
- human readable;
- machine readable;
- compliant with REST principles;
- standardized; and
- sufficient for specifying APIs in the level of details that is present in existing standards.

Many languages come into place when designing/creating a new API. Some of these are:

- **OpenAPI™** is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services. OpenAPI has a large community support base and support for many Opensource frameworks.
- **RAML** stands for RESTful API Modelling Language and is based on YAML for describing RESTful APIs. RAML focuses on modelling more than specification. The tooling for the latest version of RAML (1.0) is limited. Mulesoft (RAML developers) joined OpenAPI consortium to support OpenAPI™ Specification (OAS™).
- **API Blueprint** comes with a syntax closer to markdown to describe the Web APIs. API blueprint has low adoption and limited community support.
- **WADL** stands for the Web Application Description Language comprising an XML vocabulary for expressing the behaviour of HTTP resources. This approach is not widely adopted due to the complexity of the specification, vendor-specific limitations, and lack of available tooling.
- **OData** stands for Open Data Protocol, which is an OASIS standard for the creation and consumption of queryable RESTAPIs. It supports ATOM and JSON format. OData is strongly coupled with data structures and less focused on interface specifications.

Of the languages analysed, OpenAPI was chosen to be used in the API specification process within ETSI as it fulfils all the requirements, fits well into the specification process, and is considered the de facto standard for API design according to "The State of API 2019 survey" [i.7]. The usage of OpenAPI is described in the following clauses.

4.3.2 Standardizing RESTful interfaces using OpenAPI

4.3.2.1 OpenAPI Overview

An OpenAPI document specifies a single API as JSON object that may be represented either in JSON or YAML format [i.14]. The present document is based on OpenAPI™ Specification version 3.0.3 [i.15].

Not that object names in OpenAPI™ specifications are case sensitive. Line comments start with "#".

4.3.2.2 Document

Each OpenAPI document specifies the title and the version of the API in the *info* object. The version of OpenAPI™ standard that the document conforms to is also required.

Following example describes the structure of an OpenAPI document. The *components* object is a container for all reusable objects.

The description of the *externalDocs* should specify issuing organization, document number and title. Within ETSI, the *externalDocs* element should be used to indicate which standardized deliverable is related to the OpenAPI definition, and the URL should indicate the link to the active work item information (e.g. on the ETSI Portal) or (when present) to the download location of the related document.

It is strongly recommended that the description of the *externalDocs* contains the exact drafting or publication version of the related document.

An example of document definition is shown in Figure 4.3.2.2-1.

```
openapi: 3.0.3
info:
  title: 'Examples for RESTful API guide'
  version: '1.0.0'
externalDocs:
  # Reference to the base document
  description: 'ETSI EG 203 647 ...'
  url: 'https://docbox.etsi.org/...'
paths: {}

# Optional definitions
servers: []
security: []
components:
  schemas: {}
  securitySchemes: {}
```

Figure 4.3.2.2-1: Example of document definitions

4.3.2.3 Data types

OpenAPI specifies a set of primitive data types and rules for defining structured data types. The specification is an extension of the JSON schema [i.16]. Data type schemas may be defined inline or in a *schemas* object which enables reuse of those definitions.

Structured type is either an *array* with member type declaration (*items* object) or an object with properties (*properties* object).

For object types the *required* array lists names of properties that are mandatory in the given data structure. A property may be constrained by a set of allowed values using an *enum* array.

Cardinality of the data in message body or parameter value is defined by the data type schema. *minItems* and *maxItems* values may be specified for array types. If an array of certain member type is used with different cardinalities, then it is recommended to define the array schemas inline.

OpenAPI defines *string*, *integer*, *number* and *boolean* primitive types and a number of well-known *format* identifiers that are used to encode structured data as primitive value. Primitive types may be extended by specifying additional non-standard formats as long as encoding of those formats is defined in the API specification.

An example of data type definitions is shown in Figure 4.3.2.3-1.

```
components:
  schemas:
    # Name of data type
    SearchResults:
      # Array type
      type: array
      items:
        # Type of array members, reference to ResourceData
        $ref: '#/components/schemas/ResourceData'
      # No more than 10 results
      maxItems: 10
    ResourceData:
      # Structured type
      type: object
      properties:
        # Property name
        id:
          # Property type
          type: string
        size:
          type: string
          enum:
            # Set of allowed values
            - big
            - bigger
            - biggerer
          # Default value for non-required property
          default: big
        created:
          # Date-time value encoded as string
          type: string
          format: date-time
      required:
        # Set of required properties
        - id
```

Figure 4.3.2.3-1: Example of data type definitions

Refer to the OpenAPI™ specification for additional means of constraining the data types.

4.3.2.4 Operations

Resource operations are grouped by paths and specified in the *paths* array. A path is a relative URL that starts with a "/" and may include parameters. The format for URLs is specified in IETF RFC 3986 [i.17].

Paths are unique within the document which means that all allowed operations on a given path are grouped together. An operation is defined by request method, parameter declarations and one or more responses.

Method name is one of HTTP methods standardized by IETF RFC 7231 [i.5], but in lower case as required by OpenAPI™ specification.

The *summary* contains the text that is used as the title of the respective clause in written specification. The *description* contains the contents of the clause.

All parameters used in the path are defined in *parameters* array for every operation. This can be done using associative array syntax. It is recommended to use only *string*, *number* or *boolean* as type of path parameter. Parameters may be defined in *components:parameters* object for reuse.

An example of operations definitions is shown in Figure 4.3.2.4-1.

```
paths:
  # Resource path relative to server, parameters in curly braces
  '/resource/{id}':
    # Method
    get:
      # Unique, case-sensitive identifier
      operationId: getResource
      summary: 'Read a resource'
      description: 'Read full contents of a resource with specific ID'
      parameters:
        # Parameter name used as the key in associative array of parameters
        - name: 'id'
          # The location of parameter: path, query, header or cookie
          in: path
          required: true
          description: 'Resource ID'
          schema:
            # Primitive type
            type: string
      responses: # Further content not shown, it includes responses 200, 401, 404, ...
```

Figure 4.3.2.4-1: Example of operations definitions

4.3.2.5 Requests

Request message is defined by request body and header values. Request body is specified by content media type and data type schema in a *requestBody* object. The schema is either defined inline or as a reference to a schema object.

Headers convey protocol specific information. Some header values such as authentication tokens and message body content type are specified implicitly by other constructs of OpenAPI.

An example of resource definition supporting a POST request is shown in Figure 4.3.2.5-1.

```
paths:
  '/resource':
    # POST a JSON object
    post:
      # Info excluded
      operationId: postResource
      parameters:
        # Reference to (reusable) parameter definition
        - $ref: '#/components/parameters/resourceId'
        # Reference to (reusable) header definition
        - $ref: '#/components/parameters/Version'
      requestBody:
        description: 'Data for new resource'
        required: true
        content:
          # Content media type (Content-Type header value)
          application/json:
            schema:
              # Reference to data type
              $ref: '#/components/schemas/ResourceData'
      responses: # Further content not shown, it includes responses 201, 400, ...
```

Figure 4.3.2.5-1: Example of resource definition supporting a POST request

It is recommended to use multiple part messages when sending files or other binary content. The encoding for parts may be specified in an *encoding* object if the default (*application/octet-stream*) is not applicable.

The usage of multipart/form-data is specified in IETF RFC 2388 [i.18].

An example of resource definition supporting multipart data in the request is shown in Figure 4.3.2.5-2.

```
paths:
  '/resource/{id}/file':
    # Upload a resource file
    put:
      # Info excluded
      operationId: uploadResourceFile
      parameters:
        - $ref: '#/components/parameters/resourceId'
      requestBody:
        description: 'An image file to be attached to the resource'
        content:
          multipart/form-data:
            schema:
              type: object
              properties:
                # Property name (also the name applied to content disposition)
                file:
                  type: string
                  # Sets content type to application/octet-stream
                  format: binary
            encoding:
              # Applies custom encoding to "file" property
              file:
                # Override default content type
                contentType: image/png
      responses: # Further content not shown, it includes responses 200, 201, 204, 400, ...
```

Figure 4.3.2.5-2: Example of resource definition supporting multipart data in the request

Custom headers are specified the same way as other parameters. Either inline or defined in a *components:parameters* object. It is recommended to use only *string* and *number* as the type of a header parameter.

An example of definition of custom request headers is shown in Figure 4.3.2.5-3.

```
components:
  parameters:
    Version:
      name: 'Version'
      description: 'API version'
      in: header
      required: true
      schema:
        type: string
```

Figure 4.3.2.5-3: Example of definition of custom request headers

4.3.2.6 Responses

For each request, all acceptable responses should be specified. Response is identified by response code. The body is optional and specified in a *content* object similar to request body.

Custom headers may be specified in a *headers* object.

Examples of response definitions are shown in Figure 4.3.2.6-1.

```yaml
paths:
  '/resource/{id}':
    get:
      # Request and parameters excluded
      responses:
        # Response code
        200:
          description: 'The requested resource'
          # Custom headers
          headers:
            ETag:
              # Reference to (reusable) header definition
              $ref: '#/components/headers/ETag'
          # Response body
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ResourceData'
        401:
          # Reference to (reusable) response definition
          $ref: '#/components/responses/401'
        404:
          $ref: '#/components/responses/404'
components:
  responses:
    # Common responses with response code as identifier
    204:
      description: 'No content'
    401:
      description: 'Unauthenticated'
    404:
      description: 'Not found'
  headers:
    # Definition of ETag header
    ETag:
      description: 'Identifier for a specific version of a resource'
      schema:
        type: string
```

Figure 4.3.2.6-1: Examples of response definitions

4.3.2.7 Callbacks

One or more callbacks may be specified for a request allowing to describe more complicated scenarios. Nota that the order of responses and callbacks is unspecified. Callbacks may be defined in *components:callbacks* object for reuse.

Callback path may contain runtime expressions referring to properties of the request (such as body or header) to identify the callback URL.

See the examples for using the callback objects in clause 4.3.

An example of callback definition is shown in Figure 4.3.2.7-1.

```
'/subscription':
  post:
    summary: 'Subscribe to authenticated notifications'
    # Description excluded
    operationId: subscribeNotifications
    requestBody:
      content:
        application/json:
          schema:
            # Subscription containing callbackUrl property
            $ref: '#/components/schemas/Subscription'
    responses:
      # Subscription was created
      201:
        $ref: '#/components/responses/201'
    # Out-of-band notifications from server
    callbacks:
      # Named callback object (inline or reference)
      auth:
        # Local path used by server for callback(s)
        '{$request.body#/callbackUrl}/incoming':
          post:
            requestBody:
              content:
                application/json:
                    schema:
                      $ref: '#/components/schemas/AuthenticatedNotification'
            responses: # Content excluded
```

Figure 4.3.2.7-1: Example of callback definition

4.3.2.8 Query parameters

Query parameters are used to customize operations and specify conditions on the data that is requested or provided.

Query parameter values are not limited to primitive types. Various formats exist for converting structured or array values into strings values suitable for query parameters. Chosen format should be specified in API specification and it should conform to IETF RFC 1738 [i.19].

An example of supported query parameters definition is shown in Figure 4.3.2.8-1.

```
paths:
  # Example search path /search?text=rest&min=5
  '/search':
    # Example search path /search?text=rest&max=5
    get:
      summary: 'Search resource'
      # Description excluded
      operationId: searchResource
      parameters:
        - name: 'text'
          in: query
          required: true
          description: 'Text to search for'
          schema:
            type: string
        - name: 'max'
          in: query
          # Optional parameter
          required: false
          description: 'Maximum number of results expected'
          schema:
            type: number
      responses: # Content excluded
```

Figure 4.3.2.8-1: Example of supported query parameters definition

4.3.2.9 Extensions

OpenAPI documents may contain non-standard objects. The names of custom objects are prefixed by "*x*-" and should be defined in the API specification.

An example of usage of extensions is shown in Figure 4.3.2.9-1.

```yaml
/service:
  get:
    # Info excluded
    operationId: getService
    # Support for GET request is mandatory for API provider
    x-etsi-provision: mandatory
    parameters:
      - name: 'circuitswitching'
        in: query
        required: false
        schema:
          type: string
        x-etsi-capabilities:
          # Parameter only applies to "3G" capability
          - 3G
    responses:
      200:
        description: 'The requested service'
        content:
          application/json:
            schema:
              type: object
              properties:
                speed:
                  type: string
                  enum:
                    - fast
                    - superfast
                  x-etsi-enum:
                    # Enum value "superfast" is optional and
                    # only applies to "4G" and "5G" capabilities
                    superfast:
                      required: false
                      x-etsi-capabilities:
                        - 4G
                        - 5G
```

Figure 4.3.2.9-1: Example of usage of extensions

Note that OpenAPI editing tools may not support proper representation of custom objects.

Recommended extensions:

- **x-etsi-ref**: reference to an item in a document. If the document is omitted, then it is a reference to the document specified in the *externalDocs* property. It is recommended to follow the referring format described in ETSI Drafting Rules [i.20], clause 2.10.
- **x-etsi-note**: used for giving additional information in the same way as notes in ETSI deliverables.
- **x-etsi-provision**: provision on implementation of operation, request, response, parameter or property indicated as *mandatory*, *optional* or other values described in ETSI EG 201 058 [i.9], clause 8.3. The use of the *required* property from OpenAPI (where applicable) with value *true* has the same meaning as the value *mandatory* of this extension. If the value of this extension is anything else, then the *required* property should be set to *false* (or excluded).
- **x-etsi-capabilities**: an array containing the names of capabilities for which the containing object is applicable for, or an indication of.
- **x-etsi-enum**: an object that describes applicability of enumeration literals (see Figure 4.3.2.9-1).

If additional extensions are defined, then it is recommended to prefix the names with *x-etsi-TBNAME-* where *TBNAME* is the name of the technical body authoring the document.

4.3.2.10 Other

Reference objects may be used throughout OpenAPI document instead of inline definitions (see for example Figure 4.3.2.7-1). Reference object contains a *$ref* property with relative or absolute URL. Relative URLs are relative to current document. The path is omitted for references in the same document. The fragment part indicates the location of the referenced object in the document structure followed by the name of the object.

An example of usage of JSON references is shown in Figure 4.3.2.10-1.

```
# Reference to ResourceData object in the (components/schemas object of) the same document
$ref: '#/components/schemas/ResourceData'

# Reference to ResourceData object in a document named types.yaml
$ref: 'types.yaml#/components/schemas/ResourceData'
```

Figure 4.3.2.10-1: Example of usage of JSON references

File naming conventions and directory structure for file storage is outside the scope of present document.

Server object may be used to describe the structure of API URL. This provides the base for relative URLs of the operations.

An example of servers object definition is show in Figure 4.3.2.10-2.

```
servers:
  # Recommended structure for API paths
  - url: '{apiRoot}/{apiName}/{apiMajorVersion}/'
    variables:
      apiRoot:
        default: https://example.com
      apiName:
        description: 'Interface name from the base document'
        default: rest-api-guide
      apiMajorVersion:
        description: 'Major version of the API from the base document'
        default: v1
```

Figure 4.3.2.10-2: Example of servers object definition

Tags may be used to indicate which interface a specific operation belongs to. They are used for grouping by visual tools.

An example of usage of tags is shown in Figure 4.3.2.10-3.

```
paths:
  /resource/{id}:
    get:
      # Grouping
      tags:
        - Resource management
tags:
  # Optional descriptions of tags
  - name: Resource management
    description: Operations for managing resources
```

Figure 4.3.2.10-3: Example of usage of tags

It is recommended to specify unique *operationId* for every operation (method at a path). Operation identifiers can be used as explicit references to the OpenAPI document in derived specifications such as ICSs or test suites. Many tools use *operationId* values as implementation names for the operations, thus appropriate naming conventions should be established.

