DCSA API Design & Implementation Principles 2.0

June 2024

Change history

Rev	Issue	Contributors	Description
1.0	September 2020	DCSA	First release
1.1	September 2021	DCSA	Subscription API
2.0	June 2024	DCSA	New approach. Consolidation of many recommendations by DCSA Technology Subject Matter Experts, and the DCSA Engineering Team.

About This Document

Notes on this publication

This version of the document is the final version 2.0 release. It was approved at a vote by DCSA members that completed on 28th June 2024.

The goal is to clearly describe the criteria for conformance with each principle from three different perspectives -

- Standards Development
- API Provider
- API Consumer

This document functions both as a description of the principles and as a conformance checklist. This will ensure current and future alignment between the standardisation and the conformance process. The principles only have value when it is possible to verify their correct use.

Change log

The table below describes the most important changes to this document, compared to the previous published version of this document, which was “Design Principles 1.1”.

Area	**Difference in “API Design & Implementation Principles 2.0” compared to “Design Principles 1.1”**	**Motivation**	**Location in Document**
XML	XML is no longer supported, structured requests and responses MUST use JSON, although binary responses (e.g. images, PDFs) MAY also be appropriate in some cases.	Other encodings have not been used. Simplification of standards and alignment with current wider thinking on APIs.	Section 2 / JSON
Collation on String sort	Narrowed to the Unicode standard sort.	If this were not present, API consumers would not be able to reliably page through a result set in order to locate a particular record.	Section 2 / Sorting
Pagination Link headers	Approach is simplified so that only the Next-Page header is now mandatory.	Agreed by DCSA members.	Section 2 / Pagination
Pagination semantics.	It is mandatory to specify the semantics (e.g. contiguous paging). Different semantics can exist in different standards.	Provide consistency between API providers and therefore interoperable behaviour. Does not force a single approach on all APIs.	Section 2 / Pagination
Naming of arrays	Names of Arrays MUST be plural (was a SHOULD)	Naming conventions agreed with DCSA members. Consistency between arrays and collections.	Section 2 / Property names

Enums	Enum-like String approach is described in considerable detail, but is not a difficult change in practice.	Backward compatibility concerns.	Section 2/ Property values
Dates and Times	Time zones are mandatory.	Interoperability and semantics concerns. Ensure that a DateTime has a clear meaning.	Section 2 / Property values
limit query parameter	Clarification on allowed ranges of values and defaults for them.	Clarification requested by DCSA members.	Section 2 / Pagination
API-Version header	Format is clarified.	Formalise existing conventions.	Section 3 / Versioning
Error handling	More detail and standard error response object.	Clarification requested by DCSA members.	Section 2 / Error handling
Subscription model	Removed and delegated to each standard. However, there are extensive notes on the use of callbacks and webhooks.	This area is not yet stable, therefore placed outside of this document.	N/A
OAS	Option to support OAS 3.1 is included. When it is used, requests and responses must be specified using JSON Schema.	New standard from OAS. Define approach to upgrade.	Section 2 / OpenAPI Specification.
Backward Compatibility	Rules updated according to those agreed with the DCSA members.	Agreed with DCSA members.	Section 3 / Backward compatibility
Extensions	The approach to providing API extensions is documented, as agreed with DCSA members.	Agreed with DCSA members.	Section 3 / API Extensions
HTTPS	Stricter definition, it is not allowed to use a version with known vulnerabilities.	Clarification of existing intent, align with industry best practices.	Section 4 / Secure
HATEOAS	Removed as a best practice. This is to allow looser coupling when multiple complex APIs are in use. A given microservice may not know the full path to another microservice, which may be specific for a given API consumer.	Statement was incorrectly made in the 1.1 version of the document that HATEOAS is in use. It is believed that not using this can increase loose coupling in the DCSA case.	Section 2 / Patterns
Checklists	Includes checklists for the different roles of API Designer, API Provider, API Consumer.	Make the document clearer and easier to use.	Throughout
Additional Properties	Clarification that it is not generally allowed to set additional properties in DCSA APIs. Extensions do provide a mechanism for this, however.	Clarification of existing intent.	Section 2 / OpenAPI Specification
PATCH	Clarification on approaches to implementation of PATCH.	Provide guidance and align with the “Zalando” approach.	Section 2 / HTTP Methods

Underlying Zalando Guidelines	Where the Design and Implementation Principles are ambiguous, the Zalando guidelines, “Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando” can be used.	Leverage existing work of others. Allow resolution of many questions that are not explicit in the document.	Introduction
Code Lists from external bodies	Various approaches described; each standard must make the approach taken clear.	Provide an approach to increase clarity for greater interoperability.	Section 2 / Property values
Naming conventions	Naming conventions as agreed with DCSA members added.	Agreed with DCSA members.	Section 2 / Property names
Null values	Null values are not allowed, as agreed with DCSA members.	Agreed with DCSA members.	Section 2 / Property values
Clarifications	Extensive clarifications and usage guidelines are provided in the document, but it is explicitly noted that these are “not normative”.	Provide clarification and make the document more useful.	Throughout
HTTP and REST	There is extensive clarification about contemporary usage of HTTP and REST. For example - UTF-8, response codes, accept headers, content type negotiation, HTTP methods. This was implied in the Design Principles 1.1 document – it is made explicit in this version of the document.	Provide clarification and make the document more useful. This is simply making rules explicit that are already specified by IETF, W3C, Unicode Consortium. It addresses questions that have been raised during DCSA standardisation.	Throughout
Concurrency Control	So far no DCSA API has included explicit concurrency control. Should it be required in the future, then possible valid approaches are described.	Without imposing a single approach, provide options for concurrency control in the future.	Section 2 / HTTP Methods
JSON-LD	Explicitly not supported in DCSA APIs. This is to provide consistency across all APIs. Similar arguments to HATEOAS are also relevant here.	Clarification of existing intent.	Section 2 / Patterns
Unicode	Clarification about how to use Unicode, for example the exact version to be used, how to handle sorting, how newlines are handled.	Issue raised in standardisation discussions. Clarification of existing intent.	Section 2 / Property values
Regular expressions	Align with the OAS approach to use ECMA262 regular expressions (note this is mandatory in OAS 3.1).	Align with OpenAPI Specification direction. Provide consistency.	Section 2 / Property values
Versioning Tightly-Coupled Standards	Policies and considerations on versioning tightly coupled standards	Without imposing a single approach, provide guidance on	Section 3 / Versioning

	clarified.	handling related standards.	

Terms, acronyms, and abbreviations

Term	Definition
API	Generally, an “Application Programming Interface”. In the context of this document, specifically REST APIs using JSON.
REST	Representational State Transfer. The paradigm or set of patterns that underpin DCSA REST APIs. REST is curious in that it is not formally standardised, but it is still well-understood within the software engineering community. A possible (but not normative) definition can be found here → https://www.redhat.com/en/topics/api/what-is-a-rest-api
API Provider	An organisation operating an application that provides a service delivered using APIs. Note that when the HTTP protocol roles are reversed to provide webhooks or callbacks, the party acting as the API provider remains the same. Being an API provider is defined by being a *service provider* in terms of the business role not (only) by offering an HTTP server interface to the Internet.
API Consumer	An organisation operating an application that consumes a service delivered using APIs. This can include offering an HTTP server interface in order to receive webhooks or callbacks. Being an API consumer is defined by being a *service consumer* in terms of the business role.
camelCase	In this document camelCase is defined as lower camel case. Definition of lower camel case is that initial letter is lower and every subsequent word is Capitalised.
kebab-case	kebab-case combines words by replacing each space with a dash (-). All letters are lower case. For example, this-is-a-kebab-case-example.
UPPER_SNAKE_CASE	UPPER_SNAKE_CASE combines words by replacing each space with an underscore (_). All letters are in upper case.
Property	In JSON objects, a key/value pair is conventionally referred to as a “property”. The term “attribute” is not used in this document. This is to align with the terminology used in OpenAPI Specification and JSON Schema → {x} JSON Schema - object

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (https://www.ietf.org/rfc/rfc2119.txt).

Audience

This document is intended to be used by software engineers and architects who are already experienced in the use of REST APIs using JSON.

These can be engineers active in different phases of REST API standards development -

- Working directly for DCSA on standards development.

- Acting as subject matter experts, advising DCSA on standards development.
- Implementing applications that act as API providers.
- Implementing applications that act as API consumers.
- Working on related applications or for related standards bodies with a general interest in DCSA standardisation.

Introduction

This chapter describes the purpose, scope and structure of this document.

This document is influenced by the work provided by Zalando, *"Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando"*, available under the Creative Commons Attribution 4.0 International license → restful-api-guidelines/chapters at main · zalando/restful-api-guidelines. Formally there is alignment to commit number 8158217, the active version at the time that this document was finalised → GitHub - zalando/restful-api-guidelines at 815821712a263a1afa0bd8c1e7785a4222b5efbe. Guidelines in this document have priority over guidelines in the Zalando repository.

Despite the existence of that work, and many other guides on API Design Principles, DCSA maintains this document for the following reasons -

1. A vertical standards body differs from a general API provider, with a focus on *interoperability* across implementations by different API providers.
2. A standards body is concerned with the concept of *conformance*. This document also functions as a conformance checklist.
3. DCSA needs to be specific, or to diverge from other opinions, on certain topics. This document provides a means to do so.

Where there is a question regarding some aspect of API Design Principles that is not covered in this document, it is recommended to follow the Zalando guidelines. Where the two documents differ, this document takes precedence.

Purpose and Scope

The purpose of this document is as follows:

1. Establish principles on how APIs should be designed in the digital container shipping context. The guidelines describe the requirements of DCSA-conformant APIs.
2. Establish principles for the implementation of APIs in the digital container shipping context. The guidelines provide a "common rule book" that allows API providers and API consumers to collaborate in an interoperable API ecosystem.

The context of this document is that it describes principles for "integration APIs", sometimes referred to as "business-to-business APIs", "server-to-server APIs" or "partner APIs". This is aligned with the scope and purpose of DCSA standardisation. This document does not describe APIs for other purposes, and specifically is not intended for "backend for frontend" APIs.

Note that the title of this document includes the term "implementation" to emphasise this dual role. DCSA has intentionally combined the two into one document in order to make the division of responsibilities clear. This division of responsibilities is intended to encourage and facilitate an interoperable ecosystem for APIs.

Transitioning from "DCSA API Design Principles 1.1" to This Document.

When an API transitions from "DCSA API Design Principles 1.1" to this document, this MUST be performed with a new version of the API.

Whether this is a breaking change requiring a new <MAJOR> version of the API is to be determined by applying the conformance criteria stated in this document. In most cases the transition will occur on a new <MAJOR> version boundary.

Conformance of Implementations

An implementation is considered to be conformant to a particular DCSA API standard if it meets the following criteria -

1. Passes according to the conformance checklists in this document.
2. Passes all of the conformance criteria and tests that are defined as part of the standard in question.
3. All messages are valid according to the relevant OpenAPI Specification schema definitions that are part of the standard in question, and do not include any additional properties that are not specified in those schema definitions.

Using Conformance Checklists in this Document

This document can be used by different actors:

- API designers
- API providers
- API consumers

This document defines some of the roles and responsibilities of these actors in the realisation of an *interoperable API ecosystem*. Interoperability is ensured through a process known as conformance.

As an actor in the ecosystem, it is possible to use the checklists in this document as part of a conformance assessment. The checklists do not validate “best practices” (implementation advice or “SHOULD” recommendations), they only assess the essential requirements to create an interoperable ecosystem.

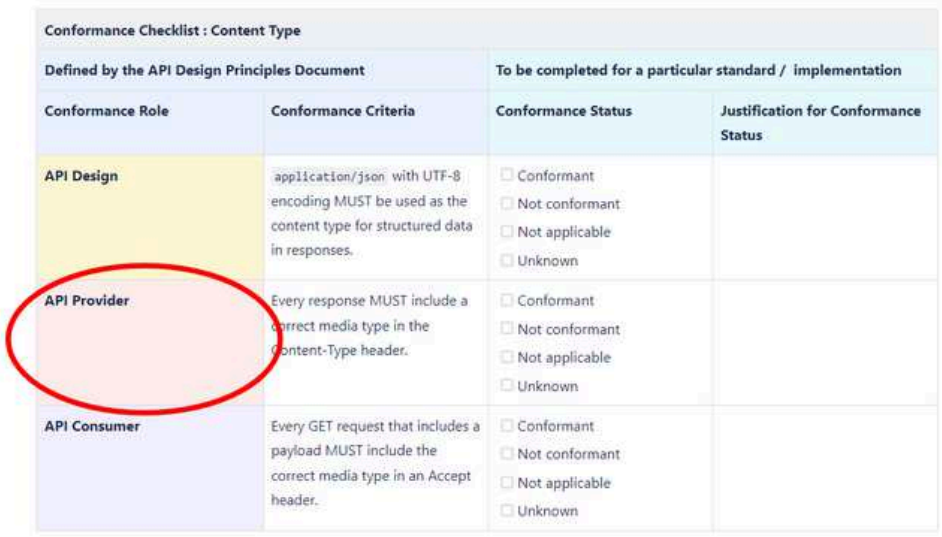
As an API provider or API consumer, there will be guidance and tooling available to evaluate conformance of each specific standard. This document provides only the generic conformance guide for all API standards.

