UN/CEFACT

UNITED NATIONS

Centre for Trade Facilitation and Electronic Business

(UN/CEFACT)

METHODOLOGY AND TECHNOLOGY PROGRAMME DEVELOPMENT AREA

SPECIFICATIONS DOMAIN

OPENAPI NAMING AND DESIGN RULES TECHNICAL SPECIFICATION

SOURCE: API TechSpec Project Team

ACTION: Ready for publication

DATE: 13 September 2022

STATUS: **v1.0**

Abstract

This OpenAPI Naming and Design Rules technical specification defines an architecture and a set of rules necessary to specify, describe and implement APIs based on an OpenAPI specification to consistently express business information. It is based on the OpenAPI specification and the UN/CEFACT Core Components Technical Specification. This specification describes the requirements that UN/CEFACT compliant APIs should fulfil. It will be used by other organisations who are interested in maximizing inter- and intra-industry interoperability.

Abstract ... *2*
1.1 Document History ... 5
1.2 Change Log ... 5
1.3 OpenAPI Naming and Design Rules Project Team ... 6
1.4 Acknowledgements ... 6
1.5 Contact information ... 6
1.6 Notation ... 6
1.7 Audience ... 7
2 INTRODUCTION ... 8
2.1 Objectives ... 8
2.2 Requirements ... 8
2.3 Dependencies ... 8
2.4 Caveats and Assumptions ... 8
2.5 Guiding Principles ... 9
2.6 Interoperability ... 10
3 API NAMING AND DESIGN RULES ... 13
3.1 Conformance and Compliance ... 13
3.2 Design Rules ... 14
3.2.1 Media type for structured data exchange ... 14
3.2.2 Endpoints ... 14
3.2.3 Discoverability ... 16
3.2.4 Date and Time ... 16
3.2.5 Using the UN/CEFACT semantics ... 16
3.2.6 Operations ... 21
3.2.7 Pagination ... 23
3.2.8 Filtering ... 26
3.2.9 Sorting ... 27
3.2.10 API Responses and error handling ... 28
3.2.11 Error Response Payload ... 33
3.2.12 Design rule examples ... 35
4 WELL-DOCUMENTED APIS ... 36
4.1 General considerations ... 36

4.2 API VERSIONING 36
4.2.1 Versioning Scheme 36
4.2.2 URI Versioning 37
4.2.3 Providing version information 37
4.2.4 Robustness 39
4.2.5 Deprecation and End of Life Policy 40
4.3 HYPERMEDIA 43
4.3.1 Hypermedia - Linked Data 43
4.3.2 HATEOAS 43
4.3.3 Hypermedia Compliant API 44
5 API SECURITY 46
6 EVENT DRIVEN DATA EXCHANGE 48
6.1 CALLBACKS 48
6.2 WEBHOOKS 49
6.3 SECURITY GUIDELINE FOR CALLBACKS (INFORMATIVE) 49
6.3.1 Subscription setup (informative) 50
6.3.2 Performing a subscription call (informative) 51
7 APPENDIX A: EXAMPLES 53
8 APPENDIX B: NAMING AND DESIGN RULES LIST (NORMATIVE) 54
9 APPENDIX C: GLOSSARY 58

1.1 Document History

Phase	***Status***	***Date Last Modified***
Draft development	First draft	06 September 2022
Ready for approval	First version	13 September 2022

Table 1 – Document history

1.2 Change Log

The change log is designed to alert users about significant changes that occurred during the development of this document.

Date of Change	**Version**	**Paragraph Changed**	**Summary of Changes**
30 May 2022	0.3		First draft TOC
07 June 2022	0.4		Drafted up to chapter 3.2.7
	0.5		Drafted up to chapter 3.2.9
20 June 2022	0.6		Completion up to chapter 6
05 Sept 2022	0.7	1.6 2.6 R 1 R 16 6.3 7 Appendix A Appendix B Appendix C	Considering public review comments
13 Sept 2022	1.0		Minor corrections

Table 2 - Document change log

1.3 OpenAPI Naming and Design Rules Project Team

We would like to recognize the following for their significant participation in the development of this Unites Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) OpenAPI Naming and Design Rules technical specification.

ATG2 Chair

Marek Laskowski

Project Lead

Jörg Walther

Lead editors

Andreas Pelekies Gerhard Heemskerk

1.4 Acknowledgements

This version of UN/CEFACT OpenAPI Naming and Design Rules Technical Specification has been created to foster convergence among Standards Development Organisations (SDOs). It has been developed in close coordination with these organisations:

- Digital Container Shipping Association
- GS1
- Odette

1.5 Contact information

ATG2 – Marek Laskowski, Marek.laskowski@gmail.com
NDR Project Lead – Jörg Walther, jwalther@odette.org
Editor – Andreas Pelekies, Andreas@pelekies.de
Editor – Gerhard Heemskerk, Gerhard.heemskerk@kpnmail.nl

1.6 Notation

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this specification, are to be interpreted as described in Internet Engineering Task Force (IETF) Request For Comments (RFC) 2119[1].

Example A representation of a definition or a rule. Examples are informative.

[1] Key words for use in RFCs to Indicate Requirement Levels - Internet Engineering Task Force, Request For Comments 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt?number=2119

[Note] Explanatory information. Notes are informative.

[R n|c] Identification of a rule that requires conformance. Rules are normative. In order to ensure continuity across versions of the specification, rule numbers “n” are randomly generated. The number of a rule that is deleted will not be re-issued. Rules that are added will be assigned a previously unused random number.
The second number “c” after the pipe symbol | identifies the conformance category of the given rule as defined in section 3.1. A `+Inf` may be added to identify rules that are informative and not normative.

`Courier` All words appearing in **`bolded courier font`** are values, objects or keywords. Representation of non-printable characters like white space are surrounded by double-quotes, e.g. `" "`.

`<<var>>` All placeholders are surrounded by double less-than and greater-than characters. The meaning of the placeholder is described in the text.

1.7 Audience

The audience for this UN/CEFACT OpenAPI Schema Naming and Design Rules Technical Specification is:

- Members of the UN/CEFACT Applied Technologies Groups who are responsible for development and maintenance of UN/CEFACT OpenAPI specifications and recommendations.
- The wider membership of the other UN/CEFACT Groups who participate in the process of creating and maintaining UN/CEFACT OpenAPI specifications.
- Designers of tools who need to design OpenAPI specifications adhering to the rules defined in this document.
- Designers of OpenAPI specifications outside of the UN/CEFACT Forum community. These include designers from other organisations that have found these rules suitable for their own organisations.

2 Introduction

2.1 Objectives

This OpenAPI NDR technical specification document forms part of a suite of documents that aim to support modern web developers to make use of UN/CEFACT semantics.

Taking any layer of the UN/CEFACT Reference Data Models to create conformant OpenAPI specifications in accordance with the UN/CEFACT Core Components Technical Specification Version 2.01. This includes comprehensive RDMs like Buy-Ship-Pay, or Accounting as well as their contextualization like the Supply-Chain-Reference-Data-Model (SC-RDM), Multi-Modal-Transport-Reference-Data-Model (MMT-RDM) down to single message implementation like the Road Consignment Note (eCMR) or the certificate of origin (COO).

2.2 Requirements

Users of this specification should have an understanding of basic data modelling concepts, basic business information exchange concepts and basic (REST) API concepts.

2.3 Dependencies

This document depends on

1. UN/CEFACT Core Components Technical Specification Version 2.01.
2. JSON Schema Naming and Design Rules Technical Specification.

2.4 Caveats and Assumptions

Specifications created as a result of employing this specification should be made publicly available as OpenAPI specification documents in a universally free, accessible, and searchable library. UN/CEFACT will make its contents freely available to any government, individual or organisation who wishes access.

Although this specification defines the data structures used in an OpenAPI specification as expressions of Reference Data Models, non-CCTS developers can also use it for other logical data models and information exchanges.

This specification does not address transformations via scripts or any other means. It does not address any other representation of CCTS artefacts – such as XML, JSON-LD, OWL, and XMI.

Standards foster interoperability. In the creation of this specification and definition of design principles, several sources were taken into account in the following order:

1. The OpenAPI 3.1.0 specification

2. Standards defined by internet standard organisations as RFCs
3. The DCSA API Design Principles 1.0
4. The json:api specification
5. Experts experience

2.5 Guiding Principles

3. OpenAPI Creation
   UN/CEFACT OpenAPI design rules support OpenAPI specification creation through handcrafting as well as automatic generation.
4. Tool Use and Support
   The design of UN/CEFACT OpenAPI will not make any assumptions about sophisticated tools for creation, management, storage, or presentation being available.
5. Technical Specifications
   UN/CEFACT OpenAPI Naming and Design Rules will be based on technical specifications holding the equivalent of OpenAPI recommendation status.
6. OpenAPI Specification
   UN/CEFACT OpenAPI Naming and Design Rules will be fully conformant with the OpenAPI specification recommendation.
7. Interoperability
   The number of ways to express the same information in a UN/CEFACT OpenAPI specification is to be kept as close to one as possible.
8. Maintenance
   The design of an UN/CEFACT OpenAPI specification must facilitate maintenance.
9. Context Sensitivity
   The design of an UN/CEFACT OpenAPI specification must ensure that context-sensitive document types are not precluded.
10. Ease of implementation
    An UN/CEFACT OpenAPI specification should be intuitive and reasonably clear in the context for which they are designed. They should allow an intuitive implementation in REST APIs, a.k.a. RESTful API, as well as other interchange appliances.

2.6 Interoperability

Decades of cross-industry and cross-national harmonisation of B2B and B2A processes have gone into the development of the semantic UN/CEFACT reference data models by thousands of experts. This tremendous achievement does not exist a second time in this scope and depth. The clear path from semantic definition to syntax - and not vice versa - means that these semantic data models are syntax-neutral and can thus be used not only with current syntaxes but also with future ones. For this purpose, either they are mapped directly into a (UN/CEFACT) syntax via NDR specifications, or they can be mapped to data models and syntaxes of other sectors.

The ideal of a REST API envisages the fully automatable connection of an API consumer to an API provider. In practice, this is often not the case today, as the corresponding standards for the design of an API, the scope and depth of the documentation and the modelling of processes and data in B2B and B2A communication via WebAPIs are still in their infancy. The keyword here is interoperability.

In classic EDI implementations (e.g. EDIFACT or XML), a variety of industry standards exist. With their help, the following dimensions of interoperability are promoted:

1. Business process interoperability: the business partners have the same understanding of the basic process flow, for example in the Order2Cash - process.
2. Semantic interoperability: the business partners have the same understanding of the technical terms. For example, the definition of consignment and shipment is the same for all business partners.
3. Syntax interoperability: a uniform syntax (e.g. UN/CEFACT XML) is used.
4. Contextualisation interoperability: Industry standards define how individual requirements are to be handled. Ideally, it is agreed that as few different contextualisations (consideration of individual requirements) as possible should take place. This means that information that is only required by some recipients will be read over by the remaining recipients instead of carrying out an individual implementation.
5. Interoperability of transmission: Business partners agree on uniform transmission methods as well as associated security measurements such as SFTP, OFTP2 or AS2. This dimension often plays a lesser role in classic EDI implementations, as the transmission of data usually takes place from one sender to one recipient at a time. EDI is usually optimised for mass data.