4.3.2.11 Process

An OpenAPI document defines technical requirements for an API implementation. It is commonly an accompaniment to written document that specifies non-technical aspects such as the context, concepts, behavioural semantics and requirements. The OpenAPI document may be included in the written document as a whole or in fragments in relevant clauses. Alternatively, it may be amended as a separate file as electronic attachment.

Clause 5.1 of the ETSI EG 201 015 [i.8] lists a number of common problems found in standards and clause 5.2 provides recommendations for avoiding those problems. One way to avoid common issues is to use modelling techniques for the technical parts of the standard. It is therefore recommended to use OpenAPI document as the single source of truth for technical aspects of the API throughout the specification process.

Compared to written document, OpenAPI tools provide following additional benefits:

- OpenAPI provides known specification format and experts working on the API do not have to get familiar with custom specification format;
- as a domain specific language, OpenAPI reduces the overhead of descriptive text;
- OpenAPI tools enforce syntactically correct specifications and provide some semantic validation; and
- experts may use the tools that they are already familiar with.

Clause 6 of the ETSI EG 201 015 [i.8] lists various explicit and implicit methods for validating standards. Using OpenAPI tools for generating prototypical implementation of the API or using OpenAPI tools for exploratory testing may be considered implicit validation methods of the API specification.

Various OpenAPI processing tools are available for producing written documents based on OpenAPI documents. Such tools may be used as the mechanism for integrating OpenAPI definitions into written document. The integration process is usually manual due to inconsistent structuring of written API specifications.

4.4 Common Patterns

4.4.1 Filtering Patterns

4.4.1.1 Overview

This clause specifies the structure and constraints of query operations that allow controlling the size of the large result set of the GET requests. Standardization groups within ETSI (ETSI GS NFV-SOL 013 [i.42] and ETSI GS MEC 009 [i.23]) and other SDOs (TM Forum guides) mentioned some of the common patterns for filtering large query result. Filtering patterns included in the present document are:

- Attribute-based filtering.
- Attribute Selector.
- Pagination.

4.4.1.2 Attribute-based filtering for collections

Filtering can be implemented as a query parameter named for the attributes to be filtered on. Attribute-based filtering allows reducing the number of objects returned by a query operation. Typically, attribute-based filtering is applied to a GET request that queries a resource that 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 is requested by adding a set of URI query parameters, the "attribute-based filtering parameters" or here refer as a filter to a resource URI. The directive *filter* is provided to retrieve a resource.

The complete resource representation (with all the attributes) is returned If no attribute is provided in the "*filter*".

The following representations may apply:

- The GET request with *filter*

  ```
  GET ../Resource?filter=(< opr>, "AttrName", "AttrValue")
  ```

- For hierarchically-structured data, filters can also be applied to attributes deeper in the hierarchy. It is separated by slash ("/").

  ```
  GET ../Resource?filter=(< opr>, "AttrName" / ["heir-AttrName"], "AttrValue")
  ```

- If a filter contains multiple sub-parts and they share the same attribute prefix, they are evaluated together per array entry when traversing an array. The filters sub-parts are separated by *";"*.

```
GET ../Resource?filter=(< opr>, "AttrName" / ["heir-AttrName"], "AttrValue") ; (< opr>,
"AttrName" / ["heir-AttrName"], "AttrValue")
```

The *AttrName* is the name of one attribute in the data type that defines the representation of the resource and *AttrValue* entry contains a scalar value of type Number, String, Boolean, Enum or DateTime. *heir-AttrName* is the resource within the hierarchy.

The operators listed in Table 4.4.1.2-1 is supported.

Table 4.4.1.2-1: Supported operators for Attribute-based filtering

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

In response to the HTTP request, the response code *200 OK* is returned for a success. In case of an invalid attribute filtering query, *400 Bad Request* is returned.

In response to the HTTP request, the response code *200 OK* is returned for a success. *400 Bad Request* is returned in case of an invalid attribute filtering query.

4.4.1.3 Attribute Selector

In certain scenarios, resource representations can become quite big, in particular, if the resource contains multiple sub-resources, or if the resource representation itself contains a deeply nested structure. It is often desirable to reduce the amount of data exchanged over the interface and processed by the API consumer application.

ETSI NFV and MEC groups specified the filtering pattern known as an attribute selector, which is typically part of a query, allows the API consumer to choose which attributes it wants to be contained in the response. Only attributes that are not required to be present, i.e. those with a lower bound of zero on their cardinality (e.g. 0..1, 0..N) and that are not conditionally mandatory, are allowed to be omitted as part of the selection process. Attributes can be marked for inclusion or exclusion. The pattern is applicable to GET methods.

Table 4.4.1.3-1 defines the valid parameter combinations in a GET request and their effect on the response body.

Table 4.4.1.3-1: Attribute Selector with valid combinations

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

NOTE 1: 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.

NOTE 2: The *exclude_default* value is recommended to be used as a default when the goal is to create a list that is a digest of the individual items.

In response to the HTTP request, the response code *200 OK* is returned for success along with resource representations. In case of an invalid attribute filtering query, *400 Bad Request* is returned.

4.4.1.4 Pagination

In some cases, Collection resources become too large that the response to the query will adversely affect the performance of the server. If it is not possible to return the complete collection due to its size, the response can be chunked up in the form of bite-sized chunks called pages.

Pagination should be configurable via query parameters. To help API consumer navigate through the pages, the API producer provides a response that contains a first page (subset) of the results to the query and provides the meta-information. This meta-information includes a *Link* HTTP header (according to the IETF RFC 8288 [i.50]) with the *rel* attribute set to *next*, which communicates a URI that allows to obtain the next page of results to the original query.

The API consumer can send a GET request to the URI communicated in the *Link* header to obtain the next page of results. The response which returns that next page contains the *Link* header to point to the next page unless there are no further pages available in which case the *Link* header is omitted.

Alternatively, if the API producer does not support pagination, the server will reject the query request with a *400 Bad Request* response and include the *ProblemDetails* (refer to clause 4.2.4.1) payload body to provide more information about the error.

Pagination can be implemented using one or both of two query parameters:

- **limit**: to define the number of items returned in the response;
- **marker**: to specify the ID of the last seen item;
- **offset**: to define the requested index for the start of resources to be provided.

Figure 4.4.1.4-1 depicts the generic flow of paging mechanism usage.

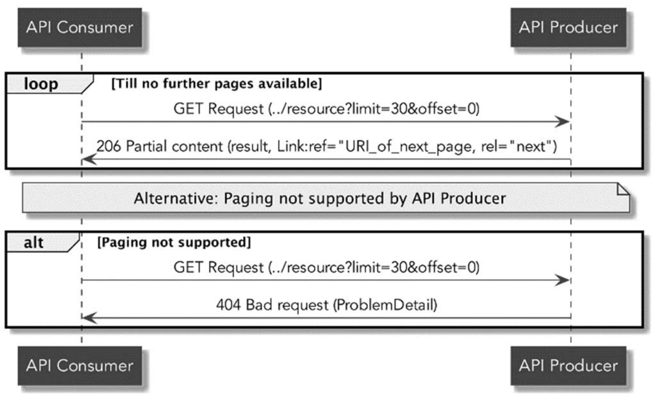

API Consumer
API Producer
loop
[Till no further pages available]
GET Request (../resource?limit=30&offset=0)
206 Partial content (result, Link:ref="URI_of_next_page, rel="next")
Alternative: Paging not supported by API Producer
alt
[Paging not supported]
GET Request (../resource?limit=30&offset=0)
404 Bad request (ProblemDetail)
API Consumer
API Producer

Figure 4.4.1.4-1: Generic flow of paged response

4.4.2 Pattern for URI Creation

4.4.2.1 Resource URI Structure

REST APIs use Uniform Resource Identifiers (URIs) to address resources. Resources are either individual resources or structured resources that can contain child resources. IETF RFC 3986 [i.17] defines the generic URI syntax shown below:

```
URI = scheme "://" authority "/" path [ "?" query ] [ "#" fragment ]
```

From the reference from standardization groups within ETSI (NFV, MEC) and its Partnership Projects (3GPP), when a resource URI is an absolute URI, the URI structure is specified as follows:

```
{apiRoot}/{apiName}/{apiVersion}/{apiSpecificResourceUriPart}
```

The URI prefix is briefly reported in Table 4.4.2.1-1.

Table 4.4.2.1-1: Path structure for URI creation

URI Parts	Descriptions
{apiRoot}	Indicates: - The scheme ("HTTP" or "https") - The fixed string "://" - authority (host and optional port) as defined in IETF RFC 3986 [i.17] - An optional deployment-specific string (API prefix).
{apiName}	Defines the name of the API or the interface name of the implementation.
{apiVersion}	Indicates the current major version of the API.
{apiSpecificResourceUriPart}	Defines a resource URI of the API relative to the base URI.

All resource URIs of the API should comply with the URI syntax as defined in IETF RFC 3986 [i.17]. It is recommended that an implementation that dynamically generates resource URI parts (individual path segments, sequences of path segments, query parameter values) should ensure that the character set that is allowed by IETF RFC 3986 [i.17] only be used in these URI parts.

4.4.2.2 Design Rules for REST API URI

The "REST API Design Rulebook" [i.41] introduced some design rules that generally apply on URI creation. This clause provides some recommendations mentioned in the Design Rulebook for consistent URI formatting:

- Use forward-slash ("/") to indicate hierarchical relationships.
- Do not use trailing forward-slash ("/") in URIs.
- Use hyphens ("-") to improve the readability of URIs.
- Do not use underscores (*"_"*).
- Use lowercase letters in URIs.
- Do not use file extensions.

NOTE: Implementation specific constraint takes precedence over these design rules.

4.4.3 Pattern to avoid Update Conflict and Data loss

4.4.3.1 Description

The concurrent update conflict or race condition mostly arises when two clients try to update a resource using the PUT or PATCH method concurrently. 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 [i.51]) supports conditional requests to detect such a situation and allows the client 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. An *ETag* header in a response does not indicate in itself that the resource requires concurrency control.

If the client wishes that the server executes the modification request only if the *ETag* has not changed since the time the GET response was generated, the client adds to the modification request the HTTP header *If-Match* with the *ETag* value obtained from the GET request. The server executes the modification 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.

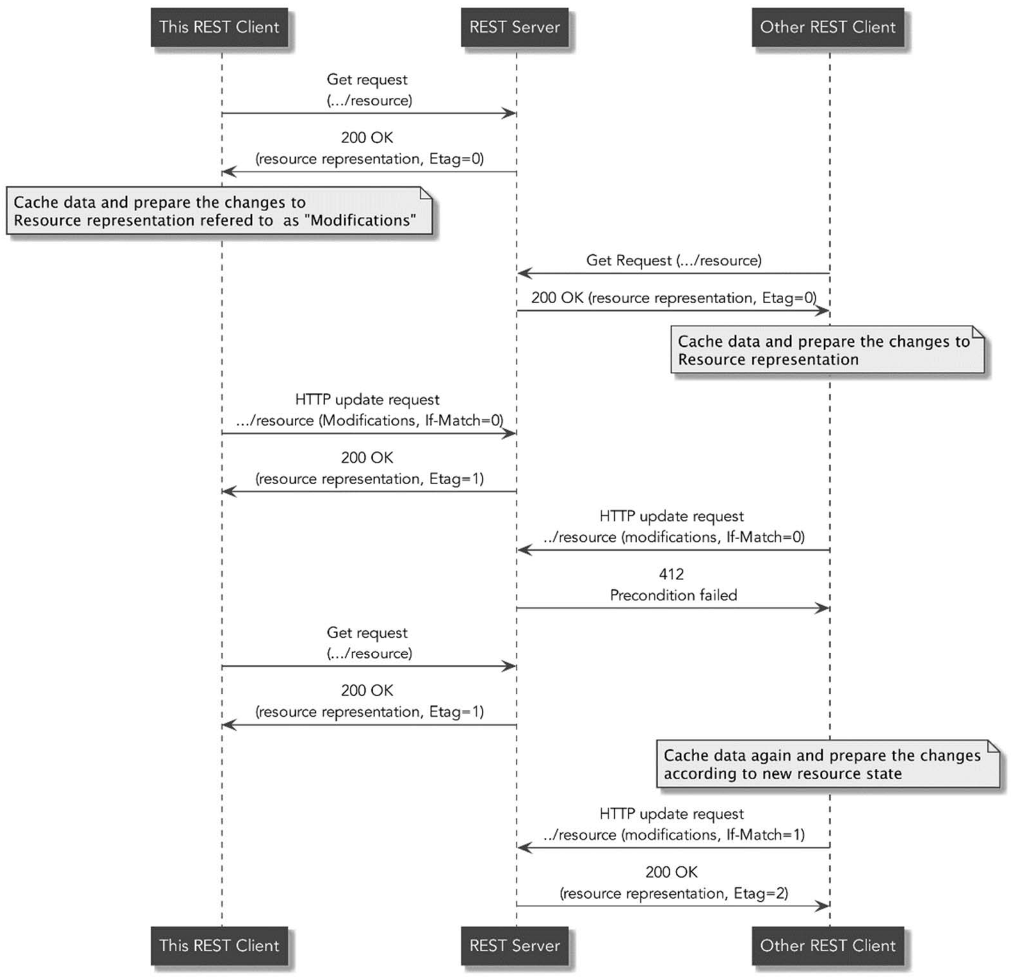

This REST Client
REST Server
Other REST Client
Get request (.../resource)
200 OK (resource representation, Etag=0)
Cache data and prepare the changes to Resource representation refered to as "Modifications"
Get Request (.../resource)
200 OK (resource representation, Etag=0)
Cache data and prepare the changes to Resource representation
HTTP update request .../resource (Modifications, If-Match=0)
200 OK (resource representation, Etag=1)
HTTP update request ../resource (modifications, If-Match=0)
412 Precondition failed
Get request (.../resource)
200 OK (resource representation, Etag=1)
Cache data again and prepare the changes according to new resource state
HTTP update request ../resource (modifications, If-Match=1)
200 OK (resource representation, Etag=2)
This REST Client
REST Server
Other REST Client

Figure 4.4.3.1-1: Flow of concurrent update of a Resource

This pattern applies to any HTTP modification request i.e. PUT and PATCH. In a particular API, it is recommended to stick to one update pattern - either PUT or PATCH.

On success, either *200 OK* or *204 No Content* is 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* is returned. Otherwise, on failure, the appropriate error code is returned.

4.4.4 Authorization and Authentication

4.4.4.1 Overview

For secure systems, APIs are only allowed to be accessed by authorized consumers. Handling of authorization differs between making an API call and sending a notification. This clause outlines several methods for authentication and authorization:

- OAuth 2.0.
- OpenID connect with JWT ID Token.
- TLS Certificates.

4.4.4.2 API Authorization using OAuth 2.0 Access tokens

The HTTP-based OAuth 2.0 framework allows REST clients to obtain access to a resource exposed by an API. To facilitate this, REST APIs may collaborate with an OAuth 2.0 Authorization Server, checking each incoming call for an access token, which it should validate with the Authorization Server. The response from the Authorization Server indicates whether the access token is valid (it was issued by the OAuth Provider and it has not expired) as well as the scope of access for which the token was issued. The API security framework assumes an Authorization Server to be available for both the API Consumer and the API Producer. This Authorization Server can be used to perform the authentication for the credentials of the REST API Consumer and the API Producer.