This document is intended as a self-assessment conformance guide.

To use this document as a conformance checklist, proceed as follows -

Conformance Process Step	**Details**
1. Identify your role	1. An **API designer** creating or maintaining DCSA APIs or governed extensions to DCSA APIs. 2. An **API provider** implementing (“adopting”) one or more DCSA APIs in order to offer a service to one or more API consumers. Note that offering this service MAY include sending HTTP requests to API consumers if the service includes webhooks or callbacks. Being an API provider is a business role, not a technical role. 3. An **API consumer** implementing (“adopting”) one or more DCSA APIs in order to use a service from one or more API providers. Note that consuming this service MAY include receiving HTTP requests if the service includes webhooks or callbacks. Being an API consumer is a business role, not a technical role. There can be situations where you are acting in multiple roles (e.g. API provider and API consumer). In this case, perform the evaluation for *all* roles that are relevant.
2. Go through the conformance checklist tables in this document, identifying the areas in the	For example, if you are an API provider, it is only necessary to evaluate the conformance criteria for rows marked as “API Provider”. Simply leave the other rows blank (it is not necessary to fill in “not applicable” for the other rows).

tables that are relevant for your role.

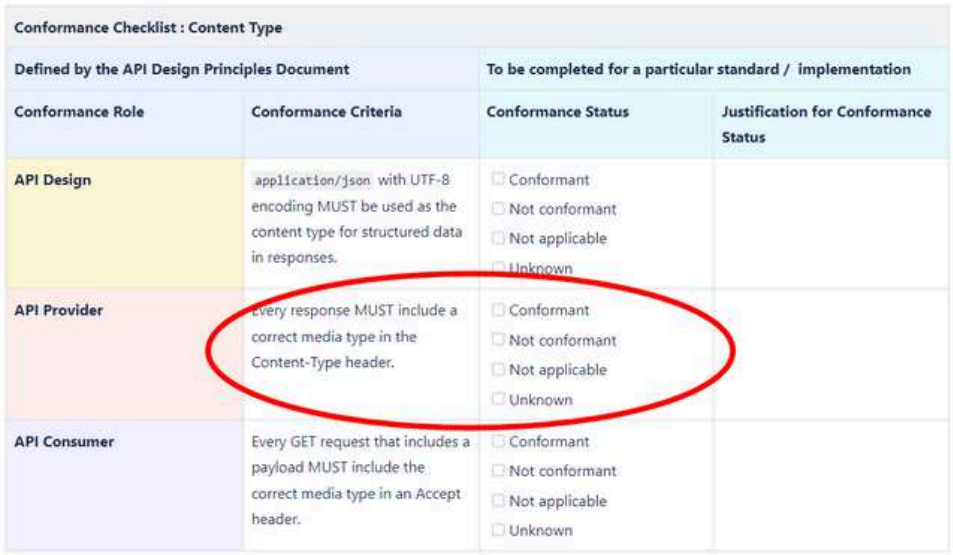


Conformance Checklist : Content Type			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	`application/json` with UTF-8 encoding MUST be used as the content type for structured data in responses.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Every response MUST include a correct media type in the Content-Type header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Every GET request that includes a payload MUST include the correct media type in an Accept header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

For some conformance checklists, there may be no evaluation required for your particular role.

3. Evaluate each criterion.

Consider each criterion in turn.



Conformance Checklist : Content Type			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	`application/json` with UTF-8 encoding MUST be used as the content type for structured data in responses.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Every response MUST include a correct media type in the Content-Type header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Every GET request that includes a payload MUST include the correct media type in an Accept header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

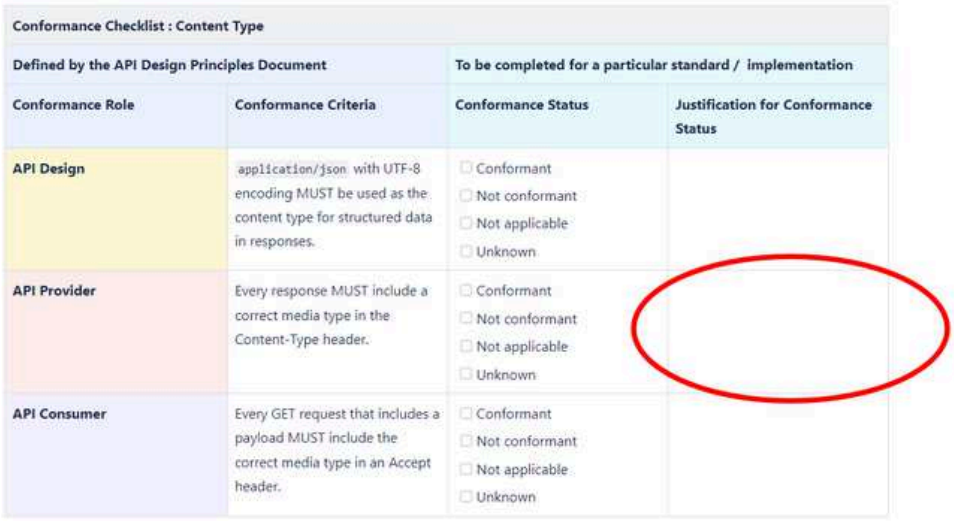
This document does not specify *how* to perform the evaluation, simply that the evaluation must be performed. Some criteria may not be applicable because they specify how to handle a situation that does not occur for the standard or implementation in question.

Depending on the outcome of the evaluation, select the appropriate checkbox -

- ☑ Conformant - The criterion is fully and completely met by the standard or implementation.
- ☐ Not conformant - The criterion is not achieved or only partially met by the standard or implementation.
- ☐ Not applicable - The situation described by the criterion does not apply to the standard or implementation.
- ☐ Unknown - It is not possible to evaluate conformance for this criterion. Either the criterion is not specific enough or it is not possible to perform the evaluation due to constraints.

4. Justify the evaluation.

For each point of evaluation, explain why a particular conformance status has been chosen. An explanation can be given describing how the evaluation took place. An explanation can be given if there is non-conformance under certain circumstances.

	
5. Finalise and take decision on conformance to the principles.	If the standard or application is evaluated as “Conformant” or “Not applicable” for all criteria, then the situation is clear - conformance to the principles has been achieved. There may be grounds for considering the standard or application to be conformant if some criteria are evaluated as “Unknown” or “Non conformant”. For example, the “non conformant” may be considered to not impact interoperability in practice. This document is a self-assessment guide, so designers and implementers will need to take a decision in such cases. At some later date DCSA may provide guidance on how to come to a conformance “score” or “rating”.
6. If you are acting as an API provider or API consumer, continue with conformance evaluation of criteria that are specific to each standard you are implementing.	If you are acting in the role of API provider or API consumer, conformance with this document is only one part of the conformance process. There will be guidance and tooling available to evaluate conformance to the specific criteria for each standard. Evaluating the criteria for each standard is a requirement in order to establish a conformant implementation.

Characteristics of Adoptable APIs

From multiple sources, a number of important characteristics for adoptable APIs have been identified as shown in the table below. This set of characteristics is used as the basis for the API Design and Implementation Principles.

Index	Characteristics	Descriptions
1	Suitable	APIs provide a well-defined service that provides value for API consumers by having clearly defined responsibilities and an appropriate level of abstraction.
2	Interoperable	Minimise divergence between implementations and ensure the API is interoperable in the wider ecosystem.
3	Stable	The occurrence and impact of major changes are minimised.
4	Secure	Measures and practices are implemented to counteract threats.
5	Performant	The performance consequences of API design are considered, monitored and optimised.
6	Well-documented	Up-to-date, complete and easy-to-read documentation is available to facilitate adoption.

Table: API quality characteristics

The structure of this document follows the order of the API characteristics given in the table above.

Event Push

Some DCSA APIs include or rely on event push APIs. DCSA maintains separate publications that describe how such API features are to be constructed. This document includes generic principles for webhooks and callbacks, but it does not define a detailed approach for event push.

---

1. Suitable

Good APIs are suitable regarding their functionality and responsibilities. Suitable APIs provide a well-defined service that provides value for API consumers by having clearly defined responsibilities and an appropriate level of abstraction.

This means it should be easy to understand how to interact with the APIs.

To achieve this objective, requirements are identified below:

- The API is constructed in a way that simplifies consuming applications.
- The API is constructed according to a minimalistic approach – less is more.
- The API is aligned to the use cases in the identified business process model(s).

Conformance Checklist : URLs			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Designed from an API consumer-centric perspective.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The API is constructed according to a minimalistic approach.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The API is aligned to the use cases in the identified business process model(s).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

---

2. Interoperable

Industry standard APIs need to be interoperable. This interoperability can be considered from three perspectives -

1. Minimise the possibilities for different implementations of the same API to diverge so that behaviour is predictable and consistent for all API consumers.

2. Design APIs based on widely-used underlying technical standards, so that consistent implementation is possible using a wide range of development tools and deployment platforms.
3. Design APIs based on adjacent vertical industry standards where possible, to allow for integration into the larger ecosystem.

Basis on a Specific set of Design Principles

A DCSA API MUST explicitly state in its documentation that it is based on a specific set of design principles - usually a reference to this document.

This is required because the design principles specify some of the semantics of the API, and also clarify the rules for backward compatibility.

Conformance Checklist : Design Principles			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The API documentation MUST state the set of the design principles upon which it is based.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

OpenAPI Specification

DCSA REST APIs have their basis in the OpenAPI Specification.

REST APIs MUST be based on one of the following versions of the OpenAPI Specification -

- Version 3.0.X
- Version 3.1.X

Where version 3.1 is used, the REST API MUST be aligned with JSON Schema 2020-12.

A REST API MAY be specified using both versions 3.0 and 3.1 providing that identical, valid JSON documents to be exchanged are equivalent in both cases. Typically in such cases the version 3.0 schema has a looser schema validation, but specifies in descriptions and documentation that constraints specified in the 3.1 schema are still applicable. Conformance tests are common for both API specifications.

Where this dual specification approach is used, the specification MAY be split up for different parts of the API where necessary. In particular, webhooks described using OpenAPI Specification 3.1 MAY be described in a separate API using OpenAPI Specification 3.0.3. Only the description is different - the wire protocol is identical in both cases.

If an upgrade from OpenAPI Specification 3.0 to OpenAPI Specification 3.1 is made without specifying an OpenAPI Specification 3.0 equivalent, then this is considered to be a breaking change requiring a new <MAJOR> version number of the corresponding DCSA API standard.

Note that the OpenAPI Initiative themselves do not consider that the transition from OpenAPI Specification 3.0 to 3.1 is fully backward compatible from a “semantic versioning” perspective → OpenAPI Specification 3.1.0 Released - OpenAPI Initiative

Extensibility

Any API defined using OpenAPI Specification is extensible by default. There is a keyword in OpenAPI Specification called `additionalProperties` which defaults to `true`.

In DCSA APIs, `additionalProperties` is usually set to `true`, either explicitly or implicitly but there is a very specific and limited reason for this. This is to allow API consumers to implement schema validation, while still being able to be backward compatible when API providers upgrade to a new <MINOR> version of the standard.

Even though adding properties is formally allowed according to the schema definition, API providers and API consumers MUST NOT send any properties that are not explicitly specified in the API schema.

API providers MUST implement both request generation and response handling in a way that would pass validation if `additionalProperties` were set to `false` throughout in the schema, unless there is a specific instruction to do otherwise for a given API.

API consumers MUST implement request generation in a way that would pass validation if `additionalPropeties` were set to `false`. API consumers MUST implement any response validation (if performed) exactly matching the actual published schema (therefore allowing additional properties because this allows API providers to upgrade to a new minor version). Response validation is optional, but recommended, for API consumers.

Conformance Checklist : OpenAPI Specification			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The API specification MUST be conformant with at least one of OpenAPI Specification 3.0.3 or OpenAPI Specification 3.1.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If the API specification is conformant with OpenAPI Specification 3.1, it MUST also be conformant with JSON Schema 2020-12.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All properties present in responses MUST be explicitly described in the DCSA API specification. Responses MUST NOT include any other properties unless the extension mechanism (see below) is used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	All properties present in requests MUST be explicitly described in the DCSA API specification. Requests MUST NOT include any other properties unless the extension mechanism (see below) is used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

JSON

JSON is the format used for structured data in DCSA REST APIs.

Requests with methods other that PATCH, where they include a body, MUST be made using the `application/json` media type for JavaScript Object Notation (JSON).

Requests with the method PATCH, where they include a body, MUST be made using one of the following media types for JavaScript Object Notation (JSON) -

- `application/json`
- `application/json-patch+json`
- `application/merge-patch+json`

Responses, except for those responses consisting only of a single binary payload (e.g. retrieving a JPEG image or a PDF document), MUST be constructed using the `application/json` media type for JavaScript Object Notation (JSON).

Conformance Checklist : JSON			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
This conformance area is covered by the conformance checklist on content type.			

UTF-8

Encoding of structured data using JSON MUST be UTF-8.