When implementing a WebAPI, the same requirements for interoperability exist in principle. An essential difference of previous WebAPIs is the approach to connect mass users to an API. For example, a map, route or booking service should be used by as many users as possible at the same time. The REST principle of composability also means that different services (possibly from different providers) are often combined into an overall solution for processing with WebAPIs. For example, in a flight booking service, the capacities, conditions and tickets are allocated by the airlines, payment service providers are connected, and often a specialised billing service that correctly calculates the different tax constellations for cross-border flights. The aspect that many consumers have to use one API (billing service) as well as one consumer has to use many APIs with the same processes (contingent with airlines) extends the interoperability requirements for WebAPIs.

6. Interoperability of API design: This specification deals with the aspect of API design interoperability. Uniform methods and rules in API design simplify the understanding of APIs, errors during implementation are minimised, the handling of error messages is standardised and the implication of similar APIs in a cross-organisational (B2B) network is promoted.

7. Service interoperability: uniform endpoints in mapping the same process requirements promote B2B communication via WebAPIs.

The following table shows how the seven dimensions of interoperability can be achieved in WebAPIs:

Dimension of interoperability	Guideline
Business process interoperability	Within UN/CEFACT, business process interoperability is achieved by implementing the harmonised business requirement specifications (BRS).
Semantic interoperability	The CCTS and its derived semantic Reference Data Models (RDMs) are the basis for this dimension for UN/CEFACT users. The UN/CEFACT Vocabulary, the JSON Schema artefacts, and the UN/CEFACT XML standards implement these semantic requirements in the respective syntax.
Syntax interoperability	When a user group agrees on the use of a uniform data exchange syntax, this dimension is achieved. When creating an OpenAPI specification, it should be noted that the syntax to be used must always be modelled as a JSON schema, even if the later exchange syntax is an XML format, for example. It is defined in an OpenAPI specification.
Contextualisation interoperability	An implementation guideline (for example, of a particular industry) defines how contextualisations are to be applied to the data or message structures to be exchanged.
Interoperability of transmission	This dimension is also specified in an implementation guideline. In particular, it also includes security aspects including authorisation and authentication.
Interoperability of API design	This NDR specification defines the interoperability of API design. Among others, it includes rules for filtering, pagination and error handling.
Service interoperability	A good OpenAPI specification especially focuses on service interoperability. The interoperability of APIs designed to be implemented by several business partners can be fostered if the services are well designed. For instance, a user group agrees on a

	set of services with a minimum subset. If a provider does not support a specific service it is still implemented, but always responds with a `501 Method not implemented` HTTP response code that includes a `HTTP Link Header` to the corresponding documentation

Table 3: Interoperability of WebAPIs

3 API Naming and Design Rules

3.1 Conformance and Compliance

Designers of OpenAPI specifications in governments, private sector, and other standards organisations external to the UN/CEFACT community have found this specification suitable for adoption. To maximize reuse and interoperability across this wide user community, the rules in this specification have been categorised to allow these other organisations to create conformant OpenAPI specifications while allowing for discretion or extensibility in areas that have minimal impact on overall interoperability.

Accordingly, applications will be considered to be in full conformance with this technical specification if they comply with the content of normative sections, rules and definitions.

[R 1|1]
Compliance and conformance SHALL be determined through adherence to the content of the normative sections and rules. Furthermore, each rule is categorised to indicate the intended audience for the rule by the following:

Category	Description
1	Rules, which must not be violated. Else, compliance and interoperability are lost.
2	Rules, which may be modified, while still conformant to the NDR structure. If all rules of categories 1 and 2 are followed, the API is fully compliant. If rules of category 2 are modified the API is not compliant anymore, but still conformant.
Inf	Rules that are informative only. If a different implementation is chosen this does not have any impact on the compliance and conformance of the implementation towards this specification.

Table 4 - Conformance categories

[R 2|1]
All API specifications based on this OpenAPI Naming and Design Rules technical specification SHALL be compliant to the OpenAPI 3.1.x specification.

[R 3|1]
An API specification claiming conformance to this specification SHALL define schema components as described in the JSON Schema Naming and Design Rules Technical Specification.

3.2 Design Rules

3.2.1 Media type for structured data exchange

[R 4|1]
Request body content and Response content used to transfer structured data information SHALL use the **`application/json`** media type for JavaScript Object Notation (JSON). This rule MAY only be deviated from, if the API implements a conversion service from or to JSON in another media type. Additional media types (e.g. **`text/xml`**) to transfer structured data information MAY be used. If non-structured information is transferred any valid media type MAY be used.

[R 5|1]
Encoding SHALL be UTF-8.

3.2.2 Endpoints

[R 6|2]
The structure of the paths defined within APIs SHOULD be meaningful to the consumers. Paths SHOULD follow a predictable, hierarchical structure to enhance understandability and therefore usability.

[R 7|1]

The API URLs SHOULD follow the standard naming convention as described below:

```
https://{env}.api.{dnsdomain}/v{m}/{service}/{resource}/{id}/{sub-resource}?{query}
```

The components are described as follows. If a rule is mandatory for a specific component of the URL it SHALL be applied to any conformant API specification, even if the basic URL structure is different from the one described above (e.g. if **`api`** is not used as a prefix to the **`dnsdomain`**).

- https:// SHALL be used as the web protocol.
- {env} indicates the environment (e.g. **`test`**, **`sandbox`** or **`dev`**) and is usually omitted for production environment.
- {dnsdomain} is the DNS domain of the API implementer (e.g. **`unece.org`**)
- {service} is a logical grouping of API functions that represent a business service domain (e.g. **`transport`**). The {service} component is optional.
- v{m} is the major version number of the API specification. This component SHALL be stated in the URL. It MAY be provided at a different place in the URL (e.g. as a prefix to the domain).
- {resource} is the plural noun representing an API resource (e.g. **`consignments`**)

- {id} is the unique identifier for the resource defined as a path parameter. Path parameters SHALL be used to identify a resource. This component is not part of the path if an operation is performed on a collection of the resource.
- {sub-resource} is an optional sub-resource. Only used when there are contained collections or actions on a main resource (e.g. **`consignmentItem`**).
- {query} is a list of additional parameters like filters that determine the results of a search (e.g. **`consignments?loadingPort=AUSYD`**).

[R 8|1]
The total number of characters in the URL, including the path and the query, SHALL NOT exceed 2000 characters in length including any formatting codes such as commas, underscores, question marks, hyphens, plus or slashes.

[R 9|1]
Endpoints SHALL NOT be actions. Services and resources SHALL consist of nouns. HTTP verbs SHALL be used for actions (See chapter 3.2.6).

[R 10|1]
Kebab-case[2] SHALL be used in services.

[R 11|1]
Lower camelCase[3] SHALL be used in resources, path parameters and query parameters.

[R 12|1]
Path parameters and query parameters with a relation to property names SHALL be consistent with property names.

[R 13|1]

[2] Kebab-case is a naming rule for a technical representation of identifiers consisting of several words. Hyphens are used to connect words. Example: **`this identifier`** is written as **`this-identifier`** in kebap-case.
[3] CamelCase is a naming rule for a technical representation of identifiers consisting of several words. White spaces are removed and every new word begins with a capital letter. Example: **`this identifier`** is written as **`thisIdentifier`** in camelCase. Lower camelCase means that the identifier must start with a small letter.

Query parameters SHALL be URL safe[4].

[R 14|1]
Resource names SHALL be pluralised. Resource names SHOULD be consistent with schemas. If a schema is defined in singular, nevertheless the resource SHALL be pluralized. If the plural of a resource is non-standard, you MAY choose a more appropriate noun in its plural form.

Examples for good endpoints:

- `/employees`
- `/customers`
- `/products`

3.2.3 Discoverability

One of the REST design principles is service discoverability. The OpenAPI specification supports them via links. They SHALL be implemented via HTTP headers.

3.2.4 Date and Time

The date and time representation in the CCL supports an ISO8601 subset with only a few exceptions. Those exceptions may be present in the content body of a request or a response.

[R 15|1]
Query parameters SHALL use ISO8601 compliant date and time representations that are defined in **`UNTDID 2379 json`** as defined in the JSON schema NDR technical specification. To represent a specific date, time or date-time the format SHALL comply with the JSON schema definition for date, time or date-time.

3.2.5 Using the UN/CEFACT semantics

Decades of harmonisation and standardisation of business requirements resulted in the UN/CEFACT reference data models (RDM). These exist across different domains like Buy-Ship-Pay, Agriculture, Regulatory or Audit and Accounting.

As one example the Buy-Ship-Pay RDM contains subsets e.g. for multimodal transport (MMT-RDM) and the supply chain (SC-RDM). Over time, hundreds of business document structures were harmonised and standardised on a semantic model level. Different Syntax Naming and Design rules allow an automated creation and mapping of those semantic models to certain syntaxes such as XML.

[4] See https://www.w3schools.com/tags/ref_urlencode.ASP Example: **`https://unece.org/this url`** is invalid because of the space. Correct it looks like **`https://unece.org/this%20url`**

In the world of web APIs, the transmission of document structures is considered obsolete. If the limitations of REST principles are to be applied to a web API, business document structures are unsuitable for a RESTful implementation. These structures contradict the basic principle of loose coupling of resources. Instead, the exchange of information should be resource-based, where resources are information blocks leading in their combination to the complete information (e.g. business document).

Nevertheless, there are often limitations in B2B information exchange that make it difficult to completely move away from document structures. This includes technical reasons, procedural reasons, but also legal reasons. If the basic processes of communication between organisations are not changed, a shift purely to resource-based information exchange leads to a new level of media disruption and consistency challenges. If both the sending and receiving systems work on the basis of document structures (e.g. an invoice), then an intermediate, purely resource-based transmission leads to a number of challenges, such as the archiving obligation of such documents that exists in many countries to ensure subsequent verification.

On the other hand, if networks of platforms (e.g. for logistics) are established, a resource-based exchange can still be useful for certain purposes. For example, a platform could exist for a marketplace where free delivery capacities by carriers can be offered and booked. The division by resources usually leads to the need for identity providers and the clarification of the question of the single source of trust for individual resources.

At UN/CEFACT, there are two basic JSON-based publications of semantic data models: the UN/CEFACT vocabulary, and the UN/CEFACT JSON schema publication.

3.2.5.1 Using the UN/CEFACT JSON schema publication

JSON schema is the natural partner of an OpenAPI specification, as OpenAPI relies on JSON schema. The UN/CEFACT JSON schemas are published in two variants:

8. Streamlined stand-alone JSON schemas for the individual business documents. Those schemas contain every definition relevant for a specific business document and its applied contextualisation.
9. A JSON schema library of the different RDMs and their related business document structures. This variant uses an inheritance and validation technique supported by JSON schema. The basic data structures define the information blocks needed together in the reference data model. Subsets and contextualisation for the individual applications (e.g. MMT-RDM, SC-RDM, Invoice ...) are then formed on this basis.