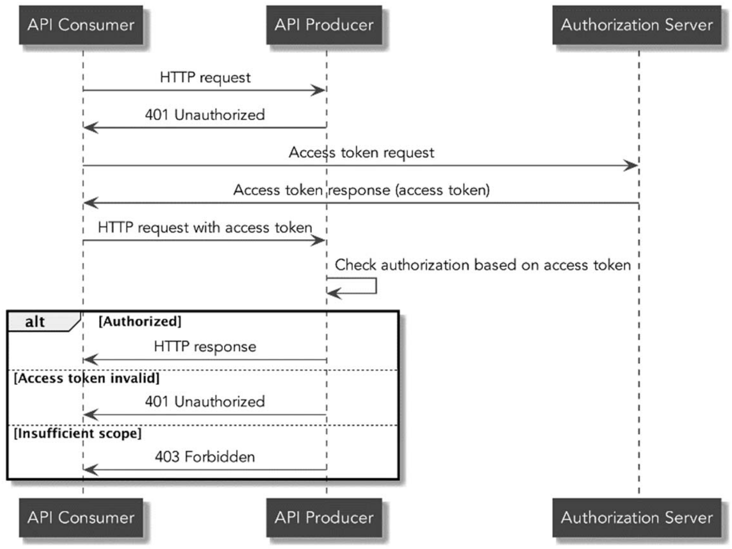


Figure 4.4.4.2-1: Flow of API authorization using OAuth 2.0 access token

To ensure that no third party can eavesdrop on sensitive information such as client credentials or access tokens, HTTP over TLS can be used to protect the transport.

4.4.4.3 API Authorization using TLS Certificates

The security of the API keys, HTTP basic authentication and OAuth 2.0 depends on TLS. The authentication and authorization are defined herein based on TLS certificates, applying the IETF RFC 5246 [i.44]. To facilitate mutual authentication during the TLS tunnel setup process, the server requests a client certificate as described in section 7.4.4 in IETF RFC 5246 [i.44]. Before the API request, it is assumed that the Authorization Server is configured with the authorization policy and access rights against the certificates. The API Consumer accesses a resource provided by the API Producer using a TLS tunnel where the certificates of both the API Producer and API Consumer are used to establish the secure tunnel. The API Producer checks authorization based on the TLS certificate of the API Consumer. The TLS certificate of the API Consumer is obtained during the TLS handshake.

TLS ensures the confidentiality and integrity of the input and output of the API Producer and API Producer i.e. the information in the HTTP body, header and in the URI. API producers can reject the requests without TLS by sending the HTTP response code *403 Forbidden*.

It is assumed that the certificates should be already enrolled in the communicating entities and the Authorization server should be configured with the authorization policy and access rights against the certificates.

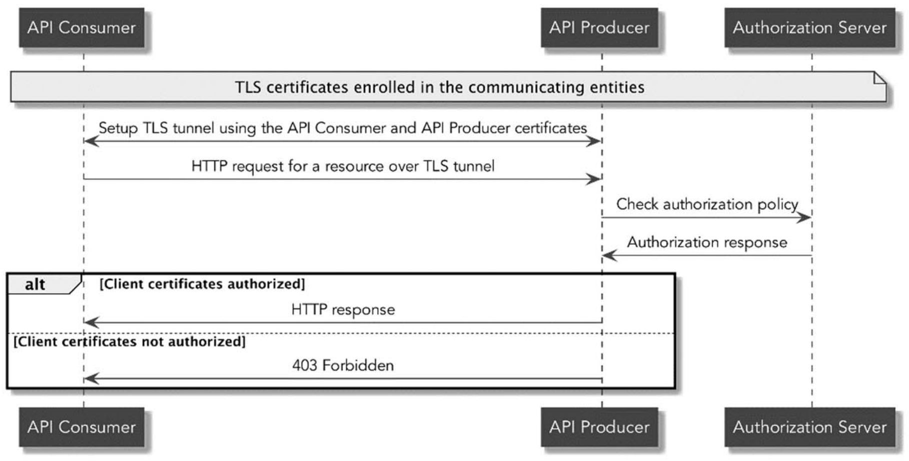

API Consumer
API Producer
Authorization Server
TLS certificates enrolled in the communicating entities
Setup TLS tunnel using the API Consumer and API Producer certificates
HTTP request for a resource over TLS tunnel
Check authorization policy
Authorization response
alt
[Client certificates authorized]
HTTP response
[Client certificates not authorized]
403 Forbidden
API Consumer
API Producer
Authorization Server

Figure 4.4.4.3-1: Flow of API authorization using TLS certificates

Two-way TLS protection is well suited for point-to-point integration with trusted partners. It is recommended to upgrade the API platform to support the latest TLS version.

4.4.4.4 API Authorization using OpenID connect with JWT ID Token

OpenID Connect is a standardized identity layer on top of the OAuth 2.0 protocol. By design, OAuth does not expose the identity of the end user towards the API Consumer. Only the access token is provided to the API Consumer i.e. it only acts as a random identifier. Thus, the rights for access are not encoded into the access token, but only associated with this identifier.

OpenID Connect extends OAuth by using the additional token, the ID token. It allows clients to verify the identity of an end-user based on the authentication performed by an authorization server as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner, by means of an additional API, the *userInfo* API. The *userInfo* API is protected by OAuth 2.0 and provides additional identity information about the user. The ID token holds identity information (or "claims") about the user i.e. name, email address, etc. OpenID Connect specifies a RESTful HTTP API, using JSON as a data format.

The ID token is formatted according to the JSON Web Token (JWT) standards. JSON Web Tokens are an open, industry-standard IETF RFC 7519 [i.45] method for representing claims securely between the parties. Claims or user information are represented as JSON data structures. The information can be signed according to the JSON Web Signature (JWS) to ensure the integrity and can be encrypted according to JSON Web Encryption (JWE) to ensure privacy. The mechanism of work for JWS and JWE is out of the scope of the present document.

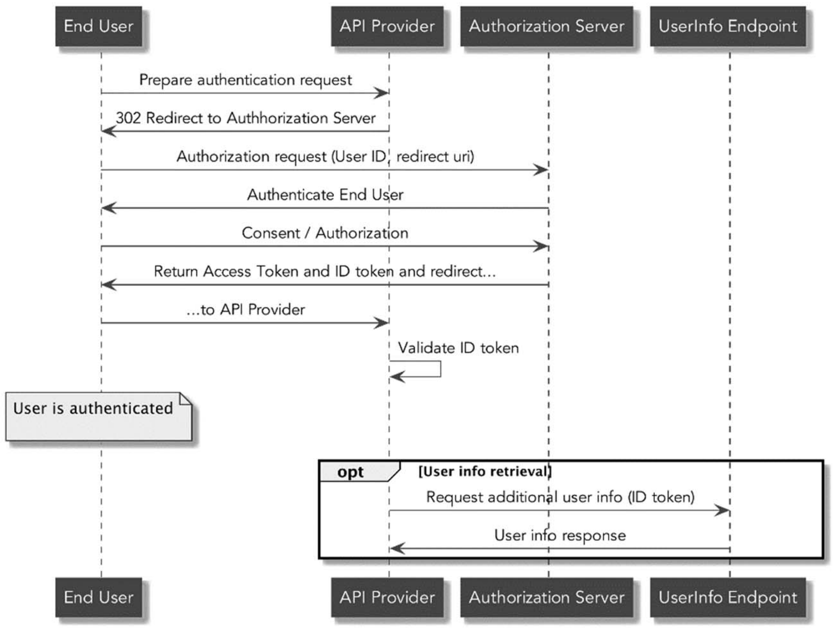

End User
API Provider
Authorization Server
UserInfo Endpoint
Prepare authentication request
302 Redirect to Authhorization Server
Authorization request (User ID, redirect uri)
Authenticate End User
Consent / Authorization
Return Access Token and ID token and redirect...
...to API Provider
Validate ID token
User is authenticated
opt
[User info retrieval]
Request additional user info (ID token)
User info response

Figure 4.4.4.4-1: Flow of API authorization using OpenID connect

4.4.5 Non-CRUD operations

4.4.5.1 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 several cases, actual operations needed in system design are difficult to model as CRUD operations, be it because they involve multiple resources, or because that they are processes that modify a resource and that take several input parameters that do not appear in the resource representation. Such operations can be modelled as "task resources".

A task resource is a child resource of a primary resource that is intended as an endpoint to invoke a non-CRUD operation. That non-CRUD operation executes a procedure that modifies the state of that actual resource in a specific way or performs computation and returns the result.

NOTE: The term "task resource" is commonly used in ETSI's NFV (ETSI GS NFV-SOL 013 [i.42]) and MEC (ETSI GS MEC 009 [i.23]) specifications as well as in TM Forum guides. The same approach can also be called as a "controller resource" (refer to "REST API Design Rulebook" [i.41]) or as "sub-resource".

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. HTTP calls the POST request method unsafe and non-idempotent, thus, to trigger the operations that cannot be intuitively mapped to one of the other core HTTP methods.

Different responses to a POST request to a task resource are possible, such as:

- **200 OK** to provide a computation result based on the state of the resource and additional parameters.
- **204 No Content** to signal success but not return a result.
- **303 See Other** to indicate the operation modifies another resource other than primary resource. In case the operation modifies a primary resource, the *Location* HTTP header points to the primary resource.
- **202 Accepted** for asynchronous invocation.

On failure, the appropriate error code is returned.

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
```

Figure 4.4.5.1-1 illustrates the flow for Non-CRUD operation.

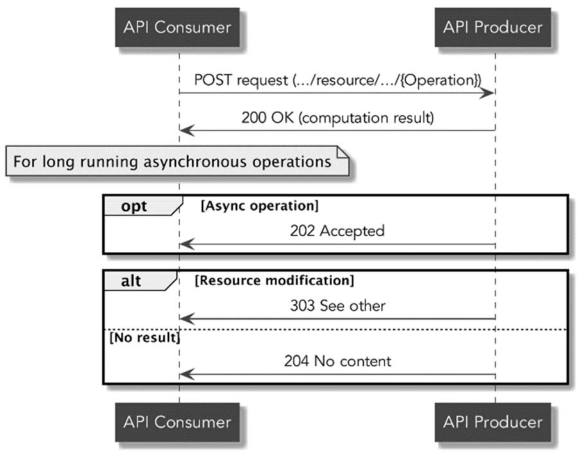


Figure 4.4.5.1-1: Generic flow of Non-CRUD operations

4.5 Naming conventions

Naming conventions help keeping different specifications consistent, thus easier to be understood and adopted.

In drafting an API specifications, coordination and conventions may be applied on several aspects. For this reason, several standardization groups within ETSI and its Partnership Projects has specified their own internal guidelines for naming conventions.

The aim of the present guide is not to replace the practices and conventions already established in those groups, but to offer a framework for novel activities and to emphasize the common rules and guidelines already present.

First of all, the list identifies in Table 4.5-1, the aspects of an API that require conventions.

Table 4.5-1: API aspects which may require naming conventions

Id	API Aspect	Description
1	Folder and file names	YAML or JSON files containing the API definition. A set of several files may be used and organized to isolate reusable definitions.
2	API names	Name of the API.
3	URI path segments	Identifiers of resources and sub-resources, mapped to the entities manipulated by the API.
4	URI Query segments	Parameters available as query segments
5	Data types names	Reusable data definitions.
6	Data attributes names	Data type instances used as parameters or members of other data definitions or API calls.

Moreover, a definition of letter-casing styles is provided in several guides, to clearly identify styles to be used to define the naming conventions. The case conventions defined in ETSI TS 129 501 [i.21], ETSI GS NFV-SOL 015 [i.22] and ETSI GS MEC 009 [i.23] are briefly reported in Table 4.5-2.

Table 4.5-2: Case definitions from 3GPP, ETSI NFV and ETSI MEC specifications

ID	Name	Description	Reference	Examples
1	UPPER_WITH_UNDERSCORE	All letters of a string are capital letters. Digits are allowed. Word boundaries are represented by the underscore "_" character. No other characters are allowed.	3GPP, MEC,	DATA_MANAGEMENT, CELL_CHANGE
2	lower_with_underscore	All letters of a string are lowercase letters. Digits are allowed. Word boundaries are represented by the underscore "_" character. No other characters are allowed.	3GPP, MEC	data_management, cell_change
3	UPPER-WITH-HYPHEN	All letters of a string are capital letters. Digits are allowed. Word boundaries are represented by the hyphen "-" character. No other characters are allowed.	3GPP	DATA-MANAGEMENT, CELL-CHANGE
4	lower-with-hyphen	All letters of a string are lowercase letters. Digits are allowed. Word boundaries are represented by the hyphen "-" character. No other characters are allowed.	3GPP	data-management, cell-change.
5	UpperCamel	A string is formed by concatenating words. Each word starts with a letter or a digit. The first letter of each word is an uppercase letter; all other characters in the word are lowercase letters or digits. Abbreviations follow the same scheme (i.e. first letter uppercase, all other letters lowercase).	3GPP, MEC	DataManagement, CellChange, 5QiPriorityLevel, Amf3GppAccessRegistration
6	lowerCamel	A string is formed by concatenating words. Each word starts with a letter or a digit. The first letter of the first word is a lowercase letter; the first letter of the rest of the words is an uppercase letter. All other characters in the words are lowercase letters or digits. Abbreviations follow the same scheme.	3GPP, MEC	dataManagement, cellChange, 5qiPriorityLevel
7	ALLUPPER	All letters of a string are uppercase letters. Digits may be used except at the first position. Other characters are not used.	NFV	MANAGEMENTINTERFACE, ETSINFVMANAGEMENT2
8	alllower	All letters of a string are lowercase letters. Digits may be used except at the first position. Other characters are not used.	NFV	managementinterface, etsinfvmanagement2

4.6 Versioning

4.6.1 Specifications and OpenAPI definitions versions

In the lifespan of certain standardized technology, from design to decommissioning, several versions of interfaces specifications and implementations are cyclically defined, following developments and maintenance of the standardized systems.

In order to track consistently and precisely the version information among interoperable solutions, a unified versioning scheme should be designed as part of the communication system itself. RESTful APIs are no exception.

In the context of standardized machine-readable definitions, version information may refer to:

- the version of the standardized interface, following the versioning rules of the related SDO;
- the version of the OpenAPI™ Specification (OAS™) definition w.r.t to the interface it defines, to allow incremental development; and
- the version of implementations for producers and consumers communicating over the standardized interface.

A complete versioning scheme should take into account these three components, which will be referred to as *doc_version*, *oas_version*, *impl_version*. The model of relationship between the three versions is depicted in Figure 4.6.1-1 by means of a class diagram.

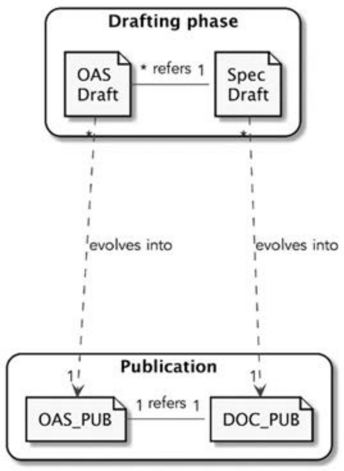


Figure 4.6.1-1: Versioning model

The following recommendations should apply:

- Each version of an OpenAPI should refer univocally to a specific version of the specification.
- Multiple versions of the OpenAPI definition for the same specification version are allowed in the drafting phase, i.e. when the specification is not in the Published state.
- Each published specification should refer univocally to a specific OpenAPI definition.

Following the recommendations above, it is not be allowed to create maintenance versions of an OpenAPI™ specification after the related interface specification has been published. In this situation, if changes are needed in the OpenAPI it is recommended that a new version of the specification is edited.

Similarly, if a new version of the base specification is drafted, the referenced OAS version should always be incremented. When an update of the base specification does not introduce provisions that would be expressed in the OAS itself and resulting in changes of the OAS definition file, the base document may nonetheless set provisions relevant on the interface implementations. Therefore at least the *externalDocuments* element should be updated in the OpenAPI definition file to refer to the new version of the related specification, and the version of the file should be incremented as well.

4.6.2 Modelling version information

The OpenAPI metamodel identifies a specific attribute to specify version information. Moreover, the API may define other ways to negotiate and identify implementation versions at design-time or runtime that may be modelled in the OpenAPI definition as well.

A (incomplete) list of such mechanisms is:

- specifying the version field in the information object of the OpenAPI metamodel;
- using a URI parameter to identify the version of the specification (e.g. as a path segment or query parameter);
- using HTTP Headers;
- modelling specific version management endpoints as part of the interface itself (i.e. resources and related operations to discover and invoke specific versions of the API).

4.7 Implementation

In the specification approach presented in the previous clauses, the OpenAPI™ specification language is recommended to be used, in order to provide a clear description of the standardized protocol interface.

Nevertheless, the original design of OpenAPI language (previously known as Swagger language) aimed at defining contract documents to be served together with the API implementation itself, at the same location where the service is exposed.

This use case does not directly apply to standardization of APIs, as they are meant to serve as blueprints for implementation and testing, but they do not identify a specific implementation.

Moreover, a standardized API may be (and often is) expanded with custom and proprietary features which would not be described in the OpenAPI file.

Therefore, the implementers of a standardized API should take care to complete or finalize the OpenAPI definition before it can be served at the point of service for the API.

In particular three aspects should at least be taken in consideration:

- redefinition of the *server* element in the OpenAPI definition file;
- elaboration of the *version* element, which could be extended to identify the proprietary implementation while stating which standardized version it conforms to. More information on versioning approaches is available in clause 4.6;
- documentation of custom or proprietary features of the API (resources, methods, data elements) that extend the standardized API. Such extensions may be decorated in OpenAPI with a schema extension such as *x-etsi-proprietary-capability* or - when the extension comprises many features - a custom complementary OpenAPI file. The use of the first solution is shown in the example in Figure 4.7-1.