Clarification and usage guidelines (not normative)
UTF-8 does *not* imply 8-bit or 16-bit characters - it implies support for all valid Unicode code points within the range U+0000..U+10FFFF. For example, it can encode all Chinese characters defined in the Unicode standard. There is a complete description of this provided by the Unicode Consortium→ {x} FAQ - UTF-8, UTF-16, UTF-32 & BOM The use of Unicode and UTF-8 encoding are both implied by the use of the `application/json` media type as the default encoding defined in RFC4627. For this reason there are no additional conformance requirements stated in this section for UTF-8.

Conformance Checklist : UTF-8			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
This conformance area is covered by the conformance checklist on content type.			

Content Type

The term “content type” is used in this section as opposed to “media type”. This is because this section aligns the HTTP standard, where this convention is used → {x} Content-Type fields in MIME

Handling of different content types in HTTP occurs via content negotiation. When sending a payload in a particular format or responding with a payload in a particular format the format media type MUST be set in the Content-Type header.

API providers MUST use a strict content negotiation scheme for all requests.

When the payload is only binary data (e.g. a PDF document), it MUST NOT be encoded. The format returned will be made available during content negotiation.

When binary data is part of the payload (typically a property in a JSON object), it MUST be Base64 encoded.

The content types required for different PATCH semantics are described in the section “HTTP Methods”.

Conformance Checklist : Content Type			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	`application/json` with UTF-8 encoding MUST be used as the content type for structured data in responses.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If a response payload includes binary data as *part* of a larger structured response, it MUST be Base64 encoded.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If a response payload is *only* binary data it MUST not be (re)encoded. It MUST be returned according to the native format for the content type in question.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Every response MUST include a correct media type in the Content-Type header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	When a PUT, POST or PATCH request is received with a request body in a content type that is not supported the API Provider MUST respond with HTTP response status code 415.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	When a GET request is received without an appropriate Accept header, the API Provider MUST respond with HTTP response status code 406.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Every GET request MUST include the correct media type in an Accept header.	☐ Conformant ☐ Not conformant ☐ Not applicable	

		☐ Unknown	
API Consumer	Every request that includes a body MUST include the correct media type in the Content-Type header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

URLs

URLs have a prefix determined by the API provider, which is not standardised. This URL prefix determines the “location” from which the standard API is accessed by the API consumer.

The standard part of the URL MUST follow the design and naming conventions below.

Version Identifier

The first part of the standard URL (after the prefix) MUST indicate the <MAJOR> version of the standard in question.

Design

URLs MUST point to a resource (entity, process).

URLs MUST use the path separator “/” to indicate hierarchical relations.

URLs MUST consist of nouns and MUST NOT be verbs.

Naming Convention

Kebab-case MUST be used in URLs.

Path parameters MUST be consistent with property names. Path parameters referring to property names MUST use camelCase.

Query parameters MUST use camelCase.

Clarification and usage guidelines (not normative)
Example syntax for API URLs in production, Internet-facing applications -  It is important for interoperability that the <MINOR> version or <PATCH> version is not included. The URL identifies the contract, not the application version. DCSA APIs are intended to be used within a large ecosystem where it is not practical to align upgrades by all API consumers and API providers. It is expected behaviour that API providers can upgrade (but not downgrade) their <MINOR> version independently of, and prior to upgrade by, their API consumers. A given API provider endpoint is expected to correctly offer service to requests that conform to <MINOR> versions of the relevant standard that are equal to or lower (but not higher!) that the currently implemented <MINOR> version.

Conformance Checklist : URLs	
Defined by the API Design Principles Document	**To be completed for a particular standard / implementation**

Conformance Role	Conformance Criteria	Conformance Status	Justification for Conformance Status
API Design	URLs MUST point to a resource.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	URLs MUST consist of nouns.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	URLs MUST use the path separator “/” to indicate hierarchical relations.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	`kebab-case` MUST be used in URLs	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Path parameters MUST be consistent with property names	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Path parameters referring to property names MUST use `camelCase`.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Query parameters MUST use `camelCase`.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	The API MUST be deployed to a path that, after the prefix for the location of the API, includes only the <MAJOR> version number of the standard, preceded by a `v`, followed only by the URL path defined according to the standard. For example, version 3.1 MUST include `/v3/` in the path.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

HTTP Methods

Defined HTTP Methods

HTTP methods (GET, PUT, PATCH, POST, DELETE, HEAD) MUST be used to indicate the required action, following the general conventions of REST APIs.

HTTP response status codes used in API designs MUST follow the conventions set down in the Zalando API Guidelines. (restful-api-guidelines/chapters/http-requests.adoc at 815821712a263a1afa0bd8c1e7785a4222b5efbe · zalando/restful-api-guidelines)
Implementers MAY use other parts of the HTTP protocol, for example redirect, as is conventional. It is not the function of this document to specify the HTTP protocol or assess conformance with it - this is an implicit lower layer of REST APIs and it is expected that both API providers and API consumers follow the conventions and standards of HTTP.

HTTP Method	**Purpose**	**Non-normative Guidance on Successful Response Status Code** ***(Zalando guidelines are leading)***	**Mandatory Idempotency**
GET	Read either a single resource or a collection resource.	200 OK 202 with empty body if there is a pending asynchronous process preventing a full response.	Not applicable because a GET method MUST NOT cause a state change that is observable to API consumers.
PUT	Update (or sometimes create) entire resources – single or collection resources.	200 if there is no new resource created and there is a response message representing the updated state. 201 if there is a new resource created. 202 if the PUT has been accepted and is likely to be enacted but not yet enacted. 204 if the PUT has been enacted and no further information is supplied.	Yes Putting the same resource more than once is required to be idempotent and to act on the same single resource instance.

PATCH	Update parts of the resource objects. PATCH semantics have a risk of being ambiguous, which represents a risk to interoperability. For this reason it is preferred to restrict updates to use PUT if possible. The semantics of every PATCH operation MUST be clear in API documentation. Simple PATCH semantics are preferred. Updating a single, pre-defined property to indicate a state change is a typical example of such semantics. Such semantics are preferred over allowing updates of arbitrary properties because interoperability is easier to demonstrate (fewer test cases) and there is a more direct alignment to a defined step in a business process. If PATCH is used to provide general updates of some arbitrary combination of properties, then the conventions specified in *"Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando"* MUST be followed → restful-api-guidelines/chapters/http-requests.adoc at 815821712a263a1afa0bd8c1e7785a4222b5efbe · zalando/restful-api-guidelines. Note the use of either the JSON Merge Patch or JSON Patch semantics and media types, corresponding to IETF RFC7396 and IETF RFC6902 respectively. APIs MUST NOT accept multiple types of patch semantics or media types on the same endpoint.	200 if the PATCH has been enacted and there is a response message representing the updated state. 202 if the PATCH has been accepted and is likely to be enacted but not yet enacted. 204 if the PATCH has been enacted and no further information is supplied.	No Patching the same resource more than once is not required to be idempotent. A good example where idempotency is not expected would be a PATCH request to add an element to an array. If idempotent semantics are necessary for a given API, this MUST be specified in API descriptions.
POST	Used to create single resources on a collection resource endpoint, but other semantics on single resources endpoint are equally possible. The semantic for collection endpoints is best described as "please add the enclosed representation to the collection resource identified by the URL". The semantic for single resource endpoints is best described as "please execute the given well specified request on the resource identified by the URL".	200 if a resource is updated and new state is returned. 201 if the resource is created (this SHOULD be the usual successful response in DCSA APIs) 202 if the POST has been accepted and is likely to be enacted but not yet enacted. 204 if a resource is updated and no	No Posting the same resource twice is not required to be idempotent and MAY result in multiple resources being created. If idempotent semantics are necessary for a given API, this MUST be specified in API descriptions.

		further information is supplied.	
DELETE	Delete resources.	200 if the DELETE has been enacted and there is a response message representing the updated state. 202 if the DELETE has been accepted and is likely to be enacted but not yet enacted. 204 if the DELETE has been enacted and no further information is supplied.	Yes, at least by a strict definition of idempotency. It is expected that any calls following the first would result in an error status code, but the state of the system of record would be the same with one call or with more calls.
HEAD	Returns headers only. In DCSA APIs this MAY be used as a basic health check, or to validate the existence or version (via ETag) of a resource. Where HEAD is required for correct functioning and interoperability, this MUST be explicitly stated in API documentation.	200 OK	Not applicable because a HEAD method MUST NOT cause a state change that is observable to API consumers.

Clarification and usage guidelines (not normative)

The “202 Accepted” response status code is appropriate when the record(s) in question are not yet committed. It may be necessary to reply with some content, in particular to provide an identifier for a newly created record that is not yet committed.

The state of the record may then remain undefined for a period of time. During this period of time other API calls to access the record MAY return a “202 Accepted” response status code.

In the case that there is a further call to update the record while the previous update is not yet committed, applications MAY either accept the update (presumably queuing it) or return a “409 Conflict”.

Control of concurrent updates

Any given sequence of GET / PUT / PATCH where concurrency is possible can lead to results that API consumers do not intend. Especially PATCH of a resource that was modified by a certain API consumer since the previous operation by a different API consumer can lead to unexpected results and incorrect behaviour. Where API designers are concerned about this, one of the following two approaches MUST be used:

1. Use the ETag header to identify the version held by the API consumer, following the conventions specified in “*Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando*” → restful-api-guidelines/chapters/http-headers.adoc at 815821712a263a1afa0bd8c1e7785a4222b5efbe · zalando/restful-api-guidelines
2. Use a single property that serves as a version identifier. The value of this property MUST be unique for a given version, within API provider scope, under all circumstances. The recommended name for such a property is versionReference.

Option 1 is preferred because it can take advantage of generic HTTP infrastructure for caching and similar. It MAY be optional for API providers to implement ETag but where it is allowed in the specification it MUST be implemented by API consumers.

Whichever option is selected, this MUST be explicitly stated in the API specification.

The version identifier MUST be opaque to the API consumer, whichever option is selected.

Conformance Checklist : HTTP Methods			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	HTTP methods GET, PUT, PATCH, POST, DELETE, HEAD are used according to the common semantics.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	A GET endpoint MUST NOT cause any state change observable to any API consumer.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If idempotent behaviour is needed for a PATCH operation, this is explicitly stated in API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where JSON Patch semantics are specified, the media type `application/json-patch+json` is also specified.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where JSON Merge Patch semantics are specified, the media type `application/merge-patch+json` is also specified.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Every PATCH endpoint MUST only support one media type and MUST also only support one type of semantics.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If idempotent behaviour is needed for a POST operation, this is explicitly stated in API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Design	If concurrency control is required, this MUST be specified either using ETag or a specific property.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If support of the HEAD method is required for correct interoperability, this MUST be stated in API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The conventions stated in the Zalando guidelines MUST be followed in response status codes.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All GET endpoints MUST NOT result in any observable state change for any API consumer.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All PUT endpoints MUST be idempotent.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	PATCH semantics and idempotent behaviour MUST be implemented as described in the API documentation provided.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where a PATCH method uses the media type `application/json-patch+json`, only this media type MUST be accepted.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where a PATCH method uses the media type `application/merge-patch+json`, only this media type MUST be accepted.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	POST semantics and idempotent behaviour MUST be implemented as described in the API documentation provided.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	If ETag is implemented, it MUST follow the conventions specified in *"Developing Restful*	☐ Conformant ☐ Not conformant ☐ Not applicable	

	APIs: A Comprehensive Set of Guidelines by Zalando".	☐ Unknown	
API Provider	If ETag is implemented and the API documentation states that the HEAD method is required for correct operation, then a HEAD request MUST correctly return the ETag header.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	If ETag is specified as being allowed in the API specification then this MUST be supported, following the conventions specified in *"Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando".*	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Where a PATCH method uses the media type `application/json-patch+json`, only this media type MUST be provided in a request.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Where a PATCH method uses the media type `application/merge-patch+json`, only this media type MUST be provided in a request.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Custom headers

Custom headers MUST NOT use the prefix `x-` or `X-`. Please see Deprecating the "X-" Prefix and Similar Constructs in Application Protocols.

The following table contains assigned DCSA headers:

Header	Description
API-Version	In communication from API consumer to API provider: used to specify the <MAJOR> version of the standard requested in use. In communication from API provider to API consumer: used to indicate the <MAJOR>.<MINOR>.<PATCH> version of the standard in use.

Clarification and usage guidelines (not normative)
Communicating the <MAJOR> version in both the header and in the URL path is redundant. This is consequence of being a standards body that does not dictate the approach of API providers. Different API providers have different approaches to request routing within their enterprise architecture, therefore both are provided to allow flexibility in this area. It is a responsibility of API Consumers to ensure that both are consistent. If they are not then the result is not defined.

Table: Custom headers

Conformance Checklist : Custom Headers			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The API specification MUST NOT include any headers using the `x-` or `X-` prefix.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
See also section below on "Versioning".			