The JSON schemas are published in the official UN/CEFACT repository. They can be used in two different ways:

First by referencing the needed data types directly from the repository. This leads to a maximum on interoperability. In an OpenAPI specification, it is easily possible to further contextualise (including extension) the JSON subschemas to the needed requirements of the specific process. This explicitly lets the users "tick off" unneeded optional attributes or supplementary components, restrict code lists or add user defined properties in a standardised and transparent way.

Additionally, maintenance becomes quite easy. If the API is to be updated to a newer version of the JSON schema publication, only the reference needs to be updated.

Alternatively, the JSON schemas can be downloaded to a local system or repository. In that case it is needed to update or remove the **`"$id"`** properties of the schemas, as they link to the official UN/CEFACT repository.

The way in which the JSON schemas are defined allow a very simple transmission from using document-based structures to resource-based structures. On the RDM level, all ABIEs (data classes) are defined. For every RDM exists a master document structure. All of the business documents are derived from this. The hierarchic structure connects the different ABIEs through ASBIEs including cardinality information. At every single ASBIE node, the JSON schema publication allows to replace the provision of a substructure by the URN of the corresponding resource:

Let us assume you want to define an API to manage transport capacity booking. In a classic message-based scenario, you would define how those messages are interchanged. In many case you would design a **`POST`** and **`GET`** or **`POST`**, subscribe and **`GET`** scenario. Those scenarios need envelope-information around the message information in order to tell the API who the ultimate receiver is, who the sender is etc. In addition the message is quite complex and contains many sub-resources with details. Those include for instance "requester", "shipTo", "receiver", "carrier", "consignment-items" etc. If this scenario is planned to move towards a (more) resource-based information exchange it is very easy to do so. First, you have to identify which of your sub-resources should become stand-alone. Let us assume you want to manage trade party information master data as a single resource. In that case, you can specify a schema under **`components/schemas`** named **`tradePartyType`** and simply define it as a reference to the contextualised data type of the corresponding RDM or even the corresponding business document structure. The following example shows, how the document structure can be restricted to resource usage as well.

Example for a `tradePartyType` under `components/schemas`:

```
"tradePartyType": {
  "description": "Trade party definition according to MMT-RDM",
  "$ref": "https://raw.githubusercontent.com/uncefact/spec-
JSONschema/main/JSONschema2020-12/library/BuyShipPay/D22A/UNECE-
```

```
MMTContextCCL.json#/$defs/tradePartyType"
}

"tradePartyType": {
  "description": "Trade party definition according to the Multimodal
                  Transport Booking Recipient",
  "$ref": "https://raw.githubusercontent.com/uncefact/spec-
JSONschema/main/JSONschema2020-12/library/BuyShipPay/D22A/UNECE-
MultimodalTransportBooking.json#/exchangedDocument/recipient"
}

"multimodalTransportBooking": {
  "title": "Multimodal Transport Booking",
  "description": "Restrict business document to resource usage for
                  recipient",
  "allOf": [
    { "$ref": "https://raw.githubusercontent.com/uncefact/spec-
JSONschema/main/JSONschema2020-12/library/BuyShipPay/D22A/UNECE-
MultimodalTransportBooking.json/#" },
    {
      "properties": {
        "exchangedDocument": {
          "properties": {
            "recipient": { "type": "string", "format": "uri" }
          }
        }
      }
    }
  ]
}
```

3.2.5.2 Using the UN/CEFACT vocabulary

The UN/CEFACT vocabulary uses the JSON-LD format in order to be conformant with the publication on schema.org.

The publication in JSON-LD follows a different approach. JSON-LD is a graph representation of context-enhanced semantic ABIE-representations derived from the combination of the corresponding RDMs. By applying the appropriate context, the subset of the defined graph can be used.

JSON-LD cannot directly be used and linked to in an OpenAPI specification. According to the maintenance body of the OpenAPI specification, this is not intended to change in the near future. In addition, the JSON-LD does not specify the cardinalities and subsets for the different contexts of business document structure definitions. Therefore, a web developer implementing an API for business related intra-organisational information exchange needs a reasonable knowledge of the underlying processes. On the other hand, JSON-LD unfolds immense power wherever (publicly) available data is to be automatically crawled, filtered and evaluated. Examples of this are applications such as flight-radar, online search for recipes or searches for goods over the boundaries of online shops with specific criteria. In those scenarios, the individual resources get into focus, as well as their relationships (links) to other resources. The business-related-interdependencies are not part of the definitions themselves. Adding state machines in definitions could help with this. Unfortunately, currently there does not exist a widely supported exchange format for this kind of information[5].

In order to use the JSON-LD vocabulary, additional tooling must be used, as there does not exist a direct support in OpenAPI specifications. As a proof-of-concept, in the JSON-LD vocabulary publication, a sample implementation is included to import the vocabulary into a UML design tool. Here the first conversion from JSON-LD to UML is performed. Now the designing of the API can be performed within the UML-Tool. Some assumptions are made how to define which operations should be supported for each of the specified endpoints. Having defined this a second conversion from the UML-Tool to the OpenAPI specification format is performed.

3.2.5.3 Using other (standardised) data structures

In chapter 2.6 seven dimension of interoperability for WebAPIs are defined. From a global cross-industry perspective, full interoperability can only be achieved if for all of the dimensions the implementation rules are clearly defined. In the context of UN/CEFACT, this means that the UN/CEFACT semantic definitions as well as the UN/CEFACT syntaxes must be used to be fully compliant.

However, this NDR specification is syntax-neutral, as it defines basic requirements for the design of an OpenAPI specification in a B2B context. The stipulations in this specification can thus also promote interoperability between APIs that use a different syntax or divergent semantic specifications within a (closed) user group. Therefore, the following rule is defined as a conformance criterion:

[R 16|1]
A prerequisite for an OpenAPI specification and its implementation to be fully compliant with this NDR TS is the use of UN/CEFACT semantics and UN/CEFACT syntax (e.g. UN/CEFACT XML, UN/CEFACT JSON Schema, and UN/CEFACT Vocabulary). An OpenAPI specification that does not use UN/CEFACT syntax or UN/CEFACT semantics may still be conformant to this NDR TS if it meets the criteria specified in [R 1|1].

[5] See for example the JSON Finite State Machine in JSON schema format at https://github.com/ryankurte/jfsm

3.2.6 Operations

[R 17|1]

Endpoints are RECOMMENDED to support CRUD operations. (Create, Read, Update, Delete). If an endpoint is not intended to support e.g. a delete operation, it SHALL return the HTTP response codes as defined in chapter 3.2.10.

HTTP Method	Description
GET	To *retrieve/read* a resource.
POST	To *create* a new resource or to *execute* an operation on a resource that changes the state of the system e.g. send a message.
PUT	To *replace* a resource with another supplied in the request.
PATCH	To perform a *partial update* to a resource.
DELETE	To *delete* a resource.
HEAD	For retrieving metadata about the request, e.g. how many results *would* a query return? (Without actually performing the query). This can be used to follow a link-chain in an HATEOS implementation as well. An example is shown in chapter 4.3.2.
OPTIONS	Used to determine if a CORS (cross-origin resource sharing) request can be made. This is primarily used in front-end web applications to determine if they can use APIs directly.

3.2.6.1 Collection of Resources

The following operations are applicable for a collection of resources:

HTTP method	Resource Path	Operation	Examples
GET	*/resources*	Get a collection of the resource	GET */employees* or GET */employees?status=open*
HEAD	*/resources*	Get header and link information of the resource collection, e.g. for pagination	HEAD /employees or HEAD /employees?birthday=2022-04-16

> **Note**
>
> Creating or updating multiple resource instances in the same request is not standardised and thus should be avoided. There are factors such as receipt acknowledgement and how to handle partial success in a set of batches that must be considered on a case-by-case basis.

3.2.6.2 Single Resource

The following operations are applicable for a single resource:

HTTP method	Resource Path	Operation
GET	*/resources/{id}*	Get the instance corresponding to the resource ID
PUT	*/resources/{id}*	To update a resource instance by replacing it – "*Take this new thing and* _ **put** _ *it there*"
DELETE	*/resources/{id}*	To delete the resource instance based on the resource e.g. id
HEAD	*/resources/{id}*	Get header and link information of the resource.
PATCH	*/resources/{id}*	Perform changes such as add, update, and delete to the specified attribute(s). Is used often to perform partial updates on a resource

3.2.6.3 Idempotency

An idempotent HTTP method is an HTTP method that can be called many times without different outcomes. In some cases, secondary calls will result in a different response code, but there will be no change of state of the resource.

As an example, when you invoke N similar DELETE requests, the first request will delete the resource and the response will be 200 (OK) or 204 (No Content). Further requests will return 404 (Not Found). Clearly, the response is different from first request, but there is no change of state for any resource on server side because the original resource is already deleted.

HTTP Method	Is Idempotent
GET	True
POST	False

HTTP Method	Is Idempotent
PUT	True
PATCH	False
DELETE	True
HEAD	True
OPTIONS	True

Table 5 – Idempotency of operations

[R 18|1]
APIs SHALL adhere to the idempotency of operations specified in the list above.

[R 19|1]
APIs SHOULD implement the `Idempotency-Key`[6] HTTP header field and the corresponding implementation advice in order to make non-idempotent operations like POST and PATCH fault-tolerant.

3.2.7 Pagination

Querying an API with a GET can theoretically result in a huge return collection. Image querying the API of one of the big internet search engines without pagination. Hundreds of millions of results would have to be downloaded and displayed on a single page. That API would be unusable. Pagination helps to keep the data load to a reasonable amount and at the same time supports security aspects.

Historically, many APIs use offset pagination. A maximum page size (e.g. 20) is specified and the clients requests the starting record or the page number. However, this approach leads to fuzzy results: Suppose an API is supposed to return a list of all planned transport movements of a certain carrier ordered by destination. The first page of results is returned accurately. Before the client requests the next page or set of records, three possible things can happen.

- The databank does not change at all. Then the next page of records is accurate.

[6] https://www.ietf.org/archive/id/draft-ietf-httpapi-idempotency-key-header-01.txt

- A record is added to the database, which falls under the result list of the first page, which the client already received. In that case, the last result of the previous page is returned as the first result of the second page. The list therefore contains a duplicate.
- In the opposite case, a planned transport movement that has already been returned to the client on the first page is deleted. The first data record of the second page therefore moves to the previous page. If the client now queries the next page, this data record is not transmitted at all.

As an inter-organisational data exchange cannot accept this type of results, an alternative solution for pagination is needed. The solution to this problem is the so called keyset-based or cursor-pagination[7]. In addition, cursor-pagination is much more time-efficient on large datasets than offset-pagination.