```yaml
/res1: # Standardized API feature
  # Content excluded
/res2: # Proprietary API feature
  description: ...
  x-etsi-proprietary-capability:
  - ACMEAuthSystem
  # Content excluded
```

Figure 4.7-1: Usage example for the proprietary capability extension

5 Testing methodology for REST APIs

5.1 Overview

Testing methodologies are well established within ETSI and beyond. With the adoption of RESTful APIs in various domains, ISGs, and partnership projects, such as ETSI ISG NFV, ETSI ISG MEC, and oneM2M™, guidelines and frameworks for the testing of RESTful APIs specific to each group started to emerge, based on more general principles and guidelines established by TC MTS and ISO/IEC. Within these newly emerging guidelines, there is no clear distinction between the general guidelines, guidelines specific to the testing of RESTful APIs, and guidelines specific to the domain. Additionally, some of the guidelines within one ISG (e.g. ETSI GR MEC-DEC 025 [i.24]) refer to guidelines by another ISG (e.g. ETSI GS NFV-TST 002 [i.25]). While this may reduce some redundancies, it also creates complex interdependencies between the specifications by the different ISGs that may result in unforeseen challenges in the evolution and use of these guidelines.

Clause 5 of the present document collects principles, conventions, and guidelines for the testing of RESTful APIs from ETSI groups and from the industry in order to provide consolidated guidelines serving as a unified future reference for ETSI groups to streamline the process of identifying relevant guidelines and providing domain specific extensions where applicable.

In the following, information and guidelines gathered from the existing sources is consolidated and reorganized across the following dimensions:

- **General:** fundamental testing principles and guidelines, applicable to testing in a broader scope, regardless of technology and architectural style, primarily based on MTS guides.
- **RESTful API-specific:** principles and guidelines specific to the testing of RESTful APIs, in particular also such that are applicable to RESTful APIs specified by means of OpenAPI. as recommended by the methodology described in clause 4.3.
- **Domain-specific:** principles and guidelines related to a specific technology and application domain, including specialization and extension points for the general and the RESTful API-specific guidelines, including pointers to further information examples in existing domain-specific guidelines that can be used as a starting point for guidelines in new domains.

5.2 Testing Frameworks and Methodologies

Given the frequent mixed use of the terms *testing frameworks* and *testing methodologies*, there is some need for establishing the basic terminology. Beyond the specifics of test implementation and automation frameworks, a *testing framework* comprises general guidelines of *what* needs to be done without a particular focus on the *how*. As such, while frameworks provide some structure and direction, they can be inherently ambiguous, leaving a lot of freedom without specific guidance on the individual activities necessary to obtain the desired results. A *testing methodology*, on the other hand, adds the necessary detail describing *how* individual tasks should be performed, including processes, deliverables, rules, methods, notations, conventions, etc. As such, a methodology helps ensure that the necessary activities can be performed in a coherent, consistent, and repeatable manner, resulting in predictable quality of the outcomes, while reducing the impact of variability in the development process and the risks of potential defects as resulting from it. Both frameworks and methodologies can be tailored to various extents to accommodate the needs of a specific context or a specific domain, while maintaining the fundamental principles.

Fundamentally, a general testing framework covers the following aspects:

- **Test subject and test environment:** Identification of the implementations under test (IUT) for conformance testing and the device under test (DUTs) for interoperability, i.e. answering the question *"what is to be tested"*.
- **Test procedures:** Definition of the applicable test procedures, i.e. answering the question *"how is it to be tested"*.
- **Test development:** Definition of the procedure for development of test specifications and deliverables (for instance: Test Purposes (TPs) in case of conformance testing and Test Descriptions (TDs) in case of interoperability testing, documentation, etc.).

- **Test deployment and execution:** Definition of the procedures for the deployment, execution, and evaluation of tests.

Within this framework, a testing methodology applied to a specific domain typically provides further domain-specific guidelines and refinements, including the following:

- Definition of a **test architecture**:
    - definition of a generic System Under Test (SUT) architecture;
    - selection of SUT and specification of SUT configurations for different scenarios;
    - identification of reference points, test interfaces, test environment, and test drivers.
- Definition of **notations and conventions**:
    - selection and/or definition of structured notations for test architectures, TPs. TDs, TCs;
    - style guidelines and examples;
    - naming and notation conventions;
    - traceability links between the individual elements of a test specification.
- Definition of a **documentation structure and process**:
    - catalogue of capabilities in the form of ICS and/or IFS;
    - high-level grouping of tests in the Test Suite Structure (TSS);
    - high-level test designs in the form of TPs and/or TDs;
    - detailed test specifications in the Abstract Test Suite (ATS).
- Definition of the **testing and validation process**:
    - steps for deployment;
    - steps for execution;
    - steps for evaluation of the test results.

Based on the general guidance, the individual guidelines can be tailored further to accommodate the needs of the specific domain. The specifics of the tailoring need to be documented as part of the test development process. For the testing of RESTful APIs, in particular also for the testing of RESTful APIs specified by means of OpenAPI, the guidelines include recommendations to make use of additional assets in order to streamline the overall process.

5.3 Conformance and Interoperability Testing

5.3.1 General

Conformance Testing and Interoperability Testing are the two main and complementary testing methodologies to test standardized systems and products implementing standardized services. The basic concepts for Conformance Testing and Interoperability Testing are defined as follows:

- **Conformance Testing** can show that a product or service correctly implements a particular standard, that is, it establishes whether or not the product or service meets the requirements regarding protocol message contents and formats as well as the permitted sequences of messages.
- **Interoperability Testing** can demonstrate that a product or service will work with other similar products in a defined environment. It proves that **end-to-end functionality** between (at least) two products or services embedded in a defined environment is as required by the standards on which those products or services are based.

- **Interoperability Testing with Conformance Checks** can provide both the proof of conformance and the guarantee of interoperation, where interfaces between the products or services are monitored to verify the appropriate sequence and contents of protocol messages, API calls, interface operations, etc.

5.3.2 RESTful API-specific

Conformance testing for RESTful APIs focuses on validating the correct implementation of an API with respect to the correct functionality including URIs, accepted requests, provided responses, as well as concerns related to authorization and authentication, filtering, pagination, and concurrent updates. Test abstractions may focus on the functional aspects where details related to serialization and adaptation are left to adaptation layers. It is recommended that conformance test include both valid request and erroneous ones. OpenAPI™ specifications can serve as the basis for the specification of conformance tests.

Interoperability testing for RESTful APIs goes beyond the individual API implementations, assuming that they are conforming to the specifications, and focuses on validating the correct functionality of different implementations of the same API embedded in a larger operational context. Tests trigger a chain of API requests and responses between the involved components including the implementations under test to ensure that the end-to-end functionality is not affected by the different implementations of the API. OpenAPI™ specifications do not provide any indication of how implementations are embedded in an operational context. While OpenAPI™ specifications can provide the specification of the end-user APIs, additional artefacts are necessary to capture the operational context.

5.3.3 Domain-specific

Domain-specific refinements focus on specific aspects pertaining to the application domain with regard to the needs of conformance testing and interoperability testing. As such they should be defined on top of the general methodology and captured in the resulting documents.

5.4 Test Specification Development

5.4.1 General

Traditionally, the test specification development process involves multiple stages of refinement as outlined in Figure 5.4.1-1 (based on ETSI EG 203 130 [i.4]). As in the original figure, the boxes describe development steps and the ellipses describe the corresponding outcomes, e.g. "Requirements Catalogue" is the result of the step "Identification of Requirements". Typically, each step builds on the outcomes of the previous step. In some scenarios, certain steps, e.g. "Specification of Test Purposes" or "Specification of Test Descriptions" may be skipped based on the guidelines within the specific context or the type of test specifications, e.g. interoperability testing or conformance testing. Refer to ETSI EG 203 130 [i.4] and ETSI EG 201 015 [i.8] for a detailed description of the individual steps and their outcomes.

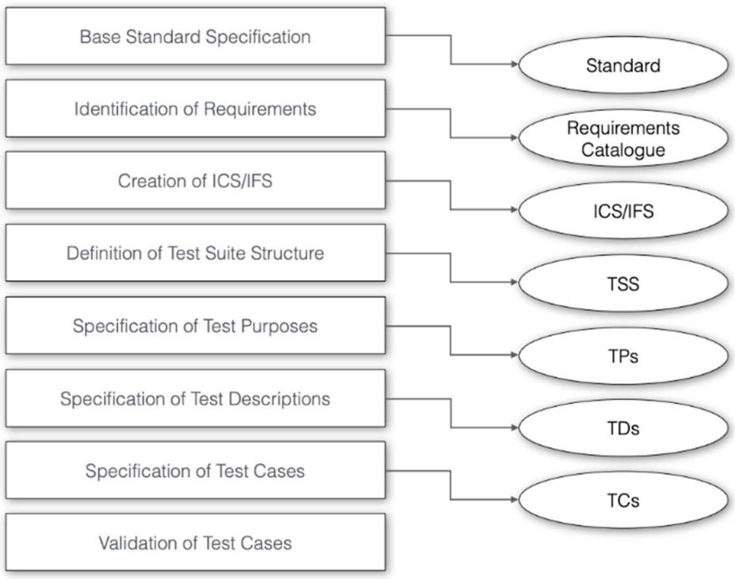

Base Standard Specification
Identification of Requirements
Creation of ICS/IFS
Definition of Test Suite Structure
Specification of Test Purposes
Specification of Test Descriptions
Specification of Test Cases
Validation of Test Cases
Standard
Requirements Catalogue
ICS/IFS
TSS
TPs
TDs
TCs

Figure 5.4.1-1: ETSI Test Specification Process (based on ETSI EG 203 130 [i.4])

Given a base specification, the recommended first step is to identify, collect, and categorize requirements in a structured requirements catalogue. This step may include collecting excerpts from the base specification indicating specific statements regarding the expected behaviour, making implicit requirements explicit, grouping and/or tagging the requirements based on various criteria, indicating their applicability and relationships among them, and assigning unique identifiers for referencing and traceability. Detailed guidelines can be found in ETSI EG 202 568 [i.26].

The Implementation Conformance Statements (ICSs) and Interoperable Function Statements (IFSs) identify the IUT features and options to be tested for conformance and interoperability respectively. They are created based on the requirements and provisions in the specifications. The capabilities are specified in the ICS to indicate which options need to be tested and to assess the achieved coverage by an IUT. In addition to the required capabilities, supported optional and conditional capabilities may be indicated by the provider of the IUT. The indicated capabilities may influence the selection and the parameterization of tests during the test deployment and execution.

NOTE: The acronym IFS may also refer to Interoperable Feature Statement, Implementable Functions Statement and other similar terminology, all referring to the identification of a communication behaviour which has relevance for successful interoperability among communicating entities. The list of usages of IFS in different ETSI specification may be retrieved using the TEDDI tool at the ETSI Portal [i.46].

The ICSs are typically structured in a tabular form and categorized based functional groups and other aspects. A standardized template for specifying and indicating ICS is recommended, which should follow the recommendations contained in ETSI EG 201 058 [i.9].

The Test Suite Structure (TSS) defines a tree-like structure where the nodes represent test groups which either contain subgroups (i.e. other non-leaf nodes) or test documents (i.e. leaf nodes), such as test purposes, test descriptions, or test cases. The TSS may be based on functional grouping in the requirements catalogue, but also consider further criteria, such as the kind of testing (valid, invalid, timing, etc.)

The Test Purposes (TPs) describe the objective of a test for a given requirement or a set of related requirements in a well-defined manner, i.e. what needs to be tested and under which conditions. The TPs do not provide additional details on how the tests need to be performed. TPs should be understandable for a wide spectrum of involved stakeholders. As such, TPs are typically specified in a prose-like (semi-) structured text, often presented in a tabular form for better readability. Message sequence charts and/or additional tables may be included for further clarification.

A TP typically contains a TP header and a behaviour description. The TP header may contain a TP identifier, a description of the objective or the requirement, external references, as well as applicability with regard to ICSs. The behaviour description may contain initial conditions, the expected behaviour, and the final conditions. The TDL-TO ETSI ES 203 119-4 [i.27] specification is the recommended way to specify TPs as it provides a suitable notation to capture TPs in a consistent, structured, machine-readable manner. The TOP project ([i.13]) provides tool support for working with TPs specified in TDL-TO which can be exported in the desired tabular presentation based on customisable templates.

Test Descriptions (TDs) are specified in addition to (or instead of) TPs to provide more detailed description of how a test is to be performed, including individual test steps that need to be followed. TDs can be an intermediate test design step between the higher-level declarative TPs and the very detailed test case implementations. TDs are typically used in the context of interoperability testing to describe actions that need to be performed on/by endpoint devices. Regardless of whether the tests are to be performed manually or in an automated manner, a precise and unambiguous notation is recommended for the specification of TDs. The steps in the test description can be of different nature, not strictly related to the IUT as they may involve different participants in a testing scenario. The TDL ETSI ES 203 119-1 [i.28] specification is the recommended way to specify TDs as it provides suitable notations to capture TDs in a consistent, structured, machine-readable manner. The TOP project provides tool support for working with TDs specified in TDL. Different presentations can be derived from the underlying TDL models. The more detailed TDs can then be transformed into executable test cases more easily as the abstraction level gap is much smaller than that of TPs and test case.

Based on the TPs and/or TDs, detailed Test Cases (TCs) are defined as part of an Abstract Test Suite (ATS). ATSs are typically specified in TTCN-3 ETSI ES 201 873-1 [i.29]. The test suite is the basis for automated conformance and interoperability testing. With the precise specification of the functional details of the tests, there is an implicit evaluation of whether individual requirements are expressed in a clear, precise, and unambiguous manner.

As more and more test-related details are specified along the process, it is often necessary to define generic test architectures and test configurations specific to certain scenarios. The overall architecture may already be provided in the base specification indicating the involved components and the communication paths between them. Test configurations indicate the roles of the involved components for specific test scenarios. Test configurations may also be useful in determining the TSS.

With the growing need for more streamlined and faster specification and testing cycles in order to respond to dynamic market and user needs, the fundamental test specification development process needs to be adapted as well. While the existing stages are well understood and widely established, the ways in which they are implemented are in need of improvement. With the increasing adoption of machine-readable documents, new opportunities for faster and more efficient test specification development begin to emerge. It is recommended to capitalize on the availability of such machine-readable documents and expand their use throughout the test specification development process so that traceability, consistency checking, and maintainability can be improved by automated and tool-supported means, reducing the need for manual interventions to a minimum.

5.4.2 RESTful API-specific

The introduction of OpenAPI™ specifications as an outcome of the *Base Standard Specification* step can benefit all subsequent test specification development steps. Normative OpenAPI™ specifications provide machine readable description of the essential requirements, along with additional annotations to add structured domain-specific information. An overview of how the availability of OpenAPI™ specification impacts the development steps is shown in Figure 5.4.2-1.

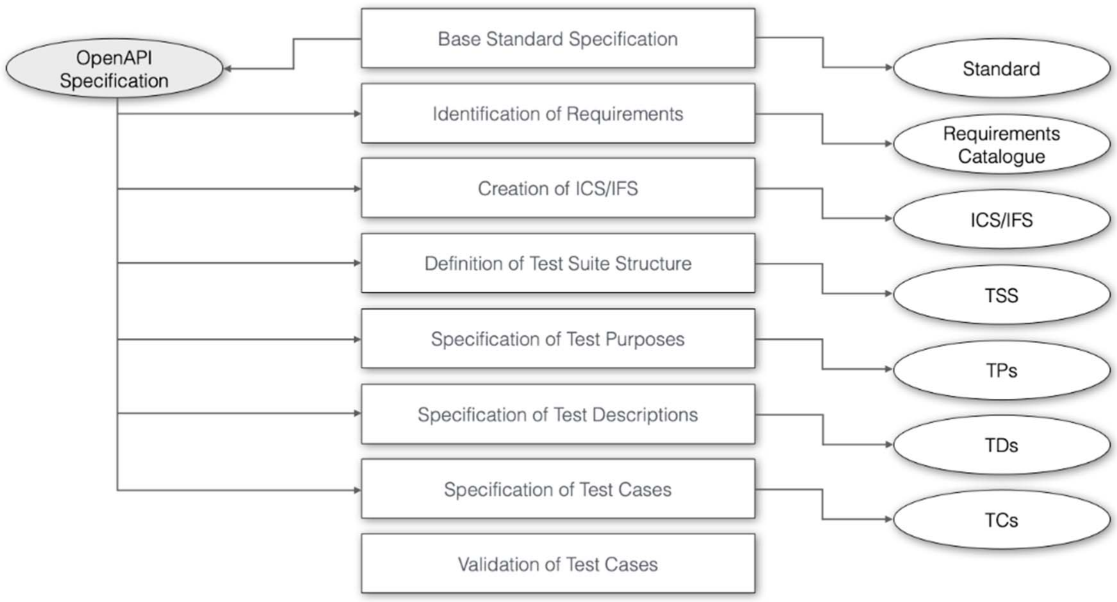

OpenAPI Specification
Base Standard Specification
Identification of Requirements
Creation of ICS/IFS
Definition of Test Suite Structure
Specification of Test Purposes
Specification of Test Descriptions
Specification of Test Cases
Validation of Test Cases
Standard
Requirements Catalogue
ICS/IFS
TSS
TPs
TDs
TCs

Figure 5.4.2-1: RESTful API Test Specification Process with OpenAPI

Given that the requirements are already indicated in a structured format with the help of the OpenAPI™ specification, the OpenAPI artefacts can be utilized to streamline the subsequent steps in the process. Consistent specifications based on well-defined conventions can be processed by tools to extract and transform the relevant information into templates for manual refinement or complete documents as input for or output from the individual steps in the process. Thus, the requirements can be derived (semi-) automatically from the OpenAPI™ specifications, thereby reducing the effort required for the identification of requirements. The resulting requirements catalogue can either be a by-product of the process for reference, or also be captured in a machine-readable format in order to ease the subsequent steps in the process. Following the guidelines in ETSI EG 202 568 [i.26], based on the OpenAPI™ specification fragments in clauses 4.3.2.4 to 4.3.2.5, the requirements can be extracted as shown in Table 5.4.2-1.

Table 5.4.2-1: Extracted requirements

Identifier	Reference	Type	Applicability	Requirement	Context
RQ_RESOURCE_GET_001	RESTful API Guide, clause 4.3.2.4; /resource/{id}	M	RESOURCE	Read full contents of a resource with specific ID (description from OpenAPI™)	(relevant OpenAPI™ specification fragment)
RQ_RESOURCE_POST_001	RESTful API Guide, clause 4.3.2.5; /resource	M	RESOURCE	Create new resource (description from OpenAPI™)	(relevant OpenAPI™ specification fragment)
RQ_RESOURCE_PUT_001	RESTful API Guide, clause 4.3.2.5; /resource/{id}/file	M	RESOURCE	Upload a file for a resource (description from OpenAPI™)	(relevant OpenAPI™ specification fragment)

Beyond the requirement identifiers, all the information is essentially already contained in the OpenAPI™ specification in this case. The references in this case point to the relevant clauses in the present document. They may also refer to base documents, RFCs, or other documents and artefacts. One requirement may also contain multiple references. This applies also to references in ICSs, TPs, TDs, and TCs. The presence of extensions as illustrated in clause 4.3.2.9 needs to be interpreted accordingly. If no additional provisions are indicated by means of extensions, the requirements are assumed to be mandatory.

During the creation of the ICSs, with appropriate tooling ICS templates can be generated based on information extracted from OpenAPI™ specification. Assuming that IUT providers supply an OpenAPI™ specification of the supported capabilities, these can be matched against the standardized OpenAPI™ specification to indicate the supported optional and conditional in the ICS.

Following the guidelines in ETSI EG 202 568 [i.26], based on the OpenAPI™ specification fragments in clauses 4.3.2.4 to 4.3.2.5, the ICSs can be extracted as shown in Table 5.4.2-2.

Table 5.4.2-2: Extracted ICSs

ID	Resource	Reference	Method	Type	Request	Response
1	/resource/{id}	Clause 4.3.2.6	GET	M	(request with prescribed id)	200
2	/resource/{id}	Clause 4.3.2.6	GET	M	(request with protected id)	401
3	/resource/{id}	Clause 4.3.2.6	GET	M	(request with non-existent id)	404

Similar to the requirements, all the information for the ICSs is essentially provided in the OpenAPI™ specification with the exception of the request descriptions. They may be populated with placeholders with concrete values defined in separate tables. The request may be filled with generic descriptions derived from the resource, method, and response code. It may need to be refined manually subsequently to provide more precise description or example. The presence of extensions as illustrated in clause 4.3.2.9 needs to be interpreted accordingly. If no additional provisions are indicated by means of extensions, the ICSs are assumed to be mandatory.

It may be desirable to provide ICSs at different levels of detail. In their most detailed form, the ICSs would include a detailed breakdown of all methods and all responses for all resources, as exemplified in Table 5.4.2-2. More concise representations of the ICSs may group together all responses for a given method on a given resource, or even all methods and responses for a given resource in a hierarchical manner, such that only in the case of deviations or additional provisions indicated by means of extensions more detailed information needs to be provided. A more concise ICS grouping all responses for the different methods is shown in Table 5.4.2-3. Similarly, all methods for a given resource may be grouped under a single ICS as shown in Table 5.4.2-4, or represented hierarchically as a tree, as shown in Table 5.4.2-5.

Table 5.4.2-3: Extracted higher (method) level ICSs

ID	Resource	Reference	Method	Type	Request	Response
M1	/resource/{id}	Clause 4.3.2.6	GET	M	(request resource with id)	200, 401, 404
M2	/resource	Clause 4.3.2.5	POST	M	(create new resource with id and version)	201, 400
M3	/resource	Clause 4.3.2.5	PUT	M	(update a resource with id)	200, 201, 204, 400

Table 5.4.2-4: Extracted higher (path) level ICSs

ID	Resource	Reference	Method	Type	Request	Response
P1	/resource	Clause 4.3.2.6	GET, POST, PUT	M	(request, create, and update resource)	200, 201, 204, 401, 404
P2	/resource/{id}/file	Clause 4.3.2.5	PUT	M	(upload a resource file)	200, 201, 204, 400

Table 5.4.2-5: Extracted hierarchical ICSs

ID	Resource	Reference	Method	Type	Request	Response
P1	/resource	Clause 4.3.2.6	GET, POST, PUT	M	(request, create, and update resource)	200, 201, 204, 401, 404
P1_M1	/resource/{id}	Clause 4.3.2.6	GET	M	(request resource with id)	200, 401, 404
P1_M1_R1	/resource/{id}	Clause 4.3.2.6	GET	M	(request with prescribed id)	200
P1_M1_R2	/resource/{id}	Clause 4.3.2.6	GET	M	(request with protected id)	401
P1_M1_R3	/resource/{id}	Clause 4.3.2.6	GET	M	(request with non-existent id)	404
P2_M2	/resource	Clause 4.3.2.5	POST	M	(create new resource with id and version)	201, 400
P2_M2_R1	/resource	Clause 4.3.2.5	POST	M	(create new resource with id and version)	200
…	…	…	…	…	…	…
P2_M3	/resource	Clause 4.3.2.5	PUT	M	(update a resource with id)	200, 201, 204, 400
P2_M3_R1	/resource	Clause 4.3.2.5	PUT	M	(update a resource with id)	200
…	…	…	…	…	…	…
P3	/resource/{id}/file	Clause 4.3.2.5	PUT	M	(upload a resource file)	200, 400
P3_M1	/resource{id}/file	Clause 4.3.2.5	PUT	M	(upload a resource file)	200, 201, 204, 400
…	…	…	…	…	…	…

A hierarchical presentation allows supported capabilities to be bundled together and only exceptions to be marked as such explicitly. For example, if *P1* is only partially implemented, the implemented parts in the sub-tree need to be marked as such explicitly, while *P1* itself is not marked. If *P2* on the other hand is fully implemented, only *P2* is marked as such explicitly and all parts in the sub-tree do not need to be marked.

The TSS may consider mirroring inherent structures in the OpenAPI™ specification, e.g. grouping by APIs, resources and/or methods. Appropriate tooling can generate an initial TSS based on the machine-readable specification and pre-set criteria for the desired TSS format.

Based on the OpenAPI™ specification fragments in clauses 4.3.2.4 to 4.3.2.5, the following TSSs shown in Figure 5.4.2-2 can be extracted. It shows the API as the (sub-) tree root, followed by the paths, methods, and method-response combinations. The exact representation may vary depending on the constraints and conventions in use (e.g. not using spaces, all capitals, etc.). For easier traceability, it is recommended to keep the representation as close to the OpenAPI as possible. TSSs covering multiple APIs and/or broader application contexts may embed the TSS derived from the OpenAPI in a larger structure.