Sorting

Sorting of result sets MUST be limited to specified properties. The parameter MUST be restricted to the values `ASC` (ascending order) and `DESC` (descending order) describing the sort direction. The direction MAY be omitted. If omitted ascending order MUST be used. A colon `:` is used to separate the property to sort by and the direction. Multiple properties can be used, in which case each property is separated by a comma `,` . The following table shows some examples:

Query Parameter	**Explanation**
`sort=name`	Sort by the name property in ascending (default) order.
`sort=name:DESC`	Sort by the name property in descending order.
`sort=name:ASC`	Sort by the name property in ascending order.
`sort=name,age:DESC`	Sort first by the name property in ascending order (default when order is not specified). If two equal names exist, then sort those items by age in descending order.
`sort=name:DESC,age:ASC`	Sort first by the name property in descending order. In the example - if two equal names exist, then sort those items by age in ascending order.

Table: Examples of sorting

The sort parameter is optional in API definitions. If it is not present, then there are two options -

1. Ordering is undefined meaning that API consumers MUST NOT be dependent on the order for correct behaviour.
2. Ordering is fixed and the property or properties used for sorting are specified in the API definition.

Option 1 is the default.

Sort order of date, time and datetime properties MUST be based on their relative positions represented on a timeline in a single timezone.

Clarification and usage guidelines (not normative)
Consider a collection of datetime with the following elements - `["2020-07-05T08:30+01:00", "2020-07-05T08:15+00:00", "2020-07-05T08:45+02:00"]` The correct *ascending* order sort order is as follows -

```
["2020-07-05T08:45+02:00", "2020-07-05T08:30+01:00", "2020-07-05T08:15+00:00"]
```

This is because an event at 08:45 in time zone +02:00 occurs before an event at 08:15 in time zone +00:00.

Sort order of Strings and Enums MUST be based on Unicode code point order.

Conformance Checklist : Sorting			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Where sorting behaviour is expected to be repeatable, API documentation MUST specify the expected behaviour.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where sorting is supported, the API documentation MUST specify which properties can be used for sorting.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where sort order can be specified by the API consumer, the query parameter “sort” MUST be used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where sort order can be specified by the API consumer, the possible values of the “sort” query parameter MUST correspond to the names of properties in the collection in question.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where date, time or datetime properties are used for sorting, ordering MUST NOT be affected by the representation of the collection elements in (potentially different) time zones. The ordering MUST be based on their relative positions represented on a timeline in a single timezone.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where string properties are used for sorting, ordering MUST be determined based on Unicode code point order.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Design	Where sorting is possible based on multiple properties, the properties form a comma-separated list as possible values of the “sort” parameter.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	When a “sort” query parameter is specified in API documentation, this query parameter MUST be parsed and used to order the returned collection.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where date, time or datetime properties are used for sorting, ordering MUST NOT be affected by the representation of the collection elements in (potentially different) time zones. The ordering MUST be based on their relative positions represented on a timeline in a single timezone.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where String properties are used for sorting, the ordering MUST be determined according to the Unicode code point order.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Where the ordering of a collection is undefined, there MUST NOT be a dependency on the order returned by the API provider for correct behaviour of the application.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Filtering

There is no overarching approach to filtering. API designers are free to implement filtering as they see fit.

Pagination

API definitions of GET requests MAY specify pagination. This is not mandatory for all collection result sets in an API design, but where this is specified it MUST NOT be optional for API providers. Where pagination is not specified it MUST NOT be provided.

Pagination is controlled by two query parameters as shown in the table below -

Query Parameter	Description	Example
limit	Limits the number of items returned in a collection-response.	limit=20

	For any given endpoint, three values can be specified in API definitions with respect to limit - 1. Minimum. The minimum possible value of limit. 2. Default. The default value of limit if the API consumer does not specify the limit. 3. Maximum. The maximum possible value of limit. If an API endpoint does not specify any of these values then the following standard values MUST be used. • The standard minimum value for limit is 1. • The standard default value for limit is 100. • The standard maximum value for limit is 1000.	
cursor	An API provider-generated pointer to a page in a collection response. This is never generated by the API consumer, is opaque to the API consumer and MUST NOT be modified by the API consumer.	cursor=f8dcb4c4-3fba-41ef-9991-2be97dad57dc

Table: Query parameters for controlling result set pagination

Clarification and usage guidelines (not normative)
The difference between a cursor and an offset is about semantics and acceptable behaviour of API consumers. It isn't about the format of the cursor. In an API that did use the concept of offsets (which MUST NOT be used in DCSA APIs), there might be a property such as `offset=100`, which would mean to start the page from the 101st element in the result set (counting from element zero). Importantly, API consumers would be free to set this parameter to any value they chose, so they could freely jump around in the result set. This behaviour MUST NOT be used in DCSA APIs because implementation is too dependent on using certain application architectures, and it is desired to allow a great deal of freedom in application architecture. It is allowed that the value of the `cursor` property is, in reality, an offset. It is allowed to have a property such as `cursor=100`, that also results in selecting a page from the 101st element in the result set. However, API consumers MUST NOT parse or set this value; it is opaque to them. The format of the cursor MAY change at any time. API consumers are only allowed to use the cursor values found in `Link` headers. The main point about `cursor` is the fact that the value MUST NOT be manipulated or interpreted by API consumers.

It is typical, but not mandatory, that pagination is combined with sorting (which is described in the section above).

An endpoint MAY be defined that supports the limit parameter without providing pagination. This is an unusual but possible case. This MUST be described in API documentation.

Pagination is provided by API providers in responses through the use of Link headers as defined by IETF RFC5988 → https://www.rfc-editor.org/rfc/rfc5988.txt.

If pagination is implemented and there is sufficient data available then a link header to the next page MUST be provided.

Link headers to current, previous, first and last page MAY be provided.

Links MUST include a cursor value in order to allow the API provider to identify the relevant result set.

URLs used as pagination links are opaque for API consumers and MUST NOT be modified by API consumers. Note that this implies that API consumers cannot change pagination or sorting parameters when making a series of pagination requests for the same result set.

URLs used for pagination links MUST NOT be stored for reuse by API consumers outside of the context of the original request. These URLs are not long-lived and are intended for use only in the immediate context following on from the original request.

Link header relation-type	Explanation	Example Link header value
First-Page	An optional link to the first page.	<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="First-Page"
Previous-Page	An optional link to the previous page. Previous-Page header link MUST be omitted if the current page is the first page.	<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Previous-Page"
Next-Page	A link to the next page. Next-Page header link MUST be omitted if the current page is the last page, otherwise it MUST be provided.	<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Next-Page"
Last-Page	An optional link to the last page. Last-Page header link MAY be omitted if the current page is the last page.	<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Last-Page"
Current-Page	An optional link to the current page.	<https://api.dcsa.org/events?limit=20&cursor=xxx>; rel="Current-Page"

Table: Pagination link headers

The API provider generates all links to pages relative to the current page requested. It is not possible for the API consumer to request a specific page – the API consumer MUST select a page provided by the API provider. The API provider MUST provide a link to the next page where there is sufficient data available.

Pagination can have different consistency semantics for different endpoints. There are two options -

1. A series of calls using pagination links from a single initial query are guaranteed to form a consistent and contiguous result set.
2. A series of calls using pagination links from a single initial query are not guaranteed to form a consistent and contiguous result set - there can be gaps, duplications, deletions etc. between the responses.

These two different options for semantics form part of the API definition. Option 1 is the default. Where the semantics of option 2 are provided, this MUST be explicitly documented.

There is no requirement on API providers to support pagination links for a specific period of time from the original request or from the last request for the result set. It is expected behaviour that pagination links will expire after a period of time. The API provider MUST return 404 Not Found if the result set has expired. API consumers SHOULD anticipate this possibility.

Clarification and usage guidelines (not normative)
From an HTTP protocol standpoint, it would be more accurate to return “410 Gone” as the status code for an expired pagination link, instead of “404 Not Found”. However, that would require API providers to be able to determine the difference between a pagination URL that had once existed and some other arbitrary URL that is similar to a pagination link but never existed.

That would mean that an API provider would still need to retain some state information about expired links, while the reason for allowing expiration is exactly to prevent the need to retain such state. This is the reason that it was decided that “404 Not Found” is a more practical choice.

Conformance Checklist : Pagination			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Where pagination is required this MUST be specifically stated in the API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where pagination is required this MUST be specified using HTTP response link headers with the relation-types First-Page, Previous-Page, Next-Page, Last-Page, Current-Page.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Pagination MUST be based on the use of cursors and MUST NOT be based on the use of offsets.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The value of the cursor MUST NOT be specified. It MUST be open to the API provider to determine the format for this value.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where pagination is required, the response link header with relation-type Next-Page MUST be provided in all responses except for the last page where this header MUST NOT be provided.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where pagination semantics do not guarantee a consistent and contiguous result set this MUST be specified in API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Design	Where minimum, default or maximum values for limit are different from the standard values, this MUST be explicit in the API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where pagination is specified, this MUST be implemented.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where pagination is not specified, it MUST NOT be implemented.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Pagination MUST NOT be based on the use of offset values that the API consumer can modify. It is permitted to use offset values as part of the cursor value, providing that this is opaque for the API consumer.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where pagination is specified for a given endpoint, that endpoint MUST include a response link header with relation-type Next-Page except for the last page where this header MUST NOT be provided.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where pagination semantics guarantee a consistent and contiguous result set this MUST be implemented as such. This might require state to be maintained by the API provider and a reference to that state to be provided using the cursor, but this might not be the case for some immutable data sets.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	The URLs in link headers for pagination MUST use the same base URL that was used for the initial request.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Link headers MUST be generated according to the	☐ Conformant	

	syntax and encoding specified in RFC5988.	☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where the value of limit provided by an API consumer in a request is outside of the allowed range this MUST result in a “400 Bad Request” error response.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where the syntax for the value of the sort parameter provided by an API consumer in a request does not conform to the standard this MUST result in a “400 Bad Request” error response.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When a request is received for a pagination link that can no longer be retrieved (typically because it has expired), this MUST result in a “404 Not Found” error response.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	URLs provided in pagination link headers MUST NOT be modified.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Where pagination semantics do not guarantee a consistent and contiguous result, correct behaviour MUST be maintained if the result set is modified between pagination calls.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	URLs provided in pagination link headers MUST NOT be stored for reuse in some other context.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Use MAY be made of the pagination link headers with relation-types First-Page, Previous-Page, Last-Page, Current-Page to improve performance. If the API provider does not set these link headers, correct behaviour MUST be maintained. For example, by repeating the initial	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

	query and/or using a series of Next-Page link headers.		

Patterns

The HATEOAS pattern (“Hypermedia as the Engine of Application State”) MUST NOT be used in API designs, with some limited exceptions such as the Link headers for pagination. Resources are identified in responses by identifier properties, and not by URLs. The API consumer must construct the URL based on these identifier properties. This allows important decoupling of standard and version life-cycles between different APIs, and allows references to cross organisational boundaries where required.

JSON-LD and similar “semantic web” concepts MUST NOT be used in API designs.

Clarification and usage guidelines (not normative)
There is nothing wrong with these patterns but mixing and matching paradigms makes interoperability harder.

Conformance Checklist: Patterns			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The HATEOAS pattern (“Hypermedia as the Engine of Application State”) MUST NOT be used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	JSON-LD and similar “semantic web” concepts MUST NOT be used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Property names

Naming conventions based on types

Type of Property	**Naming Convention**
object	Property names of objects MUST be in camelCase.
array	Property names of arrays MUST be plural.
collection	Property names of collections MUST be plural.
boolean	Boolean properties MUST be prefixed by either is or has for singular properties. Boolean properties MUST be prefixed by either are or have for plural properties.
time	Properties that represent a time that is separate from any particular date (e.g. a regular time in a timetable - but be careful with time zones!) MUST use the suffix Time.
datetime	Properties that represent an exact moment in time (a calendar date, with a time and a

	time zone) MUST use the suffix DateTime.
date	Properties that represent a calendar date (representing a specific date in a specific time zone) MUST use the suffix Date.

General principles of naming

Property names MUST NOT imply a specific technology for persistence (e.g. a certain database type or a certain database design).

Where a property name pertains to a particular group or object (in the general sense), this SHOULD be indicated by a prefix only when it is necessary to resolve ambiguity or where there is a general convention in the container shipping industry to refer to a property in a certain way. For example, usually “length” would be preferred to “vesselLength” where the context in a vessel object is already clear.

The suffix Number (e.g. VesselIMONumber, VoyageNumber) MAY be used where this is a pre-existing, recognised term in the container shipping industry. For other properties this suffix MUST NOT be used. A property with this suffix MAY be a String and MAY include characters that are not numeric.

The suffix Code MUST be used to specify a set of abbreviations or reserved words (which MAY be an enum although this is not recommended), except where this identifies a unit of measurement.

The suffix Unit MUST be used to specify a unit of measurement e.g. kilogram, metre, nautical mile.

The suffix Units MUST be used to specify a quantity, especially a quantity of shipping containers.

Where there is a conflict between pre-existing, commonly understood terms in the container shipping industry and this set of conventions, the commonly understood term SHOULD be used.