[R 20|1]
If pagination is used in an API, keyset-based pagination (cursor-pagination) SHALL be used. This means that the consumer cannot request a specific page, instead the consumer has to select a page-link provided by the server. The server SHALL provide links in the HTTP response header to the previous and next page and SHOULD provide links to the first and last page. More links MAY be provided. The cursor-value is a string, created by the server using whatever method it likes. It identifies a point in a list of results for a query containing filters and sorting parameters for a specific moment in time. Therefore, it divides the list into those that fall before the cursor and those that fall after the cursor. There may optionally be one result that falls "on" the cursor.

Cursor-pagination assures a consistent data set for a query with filtering/sorting criteria at a specific moment in time. If another consumer performs the same query a moment later, he may get a different data set.

[R 21|1]
GET requests on collection results SHOULD implement pagination. The default and maximum page size SHOULD be 100, if not specified on the endpoint. If SHOULD be smaller, if the resulting page load is large. The default page size MAY be changed per endpoint. A consumer SHOULD be able to override the default page size. If the filter, sorting and/or page size used is changed when getting a result, the pagination SHALL BE reset to the first page. The query parameters described in the following table SHALL be used, rules SHALL be applied.

[7] https://jsonapi.org/profiles/ethanresnick/cursor-pagination/, https://medium.com/swlh/how-to-implement-cursor-pagination-like-a-pro-513140b65f32

Type	Explanation	Example
Page size	Overrides the default page size defined by the server / specification.	Example for the first query: GET /transportMovements? carrier=ABC &status=PLANNED &sort=estimatedTimeOfArrival &pageSize=50
Current page	A link to the current page.	Link: <https://api.unece.org/ transportMovements? cursor=XXX>; rel="current"
First page	A link to the first page. If it is the first page the link MAY be omitted.	Link: <https://api.unece.org/ transportMovements? cursor=XXX>; rel="first"
Next page	A link to the next page. If it is the last page, the link to the next page MAY be omitted. Otherwise, a `null` link shall be provided.	Link: <https://api.unece.org/ transportMovements? cursor=XXX>; rel="next" Link: <null>; rel="next"
Previous page	A link to the previous page. If it is the first page, the link to the previous page MAY be omitted. Otherwise, a `null` link shall be provided.	Link: <https://api.unece.org/ transportMovements? cursor=XXX>; rel="prev"
Last page	A link to the last page. If it is the last page, the link to the last page MAY be omitted. Otherwise, a `null` link shall be provided.	Link: <https://api.unece.org/ transportMovements? cursor=XXX>; rel="last"

When multiple links are given, they are separated by comma.

Example for a combination of Links:

```
Link:
   <https://api.unece.org/transportMovements?cursor=XXX>; rel="current",
   <https://api.unece.org/transportMovements?cursor=YYY>; rel="first",
```

```
<https://api.unece.org/transportMovements?cursor=ZZZ>; rel="next",
<https://api.unece.org/transportMovements?cursor=LLL>; rel="last"
```

3.2.8 Filtering

Providing the ability to filter and sort collections in an API allows your consumers greater flexibility and controls on how they choose to consume a conformant API.

[R 22|1]
Sorting and filtering SHALL be done using query parameters. Using a path parameter is only allowed to identify a specific resource.

3.2.8.1 Output Selection

Consumers can specify the attributes they wish to return in the response payload by specifying the attributes in the query parameters

Example that returns only the *first_name* and *last_name* fields in the response:
`?attributes=first_name,last_name`

3.2.8.2 Simple Filtering

Attributes can be used to filter a collection of resources.

`?last_name=Citizen` will filter out the collection of resources with the attribute **`last_name`** that matches **`Citizen`**.

`?last_name=Citizen&date_of_birth=1999-12-31` will filter out the collection of resources with the attribute **`last_name`** that matches **`Citizen`** and **`date_of_birth`** that matches 31st of December 1999.

[R 23|1]
As a general guide, filtering SHOULD be done with case insensitivity. Whether you choose to filter with case insensitivity or not SHALL be clearly documented.

The equal **=** operator is the only supported operator when used in this technique. For other operators and conditions next section.

3.2.8.3 Advanced filtering with LHS Operators

There are situations where simple filtering does not meet the needs and a more comprehensive approach is required. Use the reserved keyword filters to define a more complex filtering logic. The general pattern is

`/path?property[operator]=value&property[operator]=value`

The `=` sign in this case is there to maintain URL query string compatibility with RFC 3986. However, the API service will use the operator inside the brackets for the actual comparison. A logical AND combines all query conditions.

The following operators are supported:

- `[gte]` Greater than or equalled to
- `[egt]` Equalled to or greater than
- `[gt]` Greater than
- `[lt]` Less than
- `[lte]` Less than or equalled to
- `[elt]` Equalled to or less than
- `[ne]` Not equalled

Example for filtering with LHS attributes:

```
/path?creation_date[gt]=2020-11-30
```

3.2.8.4 Rich Query with Lucene Syntax

[R 24|1]

If an application needs to support a richer search and filter capability that includes logical operators, fuzzy search, grouping, and so on, API MAY apply a query string according to lucene query syntax[8]. In that case, the filtering and query parameters normally are transmitted in the request body.

3.2.8.5 GraphQL

When API implementers would like to allow their clients rich flexibility to define response data sets that might include data from multiple APIs with rich filtering capability then a GraphQL query interface could be provided. GraphQL is a different architecture to RESTful APIs, is especially tailored to queries across multiple entities, and allows clients to specify exactly which data elements they would like in the response. If you find yourself building very complex RESTful queries then you should consider GraphQL as an alternative.

GraphQL is not discussed further in this RESTful API design guide.

3.2.9 Sorting

Providing data in specific order is often the requirement from client applications and hence it is important to provide the flexibility for clients to retrieve the data in the order they need it.

[R 25|1]

[8] https://lucene.apache.org/core/2_9_4/queryparsersyntax.html

Sorting SHOULD be limited to specified fields. The sort direction MAY be omitted. The default sort direction is ascending. A colon `:` is used to separate the field name and the sort direction. Multiple sort fields are separated by comma `,` .

Query Parameter	Description
sort=name *sort=name:asc*	Sort by the name field in ascending order.
sort=name:desc	Sort by the name field in descending order.
sort=yearOfBirth,name:dec	Sort by year of birth in ascending order. If two equal years exist, sort the names by birth year in descending order.

Table 6: Sort examples

3.2.10 API Responses and error handling

[R 26|1]

HTTP response codes SHALL be used.

The following table defines HTTP response codes supported by conformant APIs. The column `Response` indicates whether an additional error response payload is RECOMMENDED to be returned as described in chapter 0.

Code	Status	Response	When to use
`200`	OK	No	The request was successfully processed
`201`	Created	No	The resource was created. The `Location` HTTP response header SHALL be returned to indicate where the newly created resource is accessible.
`202`	Accepted	No	The request was accepted, and is processed asynchronously.
`204`	No content	No	The server successfully processed the request and is not returning any content. There is no need for the client to move to a different location.
`400`	Bad Request	Yes	The server cannot process the request (such as malformed request syntax, size too large, invalid request message

Code	Status	Response	When to use
			framing, or deceptive request routing, invalid values in the request). For sensitive information, a code **`404 Not found`** MAY be returned instead.
`401`	Unauthorised	Yes	The request could not be authenticated. For sensitive information, a code **`404 Not found`** MAY be returned instead.
`403`	Forbidden	Yes	The request was authenticated but is not authorised to access the resource. For sensitive information, a code **`404 Not found`** MAY be returned instead.
`404`	Not found	Yes	The resource was not found.
`405`	Not Allowed		The method is not implemented for this resource. The response MAY include an **`Allow`** HTTP response header containing a list of valid methods for the resource.
`408`	Request Timeout	No	The request timed out before a response was received. A **`Retry-After`** HTTP response header is RECOMMENDED to be returned.
`415`	Unsupported Media Type	Yes	This status code indicates that the server refuses to accept the request because the content type specified in the request is not supported by the server
`422`	Unprocessable Entity		This status code indicates that the server understands the content type of the request entity, and the syntax of the request entity is correct, but it was unable to process the contained instructions.
`429`	Too Many Requests		There have been too many requests (by the consumer). A **`Retry-After`** HTTP response header is RECOMMENDED to be returned. A response body MAY be returned containing information about the reason for the response code. A possible reason may be if a quota of requests for the day / hour / month etc. was exceeded.

Code	Status	Response	When to use
`500`	Internal Server error		An internal server error. The response body may contain error messages. The response body SHALL not reveal any server configuration information (e.g. version, paths, database used, etc.).
`501`	Method Not Implemented		It indicates that the request method is not supported by the server and cannot be handled for the requested resource. Implementing this response code allows a higher interoperability between API implementations based on the same specification, if a specific server does not support one of the specified methods (yet). A **`Link`** HTTP response header is RECOMMENDED to point to the specific documentation.
`503`	Service unavailable		It indicates that the service is unavailable (e.g. due to maintenance reasons). A **`Retry-After`** HTTP response header is RECOMMENDED to be returned.

Table 7: HTTP response codes

[R 27|1]

The following table defines which HTTP response codes SHALL be supported for a specific HTTP request method by conformant APIs. Column **`Use`** indicates how a conformant API supports the specified http response code:

- **`M`** the code SHALL be supported
- **`MA`** SHALL be supported for requests where the response is handled asynchronous, for instance due to forwarding or processing time. In that case, a **`Location`** HTTP response header SHALL be gives that points to the respective resource. In addition, a **`Retry-After`** HTTP response header is RECOMMENDED to be returned.
- **`R`** the code is recommended to be supported.

The default response code for a positive response is marked in **bold**.

HTTP Request method	Code	Status	Use
`GET`	**`200`**	**OK**	M
	`202`	Accepted	MA
	`400`	Bad Request	R

HTTP Request method	Code	Status	Use
	401	Unauthorised	M
	403	Forbidden	M
	404	Not found	M
	405	Not Allowed	M
	408	Request Timeout	R
	415	Unsupported Media Type	M
	429	Too Many Requests	R
	500	Internal Server error	M
	503	Service unavailable	R
POST	**201**	**Created**	M
	202	Accepted	MA
	400	Bad Request	M
	401	Unauthorised	M
	403	Forbidden	M
	408	Request Timeout	R
	415	Unsupported Media Type	M
	422	Unprocessable Entity	R
	429	Too Many Requests	R
	500	Internal Server error	M
	503	Service unavailable	R
PATCH	202	Accepted	MA
	204	**No content**	M
	400	Bad Request	M
	401	Unauthorised	M
	403	Forbidden	M
	404	Not found	M
	405	Not Allowed	M
	408	Request timeout	R
	415	Unsupported Media Type	M
	422	Unprocessable Entity	M
	429	Too Many Requests	R
	500	Internal Server error	M
	503	Service unavailable	R
PUT	202	Accepted	MA
	204	**No content**	M
	400	Bad Request	M
	401	Unauthorised	M
	403	Forbidden	M

HTTP Request method	Code	Status	Use
	404	Not found	M
	405	Not Allowed	M
	408	Request Timeout	R
	415	Unsupported Media Type	M
	422	Unprocessable Entity	M
	429	Too Many Requests	R
	500	Internal Server error	M
	503	Service unavailable	R
DELETE	202	Accepted	MA
	204	**No content**	M
	400	Bad Request	M
	401	Unauthorised	M
	403	Forbidden	M
	404	Not found	M
	405	Not Allowed	M
	408	Request timeout	R
	415	Unsupported Media Type	M
	422	Unprocessable Entity	M
	429	Too Many Requests	R
	500	Internal Server error	M
	503	Service unavailable	R

3.2.11 Error Response Payload

For some errors, returning the HTTP status code is enough to convey the response. Additional error information can be supplemented in the response body. For example; HTTP 400 Bad request is considered too generic for a validation error and more information must be provided in the response body.