```
- API: Examples for RESTful API guide
  - RESOURCE
    - GET
      - GET_200
      - GET_401
      - GET_404
    - POST
      - ...
    - PUT
      - ...
```

Figure: 5.4.2-2: Extracted TSS

TPs can be derived from the OpenAPI™ specification in a straightforward manner as the specification of the expected behaviour in the structured TDL-TO notation mirrors the request-response declarations in the OpenAPI™ specification. With appropriate tooling, TP skeletons can be generated automatically from the OpenAPI™ specification and refined further manually, (if needed). This can streamline the very labour-intensive creation and maintenance of the TPs. Based on the OpenAPI™ specification fragments in clauses 4.3.2.4 to 4.3.2.5, the following TPs for valid and non-existent IDs can be defined to exemplify the TDL-TO:

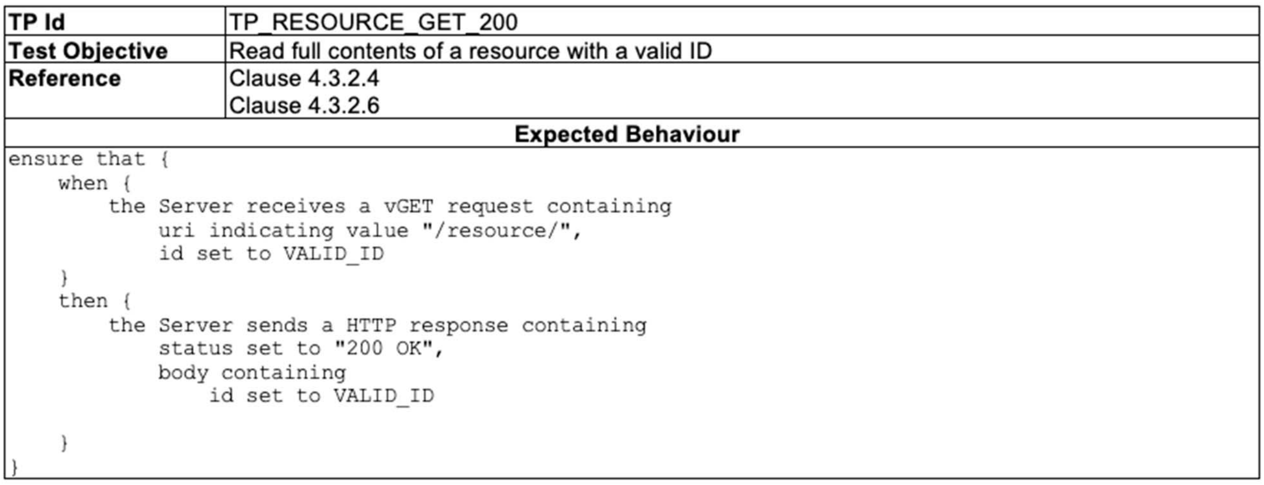
```
Test Purpose {
    TP Id TP_RESOURCE_GET_200
    Test objective "Read full contents of a resource with a valid ID"
    Reference "Clause 4.3.2.4",  "Clause 4.3.2.6"
    Expected behaviour
    ensure that {
        when {
            the Server entity receives a vGET request containing
                uri indicating value "/resource/",
                id set to VALID_ID;
        }
        then {
            the Server entity sends a HTTP response containing
                status set to "200 OK",
                body containing
                    id set to VALID_ID;
                ;
        }
    }
}
```

Figure 5.4.2-3: Example TP for valid ID

```
Test Purpose {
    TP Id TP_RESOURCE_GET_404
    Test objective "Check for correct response when requesting a non-existent resource"
    Reference "Clause 4.3.2.4",  "Clause 4.3.2.6"
    Expected behaviour
    ensure that {
        when {
            the Server entity receives a vGET request containing
                uri indicating value "/resource/",
                id set to NONEXISTENT_ID;
        }
        then {
            the Server entity sends a HTTP response containing
                status set to "404 Not found";
        }
    }
}
```

Figure 5.4.2-4: Example TP for non-existent ID

While the TP for the valid ID can be derived directly from the OpenAPI™ specification, with some placeholders (e.g. *VALID_IT*), the test objective in the TP for the non-existent ID already needs to be manually adjusted to better describe the TP based on the specific response. If a defined pattern is put in place, the descriptions can be derived automatically as well in this case. Beyond that, TPs emphasize the most important and relevant aspects of a test, leaving out additional implementation details such as authentication, content negotiation, etc., unless these are explicitly the objective of the test. The additional details may need to be provided at a later stage during the test implementation. The TPs can be presented in a tabular format as illustrated in Figure 5.4.2-5.

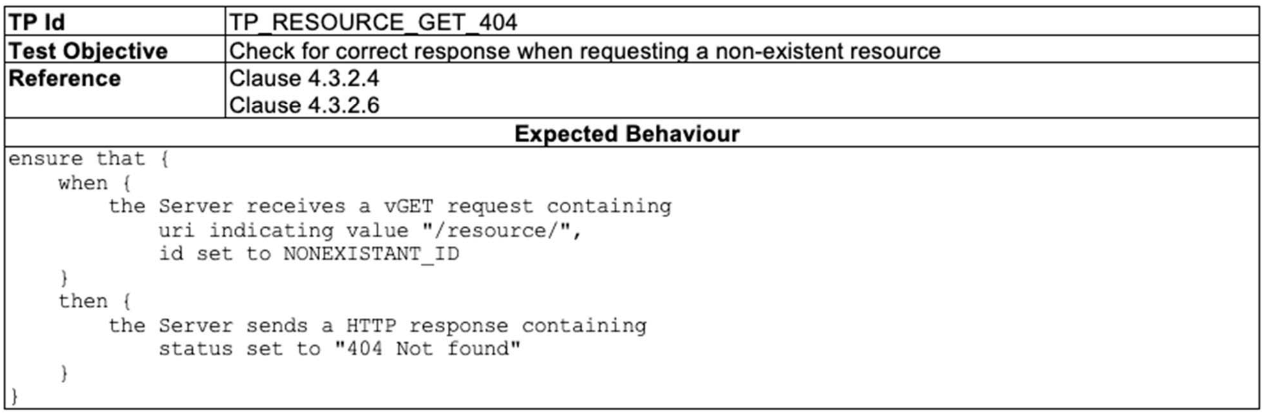


TP Id	TP_RESOURCE_GET_200
Test Objective	Read full contents of a resource with a valid ID
Reference	Clause 4.3.2.4 Clause 4.3.2.6

Expected Behaviour