Naming of Identifiers

DCSA uses the following naming conventions for identifiers in descending order of scope -

Name Part	Scope	Naming Convention
`Id`	Universally unique, including outside of DCSA scope, over all API providers.	The suffix Id is reserved unless it is a UUID. It is guaranteed to be unique in any context. This suffix MUST NOT be used for other properties.
`universal`	Unique within DCSA scope over all API providers.	The prefix universal indicates an identifier that is unique over the set of API providers. This is due to coordination of the identifier within the DCSA API ecosystem.
`Reference`	Unique for a given API provider.	The suffix Reference MUST resolve to a unique business object within the context of a particular API provider. It is not guaranteed to be unique over the set of API providers.
`Subreference`	Unique within a particular context for a given API provider.	The suffix Subreference MUST resolve to a unique business object within a narrower context that MUST be specified within a particular API standard. It is not guaranteed to be unique even within a single API provider scope.

Conformance Checklist : Property names			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**

API Design	Property names of objects MUST be in camelCase.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Property names of arrays MUST be pluralised.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Property names of collections MUST be pluralised.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Property names MUST NOT imply a specific technology for persistence (e.g. a certain database type or a certain database design).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Boolean properties MUST be prefixed by either is or has for singular properties. Boolean properties MUST be prefixed by either are or have for plural properties.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Reference MUST resolve to a unique business object within the context of a particular API provider.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Subreference MUST resolve to a unique business object within a narrower context that MUST be specified within a particular standard.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Id is reserved unless it is a UUID. It is guaranteed to be unique in any context. This suffix MUST NOT be used for other properties.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	This suffix Code MUST NOT be used for identifiers of dimensions (units of measure).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Properties that represent a calendar date (representing a	☐ Conformant ☐ Not conformant	

	specific date in a specific time zone) MUST use the suffix Date.	☐ Not applicable ☐ Unknown	
API Design	Properties that represent an exact moment in time (a calendar date, with a time and a time zone) MUST use the suffix DateTime.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Properties that represent an exact moment in time that is not associated with a particular calendar date MUST use the suffix Time.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Number MUST NOT be used for properties that do not represent an (already established) term in the container shipping industry.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The prefix universal MUST only be used for a property that is guaranteed to be unique within the entire DCSA ecosystem.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Unit MUST only be used for a property that indicates a unit of measurement.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The suffix Units MUST only be used for a property that indicates a quantity. This suffix MUST NOT be used for a property that indicates a unit of measurement.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Property values

Enum values and “Enum-like string” values

Clarification and usage guidelines (not normative)
Many properties in DCSA API standards would be typically considered to be enumerated types by software engineers. However, actually specifying these as enums in OpenAPI Specification is considered risky in terms of breaking API consumer implementations when the API version is updated. In “*Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando”*, a new type is introduced to deal with this called `x-extensible-enum`. In order to have maximum compatibility (with tools, middleware etc.), DCSA has chosen not to extend the OpenAPI Specification in this way, but simply to use Strings in most cases. These properties are identified in API documentation and often use the `Code` suffix in the

property name. API providers MAY implement these properties as enumerated types, but this is not recommended for API consumers. We refer to such properties as *“enum-like string” properties* in this document.

These properties are often similar to “code lists” as they are defined by other standardisation bodies.

Although this point may seem to make an obscure distinction, it is important with respect to backward compatibility, which in turn is important for interoperability.

Enum values MUST be declared using UPPER_SNAKE_CASE.

This convention also SHOULD be followed for “enum-like string” properties.

Null values

Null values MUST NOT be used in JSON responses. Instead of a null value omit the property.

Empty arrays MUST NOT be represented with null – but MUST be empty lists [].

Where this leads to ambiguous semantics, consider introducing an additional boolean property.

Null values MAY be used in JSON requests where JSON Merge Patch semantics are used.

Date and Time values

DCSA standards MAY represent date and time in three ways -

1. A Date property only contains a date.
2. A DateTime property contains both a date and time.
3. A Time property only contains a time.

Date, DateTime and Time properties MUST be serialised using the ISO 8601 format.

An API MUST NOT use a “special time” or “special date” such as 12:00 or 00:00 to indicate (reduced) accuracy or for any other “special” purpose. This would lead to ambiguity and risk of incorrect implementation by adopters.

Clarification and usage guidelines (not normative)

If a period of time needs to be represented then there are two recommended approaches -

1. A start property and an end property. Both properties MUST be of the same data type (Date, DateTime, Time).
2. A “centre” property (of type Date, DateTime or Time) and an indication of range, which MAY be used to indicate accuracy. How such an “range” indicator is defined MAY be different for a given API endpoint. Examples of range indicators -
   a. A boolean property indicating that the accuracy is to within the date only (e.g. isDateOnly), the time portion is then ignored.
   b. A property indicating a range of options for error margin e.g. (estimatedArrivalTimeRange) with possible values such as HOURS_12, HOURS_6, HOURS_1, MINUTES_10, MINUTES_1.
   c. A property indicating a specific error margin e.g. (estimatedArrivalTimeRange), which could indicate the accuracy in seconds. The unit is to be chosen as appropriate for each endpoint and MUST be either specified or indicated in another property.

Clarification and usage guidelines (not normative)

As an API provider or API consumer, be careful when working with datetimes.

Note the serialisation of date as YYYY-MM-DD to avoid ambiguity with DD-MM and MM-DD formats used in different countries.

Consider leap years, consider daylight saving time, consider time zones that do not have integer hour offsets from UTC, consider leap seconds, consider the international date line. This is an important area for interoperability.

Date values

Date values MUST be in `YYYY-MM-DD` format

Date properties MUST allow offset based timezones. Date MAY use `+Z` for Zulu time (UTC).

DateTime values

DateTime properties MUST allow offset based timezones. DateTime MAY use `+Z` for Zulu time (UTC).

A DateTime property MUST contain a specific date in `YYYY-MM-DD` format

A DateTime property contains a specific time of a day in 24-hour-format using the following format: hh:mm.

DateTime properties MAY include seconds in which case the following is added :ss.

DateTime properties MAY include milliseconds in which case the following is added: `.sss`

Time values

A Time property only contains a specific time of a day in 24-hour-format using the following format: `hh:mm`.

Time properties MAY include seconds in which case the following is added `:ss`.

Time properties MAY include milliseconds in which case the following is added: `.sss`

Clarification and usage guidelines (not normative)
Although it is mandatory to use a time zone in DateTime properties, it is not mandatory to use any *specific* time zone such as UTC. This is to allow for the convention in container shipping of expressing time in “local time” at the port to be used, which is easier for debugging and logging for most container shipping IT environments.

Example	**Explanation**
2020-07-13	13th of July 2020, timezone is ambiguous (which MAY be acceptable in some cases for Date but not for DateTime).
The following 5 datetimes represent the same moment in time in different Time Zones:	
2020-07-05T02:15-04:00	5th of July 2020, 02:15 in New York (EDT Eastern Daylight Time)
2020-07-05T08:15+02:00	5th of July 2020, 08.15 in Rotterdam, Netherlands (CEST Central European Summer Time). Note that the time zone changes when daylight saving time is active. Due to this a DateTime always represents an unambiguous point in time, even in a time zone where daylight saving time is used.
2020-07-05T11:45+05:30	5th of July 2020, 11.45 in New Delhi, India (IST India Standard Time)
2020-07-05T14:15+08:00	5th of July 2020, 14.15 Shanghai, China (CST China Standard Time)
2020-07-04T23:15-07:00	4th of July 2020, 23.15 San Francisco (PDT Pacific Daylight Time)

Table: Examples of time format

String values

Strings MUST NOT include leading or trailing whitespace, unless this is explicitly specified in the API description, regular expression or similar constraint definition.

Strings MUST NOT include any of the C0 control characters as defined by the Unicode Consortium → {x} none , unless this is explicitly specified in the API description, regular expression or similar constraint definition. Where String properties with length > 100 are used (e.g. text of contract clauses), then it is allowed to use Unicode code point `000A` in order to indicate a line feed. Unicode code point `000D` (carriage return) is not allowed in such cases unless this is explicitly specified in the API description, regular expression or similar constraint definition. Aligning on this is important for interoperability.

Within string values the full range of Unicode code points MUST be accepted by endpoints, except where this is explicitly limited in the API description, regular expression or similar constraint definition.

Regular expressions

OAS 3.0 states that a pattern SHOULD be a valid ECMA262 regular expression.

OAS 3.1 is based on JSON Schema where ECMA262 MUST be used to specify patterns. Pattern is more strictly formalised in OAS 3.1 by design.

In DCSA APIs regular expression patterns MUST follow ECMA262 syntax and semantics.

Where it is not possible to (fully) formulate the constraint as an ECMA262 regular expression, the following approach SHOULD be used:

1. Formulate an ECMA262 regular expression that matches a superset of the permitted strings.
2. In the property description state the full constraints as exactly as possible.

Property descriptions MAY include equivalent regular expression syntax in Java and .NET where these programming environments cannot support the specified ECMA262 syntax. Write in the comments that this is a “suggestion” . It MUST NOT be a normative part of the standard because it cannot be (easily) integrated into conformance tools or validation layers of solutions.

Care SHOULD be taken to avoid regular expressions that could be susceptible to the “ReDoS attack” → Regular expression Denial of Service - ReDoS | OWASP Foundation. API Providers SHOULD put measures in place to ensure the security of their REST APIs with respect to the ReDoS attack.

Where the regular expression is difficult for a human to interpret, due to modifications to prevent the ReDoS attack, the API specification MAY explain the regular expression in the comments noting that the explanation is “non-normative”.

Comments and similar “free text” properties

DCSA APIs MAY include properties for “comments” or similar “free text”. This MUST be used only to provide information for humans and not for machines. It is not permitted for such data to be parsed, because this will change the semantics of the API, which has negative consequences for interoperability. If there is something that a machine needs to be able to process, add it to the API as a suitable property. The extensions mechanism can be a good way to approach this.

Conformance Checklist : Property Values			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Enum values MUST be declared using UPPER_SNAKE_CASE.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Design	Null values MUST NOT be used in JSON responses. Instead of a null value omit the property.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Empty arrays MUST NOT be null.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	There are no “special values” in Date, DateTime or Time properties with different semantics.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Each property that is a Date, DateTime or Time MUST conform to the ISO 8601 format, with additional restrictions as follows - Date component serialised as YYYY-MM-DD. Time component serialised as hh:mm. Where seconds are used this MUST be serialised by concatenating with the format :ss. Where milliseconds are used this MUST be serialised by concatenating (after seconds) with the format .sss.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Time zones MUST NOT be optional for DateTime properties.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Comments and similar “free text” properties are only used for human-readable content. It MUST be explicitly stated in the API definition that such fields MUST NOT be used by API consumers or API providers in order to drive business logic.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where leading or trailing whitespace is allowed in a	☐ Conformant ☐ Not conformant	

	string value, this is explicitly stated in the API description or in a regular expression or similar constraint.	☐ Not applicable ☐ Unknown	
API Design	Where there is a limitation on the set of Unicode code points that can be used in a string value, this is explicitly stated in the API description or in a regular expression or similar constraint.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	All regular expressions are formally specified only using the ECMA262 standard.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Null values for properties MUST NOT be used in API responses.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	In responses, each property that is a Date, DateTime or Time MUST be serialised according to the ISO 8601 format, with additional restrictions as follows - Date component serialised as YYYY-MM-DD. Time component serialised as hh:mm. Where seconds are used this MUST be serialised by concatenating with the format :ss. Where milliseconds are used this MUST be serialised by concatenating (after seconds) with the format .sss.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	In responses, each property that is a DateTime MUST be serialised including an indication of time zone according to the ISO 8601 format.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Where a request including a DateTime is received without	☐ Conformant ☐ Not conformant	

	an indication of time zone, this MUST be rejected as a Bad Request.	☐ Not applicable ☐ Unknown	
API Provider	Values from comments and similar “free text” MUST NOT be used to drive business logic.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	Unless the API specifically allows it, String values must not be provided or accepted including the C0 control characters as defined by the Unicode Consortium. There is an exception where the property may have a length > 100. In those cases, it is allowed to use the Unicode code point `000A` in order to indicate a line feed.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Null values for properties MUST NOT be used in API requests, except in the case of JSON Patch or JSON Merge Patch semantics.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	In requests, each property that is a Date, DateTime or Time MUST be serialised according to the ISO 8601 format, with additional restrictions as follows - Date component serialised as YYYY-MM-DD. Time component serialised as hh:mm. Where seconds are used this MUST be serialised by concatenating with the format :ss. Where milliseconds are used this MUST be serialised by concatenating (after seconds) with the format .sss.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Consumer	In requests, each property that is a DateTime MUST be serialised including an indication of time zone according to the ISO 8601 format.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When processing responses, any valid time zone according to ISO 8601 MUST be accepted and correctly mapped to the system of record.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Values from comments and similar “free text” MUST NOT be used to drive business logic.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Unless the API specifically allows it, String values must not be provided or accepted including the C0 control characters as defined by the Unicode Consortium. There is an exception where the property may have a length > 100. In those cases, it is allowed to use the Unicode code point 000A in order to indicate a line feed.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Property Values Provided by External Parties

Values used in certain properties of DCSA standards MAY be specified by external parties. These are typically other standards bodies such as UN/CEFACT, BIC, ISO, IMO, SMDG etc. These bodies often refer to such values as “code lists”.