[R 28|1]
An API SHALL implement an error response schema to allow a standardised error handling. The response SHALL use the following JSON Schema. The JSON Schema MAY be extended.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "errors": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string",
                  "format": "uuid" },
          "code": { "type": "string" },
          "detail": { "type": "string" },
          "source": {
            "type": "object",
            "properties": {
              "parameter": { "type": "string" },
              "pointer": { "type": "string",
                           "format": "json-pointer" }
            },
            "unevaluatedProperties": false
          },
          "sourcePointer": { "type": "string",
                             "format":"json-pointer"}
        },
        "required": ["code", "detail"],
        "patternProperties": { "^x-": true },
        "unevaluatedProperties": false
      },
      "minItems": 1
    }
  },
  "required": [ "errors" ],
  "patternProperties": { "^x-": true },
  "unevaluatedProperties": false
}
```

The following definitions are applied:

Error response attributes	**Description**
id	Identifier of the specific error
detail	A human-readable explanation specific to this occurrence of the problem.
code	An application-specific error code
source	An object containing computer processable information about the origin of the error.
parameter	The (query) parameter where the error was caused.
pointer	JSON Pointer [RFC6901] to the associated entity in the request document [e.g. "/data" for a primary data object, or "/data/attributes/title" for a specific attribute].

Table 8: Error response attributes

Example for a `400 Bad Request` error response:

```
{
  "errors": [
  {
    "id": "86032cbe-a804-4c3b-86ce-ec3041e3effc",
    "code": "19283",
    "detail": "Invalid value(s) in request input",
    "source": {
      "parameter": "id"
    }
  }
  ]
}
```

Example for a `503 Service unavailable` error response:

```
Retry-After: Sat, 16 Apr 2022 15:00:00 GMT
{
  "errors": [
  {
    "id": "45786a8f-452e-492f-a779-801b5d0bd0a7",
    "code": "19284",
```

```
    "detail": "The service is unavailable due to maintenance. Come back at 15:00 GMT.",
    "source": {
      "pointer": "#/resources/12345"
    }
  }
  ]
}
```

3.2.12 Design rule examples

Good examples

Get a list of voyages:
`GET https://api.logistics.io/v1/transport/voyages`

Filtering in a query:
`GET https://api.logistics.io/v1/transport/voyages?departure_location=AUBNE&date=2022-04-16`

Get a single voyage:
`GET https://api.logistics.io/v1/transport/voyages/N234`

Create a new voyage:
```
POST https://api.logistics.io/v1/transport/voyages
{content body with voyage data in JSON format}
```

Update a voyage status:
```
PATCH https://api.logistics.io/v1/transport/voyages/N234/status
{content body status data in JSON format}
```

4 Well-documented APIs

4.1 General considerations

[R 29|1]
The following rules are RECOMMENDED: - The definitions in a conformant OpenAPI specification SHALL be considered as technical contracts between designers and developers and between consumers and providers. - Mock APIs SHOULD be created using the API description to allow early code integration for development. - The behaviour and intent of the API SHOULD be described with as much information as possible. - Operations SHOULD provide examples for request and response bodies. - Expected response codes and error messages SHOULD be provided in full. - Known issues or limitations SHOULD be clearly documented. - Expected performance, uptime and SLA/OLA SHOULD be clearly documented. - Although YAML is a supported file format of an OpenAPI specification, the JSON format SHOULD be used as the OpenAPI specification format.

4.2 API Versioning

4.2.1 Versioning Scheme

[R 30|1]
All APIs **SHALL** apply Semantic versioning 2.0.0[9]: `MAJOR.MINOR.PATCH` The first version of an API SHALL start with a **`MAJOR`** version of 1. Pre-release version[10] information and build metadata[11] version information SHALL NOT be used in API versioning.

Use the following guidelines when incrementing the API version number:

---

[9] https://semver.org/spec/v2.0.0.html

[10] https://semver.org/spec/v2.0.0.html#spec-item-9

[11] https://semver.org/spec/v2.0.0.html#spec-item-10

- **MAJOR** version when you make API changes that break backwards-compatibility,
- **MINOR** version when you add functionality in a backwards-compatible manner, and
- **PATCH** version when you make backwards-compatible bug fixes. A PATCH does not include new functionality.

4.2.2 URI Versioning

[R 31|1]

All APIs **SHALL** use URI versioning. They SHALL include the **`MAJOR`** version as part of the URI in the format of **`'v{MAJOR}'`**
 	
`Example:`
`https://api.logistics.io/transport/v1/voyages`
 	
The minor and patch version SHALL NOT be used in the URI.

4.2.3 Providing version information

APIs conforming to this technical specification are intended to be used with REST principles. Those mandate HATEOS (see chapter 4.3.2) support. On major aspect is the self-descriptiveness of an API. Although a support of HATEOS is not required, providing basic metadata about the called API including version information is useful even in not RESTful scenarios.

[R 32|1]
A custom header named **`API-Version`** SHALL be added to any response of the API. It SHALL be aligned with the URI version and SHALL state all three levels: `API-Version: 1.21.5`

[R 33|1]
An API-Version custom header MAY be added to a request. If added, it SHALL only contain the **`MAJOR`** version. `API-Version: 1`

In order to easily provide information about an API in a standardised way, the following information can be retrieved from any conformant API:

[R 34|1]

An API SHALL implement a response to a GET request to the base URI of the API. The response SHALL use the following JSON Schema:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "title": { "type": "string" },
```

```
      "version": {
        "type": "string",
        "pattern": "^\\d+(-.+)?\\.\\d+(-.+)?\\.\\d+(-.+)?$"
      },
      "status": {
        "type": "string",
        "enum": ["DRAFT", "ACTIVE", "DEPRECATED", "RETIRED"]
      },
      "effective": {
        "type": "string",
        "format": "date-time"
      },
      "specification": {
        "type": "string",
      "format": "uri"
     }
    },
    "required": [
      "title", "version", "status", "effective", "specification"
    ],
    "$comment" : "Allow extensions to the API metadata",
    "patternProperties": {
      "^x-": true
    },
    "unevaluatedProperties": false
}
```

The following definitions are applied:

- `title`: The name of the API. It SHALL be identical to the API title defined in the OpenAPI specification
- `version`: The API version
- `status`: The operation status of the API. The following values are used:
    - `ACTIVE`: The API is in its productive phase. Maintenance or deprecation of specific services SHALL be indicated at the service level. The `effective` defines the moment in the past since when API is in its productive phase.
    - `DEPRECATED`: The complete API is going to its end-of-life phase. The `effective` defines the moment in the future when the API is intended to switch to `RETIRED`. The rules of deprecation (see chapter 4.2.5) are applied additionally.
    - `RETIRED`: The complete API is to its end-of-life phase. The `effective` defines the moment in the past when the API was set to `RETIRED`. The rules of deprecation (see chapter 4.2.5) are applied additionally.
- `effective`: The moment in time corresponding to the `status`.
- `specification`: A valid URI to the OpenAPI specification of the current API. This way the available services and data types become self-descriptive from their basic structure. The OpenAPI specification SHOULD be public where possible and easily accessible to those that require it.

Additional metadata can be added to the response if required.

```
Example:

GET https://api.uncefact.unece.org/v1/

HTTP 200 OK
content-type: application/json; charset=utf-8
API-Version: 1.0.0

{
  "title": "UN/CEFACT Demo API",
  "version": "1.0.0",
  "status": "ACTIVE",
  "effective": "2022-06-02T23:00:00Z",
  "specification": "https://service.unece.org/demo/demoAPI.json",
  "x-info" : "Additional information"
}
```

During the draft, development or testing phase of an API sandbox environments are used to validate the intended functionality. For those kinds of APIs in development no additional state like `DRAFT` is provided.

[R 35|2]
APIs that are still in a `DRAFT` status SHOULD be placed in a sandbox environment. This could be done by changing the basis URL accordingly.

Example for a productive base URL:

```
https://api.uncefact.unece.org/v1/
```

Examples for a development base URL:

```
https://sandbox.api.uncefact.unece.org/v1/
https://staging.api.uncefact.unece.org/v1/
```

4.2.4 Robustness[12]

It is critical that APIs are developed with loose coupling in mind to ensure backwards compatibility for consumers.

[R 36|1]
Within a major release backward compatibility SHALL NOT be broken.

The following changes are deemed backwards compatible:

- Addition of a new optional field to a representation
- Addition of a new link to the `_links` array of a representation
- Addition of a new endpoint to an API
- Additional support of a new media type (e.g. `Accept: application/pdf`)

[12] https://en.wikipedia.org/wiki/Robustness_principle

The following changes are **NOT** deemed backwards compatible:

- Removal of fields from representations
- Changes of data types on fields (e.g. `string` to `boolean`)
- Changing semantic definitions
- Removal of endpoints or functions
- Removal of media type support

[R 37|1]
API clients and subscribers SHOULD be robust: - Be conservative with API requests and data passed as input. - Be tolerant with unknown fields in the payload, but do not eliminate them from payload if needed for subsequent `PUT` requests.

4.2.5 Deprecation and End of Life Policy

When designing new APIs one of the most important dates to consider is when the API will be retired. APIs are not intended to last forever. Some APIs are retired after a short time as they may be proving a use-case; others may be removed when better options are available for users.

The End-of-Life (EOL) policy determines the process that APIs go through to move through their workflow from `ACTIVE` to the `RETIRED` state. The EOL policy is designed to ensure a consistent and reasonable transition period for API customers who need to migrate from the old API version to the new API version while enabling a healthy process to retire technical debt.

Major API Version EOL

Major API versions **MAY** be backwards compatible with preceding major versions. The following rules apply when retiring a major API version.

[R 38|1]
An API SHALL NOT be set to `DEPRECATED` until a replacement service is running with status `ACTIVE`. The root service of the API SHALL provide the `Deprecation Header Field`[13] and the `Sunset HTTP Response Header Field`[14]. A `Link header` SHALL be added in combination with the `Deprecation header`. It SHALL provide a link to the documentation. A second `Link header` SHALL be added linking to the replacement version of the API.

Additionally, the following thoughts should be considered:

[13] https://tools.ietf.org/html/draft-dalal-deprecation-header-02
[14] https://tools.ietf.org/html/rfc8594#section-3

1. A minimum transition period of 60 days should be planned to give users adequate notice to migrate.
2. Deprecation of API versions with external users should be considered on a case-by-case basis and may require additional deprecation time and/or constraints to minimise impact to users.
3. If a versioned API is `ACTIVE` or `DEPRECATED` state has no registered users, it may move to the `RETIRED` state immediately.