```
ensure that {
    when {
        the Server receives a vGET request containing
            uri indicating value "/resource/",
            id set to VALID_ID
    }
    then {
        the Server sends a HTTP response containing
            status set to "200 OK",
            body containing
                id set to VALID_ID

    }
}
```



TP Id	TP_RESOURCE_GET_404
Test Objective	Check for correct response when requesting a non-existent resource
Reference	Clause 4.3.2.4 Clause 4.3.2.6

Expected Behaviour

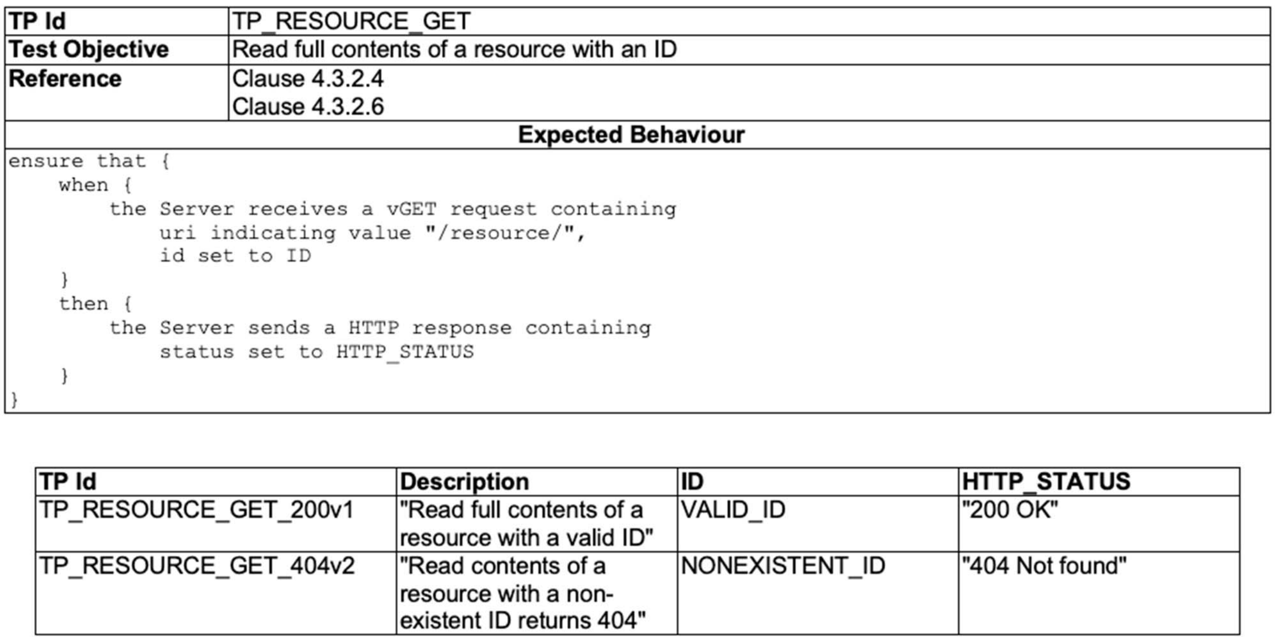
```
ensure that {
    when {
        the Server receives a vGET request containing
            uri indicating value "/resource/",
            id set to NONEXISTANT_ID
    }
    then {
        the Server sends a HTTP response containing
            status set to "404 Not found"
    }
}
```

Figure 5.4.2-5: Tabular presentation of example TPs

As large chunks of the TPs may remain identical for e.g. different request-response combinations, the use of TP variants can reduce duplication and improve conciseness and reuse by specifying a generic TP with variants providing concrete combinations of data elements and overriding the meta-information, e.g. to refine the test objective or the PICS selection. An example combining the two TPs into one TP with variants is shown in Figure 5.4.2-6. The corresponding tabular presentations are shown in Figure 5.4.2-7.

```
Test Purpose {
    TP Id TP_RESOURCE_GET
    Test objective "Read full contents of a resource with an ID"
    Reference "Clause 4.3.2.4",  "Clause 4.3.2.6"
    Expected behaviour
        ensure that {
            when {
                the Server entity receives a vGET request containing
                    uri indicating value "/resource/",
                    id set to ID;
            }
            then {
                the Server entity sends a HTTP response containing
                    status set to HTTP_STATUS;
            }
        }
    Variants
        TP_RESOURCE_GET_200v1 {
            Test objective "Read full contents of a resource with a valid ID"
            Bindings
                value ID set to VALID_ID;
                value HTTP_STATUS set to "200 OK";
        }
        TP_RESOURCE_GET_404v2 {
            Test objective "Read contents of a resource with a non-existent ID returns 404"
            Bindings
                value ID set to NONEXISTENT_ID;
                value HTTP_STATUS set to "404 Not found";
        }
}
```

Figure 5.4.2-6: Example reusable TP with variants



TP Id	TP_RESOURCE_GET
Test Objective	Read full contents of a resource with an ID
Reference	Clause 4.3.2.4 Clause 4.3.2.6

Expected Behaviour

```
ensure that {
    when {
        the Server receives a vGET request containing
            uri indicating value "/resource/",
            id set to ID
    }
    then {
        the Server sends a HTTP response containing
            status set to HTTP_STATUS
    }
}
```

TP Id	Description	ID	HTTP_STATUS
TP_RESOURCE_GET_200v1	"Read full contents of a resource with a valid ID"	VALID_ID	"200 OK"
TP_RESOURCE_GET_404v2	"Read contents of a resource with a non-existent ID returns 404"	NONEXISTENT_ID	"404 Not found"

Figure 5.4.2-7: Tabular presentation of example TPs with variants

TDs can be derived from the OpenAPI™ specification directly or through the intermediate use of TPs in TDL-TO. The request-response declarations in the OpenAPI™ specification can be represented as interactions and visualised with TDL-GR ETSI ES 203 119-2 [i.30] and the TOP tools [i.13]. With appropriate tooling, TD skeletons can be generated automatically from the OpenAPI™ specification or from the TPs in TDL-TO and refined further manually, (if needed). This can streamline the very labour-intensive creation and maintenance of the TDs. The TDs can be very detailed, including inline declarations of the requests and responses, as illustrated in Figure 5.4.2-8, or they may rely on predefined declarations which abstract away the details and enable reuse, as shown in Figure 5.4.2-9, as well as graphically in Figures 5.4.2-10 and 5.4.2-11.

```
Test Description TD_RESOURCE_GET_200_Inline uses configuration BasicClientServer
{
    client.http sends GET(
        uri = "/resource/{id}",
        parameters = {
            new Parameter(^name="id", value = ValidId, location = path)
        }
    )
        to server.http;

    server.http sends OK(
        status = "200",
        body = new ResourceData (
            id = ValidId
            //other properties (created, size) omitted as they are not relevant for the TD
        )
    )
        to client.http;
} with {
    test objectives : TP_RESOURCE_GET_200;
}
```

Figure 5.4.2-8: Example TD with inline request/response declarations

```
Test Description TD_RESOURCE_GET_200 uses configuration BasicClientServer
{
    client.http sends getValidResource to server.http;
    server.http sends okWithValidResource to client.http;
} with {
    test objectives : TP_RESOURCE_GET_200;
}

Test Description TD_RESOURCE_GET_404 uses configuration BasicClientServer
{
    client.http sends getNonExistantResource to server.http;
    server.http sends NotFound to client.http;
} with {
    test objectives : TP_RESOURCE_GET_404;
}
```

Figure 5.4.2-9: Example TDs with reusable predefined request/response declarations

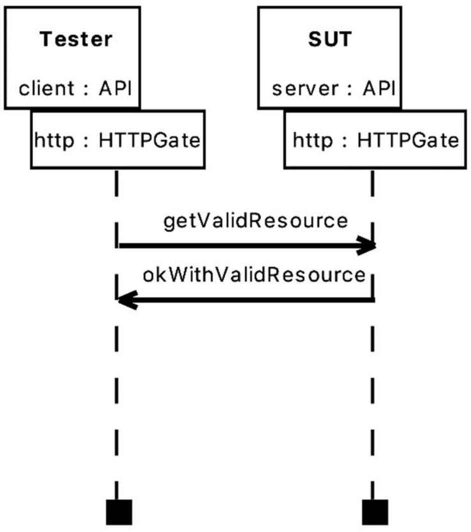


Figure 5.4.2-10: Graphical presentation of test description with valid ID

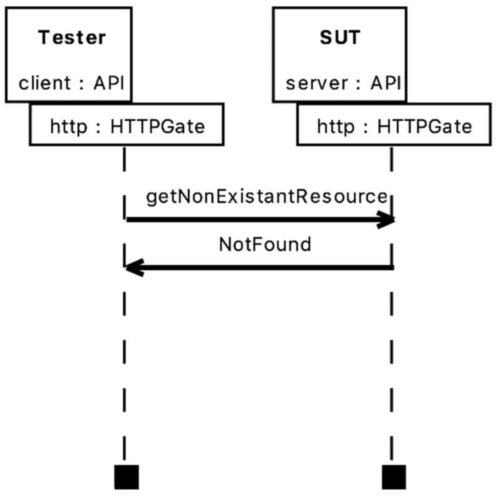


Figure 5.4.2-11: Graphical presentation of test description with non-existent ID

If the overall TD behaviour is identical with the exception of small variations in the details in the requests and responses, parameterized TDs can be defined and reused, similar to how the variants can reduce duplication in TPs. An example of a generic parameterized TD and a TD referencing the parameterized TD with different parameters is illustrated in Figure 5.4.2-12.

```
Test Description TD_RESOURCE_GET (
    ID of type String,
    STATUS of type String
) uses configuration BasicClientServer
{
    client.http sends GET(
        uri = "/resource/{id}",
        parameters = {
            new Parameter(^name = "id", value = parameter ID, location = path)
        }
    )
        to server.http;

    server.http sends new Response(
        status = parameter STATUS
    )
        to client.http;
} with {
    test objectives : TP_RESOURCE_GET;
}

Test Description TD_RESOURCE_GET_All uses configuration BasicClientServer
{
    execute TD_RESOURCE_GET(ID = ValidId, STATUS = "200") with {
        test objectives : TP_RESOURCE_GET_200;
    };
    execute TD_RESOURCE_GET(ID = NonExistantId, STATUS = "404") with {
        test objectives : TP_RESOURCE_GET_404;
    };
} with {
    test objectives : TP_RESOURCE_GET, TP_RESOURCE_GET_200, TP_RESOURCE_GET_404;
}
```

Figure 5.4.2-12: Example reusable parameterized TD

TDs can be defined and organized at different levels of detail, depending also on the TSS. TDs may target individual responses for a method on a resource, combine all responses for a method in a single TD referencing the TDs for the individual responses (as illustrated in the examples in Figure 5.4.2-12), combine all the methods for a resource, or even all the resources for an API. Additionally, they may be organized according to valid and error-handling behaviours. TDL provides the means to compose and reuse TDs according to the TSS and the specific needs in a given context.

Data types defined in an OpenAPI™ specification declare the structures of requests, parameters, and responses. The data type definitions are used as the basis for the implementation of the APIs. It is recommended that the same data type definitions are also used for the development of test specifications directly. This implies direct references to the OpenAPI™ specifications for establishing traceability, validating compatibility, and maintaining consistency as the specifications evolve. The traceability chain needs to be established and maintained throughout the test specification development process, ideally in a machine-readable manner so that appropriate tools can be utilized to support the continuous maintenance and validation of the test specifications in accordance with API specifications. Data types and data instances in TDL specified as abstract symbols. Data element mappings establish formalized relationships to concrete data representations or external specifications, which may be defined in external resources, such as TTCN-3, XML, or JSON documents. The use of external resources is declared by means of *data resource mappings*.

It is important to emphasize that data specifications in TDL and TDL-TO focus on the data elements that are of relevance for the specification of the tests, not necessarily for their implementation and operationalisation. Through data element mappings, TDL provides means to abstract away test implementation details in the data specifications. These details, such as additional headers, parameters, etc., may be necessary for the implementation, but if they have no influence on the test behaviour, they may be omitted from the TPs and TDs to keep them simple and focused on the essentials, as illustrated in some of the examples. A library of generic data type and data instance definitions for use in HTTP TPs and TDs with TDL is available as part of the examples in clause 7.

In conformance tests, the test architectures and test configurations for RESTful APIs are usually straightforward, involving the API producer as the SUT and the API consumer as the tester. In interoperability tests, different implementations of an API are embedded in often complex operational contexts, which need to be well-defined and documented. The OpenAPI™ specification only provides a description of the capabilities of the API producer. It is important to select an appropriate and consistent notation with clear semantics to indicate the involved components their interfaces and roles, and the communication paths between them. TDL provides means for the formalized specification of test configurations which are also used in the specification of the TPs and the TDs, enabling a consistent, reusable, cross-linked definitions. This can streamline the development and maintenance of test specifications, where any changes from the base specifications can be easily propagated throughout the documentation chain for the test specifications. An example for a minimal test configuration is illustrated in Figure 5.4.2-13 and Figure 5.4.2-14. The component and gate types need to be declared in advance.

```
Test Configuration BasicClientServer
{
    create Tester client of type API;
    create SUT server of type API;
    connect client.http to server.http;
}
```

Figure 5.4.2-13: Test configuration example

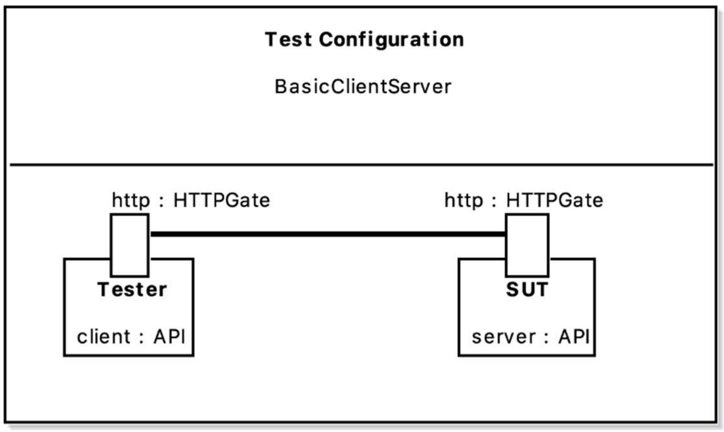


Figure 5.4.2-14: Graphical presentation of test configuration example

The detailed TCs in Abstract Test Suite (ATS) can be derived manually or (semi-) automatically from the TPs or TDs (e.g. using ETSI ES 203 119-6 [i.31] for a standardized TDL to TTCN-3 mapping). Alternative technologies for the implementation of RESTful API tests may be considered as well, however, it is important to recognize that many technologies emerge and disappear very quickly. Hence, it is recommended to use an established and standardized technology which will likely last as long as the test specifications are maintained and used. The OpenAPI™ specifications can be used to derive codecs for TTCN-3 as part of the adaptation layers which are relevant for the deployment and execution of the test cases. ETSI ES 201 873-11 [i.32] provides instructions on the use of JSON with TTCN-3 which can be helpful for the implementation test cases for RESTful API. A standardized mapping for HTTP may be of further benefit. An example illustrating the essential parts of a TTCN-3 test case based on the TDs in Figure 5.4.2-9 and the test configuration in Figures 5.4.2-13 and 5.4.2-14 is shown in Figure 5.4.2-15. The example is derived using the provisions in ETSI ES 203 119-6 [i.31] for the standardized TDL to TTCN-3 mapping. It includes a test case as well as the related functions for setting up the test configuration and for the behaviour. A library of generic data type and data instance definitions for use in HTTP TCs with TTCN-3 is available as part of the examples discussed in clause 7.