In such cases, at API design time a decision is taken per property to adhere to one of the following three policies -

1. DCSA allows the valid set of values to be determined at runtime. Any values that are specified by the external party at the moment the API call is made are considered to be valid.
2. DCSA aligns with a certain version of the set of values. In this case the version (or publication date) is specified in the DCSA standard.
3. DCSA identifies a subset of the values for standardisation, which is then specified as part of the DCSA standard. This could be directly in the description in the OpenAPI Specification or in other documentation e.g. in a reference file available for download. In this case the situation is equivalent to the case of “enums and enum-like string properties” as described above.

Policy option 1 from the list above is the default. The selected policy MUST be explicitly stated in DCSA API specifications.

Where policy option 1 is used, API providers and API consumers SHOULD regularly review the set of values that they are able to process. For policy option 1, the set of values is considered be dynamic over the life cycle of a single DCSA API <MINOR> version.

For the same set of external values, this policy MUST be consistent for all DCSA standards.

Care MUST be taken when publishing APIs to not infringe intellectual property that other standards bodies may own.

Conformance Checklist : Property Values Provided by External Parties			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Where a property is based on a set of values provided by an external party, the policy for alignment MUST be specified in the API documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	When policies 2. or 3. are used, any values that are not specified MUST NOT be generated or accepted.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When policies 2. or 3. are used, any values that are not specified MUST NOT be generated or accepted.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Items in collections

Collection items MUST be uniquely identified by a single, mandatory property.

Conformance Checklist : Items in Collections			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Collection items MUST be uniquely identified by a single, mandatory property.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Unicode

Representation of all text in all DCSA API standards MUST be based on Unicode. For conformance with this version of API Design Principles, applications MUST use version 15.0.0 of Unicode / ISO/IEC 10646:2020.

The specification of the lengths of text properties MUST be in units of Unicode characters, irrespective of where those characters lie in the Unicode code point range.

A standard MAY restrict the range of allowed values in a text property. Such restrictions MUST NOT use a different encoding of characters.

Clarification and usage guidelines (not normative)
Container shipping is an international business so the technical concept of “internationalisation” is critical for interoperability. Fully supporting Unicode is one example of this. The original Unicode standard was a 16-bit representation of characters, but this was extended in 1996 (see https://unicode.org/faq/utf_bom.html). DCSA standards are explicitly based on current Unicode standards. 16-bit character width is insufficient to support that. When implementing it is important to be aware of Unicode considerations in order to achieve correct interoperability. Many application and development platforms require careful configuration to achieve full Unicode support. See also above section on UTF-8.

Conformance Checklist : Unicode			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The specification of the length of text properties MUST be in units of Unicode characters. It specifically MUST NOT be in units of bytes.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If a text property does not allow for all Unicode code points to be used, then this MUST be explicitly documented. A regular expression, a list of allowed values or a reference to a list provided by another party is a possible way to specify such a restriction.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	If there is no explicit restriction, then text properties MUST allow for all Unicode code points defined in version 15.0.0 of Unicode to be used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All text properties that are exchanged MUST be persisted such that all valid Unicode code points for each property can be stored and retrieved.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	All text properties that are exchanged MUST be persisted such that all valid Unicode code points for each property can be stored and retrieved.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Error handling

API providers need to determine how to respond when a request is not conformant or when a request cannot be accepted due to business rules.

The API specification MUST define all JSON documents that are acceptable in requests. If the semantics of the API are to allow a partially complete or partially invalid *business* request to be made so that the API provider can evaluate it or to allow the API provider to propose a fully complete or fully valid *business* request, then the API specification MUST make this clear and MUST define all *technically* valid JSON documents that are to be processed.

Clarification and usage guidelines (not normative)

Suppose that an API provides a service whereby an API consumer can make a business request described with a complex object.

The semantics of the API are such that the consumer can make an “incomplete” or “illogical” request in terms of business constraints, so that the API provider can then propose the best possible “complete” and “logical” request, or to facilitate the “negotiation” of this request.

In that case, the API specification defines *any* possible request the API consumer can make, even though this request could be “illogical” in terms of business constraints. The response to such a request is never an error in these cases.

If a request is not conformant, API providers MUST NOT process it and MUST respond with an appropriate HTTP 4XX or 5XX error status code.

If an API provider responds with a 4XX or 5XX HTTP response code, this MUST NOT cause any state change to business objects held on the API provider (other than in logging subsystems or similar secondary functions).

These are the only requirements for conformant behaviour and are simply the expected behaviour of HTTP-based application servers.

It is noted that this will not allow an API consumer to distinguish between technical errors and functional errors. In order to make this possible, DCSA has standardised a JSON error document that MAY also be provided. This document includes an error code that can be used to determine the classification of the error.

An API specification MUST NOT allow a 2XX HTTP response to be combined with a JSON error document. The JSON error document is only to be returned in responses in combination with HTTP status codes corresponding to errors.

Clarification and usage guidelines (not normative)

By using the standard JSON error document, useful additional information can be returned to the API consumer. This information can be used to drive a sequence of interactions whereby alternatives are found - for example, perhaps a series of BIC codes are attempted until a BIC code is supplied with a correct checksum. This could be based on automated interactions or by providing feedback to users. Using the jsonPath property in the error document, it is possible to exactly identify the properties in the request that were the root cause of the error. The jsonPath property corresponds to the definition of JSONPath which can be found here → {x} JSONPath - XPath for JSON

API providers do not need to provide these JSON error documents in order to be conformant.

API consumers do not need to be able to parse the JSON error document in order to be conformant.

The error document is provided to facilitate an optional, higher level of automation and to assist in debugging consistently across API providers. A decision was taken to not make this mandatory for various reasons -

- Some errors will occur in infrastructure tiers / layers that are not concerned with DCSA standardisation (e.g. security gateways). It is not reasonable to expect these components to implement DCSA-specific features.
- There can be conflicting concerns on error handling - for example an implementation by a solution provider that also has non-DCSA endpoints and wishes to have consistent error handling across all endpoints.
- Some security experts are of the opinion that exposing details about errors may be useful to attackers and should be avoided. API providers are able to make their own choice in this matter.

Below is an example HTTP response that includes the standard HTTP response headers and a DCSA standard JSON error document.

Note that there are two codes - the standard HTTP status code (in this case 400) and the DCSA error code (in this case 7999).

```
HTTP/1.1 400 Bad Request
Server: nginx/0.8.54
Date: Mon, 02 Jan 2024 02:33:17 GMT
Content-Type: application/json
Content-Length: 524
Connection: keep-alive

{
  "httpMethod": "POST",
  "requestUri": "https://dcsa.org/dcsa/tnt/v1/events",
  "statusCode": 400,
  "statusCodeText": "Bad Request",
  "statusCodeMessage": "The supplied data could not be accepted",
  "providerCorrelationReference": "4426d965-0dd8-4005-8c63-dc68b01c4962",
  "errorDateTime": "2019-11-12T07:41:00+08:30",
  "errors": [
    {
      "errorCode": 7999,
      "property": "facilityCode",
      "value": "SG SIN WHS",
      "jsonPath": "$.location.facilityCode"

      "errorCodeText": "invalidQuery",
      "errorCodeMessage": "Spaces not allowed in facility code"

    }

  ]
}
```

Note that this example does not define the error associated with code 7999. It is included here only for illustration.

Standard HTTP status codes MUST be used. The table below extends the standard HTTP status codes with extra information. The extended status codes SHOULD be used together with standard DCSA error codes for easier debugging during development and maintenance. This is not an exhaustive list; other status codes MAY be used in implementations.

HTTP Status Code	Status Code Text	Example information for Status Code Message
`400 Bad Request`	`invalidParameter`	An invalid parameter or parameter value was supplied in the request.
`400 Bad Request`	`missingParameter`	The API request is missing a required parameter.
`400 Bad Request`	`invalidQuery`	The request query was invalid. Check the documentation to ensure that the supplied parameters are supported, and check if the request contains an invalid combination of parameters, or an invalid value.

`400 Bad Request`	`invalidUri`	The requested URI is not represented on the server.
`401 Unauthorized`	`missingCredentials`	The user is not authorised to make the request.
`401 Unauthorized`	`invalidCredentials`	The supplied authorisation credentials for the request are invalid.
`401 Unauthorized`	`expiredAccessToken`	The supplied Access Token has expired.
`403 Forbidden`	`accessDenied`	The requested operation is forbidden.
`403 Forbidden`	`insufficientPermissions`	The authenticated user is not permitted to execute this request.
`404 Not found`	`notFound`	The requested resource could not be found. 404 Not found MAY be used to mask sensitive information for both 400, 401 and 403.
`405 Method not allowed`	`httpMethodNotAllowed`	The HTTP method for the request is not supported.
`409 Conflict`	`resourceAlreadyExists`	The resource already exists. Used when a “Unique constraint violation” occurs.
`415 Unsupported Media Type`	`unsupportedMediaType`	A request was performed with a content type that is not supported. For example an XML payload was sent to the API provider but only a JSON payload is supported.
`429 Too many requests`	`rateLimitExceeded`	Too many requests have been sent recently. A Retry-After header SHOULD be added to notify consumer when to try again.
`500 Internal server error`	`internalError`	The request failed due to an internal error.
`503 Service Unavailable`		Used for maintenance purposes A Retry-After header SHOULD be added to notify consumer when to try again.

Table: Error handling

JSON Error Document

The standard JSON error document is described by the following JSON schema, according to JSON Schema 2020-12:

```
{
  "$id": "https://dcsa.org/api-design-principles/2.0/error-message-schema",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
```

```
  "title": "Error Message",
  "required": [
    "httpMethod",
    "requestUri",
    "statusCode",
    "statusCodeText",
    "errorDateTime",
    "errors"
  ],
  "type": "object",
  "properties": {
    "httpMethod": {
      "type": "string",
      "description": "The HTTP method used to make the request."
    },
    "requestUri": {
      "type": "string",
      "description": "The URI that was requested."
    },
    "statusCode": {
      "description": "The HTTP status code returned.",
      "type": "integer",
      "minimum": 400
    },
    "statusCodeText": {
      "type": "string",
      "description": "A standard short description corresponding to the HTTP status code."
    },
    "statusCodeMessage": {
      "type": "string",
      "description": "A long description corresponding to the HTTP status code with additional information."
    },
    "providerCorrelationReference": {
      "type": "string",
      "description": "A unique identifier to the HTTP request within the scope of the API provider."
    },
    "errorDateTime": {
      "type": "string",
      "description": "The DateTime corresponding to the error occurring. This MUST be formatted according to
    },
    "errors": {
      "type": "array",
      "description": "An array of errors providing more detail about the root cause.",
      "items": {
        "$ref": "#/$defs/errors"
      }
    }
  },
  "$defs": {
    "errors": {
      "type": "object",
      "required": [
        "errorCodeText",
        "errorCodeMessage"
      ],
      "properties": {
        "errorCode": {
```

```
          "description": "The detailed error code returned.",
          "type": "integer",
          "minimum": 7000
        },
        "property": {
          "description": "The name of the property causing the error.",
          "type": "string"
        },
        "value": {
          "description": "The value of the property causing the error serialised as a string exactly as in th
          "type": "string"
        },
        "jsonPath": {
          "description": "A path to the property causing the error, formatted according to JSONPath (https://
          "type": "string"
        },
        "errorCodeText": {
          "description": "A standard short description corresponding to the error code.",
          "type": "string"
        },
        "errorCodeMessage": {
          "description": "A long description corresponding to the error code with additional information.",
          "type": "string"
        }
      }
    }
  }
}
```

Three classes of error codes have been defined, with corresponding ranges of values.

- 7000-7999: Technical error codes
- 8000-8999: Functional error codes
- 9000-9999: API provider-specific error codes

Standardised error codes are listed in a separate document, that DCSA makes available.

Conformance Checklist : Error Handling			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The API MUST be designed such that an error response is only made where there is no change in state on the API provider.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The API specification MUST recommend the use of the DCSA standard JSON error document.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Design	Wherever possible, the API specification MUST map error cases to both HTTP status codes (4XX, 5XX) and DCSA standard error codes (7XXX, 8XXX, 9XXX).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	The API specification MUST NOT combine the DCSA standard JSON error document with HTTP status codes other than 4XX and 5XX.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	If a 4XX or 5XX status code is used in a response then there MUST NOT be any change in state of business objects.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	If the jsonPath property is included in the standard JSON error object, this MUST be conformant with JSONPath as defined here → {x} JSONPath - XPath for JSON .	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

3. Stable

APIs need to be stable over time, which means the occurrence and impact of major changes are minimised.

The requirements to achieve stability are described in this chapter.

Versioning

Version information

Semantic Versioning 2.0 MUST be used to specify version information.

For an API standard this is restricted to the format <MAJOR>.<MINOR>.<PATCH>

- MAJOR to be increased when backward compatibility is broken.
- MINOR to be increased when new functionality (e.g. new operations, new optional properties) is added in a backward-compatible manner
- PATCH to be increased when backward-compatible bug fixes are made. PATCH does not include new functionality.