[R 39|1]
Deprecated endpoints SHALL be documented in the OpenAPI specification using the `DEPRECATED` property introduces since OpenAPI 3.0.0. Deprecated endpoints SHOULD provide the Deprecation Header Field and the Sunset HTTP Response Header Field. A Link header SHALL be added in combination with the Deprecation header. It SHALL provide a link to the documentation. Where possible, communication SHOULD be sent to consumers of deprecated endpoints.

[R 40|1]
The introduction of a major version SHOULD be avoided, whenever possible. This MAY be achieved as follows: - Create a new service endpoint, if the process is changed. Duplicate and Deprecate: add a `Deprecation Header` to the old service including a `Link Header` to documentation and to the new service. Eventually add a `Sunset Header`. - Create a new resource (a variant of the old) in addition to the old.

Minor API Version EOL

Due to the specified URL versioning the URL does not change if the minor version of an API changes. Minor API versions are backwards compatible with preceding minor versions within the same major version.

Therefore, the status before, during or after a minor API version update does not change. The change should have no impact on existing subscribers so there is no need to transition through a `DEPRECATED` state to facilitate client migration.

[R 41|2]
New resources or service endpoints can be added during a minor release. In order to support the implementation of those new services a sandbox environment SHOULD be provided to the interested or affected consumers.

[R 42|1]

It is RECOMMENDED that no more than 3 parallel MAJOR versions are available. Implementers of the API SHALL NOT be more than 1 major version behind the latest version.

Example

Version 1 is **`RETIRED`**
Version 2 is **`DEPRECATED`**
Version 3 is **`ACTIVE`**

4.3 Hypermedia

4.3.1 Hypermedia - Linked Data

An API becomes RESTful by meeting the requirements of the REST principles. A key principle is the discoverability of the API. Ideally, this is achieved by an API being completely self-describing. According to the inventor of REST, Roy Fielding[15], the use of hypermedia is a prerequisite for designing a RESTful API.

Hypermedia means that links are provided together with the response payload. They inform the consumers what options are available according to their original request. Though simple in concept hypermedia links in APIs, allow consumers to locate resource without the need to have an upfront understanding of the resource and its relationship.

This is similar to the navigation of a web page. The user is not expected to know the structure of the web page prior to visiting. They can simply browse to the home page and the navigation lets them browse the site as required.

APIs that do not provide links are more difficult to use and expect the consumer to refer to the documentation.

4.3.2 HATEOAS

Hypermedia As The Engine Of Application State is the concept of representing allowable actions as hyperlinks associated with resource. Similar to Hypermedia Linked Data concept the links defined in the response data represents state transitions that are available from that current state to adjacent states.

Example:

```
HEAD /v1/accounts/4711

HTTP/1.1 200 OK
Link: <https://api.unece.org/v1/accounts/4711>; rel="self",
      <https://api.unece.org/v1/accounts/4711/deposit>; rel="deposit",
      <https://api.unece.org/v1/accounts/4711/withdraw>; rel="withdraw",
      <https://api.unece.org/v1/accounts/4711/transfer>; rel="transfer"
```

If the same account is overdrawn, the only allowed action could be to deposit:

[15] https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf

Example:

```
GET /v1/accounts/4711

HTTP/1.1 200 OK
Link: <https://api.unece.org/v1/accounts/4711>; rel="self",
      <https://api.unece.org/v1/accounts/4711/deposit>; rel="deposit"
Content-Type: application/json
Content-Length: ...
{
  "accountId": "4711",
  "balance": {
    "currency": "EUR",
    "value": -25
  }
}
```

4.3.3 Hypermedia Compliant API

In APIs, request methods such as *DELETE*, *PATCH*, *POST* and *PUT* initiate a transition in the state of a resource. A *GET* request never changes the state of the resource that is retrieved.

[R 43|1]
In order to provide a better experience for API consumers, APIs SHOULD provide a list of state transitions that are available for each resource. As possible values for link relation types the official IANA registry list[16] SHALL be used. It MAY be extended. Any extension SHALL be documented in the API specification.

An example of an API that exposes a set of operations to manage a user account lifecycle and implements the HATEOAS interface constraint is as follows:

A client starts their interaction with a service through the URI */users*. This fixed URI supports both *GET* and *POST* operations. The client decides to do a *POST* operation to create a user in the system.

<u>Request</u>

```
POST https://api.unece.org/v1/v1/users

{
  "firstName": "John",
  "lastName" : "Smith",
```

[16] https://www.iana.org/assignments/link-relations/link-relations.xhtml

```
  ...
}
```

The API creates a new user from the input and returns the following links to the client in the response.

- A link to the created resource in the *Location* header (to comply with the 201 response spec)
- A link to retrieve the complete representation of the user (a.k.a. *self*-link) (*GET*).
- A link to update the user (*PUT*).
- A link to partially update the user (*PATCH*).
- A link to delete the user (*DELETE*).

```
HTTP/1.1 201 CREATED
Location: https://api.unece.org/v1/users/JFWXHGUV7VI
Link: <https://api.unece.org/v1/users/JFWXHGUV7VI>, rel="self",
      <https://api.unece.org/v1/users/JFWXHGUV7VI>, rel="delete",
      <https://api.unece.org/v1/users/JFWXHGUV7VI>, rel="replace",
      <https://api.unece.org/v1/users/JFWXHGUV7VI>, rel="edit"
```

A client can store these links in its database for later use.

In summary:

- There is a well-defined index or navigation entry point for every API, which a client navigates to in order to access all other resources.
- The client does not need to build the logic of composing URIs to execute different requests or code any kind of business rule by looking into the response details that may be associated with the URIs and state changes.
- The client acknowledges the fact that the process of creating URIs belongs to the server.
- Client treats URIs as opaque identifiers.
- APIs using hypermedia in representations could be extended seamlessly. As new methods are, introduced responses could be extended with relevant HATEOAS links. These way clients could take advantage of the functionality in incremental fashion. For example, if the API starts supporting a new *PATCH* operation then clients could use it to do partial updates.

The mere presence of links does not decouple a client from having to learn the data required making requests for a transition and all associated link semantics particularly for *POST*/*PUT*/*PATCH* operations.

5 API Security

[R 44|1]
All API endpoints SHALL be secured. HTTPS SHALL be used. The OAUTH2 security scheme is RECOMMENDED. Other security schemes MAY be used. The receivers' endpoints of subscription callbacks MAY be designed with different security measures like those described in chapter 6.3. The following aspects of API security are RECOMMENDED to be implemented:

Rate Limiting

Rate limiting and throttling policies are introduced to prevent abuse of your API. Appropriate alerts should be implemented and respond with informative errors when thresholds are nearing or have been exceeded. See https://greenbytes.de/tech/webdav/draft-ietf-httpapi-ratelimit-headers-latest.html for implementation details.

Error Handling

When your application displays error messages, it should not expose information that could be used to attack your system. You should establish the following controls when providing error messages:

- Your API MUST mask any system related errors behind standard HTTP status responses and error messages e.g. do not expose system level information in your error response
- Your API MUST NOT pass technical details (e.g. call stacks or other internal hints) to the client

Audit Logs

An important aspect of security is to be notified when something wrong occurs, and to be able to investigate it. It is RECOMMENDED to implement logging.

- Write audit logs before and after security related events which can trigger the alerts
- Sanitizing the log data to prevent log injection attacks

Input Validation

Input validation is performed to ensure only properly formed data is received by your system, this helps to prevent malicious attacks

- Input validation should happen as early as possible, preferably as soon as the data is received from the external party
- Define an appropriate request size limit and reject requests exceeding the limit
- Validate input: e.g. length / range / format and type
- Consider logging input validation failures. Assume that someone who is performing hundreds of failed input validations per second has a malicious intent.
- Constrain string inputs with regular expression where appropriate

Content Type Validation

Honour the specified content-type. Reject requests containing unexpected or missing content type headers with HTTP response status *415 Unsupported Media Type*.

Gateway Security Features

It is RECOMMENDED to use the security policy features available in the gateway rather than to implement the policies in your back-end API.

6 Event driven data exchange

Classic B2B data exchange scenarios reach their limits especially when it comes to processing real-time data. For example, one of the most important pieces of information in just-in-time production is the expected arrival time (ETA) at the factory. PULL scenarios are often implemented, where the consumer periodically asks the data sender for the current status of the delivery. Alternatively, the carrier sends a status message at regular but short intervals on the current status of the delivery with detailed information for each consignment item. This leads to tremendous amounts of data, so that in practice the minimum interval of such updates is about 15 minutes. Thus, in such scenarios, real-time information is a long way off.

One approach to solving this problem is now to define events when they occur and exchange the data instead of constantly exchanging (less relevant) information. This could be the case, for example, if a geo-fence is crossed, a temperature is exceeded or not reached, or a clearance takes longer than it is intended. In the consumer space, such scenarios are already familiar, for example, when the buyer of an online delivery is notified that the package is only 10 stops away from delivery.

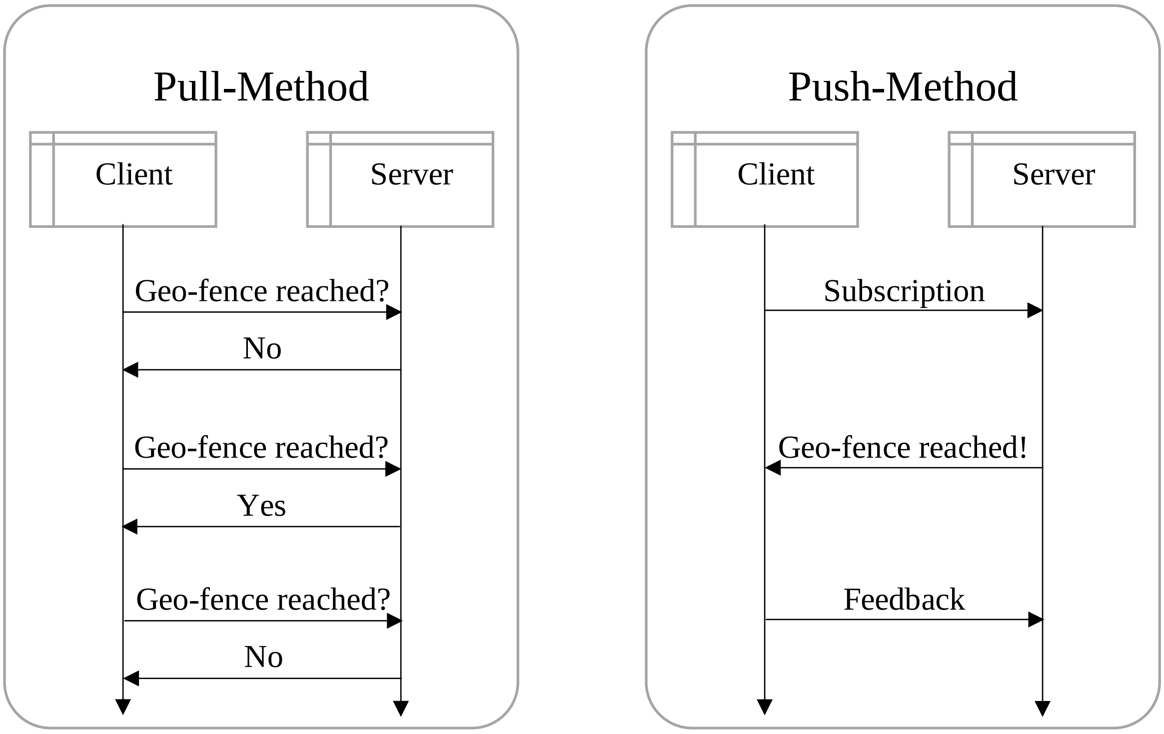

Figure 1: Event driven data exchange – pull versus push method

6.1 Callbacks

In OpenAPI, you can define callbacks. Those are asynchronous requests to a consumer specified URL that are called in response to a specific event. An example is that a carrier is informed if a specific vessel approaches a port.