```
function setupTestConfiguration_BasicClientServer ( ) runs on MTC_BasicClientServer {
    client := API.create ;
    map ( client : http_to_server_http , system : server_http ) ;
}

function f_RESOURCE_GET_200 ( ) runs on MTC_BasicClientServer {
    client.start ( f_RESOURCE_GET_200_client_main ( ) ) ;
}

function f_RESOURCE_GET_200_client_main ( ) runs on API {
    http_to_server_http.send ( getValidResource ) ;
    http_to_server_http.receive ( okWithValidResource ) ;
}

testcase tc_RESOURCE_GET_200 ( ) runs on MTC_BasicClientServer system SYSTEM_BasicClientServer {
    setupTestConfiguration_BasicClientServer ( ) ;
    f_RESOURCE_GET_200 ( ) ;
    all component.done ;
}
```

Figure 5.4.2-15: Derived test case example and supporting functions for behaviour and configuration

5.4.3 Domain-specific

The RESTful API testing guidelines can be refined and tailored further to accommodate the needs of specific domain. This adds another domain-specific layer of guidelines, which need to be documented as part of the test development process. Domain-specific guidelines and conventions may be related to the specific technology and application domain. Examples for such conventions can be found in e.g. ETSI GR MEC-DEC 025 [i.24], ETSI GS NFV-TST 002 [i.25] and ETSI GR NFV-TST 007 [i.33], or ETSI TS 118 115 [i.34] and oneM2M TS-0018 [i.35]. They may span naming and structuring guidelines for the TSS as well as recommendations for the specific notation conventions in TPs and TDs. The present document provides guidelines at the more generic level of RESTful APIs which can serve as a baseline for future domain-specific guidelines. The existing domain-specific guidelines may serve a starting point for domain-specific refinements in new domains.

5.5 Test Deployment and Execution

OpenAPI™ specifications provide a description of the capabilities of individual API producers. Depending on the type of test, there may be multiple API producers involved. Traditionally, informal graphical representations are used for descriptive purposes, which then need to be translated into instructions for manual or automated configuration. The formalized test configurations early on can help describe complex operational contexts in a consistent way and serve as the basis for the preparation and configuration of the test environments during test deployment and execution, while also providing means to visualise the configurations for documentation and communication purposes. Test configurations may start at a very abstract level and be refined in a stepwise manner throughout the process. Known common aspects of the target deployment and execution environment may be taken into consideration already during the early design stages to avoid test designs that may be difficult to deploy. However, tests should be described in an abstract manner so that they can be deployed and executed against different target systems.

Test plan preparation is essential for the effective and efficient test deployment and execution. Test plans typically include the tests to be included, the order and grouping of the tests and potential relationships among them, the necessary configuration and preparation for each group of tests, the required technical and human resources for the test deployment and execution. Test automation can help ensure consistency in the testing environments and test results. Formalized and machine-readable assets can facilitate the adoption of integrated toolchains around standardized procedures, e.g. for generating stubs, adapters, and mappings based on OpenAPI™ specifications, as well as based on TDL and TTCN-3 specifications. Standardized libraries, e.g. for HTTP or other protocols can provide well-defined targets for tools to work with, while and facilitate interoperability between different tools at the various stages.

5.6 Test Maintenance and Evolution

As the specifications for products evolve, the corresponding tests need to be maintained as well, both to address changes to the specifications as well as to improve the tests themselves. In order to facilitate the co-evolution of tests along the systems and services they are designed to test, a structured process needs to be established. During maintenance and evolution, the process discussed in clause 5.4 typically undergoes multiple iterations. The use of OpenAPI™ specification along with the guidelines for RESTful API testing outline the foundation for the formalized linking between different assets enabling the use of toolchains to streamline the process and reduce maintenance overhead while increasing consistency between the different assets throughout the process. For every iteration, change impact assessment needs to be performed, identifying what can be reused, what needs to be updated, and which steps need to be carried out in order to minimize the risk of regressions and inconsistencies. With supporting tools, potential inconsistencies as a result of changes can be quickly identified or even prevented in case the tools are holistically integrated in the test development process where changes in the OpenAPI™ specification are directly reflected in the all the test-related assets in all steps of the process. Automation during the test deployment and execution is essential for ensuring consistent testing environments, reliable results, and quick feedback during the iterations. The automation itself may need to evolve as well over time, so it is essential to document and maintain automation-related assets as part of the process es well.

6 Tooling recommendations

6.1 Introduction

In clause 4 of the present document, the benefits of formal languages for specification of communication interfaces are highlighted. RESTful APIs are an exemplary case that shows large adoption of formal specifications, enabling several degrees of automation for a variety of use cases.

This large adoption has naturally converged towards the availability of a myriad of tools to parse, validate, process and transform formal definitions of RESTful interfaces, particularly for the most common languages such as OpenAPI.

Therefore, the authors intend to provide the readers with a few pointers to available tools to support the soundness of the proposed methodology and to serve as a reference for the interested users. Unless specified otherwise, the tools presented in the following clauses are not developed nor endorsed by the authors of the guide.

The list focuses on tools that provide support OpenAPI™ specifications.

6.2 Design and drafting

6.2.1 Overview

A major blocking issue in the drafting of an OpenAPI definition is the correctness of the syntax and the validity of the OpenAPI metamodel applied. For the user who is not often drafting OpenAPI™ specification by hand this usually causes an inefficient trial and error process. To maximize the efficiency and minimize entry barrier for new users in the standardization environment it is recommended not to draft the OpenAPI definitions by hand in their textual syntax.

Instead a graphical tool should be used to edit the specification and to serialize it into the OpenAPI textual syntax. A set of third-party tools supporting this activity are listed below.

A fairly comprehensive list of editors and design environments for OpenAPI is available at [i.36], in particular the sub-lists of graphical editors and textual editors.

6.2.2 Recommendations on editing tool selection

A tool feasible to be utilized in a RESTful APIs standardization context should cover all the following recommendations:

- It should have the capability to export the API definition in OpenAPI™ 3.0.
- It should have the capability to interact with external versioning systems, i.e. coordination should be done in a way which is independent from the tool used to design the API.
- It should have the capability to resolve remote definitions correctly (e.g. include remote data type definitions).

6.3 Coordination and collaboration

The OpenAPI definition are represented and exchanged mainly in the form of YAML or JSON text files. Therefore, in order to coordinate distributed development and maintenance of such files a Version Control Systems (VCS) should be used. VSCs enable efficient version tracking, concurrent workflows and semi-automated synchronization. Among the several VCSs available and well established in the industry, the Git [i.52] versioning system is recommended.

ETSI provides hosting for Git repositories through the ETSI Forge platform [i.11], thus offering a solution for standardization activities that need to coordinate different users in OpenAPI development. The use of ETSI Forge to collaborate upon, store and track OpenAPI™ specifications is recommended, in particular for the following benefits:

- Specific guidance is provided on how to integrate documents and digital attachments with ETSI specifications.
- The platform is integrated in the ETSI ecosystem and user services (e.g. it is accessible via the ETSI online credentials).
- The platform includes a set of application and services that are well known and documented thus easing the discovery, acquisition and adoption of digital attachment to standards by external users and general public.

Usage of public Git hosting solutions is not recommended. Instead, mirroring techniques on such public platforms are recommended, to allow wider discovery and consumption of the standardized materials.

6.4 Validation and quality check

When synchronization of different contributions happen, syntax and semantic errors may be introduced, even when the individual contributions were correct. In order to ensure the quality of the final standardized API, automated validation should be applied to every new revision of the API definition, at the moment in which the contribution happens.

EXAMPLE: In the context of API definitions tracked with VCS repositories, this implies execution of validation at each contributed commit.

The introduction of errors in individual contributions is certainly mitigated if design tools are used (as recommend in the previous clause) but this does not remove the need for a centralized and unbiased validation of the contributed definitions.

Different types of validations may be applied, including:

- Syntactical correctness (e.g. YAML or JSON syntaxes).
- Validity against the metamodel (i.e. matching the schema of OpenAPI).
- Linting or validation of conventions and best practices.

While the first two validation types are required and defined by the OpenAPI™ specification, the application and definition of the third type is optional and agreed by the group producing the API specification.

Examples of automated OpenAPI validation as applied for different standardization groups within ETSI are:

- Validation applied in 3GPP projects sources [i.37].
- Validation applied in MEC projects example [i.38].
- Validation applied in NFV projects example [i.39].

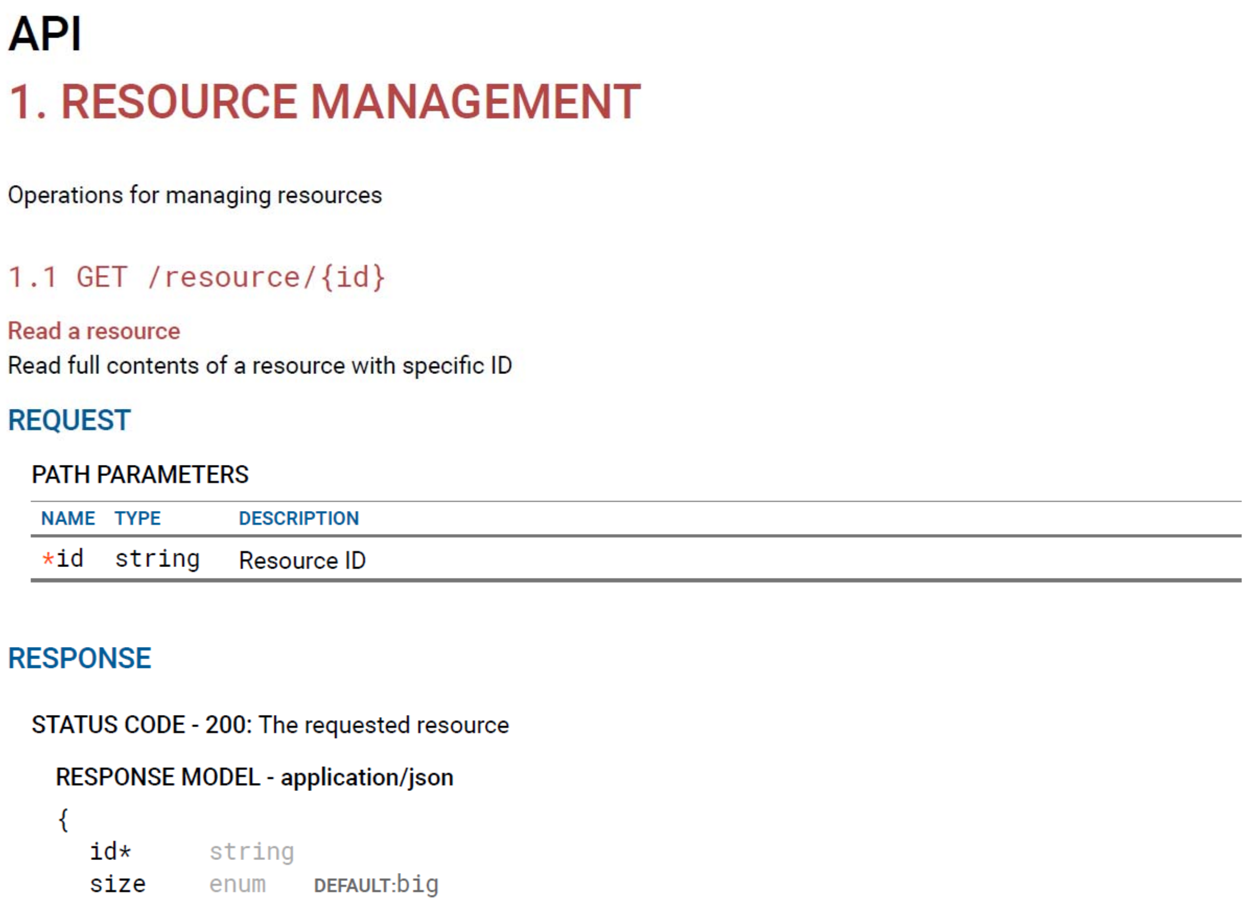
Finally, while the selection of design tools may be left to the individual contributors (as long as the tool complies with the OpenAPI™ specification), the selection of automated validators should be agreed among the contributors since it provides a common ground for valid contributions. As such, it needs to be a solution that may be acquired and executed by all the participants as well as in the common IT infrastructure such as ETSI Forge. Therefore, the usage of free and open source tools is recommended.

6.5 Post processing

Once the OpenAPI definition files are finalized by the standardization group, they may be post-processed to provide different options to consume them. Example of post-processing activities are:

- resolution of all JSON references in the OpenAPI definition file to generate a standalone and easily portable file;
- generation of human readable in-browser documentation, either static or dynamically constructed; or
- generation of human readable "print-outs" of the OpenAPI in portable document formats to facilitate offline review.

Several tools are freely available to generate documental exports of the OpenAPI definitions. An example of such an export (generated with the RapiPDF tool [i.40]) is available in Figure 6.5-1.


API

1. RESOURCE MANAGEMENT

Operations for managing resources

1.1 GET /resource/{id}

Read a resource

Read full contents of a resource with specific ID

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	string	Resource ID

RESPONSE

STATUS CODE - 200: The requested resource

RESPONSE MODEL - application/json