Pre-releases (rule 9 from Sematic Versioning) MAY be used in API version information when an API is provided that corresponds to a pre-release of the standard itself, for example during a beta phase of a standard release. It is not acceptable to use pre-release to indicate a software release that has a goal to implement a final release of a standard. The versioning information is about the version of the standard, not about the version of the application. Other headers MAY be used to indicate application version information.

Build metadata (rule 10 from Semantic Versioning) MUST NOT be used.

URI versioning MUST be used by API providers. Only <MAJOR> version is included – example: `/v2/events`. The first version of any standard MUST include `/v1/`.

An API-Version custom header MAY be added to the request; if added it MUST only contain <MAJOR> version. The API-Version header MUST be aligned with the URI version.

A custom header called API-Version MUST be added in the response to specify the full version. The API-Version custom header in the response MUST include complete version information for example: API-Version: 2.0.0-beta-3. An API Consumer MAY use this information to adapt processing of the response in order to make use of features introduced into later versions of the standard.

Versioning callbacks and webhooks within a single API Specification

In some DCSA standards, use is made of callbacks or webhooks according to the terminology in OpenAPI Specification 3.1. Equivalents MAY also be defined using OAS 3.0.3, where callbacks are present but webhooks are not.

In these cases, the API consumer is providing an endpoint to receive HTTP requests, but it is still considered to be an API consumer in the context of this document.

This is important with respect to versioning and version headers. API providers MAY upgrade to a new <MINOR> version without informing API consumers, but in their communication to API consumers they MUST include an identifier of the <MAJOR> version and the <MINOR> version in the header. This continues to be true when callbacks or webhooks are used, even though the roles are reversed from an HTTP protocol perspective.

Therefore, when an API provider makes an HTTP request to an API consumer in order to invoke a callback or webhook endpoint, the API-Version header MUST indicate the <MAJOR>.<MINOR>.<PATCH> version. This allows the API consumer to adapt their behaviour depending upon the exact version. In particular, it allows API consumers to be able to determine the difference between an optional property not being present because it is “null” and a prior version where the property was simply not yet defined.

When an API consumer responds to a callback or webhook invocation, the API-Version header MAY indicate only the <MAJOR> version of the consumer implementation for symmetry and logging; it is not mandatory because it can be difficult to implement and does not add significant value.

Callback and webhook endpoints MAY use URI versioning. This is not mandatory.

Versioning other tightly coupled standards

When two or more standard APIs have tight coupling but are not defined in a single API specification (e.g. a primary API and a *separate* callback API), their version numbers SHOULD NOT be aligned. It is better to not align the version numbers because the different APIs will be often implemented by different departments or different parties. Each needs to know whether their API is subject to change, and whether it is backwards compatible. It is often an intentional part of the standard design to allow one party’s API to be more stable than another’s, in order to make the ecosystem as a whole more stable. This SHOULD be taken into account during the process of API design.

There can be cases where one or more parties are still considered to be API consumers in terms of other requirements for conformance (e.g. implement full backwards-compatibility etc.). This MUST be clearly stated by DCSA in API documentation. This will typically be the case where small APIs are used as webhooks or callbacks.

DCSA MUST maintain compatibility documentation between APIs that are tightly coupled. This compatibility documentation SHOULD include details of particular use cases, particularly where version compatibility differs for specific use cases.

Callback and webhook endpoints MAY use URI versioning. This is not mandatory.

Conformance Checklist : Versioning	
Defined by the API Design Principles Document	**To be completed for a particular standard / implementation**

Conformance Role	Conformance Criteria	Conformance Status	Justification for Conformance Status
API Design	Semantic Versioning 2.0 MUST be followed in identifiers of API versions.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where two or more APIs are tightly coupled, documentation MUST be provided regarding version compatibility.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	Where two or more APIs are tightly coupled, documentation MUST be provided regarding the role of each party in terms of having responsibilities are an API provider or API consumer (or both).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	If an API-Version custom header is received on a request but this is not consistent with URI versioning, this MUST be rejected with a 400 Bad Request response.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	An API-Version custom header MUST be included in responses and this MUST indicate the <MAJOR>.<MINOR>.<PATCH> version of the API standard.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	The <MAJOR> version in the API-Version custom header included in responses MUST be aligned with the URI version.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	When making an HTTP request to a webhook or callback endpoint offered by an API consumer, the API-Version custom header MUST be present and this header MUST indicate the <MAJOR>. <MINOR.<PATCH> version of the API standard in used.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	If an API-Version custom header is added to a request, this MUST NOT include the <MINOR> version of the API	☐ Conformant ☐ Not conformant ☐ Not applicable	

	standard. It MUST include only the <MAJOR> version of the standard.	☐ Unknown	
API Consumer	When receiving an HTTP request on a webhook or callback endpoint, the API-Version custom header MUST be verified for compatibility.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When responding to an HTTP request on a webhook or callback endpoint, the API-Version custom header MUST be included and MUST indicate the <MAJOR> version supported only.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
See also section on URIs regarding the principle of URI-based versioning.			

Deprecation

Deprecated endpoints MUST be documented in OpenAPI specification using the ”Deprecated” property introduced in OpenAPI 3.0. They MUST only be removed when a new <MAJOR> version is initiated.

Deprecated endpoints SHOULD add Deprecation and Sunset headers to responses. See The Deprecation Header Field and The Sunset http Response Header Field.

Communication (e.g. e-mail notification) SHOULD be sent to consumers of deprecated endpoints where possible.

Conformance Checklist : Deprecation			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Deprecated endpoints MUST be documented in OpenAPI specification using the ”Deprecated” property introduced in OpenAPI 3.0.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Deployment of Up-to-date Versions of APIs

API providers MUST NOT remain more than one <MAJOR> version behind the latest version of the standard with their production deployment. This is to ensure that the interoperable, standards-based ecosystem remains healthy.

API providers SHOULD make best effort to implement and deploy the latest <MAJOR>.<MINOR> version of any standard that they support.

No more than three <MAJOR> versions of the same standard API SHOULD be deployed in parallel by the same API provider.

Clarification and usage guidelines (not normative)

Suppose that a given standard has version 6.x as the current latest standard.

It is acceptable for an API provider to offer version 5.x or version 6.x as their latest version, but it is not acceptable for an API Provider to offer version 4.x as their *latest* version (more than one major version behind).

The API provider may offer two other versions. In the example case, version 1.x, version 3.x and version 6.x could be offered - but no other versions. This is to allow the API Provider to offer longer term support where necessary, but still to encourage an upgrade cycle towards latest major versions.

Conformance Checklist : Deployment of Parallel Versions			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Provider	API providers MUST NOT remain more than one <MAJOR> version behind the latest version of the standard with their production deployment.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Backward compatibility

Goals

The goal of backward compatibility could be summarised as follows - "*Every unmodified, conformant API consumer application continues to work correctly when a backward compatible change is introduced*."

Incrementing the <MAJOR> version SHOULD be avoided where possible.

Clarification and usage guidelines (not normative)

The following strategies could be used to avoid incrementing the major version -

- Create a new resource (a variant of the old) in addition to the old.
- Create a new service endpoint (duplicate and deprecate in API documentation)

Clarification and usage guidelines (not normative)

When an API consumer decides to integrate with an API provider they are aware of the standard version supported by that API provider. Suppose that an API provider offers version 7.3 of a particular API; an API consumer could start integration based on an API consumer implementation that is conformant with versions 7.0, 7.1, 7.2 or 7.3.

An API provider MAY, at any time and without notice to API consumers, upgrade an API to a higher <MINOR> version, backward compatible API. An API provider SHOULD give notice of an upgrade to API consumers, but it is not mandatory. If an API provider currently offers version 7.3 of a standard they could, for example, upgrade to version 7.8. They MUST NOT downgrade, once API consumers are actually using new features.

An API consumer MAY use version information in responses in order to optimise processing of responses from different API providers supporting different versions of the same standard. For example, suppose an API consumer connects to two API providers, one offering version 7.0 and another offering version 7.8 of the same standard. The API consumer MAY use the version information in the response API-Version header to handle the providers differently.

An API consumer MUST NOT send an API request that includes features from a higher version to the version provided by the API provider. For example, it is not conformant to send a request defined using version 7.8 to an API provider offering version 7.1.

Backward compatibility rules

API consumers SHOULD be robust:

- Be conservative with API requests and data passed as input
- Be tolerant with unknown properties and unknown enum and “enum-like string” values in the payload, but do not eliminate them from payload if needed for subsequent PUT or PATCH requests

Backward compatibility MUST NOT be broken within a <MAJOR> release.

When adding new features, the following rules MUST be followed in order to adhere to backward compatibility:

- Add only optional and not mandatory properties
- Never change the semantics of a property
- Never change the syntax of a property in a way that could break an implementation. In particular - string lengths MAY be reduced but not increased for responses; string lengths MAY be increased but not reduced for requests.
- Enum and “enum-like string” range can be reduced (API provider MUST be able to handle values from all previous <MINOR> versions of the standard that they receive)
- Enum and “enum-like string” range can be expanded when used for properties received by an API provider
- “Enum-like string” range can be expanded when used for properties sent by an API provider (this is not allowed for Enum).
- Never change the technical semantics of a method (e.g. different idempotent behaviour).
- Never change the business semantics of a method.

Clarification and usage guidelines (not normative)

The goal is that API consumers can always continue to use business services offered by API providers whenever API providers upgrade to a new <MINOR> version in a <MAJOR> version series. This does not have to be a consecutive <MINOR> version.

If a value is removed from an enum or enum-like string, then the API consumer is still allowed to send values from *any* previous <MINOR> version (in the same <MAJOR> version series) to the API provider and the API provider MUST still handle this value. This also shows the risk and subtle complexity of removing values - a request MUST be accepted even though it does not validate against the “new“ API specification. So, although removing values is allowed, it’s an area where API designers and API providers need to proceed with caution.

Remember that being an API consumer or an API provider is a business role. API consumers MAY offer callback or webhook endpoints to API providers. Therefore, although it is an obscure case, these “removed” values could be part of responses from requests to webhook or callback endpoints.

Change	Example	Major / Minor Release
Add an optional property	Port Call Phase Type Code was added in JIT 1.1	Minor
Add a mandatory property		Major
Change the semantics of a data property	Reefer temperature: if a current version used Fahrenheit and in the next version used Celsius (without some other indication of the units).	Major

Reduce set of valid values for properties in responses from API providers.	Range: Received, Pending Update, Pending Approval, Approved, ~~Confirmed~~, Rejected – Confirmed would be replaced by Approved (that was already there) providing that semantics for both Approved and Confirmed were identical for API consumers in the previous version (otherwise an API consumer using the previous version may not have correct behaviour when processing “Approved”).	Minor
Reduce set of valid values for properties in requests to API providers, only if the API provider can still process the removed values (to support API consumers still on the previous version) and that semantics are not changed.	Range: Received, Pending Update, Pending Approval, Approved, ~~Confirmed~~, Rejected Confirmed would be replaced by Approved (that was already there)	Minor
Reduce set of valid values for properties in requests to API providers in all other cases	Remove Fahrenheit as a temperature unit from the API.	Major
Expand set of valid values for input and/or output properties in either requests or responses., providing that the “enum-like string” approach is used.	Add Kelvin as a temperature unit to the API.	Minor

Table: Backward compatibility examples

Conformance Checklist : Backward compatibility			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	The <MAJOR> version number of the API standard MUST be incremented whenever there is deviation from any of the backward compatibility rules, described in detail above and summarised as “every unmodified, conformant API consumer application will continue to work correctly when a backward compatible change is introduced.”	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All API endpoints that receive data from an API consumer MUST accept and correctly process all values for enums or “enum-like strings” that have been defined over the entire	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

	<MAJOR> version life cycle for the property in question, where such properties are present.		
API Consumer	When processing responses, an API consumer MUST be able to handle properties that were not defined at the time of implementation. Best effort MUST be made to process objects including such properties. The typical behaviour is to persist them only if they will be required in future PUT or PATCH requests, and to ignore them in business logic.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When generating PUT or PATCH requests, the API consumer MUST include all properties from previous GET requests where this is required by the specified PUT or PATCH semantics (as specified at the time of implementation).	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	When sending to an API provider, features MUST NOT be used that are specified by a higher <MINOR> version than the version supported by the API provider. The API consumer will have knowledge of the version supported by the API consumer via a channel such as a developer portal.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
See also section below on Enums and "Enum-like strings"			

Backward compatibility of enum and "enum-like string" properties

Enum properties MAY be included in API definitions under limited circumstances. This is due to their impact on backward compatibility.

Adding an allowed value to an enum risks breaking API Consumer applications. Within the context of DCSA standardisation, adding an allowed value to an enum is a breaking change that requires a new <MAJOR> version.

In order to avoid this problem, generally in DCSA APIs, "enum-like strings" SHOULD be used instead of enums.

An "enum-like string" property is simply a property that formally has the type string and that has a limited set of valid values defined in the API description and other supporting material. From a syntax perspective it behaves like a string, from a semantic perspective it behaves like an enum.