In order to be able to receive this information, the receiver first needs to subscribe to this event information in the API. When subscribing, he may pass filter criteria that define the conditions under which the consumer will be informed. Examples are a specific journey where the consumer wants to get informed if it approaches a specific port.

The basic principle is that a consumer subscribes for an event, supplies a (callback) URL and stands by for incoming HTTP requests to that URL.

6.2 Webhooks

Since OpenAPI 3.1, webhooks are supported as well. The main difference between callbacks and webhooks is that webhooks are synchronous to the process flow handled by the APIs. This means that a consumer can directly hook into the process and thus, if necessary, change the processed information before it is further processed. A webhook is used to extend the functionality of the API.

A webhook defines a clear point in the process where the consumer is enabled to react on, for example based on some external event. An example is if you want to react immediately on any incoming order/payment etc. The payload itself is given with the webhook and often allows modifications. Examples are the option to link to a GitHub push event or to define a plugin for the WordPress content management system. The latter modifies for example the displayed HTML page directly by adding new functionalities like images, tables, videos or similar to the HTML page. Such modifications would not be possible with an asynchronous callback.

6.3 Security guideline for callbacks (informative)

Since webhooks work synchronously, the same security rules apply to them as to the entire API. In contrast, the call direction is reversed for asynchronous callbacks. This makes it important to ensure that the callback URL is only called from the authorized API.

The following rules are based on the current approach of the DCSA. They are in the trial phase at the time of publication of this document. As soon as sufficient practicality has been demonstrated, this specification will be updated accordingly. Against this background, the following rules are purely informative and not normative.

[R 45|1+Inf]
All event subscriptions SHALL be secured via a Shared Secret that is used to sign every callback message as described in this section. The secret SHALL be provided BASE64 encoded. The provider SHALL NOT expose the `secret` in any endpoint. It is write-only. The provider SHALL assure that the secret fulfils the security requirements of the applied algorithm.

[R 46|2+Inf]

A sha256 signature SHALL be used computed as an HMAC-SHA246 over the request body[17]. The subscriber provided Shared Secret SHALL be of at least 32-byte length. It SHOULD not be longer than 64 byte, as longer keys do not provide additional security to that algorithm.

To improve security, it is RECOMMENDED to update the `secret` (and together with it the `callbackURL`) on a regular basis.

[R 47|1+Inf]

The publisher API SHALL provide the following endpoints for subscriptions:

- `POST …/subscriptions` to create a new subscription
- `GET …/subscriptions` to list all subscriptions the subscriber has access to
- `GET …/subscriptions/{subscriptionId}` to get details about a specific subscription
- `PUT …/subscriptions/{subscriptionId}` to update a specific subscription
- `PUT …/subscriptions/{subscriptionId}/secret` to update the secret of a specific subscription
- `DELETE …/subscriptions/{subscriptionId}` to cancel a specific subscription

6.3.1 Subscription setup (informative)

The setup of a subscription follows the following steps:

1. The subscriber defines a Shared Secret and registers with the `secret` and a `callbackURL` in the publisher's system. It is recommended to use a not-easy-to-guess[18] callback URL and to update it when the secret is changed.
2. The publisher confirms the subscription and returns the `subscriptionId` to the subscriber.
3. The subscriber records the `subscriptionId` associated with the shared `secret`.

Example for a subscription setup

1. Initiating the subscription

```
POST https://api.unece.org/v1/events/subscribe
Content-Type: application/json
Content-Length: ...
{
  "callbackURL" : "https://callback.example.com/callback/Ujh4kkQ9A",
```

[17] Compare https://docs.github.com/en/developers/webhooks-and-events/webhooks/securing-your-webhooks

[18] https://callback.example.com/callback/$RANDOM_STRING

```
  "secret":
"MDEyMzQ1Njc4OWFiY2RlZjAxMjM0NTY3ODlhYmNkZWYwMTIzNDU2Nzg5YWJjZGVmMDEyMzQz
NjU3ODlhYmNkZQ",
  ... additional filter parameters etc. ...
}
```

2.a Confirmation of the publisher if the **callbackURL** is valid

Remark: As the subscription is not setup yet, not additional headers are provided.

```
HEAD https://callback.example.com/callback/Ujh4kkQ9A
```

2.b Response of the subscriber that the **callbackURL** is valid

```
HTTP/1.1 204 No Content
```

3. Response from the publisher

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: ...
{
  "subscriptionId": "936DA01F-9ABD-4D9D-80C7-02AF85C822A8",
  "callbackURL": "https://callback.example.com/callback/Ujh4kkQ9A",
  ... additional optional content ...
}
```

6.3.2 Performing a subscription call (informative)

A subscription call follows the following steps:

1. The publisher SHALL perform a **POST** to the **callbackURL** of the subscriber.
   - A **Subscription-ID** HTTP header containing the **subscriptionId** is added.
   - A **Notification-Signature** HTTP header containing the computed signature of the request body is added.
   - The request-body is sent using the **application/json** format.
2. The subscriber SHALL validate the **POST** request. It SHOULD be done in the following order. If any of the validation steps fail, the message SHALL be rejected.
   - It is RECOMMENDED to start message parsing only if all of the validation steps are performed without an error.
   - The **Notification-Signature** HTTP header MUST be provided.
   - The **Subscription-ID** HTTP header MUST be included. It MUST be a GUID.
   - Additional provided custom information is RECOMMENDED to be validated. (e.g. in the **callbackURL**)

- The subscriber uses the stored Shared Secret to compute the signature of the request body. The signature SHALL equal the provided signature.
- In case the callback was performed due to a a subscription of an event, the occurrence time of the event MUST be in the past. It MAY be a few seconds in the future to account for minor time synchronization issues.

3. A successful callback is responded by the **`204 No Content`** response code.

Example for a subscription call using the secret from the example above

```
POST https://callback.example.com/callback/Ujh4kkQ9A
Subscription-ID: 936DA01F-9ABD-4D9D-80C7-02AF85C822A8
Notification-Signature:
sha256=66c2912069e6c9563d66fee4674cd23dd9dd00e6c08c985e964b11f92f477e48
Content-Type: application/json
Content-Length: ...
{
  "id": "84db923d-2a19-4eb0-beb5-446c1ec57d34",
  "occurrenceDateTime": "2022-04-16T16:40:00+01:00",
  "typeCode": "ARRIVAL",
  "shipmentId": "123e4567-e89b-12d3-a456-426614174000"
}
```

Response