```
{
  id*     string
  size    enum    DEFAULT:big
```

Figure 6.5-1: PDF export example

7 Working Examples

More comprehensive examples related to the snippets discussed in clause 4 and clause 5 are available in [i.10]. They include an OpenAPI™ specification, test purpose and test description definitions in TDL (including supporting data and configuration definitions), as well as test cases in TTCN-3 and Robot Framework. The examples focus on HTTP. They can be used as basis for other transports. Further examples for OpenAPI™ specifications and test purpose and test description definitions are available on the ETSI Forge [i.11] and ETSI Labs [i.12].

The examples related to the present document in [i.10] comprise the following:

- Example OpenAPI™ specification located in the *OpenAPI* folder.
- Example requirements, implementation conformance statements and test suite structure generated from the OpenAPI™ specification located in the *RQ-ICS-TSS.md* file (also available as Word document).
- Example test purposes and related resources in TDL-TO located in the *TP* folder, with automatically generated TP skeletons in the *Generated* sub-folder and manually derived TPs in the *Manual* sub-folder.
- Example test descriptions and related resources in TDL located in the *TD* folder, libraries including *Standard* and *HTTP* definitions for TDL are located in the *Library* sub-folder, with automatically generated TD skeletons in the *Generated* sub-folder and manually derived TDs in the *Manual* sub-folder, including data and behaviour packages for the example. Graphical representations with TDL-GR are also included.
- Example test cases and related resources in TTCN-3 located in the *TC* folder, with automatically generated TTCN-3 skeletons in the *Generated* sub-folder and manually derived complete TTCN-3 test cases in the *Manual* sub-folder.
- Example test cases and related resources for Robot Framework are located in *Robot* folder, with JSON schemas in *schemas* sub-folder.

To make the best out of the test purposes in TDL-TO and the test descriptions in TDL, it is recommended to use the TOP toolset available at [i.13]. Further up-to-date information regarding the examples is available in the *README.md* file.

8 Survey of Activities at ETSI and Beyond

8.1 Review of base documents

8.1.1 ETSI GS MEC 009 (V2.1.1)

ETSI GS MEC 009 [i.23] 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. While the recommendations are intended to be technology implementation independent, the focus is on HTTP which is fully specified in the recommendations.

The list of recommendations in the document include conventions and best practices related to the specification structure (including Purpose, URIs + versions, Methods, Representations, Request and Response schemas, Links, Status codes), the specification of entry points (version, supported features, collections, resources, etc.), as well as security and privacy considerations (flow, anonymisation, authorization).

Further templates and examples illustrate naming conventions, paths, and queries. The definition of supplementary OpenAPI™ specifications is recommended, however the base specifications always have precedence. Seventeen patterns for common operations are described. Normative templates and informative sequence diagrams are outlined.

8.1.2 ETSI GR MEC-DEC 025 (V2.1.1)

ETSI GR MEC-DEC 025 [i.24] specifies a testing framework defining a methodology for development of interoperability and/or conformance test strategies, test systems and the resulting test specifications for MEC standards and lists and prioritizes the testable requirements. It builds upon ETSI GS NFV-TST 002 [i.25], referencing content with applicable extensions and modifications.

It covers generic information regarding conformance and interoperability testing and indicates provisions regarding capabilities for ICS.

For conformance testing, it recommends informal TPs written in prose, optionally including graphical, tabular, or MSC contents for clarification. It recommends the use of TDL-TO for the specification of test purposes in a structured manner and provides some conventions and best practices.

For interoperability testing, it recommends the use of TDs in a tabular format and defines an interoperability test process for requirements assessment based on provisions from specifications.

8.1.3 Draft ETSI GS MEC-DEC 032-1 (V0.0.3)

ETSI GS MEC-DEC 032-1 [i.53] applies the testing methodology guidelines and framework specified in ETSI GR MEC-DEC 025 [i.24]. It is a two part document where Part 1 specifies conformance test related information for the MEC service APIs including test requirements and Implementation Conformance Statement (ICS) and Part includes the Test Suite Structure (TSS) and Test Purposes (TPs) using TDL-TO.

NOTE: ETSI GS MEC-DEC 032-1 [i.53] is still in the drafting stage as of the time of writing, therefore all comments are preliminary.

8.1.4 ETSI GS CIM 009 (V1.2.1)

ETSI GS CIM 009 [i.54] defines a standard API for Context Information Management enabling close to real-time access to information coming from many different sources, including performing updates on context, registering context providers, querying information on current and historic context information, and subscribing to receive notifications of context changes.

It outlines to three prototypical architectures (centralized, distributed, federated) where the APIs should enable efficient support for all of them. Notable aspects of the specification include managing multi-attributes, temporal representations and properties, and geospatial properties. Query languages are defined for filtering entities and context sources, as well as filtering based on temporal and geospatial properties.

The specification provides some data representation restrictions and further conventions.

8.1.5 ETSI GS QKD 014 (V1.1.1)

ETSI GS QKD 014 [i.55] describes a communication protocol and data format for a Quantum Key Distribution (QKD) network to supply cryptographic keys to an application in order to allow interoperability of equipment from different vendors. While the QKD network can consist of a single link between a single QKD transmitter and a single QKD receiver, or it can be an extended network involving many such QKD links, the API defines a single interface for the delivery of key material to applications in both scenarios. The specification includes the data formats and the methods described in tabular format but no methodological information regarding the specification of the APIs.

8.1.6 ETSI TS 129 501 (V15.3.0)

ETSI TS 129 501 [i.21] document defines the design principles and documentation guidelines for the RESTful 5GC SBI APIs. These principles are used for drafting stage 3 specifications for the 5G system. It provides the facilities for design Principles for REST implementation including Rest API designs, the requirement for secure API design and REST implementation levels. The specifications include URI Conventions, resource modelling by using 4 different archetypes and provides the information about the changes in the API that are considered as backward compatible and those that are considered as backward incompatible. Backward compatible changes are additions or changes in the API that do not break the existing Service Consumer behaviour. While backward-incompatible changes are additions or changes in the API that break the existing Service Consumer behaviour. An example of OpenAPI™ Specification files is also provided. However, the base specifications always have precedence.

8.1.7 ETSI GS NFV-SOL 013 (V2.7.1)

ETSI GS NFV-SOL 013 [i.42] specifies common aspects of RESTful protocols and common data models for NFV MANO interfaces specified in NFV SOL specifications (ETSI GS NFV-SOL 002 [i.58], ETSI GS NFV-SOL 003 [i.59] and ETSI GS NFV-SOL 005 [i.60]). It provides normative provisions regarding the HTTP usage (URI structure, Header fields), Result set control procedures, Effective error reporting mechanism, Authorization of API requests (OAuth 2.0, TLS certificate) and API versioning (Semantic versioning; Major.Minor.Patch). These provisions are referenced from the ETSI NFV SOL API specifications.

8.1.8 ETSI GS NFV-SOL 015 (V1.1.1)

ETSI GS NFV-SOL 015 [i.22] describes patterns and conventions for RESTful NFV-MANO API specifications, gives recommendations on API versioning and provides an API specification template. The provisions include the Naming conventions (Name, Strings, and URIs) and Patterns of HTTP methods related to CRUD (Create, GET, Update and Delete with HTTP methods), Non-CRUD (Task resource) and Asynchronous (with monitoring and without monitoring) operations.

ETSI GS NFV-SOL 015 [i.22] defines provisions to be followed by the ETSI NFV Industry Specification Group (ISG) when creating RESTful NFV-MANO API specifications. The provisions do not apply to implementations.

The main difference between ETSI GS NFV-SOL 013 [i.42] and ETSI GS NFV-SOL 015 [i.22] is that ETSI GS NFV-SOL 013 [i.42] specifies common implementation level details intended to be referenced from the individual API specifications while the recommendations provided in ETSI GS NFV-SOL 015 [i.22] apply to the creation of API specifications and are intended to be implementation independent.

8.1.9 ETSI GS NFV-TST 010 (V2.4.1)

The goal ETSI GS NFV-TST 010 [i.56] is to specify the methodologies of conformance test including test descriptions for NFV implementations with interfaces specified in the following NFV specifications: ETSI GS NFV-SOL 002 [i.58] for the Ve-Vnfm, ETSI GS NFV-SOL 003 [i.59] for the Or-Vnfm and ETSI GS NFV-SOL 005 [i.60] for the Os-ma-nfvo reference point. ETSI NFV SOL deliverables specify a set of interfaces built on the RESTful approach and meant to be used over the HTTP protocol. The document defines the methodologies and the procedures with test descriptions to test the conformance of the exchanged HTTP payloads and the implementation of required actions for one or more of the available interfaces within a reference point.

The purpose of general conformance testing is to determine to what extent a single implementation of a particular standard conforms to the individual requirements of that standard. The document defines the System Under Test (SUT), Test Configurations and test Descriptions for the conformance testing of NFV SOL specification.

8.1.10 ETSI TS 118 115 (V2.0.0)

ETSI TS 118 115 [i.34] specifies a testing framework defining a methodology for development of conformance and interoperability test strategies, test systems and the resulting test specifications for oneM2M standards. The oneM2M testing framework consists of a documentation structure for the main artefacts (catalogue of capabilities, test suite structure, test purposes) and a methodology with concrete guidelines, conventions, and notations. The described notation for test purposes provides some structure and keywords for the specification of test purposes but does not refer to a standardized notation. The guidelines for the use of TTCN-3 for the abstract test suite specification indicate how to implement the abstract test architecture, setup test configurations, other conventions. The guidelines for interoperability testing focus more on test architectures test descriptions. The test description notation includes notations at an abstract (primitive) level as well as different concrete protocol levels. Relying on such a notation without adequate tool support is likely to produce a substantial amount of duplicated content for the different concrete protocol levels which need to be kept consistent.

8.1.11 oneM2M TS-0018 (V3.2.0)

oneM2M TS-0018 [i.35] specifies test suite structure and test purposes that are designed to evaluate the conformity of oneM2M implementations to the oneM2M specifications. It also provides guidelines and notations for the description of test purposes, test behaviours, and test configurations for conformance testing. The guidelines and notations are based on ETSI TS 118 115 [i.34]. The test purposes make use of so-called "variants", where specific test purposes are derived from the same base test purpose where abstract placeholders (including meta-information) are replaced by more specific descriptors in each variant.

8.1.12 TM Forum Open APIs initiative

The Tele Management Forum (TM Forum) is leading an activity to design common RESTful APIs, called the Open API initiative. The list of specified APIs is available at https://projects.tmforum.org/wiki/display/API/Open+API+Table.

In support of such activity, the Forum has developed recommendations and guidelines for common REST patterns (such as CRUD and Task operations) which are documented in TMF630 API Design Guidelines 4.0.1 [i.57] https://www.tmforum.org/resources/how-to-guide/tmf630-api-design-guidelines-4-0/. The Guidelines focus on modelling different operational patterns with a REST paradigm, from CRUD operations to task operations, monitoring, notifications, polymorphism, hypermedia support and other patterns.

To define the rationale and main approach behind the Open APIs program TM Forum members drafted the Open API Manifesto https://www.tmforum.org/open-apis/open-api-manifesto/. The manifesto also includes the commitments asked to the signees, in terms of participation and adoption of TM Forum developed APIs.

TM Forum also runs a certification program for Open APIs adopters, based on a set of test suites published by TM Forum itself, defined Conformance ToolKit (CTK).

8.1.13 OMG hData RESTful Transport

The Object Management Group (OMG) is widely known for its Unified Modelling Language™ (UML®). The OMG hData RESTful Transport document https://www.omg.org/spec/Hdata/About-Hdata/ defines remote operations for accessing components of a Health Record and sending messages to an Electronic Health Record (EHR) system.

The hDatadocument defines general conventions and provides recommendations regarding default response codes, compression, content negotiation, versioning, handling intermediaries, rejecting update operations because of integrity concerns or business rules, as well as recommended HTTP headers. It does not address data modelling in any form as hData is designed to be able to transport clinical data of any type. It does not use ETags.

The specification of the operations and their semantics is provided in natural language with little structuring. A reliable operation pattern is described for scenarios where reliable transfer of information is required. Such scenarios require that *"both sender and service provider have a confirmation that the other side has successfully received the information exactly once"*. The use of the reliable operation pattern is indicated and negotiated by using designated headers. The mechanism for the implementation of the pattern requires that resources are locked until the pattern completes or a given timeout has occurred, which inevitably breaks the statelessness of the service. The document describes provisions for baseline security based on TLS, as well as recommendations for custom security mechanisms.

8.1.14 OASIS OData v4.01

OData or Open Data Protocol is an application-level protocol aims to define a way for creating and consuming queryable and interoperable RESTful APIs and interaction with data feed resources using RESTful services. It is built on technologies like HTTP, ATOM/XML, and JSON. OData is standardized by OASIS and approved as an ISO/IEC International Standard. At present, it is in version 4.01. OData services are described by an entity-relationship model and provide two data management models: Entity Data Model (EDM) and Service Model. OData tries to enable information to be accessed from a variety of sources including relational databases, file systems, content management systems, and traditional Web sites.

OpenAPI™ Specification (OAS) has the objective of creating a vendor-neutral, portable, and open specification for describing REST APIs. On the other hand, Open Data Protocol (OData), OData defines specification for creating data services over HTTP. OData services are described by an entity-relationship model. It intends to specify the format and patterns to construct the Web APIs which is convenient to expose and query data sources as REST APIs.

8.2 API adoption survey

In preparation for the development of the present guide, in December 2019 ETSI STF 576 https://portal.etsi.org/STF/STFs/STF-HomePages/STF576 has conducted an online survey on "Current activities related to the specification, implementation and testing of RESTful APIs". The survey was closed in January 2020 and the results have been presented in a public webinar in the same month.

The survey was conducted among several ETSI TBs and ISGs (including NFV, MEC, ZSM, 3GPP, SmartM2M, etc.) for the collection of information on current REST API specification activities and related guidelines specified in the individual groups. The objective of the survey has been to learn from experiences of groups within ETSI as well as from other SDOs on the Specification and Testing of REST APIs. This consolidation activity helped in developing and producing the present document meant to provide guidance to standardization groups that adopt the RESTful paradigm and to harmonize and align their REST API specification methodologies. The survey focused on the following areas:

- Importance of REST APIs
- Challenges affecting the specification of REST APIs
- Essential testing activities
- Important topics to be considered in the guide
- Specifications of common aspects of REST APIs

The survey Participants belonged to different domains i.e. telecoms, IoT, smart cities, and system engineering and management. Among the survey respondents, 95 % considered REST APIs substantially **critical** to their organizations. While the other 5 % considered REST APIs as somewhat important.

In the survey, participants highlighted some of the major **challenges** they faced in the specification of REST APIs:

- inconsistency of and between documents; and
- the growing size of documents.

71 % and 59 % of Survey respondents declared **Interoperability** and **Conformance testing**, respectively, as essential testing activities.

When asked respondents the Important **Topics** to be covered in the RESTful API Guide, participants provided a wide range of topics. Some of the highlighted topics were:

- Validation of API specifications
- Selection and usage of tools
- Examples of standard deliverables (TPs, TDs, ATS)

Several questions were included in the survey for collecting and documenting the best practices in the specifications of common aspects of REST APIs. The following responses proved the most popular:

- 97 % of the respondents reported using JSON as data serialization and exchange format;
- for version management, 68 % use API versioning as a URI path parameter with a version number;
- when asked the question about handling the large query result set, rather split of answers were recorded among several multiple choices:
  - 76 % said they use attribute-based filtering of collections;
  - 50 % preferred attribute selector (limit attributes included in the response);
  - 47 % use the paged response approach;
- 71 % of respondents considered the "OAuth 2.0" preferred authentication scheme;
- 44 % use ETag as a common practice to avoid loss in the concurrent update of the same resource, while, 35 % never considered it as a problem;

- for asynchronous communication methods, 79 % chose "Subscribe-notify" (Server-to-Server) approach while 47 % chose web sockets;
- regarding information for the specification of test configurations for conformance/interoperability, 50 % chose the "Client-server" option.

In conclusion the survey received good participation and provided precious input for the development of the present document.

Annex A (informative): Bibliography

- "Towards conformance testing of REST based Web Services", Lo Iacono and Nguyen.
- Open API™ Alliance.

NOTE: Available at https://www.openapis.org.

Annex B (informative): Change History

Date	Version	Information about changes
January 2020	0.0.1	Incorporates STF contributions as per development available at https://forge.etsi.org/rep/stf/stf-576/mts-203647-methodology-for-restful-apis-specifications-and-testing/tree/v0.0.1 (access may be required).
May 2020	0.0.2	Incorporates STF contributions as per development available at https://forge.etsi.org/rep/stf/stf-576/mts-203647-methodology-for-restful-apis-specifications-and-testing/tree/v0.0.1 (access may be required).
July 2020	0.0.3	Incorporates STF contributions as per development available at https://forge.etsi.org/rep/stf/stf-576/mts-203647-methodology-for-restful-apis-specifications-and-testing/tree/v0.0.1 (access may be required).

History

Document history		
V1.1.1	September 2020	Membership Approval Procedure MV 20201106: 2020-09-07 to 2020-11-06
V1.1.1	November 2020	Publication