```
HTTP/1.1 204 No Content
```

7 Appendix A: Examples

Printed JSON schema files of a realistic example can be very large, especially because of the code lists used. Therefore, we have not included an example here.

However, examples can be found on the web at the following address:

https://github.com/uncefact/spec-openAPI/examples

8 Appendix B: Naming and Design Rules List (normative)

Rule #	Rule
[R 1|1]	Conformance SHALL be determined through adherence to the content of the normative sections and rules. Furthermore, each rule is categorized to indicate the intended audience for the rule by the following: 1. Rules, which must not be violated. Else, conformance and interoperability is lost. 2. Rules, which may be modified, while still conformant to the NDR structure. Inf. Rules that are informative only. If a different implementation is chosen this does not have any impact on the conformance of the implementation towards this specification.
[R 2|1]	All API specifications based on this OpenAPI Naming and Design Rules technical specification SHALL be compliant to the OpenAPI 3.1.x specification.
[R 3|1]	An API specification-claiming conformance to this specification SHALL define schema components as described in the JSON Schema Naming and Design Rules Technical Specification.
[R 4|1]	Request body content and Response content used to transfer structured data information SHALL use the **`application/json`** media type for JavaScript Object Notation (JSON). This rule MAY only be deviated from, if the API implements a conversion service from or to JSON in another media type. Additional media types (e.g. **`text/xml`**) to transfer structured data information MAY be used. If non-structured information is transferred any valid media type MAY be used.
[R 5|1]	Encoding SHALL be UTF-8.
[R 6|2]	The structure of the paths defined within APIs SHOULD be meaningful to the consumers. Paths SHOULD follow a predictable, hierarchical structure to enhance understandability and therefore usability.
[R 7|1]	The API URLs SHOULD follow the standard naming convention as described below: **`https://{env}.api.{dnsdomain}/v{m}/{service}/{resource}/{id}/{sub-resource}?{query}`** The components are described as follows. If a rule is mandatory for a specific component of the URL is SHALL be applied to any conformant API specification, even if the basic URL structure is different from the one described above (e.g. if **`api`** is not used as a prefix to the **`dnsdomain`**). • https:// SHALL be used as the web protocol. • {env} indicates the environment (e.g. **`test`**, **`sandbox`** or **`dev`**) and is usually omitted for production environment. • {dnsdomain} is the DNS domain of the API implementer (e.g. **`unece.org`**) • {service} is a logical grouping of API functions that represent a business service domain (e.g. transport). The {service} component is optional. • v{m} is the major version number of the API specification. This component SHALL be stated in the URL. It MAY be provided at a different place in the URL (e.g. as a prefix to the domain). • {resource} is the plural noun representing an API resource (e.g. **`consignments`**) • {id} is the unique identifier for the resource defined as a path parameter. Path parameters SHALL be used to identify a resource. This component is not part of the path if an operation is performed on a collection of the resource. • {sub-resource} is an optional sub-resource. Only used when there are contained collections or actions on a main resource (e.g. **`consignmentItem`**). • {query} is a list of additional parameters like filters that determine the results of a search (e.g. **`consignments?loadingPort=AUSYD`**).

[R 8|1]	The total number of characters in the URL, including the path and the query, SHALL NOT exceed 2000 characters in length including any formatting codes such as commas, underscores, question marks, hyphens, plus or slashes.
[R 9|1]	Endpoints SHALL NOT be actions. Services and resources SHALL consist of nouns. HTTP verbs SHALL be used for actions.
[R 10|1]	Kebab-case SHALL be used in services.
[R 11|1]	Lower camelCase SHALL be used in resources, path parameters and query parameters.
[R 12|1]	Path parameters and query parameters with a relation to property names SHALL be consistent with property names.
[R 13|1]	Query parameters SHALL be URL safe.
[R 14|1]	Resource names SHALL be pluralised. Resource names SHOULD be consistent with schemas. If a schema is defined in singular, nevertheless the resource SHALL be pluralized. If the plural of a resource is non-standard, you MAY choose a more appropriate noun in its plural form.
[R 15|1]	Query parameters SHALL use ISO8601 compliant date and time representations that are defined in `UNTDID 2379 json` as defined in the JSON schema NDR technical specification. To represent a specific date, time or date-time the format SHALL comply with the JSON schema definition for date, time or date-time.
[R 16|1]	A prerequisite for an OpenAPI specification and its implementation to be fully compliant with this NDR TS is the use of UN/CEFACT semantics and UN/CEFACT syntax (e.g. UN/CEFACT XML, UN/CEFACT JSON Schema, and UN/CEFACT Vocabulary). An OpenAPI specification that does not use UN/CEFACT syntax or UN/CEFACT semantics may still be conformant to this NDR TS if it meets the criteria specified in [R 1|1].
[R 17|1]	Endpoints are RECOMMENDED to support CRUD operations. (Create, Read, Update, Delete). If an endpoint is not intended to support e.g. a delete operation, it SHALL return the HTTP response codes as defined in chapter 3.2.10.
[R 18|1]	APIs SHALL adhere to the idempotency of operations specified in Table 4.
[R 19|1]	APIs SHOULD implement the Idempotency-Key HTTP header field and the corresponding implementation advice in order to make non-idempotent operations like POST and PATCH fault-tolerant.
[R 20|1]	If pagination is used in an API, keyset-based pagination (cursor-pagination) SHALL be used. This means that the consumer cannot request a specific page, instead the consumer has to select a page-link provided by the server. The server SHALL provide links in the HTTP response header to the previous and next page and SHOULD provide links to the first and last page. More links MAY be provided. The cursor-value is a string, created by the server using whatever method it likes. It identifies a point in a list of results for a query containing filters and sorting parameters for a specific moment in time. Therefore, it divides the list into those that fall before the cursor and those that fall after the cursor. There may optionally be one result that falls "on" the cursor.
[R 21|1]	GET requests on collection results SHOULD implement pagination. The default and maximum page size SHOULD be 100, if not specified on the endpoint. If SHOULD be smaller, if the resulting page load is large. The default page size MAY be changed per endpoint. A consumer SHOULD be able to override the default page size. If the filter, sorting and/or page size used is changed when getting a result, the pagination SHALL BE reset to the first page. The query parameters described in the following table SHALL be used, rules SHALL be applied.
[R 22|1]	Sorting and filtering SHALL be done using query parameters. Using a path parameter is only allowed to identify a specific resource.
[R 23|1]	As a general guide, filtering SHOULD be done with case insensitivity. Whether you choose to filter with case insensitivity or not SHALL be clearly documented.

[R 24|1]	If an application needs to support a richer search and filter capability that includes logical operators, fuzzy search, grouping, and so on, API MAY apply a query string according to lucene query syntax . In that case, the filtering and query parameters normally are transmitted in the request body.
[R 25|1]	Sorting SHOULD be limited to specified fields. The sort direction MAY be omitted. The default sort direction is ascending. A colon `:` is used to separate the field name and the sort direction. Multiple sort fields are separated by comma `,`.
[R 26|1]	HTTP response codes SHALL be used. Table 6 defines HTTP response codes supported by conformant APIs. The column Response indicates whether an additional error response payload is RECOMMENDED to be returned as described in chapter 3.2.11.
[R 27|1]	Table 7 defines which HTTP response codes SHALL be supported for a specific HTTP request method by conformant APIs. Column Use indicates how a conformant API supports the specified http response code: - **`M`** the code SHALL be supported - **`MA`** SHALL be supported for requests where the response is handled asynchronous, for instance due to forwarding or processing time. In that case, a **`Location`** HTTP response header SHALL be gives that points to the respective resource. In addition, a **`Retry-After`** HTTP response header is RECOMMENDED to be returned. - **`R`** the code is recommended to be supported. The default response code for a positive response is marked in **bold**.
[R 28|1]	An API SHALL implement an error response schema to allow a standardised error handling. The response SHALL use the following JSON Schema. The JSON Schema MAY be extended.
[R 29|1]	The following rules are RECOMMENDED: - The definitions in a conformant OpenAPI specification SHALL be considered as technical contracts between designers and developers and between consumers and providers. - Mock APIs SHOULD be created using the API description to allow early code integration for development. - The behaviour and intent of the API SHOULD be described with as much information as possible. - Operations SHOULD provide examples for request and response bodies. - Expected response codes and error messages SHOULD be provided in full. - Known issues or limitations SHOULD be clearly documented. - Expected performance, uptime and SLA/OLA SHOULD be clearly documented. - Although YAML is a supported file format of an OpenAPI specification, the JSON format SHOULD be used as the OpenAPI specification format.
[R 30|1]	All APIs SHALL apply Semantic versioning 2.0.0 : **`MAJOR.MINOR.PATCH`** The first version of an API SHALL start with a **`MAJOR`** version of 1. Pre-release version information and build metadata version information SHALL NOT be used in API versioning.
[R 31|1]	All APIs SHALL use URI versioning. They SHALL include the MAJOR version as part of the URI in the format of 'v{MAJOR}'. Example: `https://api.logistics.io/transport/v1/voyages` The minor and patch version SHALL NOT be used in the URI.
[R 32|1]	A custom header named API-Version SHALL be added to any response of the API. It SHALL be aligned with the URI version and SHALL state all three levels: `API-Version: 1.21.5`
[R 33|1]	An API-Version custom header MAY be added to a request. If added, it SHALL only contain the MAJOR version. `API-Version: 1`

[R 34|1]	An API SHALL implement a response to a GET request to the base URI of the API. The response SHALL use the JSON Schema defined in R 33.

[R 35|2]	APIs that are still in a **`DRAFT`** status SHOULD be placed in a sandbox environment. This could be done by changing the basis URL accordingly.
	Example for a productive base URL:
	`https://api.uncefact.unece.org/v1/`
	Examples for a development base URL:
	`https://sandbox.api.uncefact.unece.org/v1/`
	`https://staging.api.uncefact.unece.org/v1/`

[R 36|1]	Within a major release backward compatibility SHALL NOT be broken.
[R 37|1]	API clients and subscribers SHOULD be robust: - Be conservative with API requests and data passed as input. - Be tolerant with unknown fields in the payload, but do not eliminate them from payload if needed for subsequent PUT requests.
[R 38|1]	An API SHALL NOT be set to **`DEPRECATED`** until a replacement service is running with status **`ACTIVE`**. The root service of the API SHALL provide the **`Deprecation Header Field`** and the **`Sunset HTTP Response Header Field`** . A **`Link header`** SHALL be added in combination with the **`Deprecation header`**. It SHALL provide a link to the documentation. A second **`Link header`** SHALL be added linking to the replacement version of the API.
[R 39|1]	Deprecated endpoints SHALL be documented in the OpenAPI specification using the **`DEPRECATED`** property introduces since OpenAPI 3.0.0. Deprecated endpoints SHOULD provide the Deprecation Header Field and the Sunset HTTP Response Header Field. A Link header SHALL be added in combination with the Deprecation header. It SHALL provide a link to the documentation. Where possible, communication SHOULD be sent to consumers of deprecated endpoints.
[R 40|1]	The introduction of a major version SHOULD be avoided, whenever possible. This MAY be achieved as follows: - Create a new service endpoint, if the process is changed. - Duplicate and Deprecate: add a **`Deprecation Header`** to the old service including a **`Link Header`** to documentation and to the new service. Eventually add a **`Sunset Header`**. - Create a new resource (a variant of the old) in addition to the old.
[R 41|2]	New resources or service endpoints can be added during a minor release. In order to support the implementation of those new services a sandbox environment SHOULD be provided to the interested or affected consumers.
[R 42|1]	It is RECOMMENDED that no more than 3 parallel MAJOR versions are available. Implementers of the API SHALL NOT be more than 1 major version behind the latest version.
[R 43|1]	In order to provide a better experience for API consumers, APIs SHOULD provide a list of state transitions that are available for each resource. As possible values for link relation types the official IANA registry list SHALL be used. It MAY be extended. Any extension SHALL be documented in the API specification.
[R 44|1]	All API endpoints SHALL be secured. HTTPS SHALL be used. The OAUTH2 security scheme is RECOMMENDED. Other security schemes MAY be used. The receivers endpoints of subscription callbacks MAY be designed with different security measures like those described in chapter 6.3. The aspects described after rule 32 of API security are RECOMMENDED to be implemented.

1270 9 Appendix C: Glossary

Term	Definition
ABIE	Aggregate Business Information Entity – a term from CCTS that describes an information class such as “consignment”
API	Application Programming Interface – a term that references a machine-to-machine interface.
ASBIE	Association Business Information Entity – a term from CCTS that defines a directed relationship from source ABIE to target ABIE – e.g. “consignee” as a relationship between “consignment” and “party”
B2B	Business to Business
BBIE	Basic Business Information Entity – a term from CCTS that describes a property of a class such as party.name
BRS	Business requirement specification
CamelCase	CamelCase is a naming rule for a technical representation of identifiers consisting of several words. White spaces are removed and every new word begins with a capital letter. Example: `this identifier` is written as `thisIdentifier` in camelCase.
CCL	Core Component Library
CCTS	Core Component Technical Specification – a UN/CEFACT specification document that described the information management metamodel.
CDT	Core Data Type. A value domain for a BBIE that is a simple type such as “text” or “code”
HATEOS	Hypermedia as the Engine of Application State
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation – an IETF document syntax standard in common use by web developers for APIs.
JSON-LD	JSON-Linked Data – a JSON standard for linked data graphs / semantic vocabularies.
Kebab-case	Kebab-case is a naming rule for a technical representation of identifiers consisting of several words. Hyphens are used to connect words. Example: `this identifier` is written as `this-identifier` in kebap-case.
NDR	Naming & Design Rules – a set of rules for mapping one representation (e.g. RDM) to another (e.g. JSON-LD)
OpenAPI	An open source standard, language-agnostic interface to RESTful APIs.
OWL	Web Ontology Language
RDF	Resource Description Framework – a W3C semantic web standard
RDM	Reference Data Model- a UN/CEFACT semantic output.
RESTful API	See REST API
REST API	Representation State Transfer Application Programming Interface, a.k.a. RESTful API
RFC	Request for Comments
SDO	Standards Development Organisation
UN/CEFACT	United Nations Centre for Trade Facilitation and Electronic Business
UNECE	United Nations Economic Commission for Europe
URI	Uniform Resource Identifier – a namespace qualified string of characters that unambiguously identify a resource. AURL is one type of URI.

Term	Definition
URL	Uniform Resource Locator – the web address of a resource.
UNTDID	United Nations Trade Data Interchange Directory
XML	Extensible Markup Language
XMI	Xml Metadata Interchange - a well-established OMG standard for exchange of UML models between different tools.

Table 9 - Glossary