Clarification and usage guidelines (not normative)

The problem with using enum in API specifications is that automated code generators and human programmers are likely to build API consumer components using enumerated types (in whatever programming languages they use). Such components will immediately fail whenever a new valid value is added. This is not acceptable behaviour in a standards-based ecosystem where API providers are free to perform upgrades to a new <MINOR> version of a standard at any time without coordinating with API consumers.

DCSA API Design principles are strongly influenced by *"Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando"*. As also noted earlier, these guidelines resolve this "enum backward compatibility quandary" by introducing a new type called `x-extensible-enum`. This is less acceptable within the context of an API standardisation effort. The DCSA resolution of this question is to recommend "enum-like strings". This is, to all intents and purposes, equivalent to the Zalando guidelines. It is less explicit, but more compatible with tools, middleware etc. Importantly, this approach is compatible with OpenAPI Specification and JSON Schema, which makes formal conformance easier to define and test. This is an important consideration when creating API standards.

Clarification and usage guidelines (not normative)

Where there is any doubt during API design, do not select an enum and use an "emum-like string" instead.

Use documentation to specify the allowed set of values and their meanings. The set of allowed values - and their meanings - is considered to be part of the definition of the API semantics.

API semantics cannot change between <MINOR> versions, so care is needed in the design of the set of allowed values. In particular, it is not allowed to add a value that has an intersection with an existing value with respect to the set of objects that the property can represent. This is because it will prevent correct processing of an object that previously could be processed correctly by the same API consumer.

As an example -

Suppose there is an property (an "enum-like string" property as noted above) that classifies states of matter as follows -

`SOLID, LIQUID`

It would be acceptable to expand the set of allowed values as follows without breaking backward compatibility -

`SOLID, LIQUID, GAS` (where no objects that are `SOLID` or `LIQUID` can also be a `GAS`)

because this new value does not break for any existing objects.

However, it would not be acceptable to expand the allowed values as follows -

`SOLID, LIQUID, CRYSTALLINE` (where some objects that are `SOLID` can also be `CRYSTALLINE`)

because an unchanged API consumer that can process the value "`SOLID`" correctly will not be able to process the value "`CRYSTALLINE`", even though it supports an interface that could process the object in question, when it is considered as `SOLID`.

Where these cases occur, they MUST be represented by a new optional property to retain backward compatibility. In this example, this might be a new a new property called `nonClassicalState` or it could be a Boolean `isCrystalline` property or a new array could be added (alongside the existing property) that can represent multiple states so that an object can be both `SOLID` and `CRYSTALLINE`.

As the above example demonstrates, the sets of allowed values for enums and "enum-like string" properties SHOULD not have intersections so that semantics are well-defined.

There are three circumstances where enums MAY be used:

1. **Inherent stability of the concept in the domain model**

In some cases, it is not considered possible that other values could exist, either now or in the future due to some inherent constraint of the domain.

A good example might be "WEST" and "EAST" as possible longitude classifiers or “PORT” and “STARBOARD” to indicate the two sides of a vessel.
A possible example might be "CELCIUS" and "FAHRENHEIT" as temperature units. Although even this example shows the danger of enums because other units (e.g. Kelvin) could be used.

2. **A breaking change that requires a new <MAJOR> version is acceptable if values are added to the enum**

In some cases, the set of allowed values in an enum property is fundamental to how the API standard works. If this is so fundamental that it is acceptable to update the <MAJOR> version number of the standard when a value is added, then an enum MAY be used.

3. **Due to backward compatibility concerns where there is an enum in a previous <MINOR> version.**

Enums MAY remain in place in order to support backward compatibility with standards based on previous versions of this document.

The set of valid values MUST NOT be expanded without incrementing the <MAJOR> version.

Conformance Checklist : Enums and “Enum-like strings”			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	Where a property is defined as an enum, at least one of the following three criteria MUST be met - 1. The property values are inherently stable for the long term in the domain. 2. It is acceptable that adding a value will cause a new <MAJOR> version to be required. 3. The property was already present in a previous <MINOR> version in the existing <MAJOR> version sequence, that already used enum and this previous <MINOR> version was based on the DCSA API Design Principles 1.X.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	For “enum-like string” properties received in API responses, the application MUST expect and be able to process values in API responses that were not defined at the time of	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

	implementation. Best effort MUST be made to process these values where possible and reasonable to do so.		
API Consumer	For “enum-like string” properties received in API requests to webhook or callback endpoints, the application MUST expect and be able to process values in API responses that were not defined at the time of implementation. Best effort MUST be made to process these values where possible and reasonable to do so and the presence of a new value MUST NOT be a reason to respond with a 4XX or 5XX status code.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	For “enum-like string” properties sent in API requests using PUT or PATCH methods, the application MUST include values that were previously received in responses to API GET requests, including when these values were not defined at the time of implementation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

API Extensions

Standard DCSA APIs are intended to provide a core feature set for a given business process defined in the DCSA Industry Blueprint.

This core feature set is common to a body of API providers and API consumers. This is the underlying motivation for standardisation.

Some API providers and API consumers will have requirements that are not met by DCSA standards. For this reason, DCSA provides a mechanism for API extensions.

Approach

API Providers MAY offer APIs with features that are not included in DCSA standards, while retaining alignment with DCSA.

Possible approaches are listed in the table below.

Extension Approach	Description	Motivation
Governed Standards-conformant API Extension	An API is provided that MUST provide all features of an existing DCSA standard API and some additional features. The additional features are candidates for future standardisation. These features are	Support the entire business process as defined in the DCSA Industry Blueprint, while offering additional features of some kind.

	registered and governed through DCSA processes.	The additional features are generic to the container shipping industry. The extensions may be offered by other parties in the future.
Ungoverned Standards-conformant API Extension	An API is provided that MUST provide all features of an existing DCSA standard API and some additional features. The additional features are not candidates for future standardisation.	Support the entire business process as defined in the DCSA Industry Blueprint, while offering additional features of some kind. The additional features are not generic to the container shipping industry. They are specific to the API provider and/or they are specific to one or more API consumers.
Ungoverned Non-standard API	A new API is provided that is not specified by an existing DCSA API standard. This category includes variants of DCSA APIs that do not provide all mandatory features according to the standard and therefore are not conformant APIs.	This case covers any other API that a party within the DCSA ecosystem may offer. It is explicitly not the goal of DCSA to standardise every API that organisations within the container shipping industry may provide.

Visible and Explicit

The goal of standardisation is to allow API consumers to reuse their applications and software components with a range of conformant API providers.

When APIs providers offer extensions, they provide features that are not described by the relevant standard. Therefore, an application acting as an API consumer that relies on the extension cannot be reused with other API providers.

API consumers and API providers need to clearly understand which features of APIs are part of standards, and which are not, in order to achieve *repeatable interoperability*.

Therefore, API providers need to make clear whether an API endpoint includes extensions or not. This MUST be clear both in documentation, for example in a developer portal, and at protocol level.

Conformant Extensions

DCSA standardisation is only concerned with the question of conformant extensions to APIs. Other APIs may exist that are similar to, or loosely based upon, DCSA standards. Such APIs are not the subject of DCSA API standardisation.

The general principle of conformant extensions can be summarised by the phrase "*DCSA in means DCSA out*". What is meant here is that if an API consumer makes a DCSA conformant request with no explicit request for an extension, then they only receive a fully conformant response with no extensions. Only if an API consumer explicitly requests an extension, MAY the additional features be provided.

A conformant extension to an API endpoint MAY include additional properties or ranges of values in JSON objects that are not specified in the corresponding standard.

A conformant extension to an API endpoint MAY omit properties that are mandatory according to the standard.

A conformant extension to an API endpoint MAY change the semantics of the API.

A conformant extension to an API endpoint MAY change constraints.

In general, a conformant extension MAY incorporate changes as required in order to support the relevant use case. It is noted however, that excessive divergence may be poor API design, where it would be preferred to add a new proprietary API instead.

Every feature of the extension MUST be made clear to API consumers in documentation, for example in a developer portal. All such extension features MUST only be accessible by API consumer applications if explicitly requested by including relevant HTTP headers (see below).

Non-Conformant APIs Based on DCSA APIs

It is possible for an API provider to offer an API that behaves exactly as described in the above section on “Conformant Extensions“ but that does not make additional features conditionally available according to explicit requests by API consumers. Such APIs can be deployed as useful initiatives towards an improved standard.

Such APIs can be useful when deployed alongside standards-conformant APIs. API Providers MAY use such an approach alongside a DCSA-conformant environment. This will never break the conformance requirements for backward compatibility (number of versions in production in parallel) because there are no restrictions on making non-conformant APIs available. However, it is important to note that an API Provider MUST NOT make a claim that such APIs are DCSA-conformant and MUST NOT associate them with a DCSA standard version number either in documentation (e.g. within a developer portal), URLs or in the protocol.

Such APIs are considered to be “Ungoverned Non-standard API Extensions” and cannot be conformant. This is because they do not offer predictable behaviour towards API consumers. In particular, API consumers MAY be reliant on the API-Version HTTP header in responses, which includes the <MINOR> version. An API consumer MAY reject a response that does not validate according to the published standard API specification corresponding to the version in the API-Version HTTP header.

Versioning of Extensions

When an API is conformant and also includes extensions, then it is subject to two version life cycles:

- The standardisation version life cycle
- The extension version life cycle

These version life cycles are independent. Both version life cycles MUST be made clear to API consumers both in documentation, for example in a developer portal, and at protocol level.

Clarification and usage guidelines (not normative)
To explain the two version life cycles in more detail with an example - Consider a particular DCSA API standard with a current release version of 5.2. An API provider has implemented this standard and offers a conformant API. This implies that an API consumer that has implemented version 5.1 of the same standard is interoperable with this API provider. The API provider adds an extension, which is considered to have a version number of 1.0. In some later release, the API provider changes the extension so that it is no longer backward compatible. The extension is then assigned a version number 2.0, informing API consumers via semantic versioning that applications that make use of the extension 1.0 will no longer work with this new release. However, this exact same release *remains* conformant to the DCSA standard at version 5.2. So, an API consumer that only uses standard features will continue to work unmodified.

Implementation of Extensions

Making extensions explicit and dealing with the two version life cycles is implemented through the use of HTTP headers for versioning.

The DCSA standards include HTTP headers to identify versions, both in requests and responses - the API-Version header.

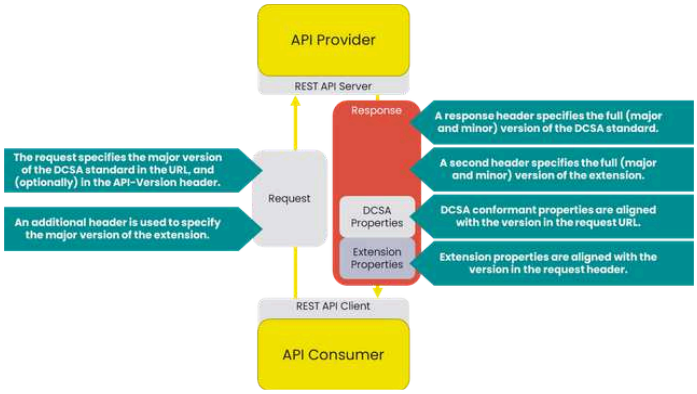
Secondary version headers act as extensions to the existing DCSA approach to API versioning. An API provider offering a conformant extension MUST use these headers in requests to determine that an API consumer is able to consume the extension and also to indicate the exact version of the extension back to the API consumer in the response.

There are requirements for conformant extensions that apply in both the governed and ungoverned cases -

- URL versioning MUST still be used. URL versioning is aligned to the standard version, not to the extension version.
- API consumers requiring features provided using extensions MUST explicitly request those extensions by adding a custom extension header to the request. This custom header MUST only contain the <MAJOR> version of the extension.
- An API provider MUST only provide extensions, such as additional properties in JSON objects, when the API consumer explicitly includes the relevant custom extension header in the HTTP request.
- An API provider MUST NOT provide any part of the response that is not covered by the standard unless the API consumer explicitly requests the extension by providing the relevant custom extension header.
- An API provider MUST NOT provide a response that includes additional values in enum or “enum-like string” properties not covered by the standard unless the API consumer explicitly requests the extension by providing the relevant custom extension header.
- An API provider MUST NOT provide a response that includes values that are outside of ranges of possible values or sets of possible values not covered by the standard unless the API consumer explicitly requests the extension by providing the relevant custom extension header.
- The custom header for the extension MUST NOT be called API-Version, because this name is already reserved within DCSA standardisation to indicate the standard version.
- A custom header with the same name used in the request MUST be added in the response that includes complete version information – example: XXX-DCSA_BKG-Extension-Version: 2.0. An API consumer MAY use this information to adapt processing of the response in order to make use of features introduced into later versions of the extension.
- The custom header MUST NOT be a name that has already been reserved for another purpose within the DCSA governed extensions process.
- DCSA members will adopt a header name prefix for ungoverned extensions based on their SMDG liner code. This prevents naming conflicts between the DCSA members. These prefixes may not be used by other organisations as part of a conformant DCSA API. Any organisation that has been assigned a Liner Code by SMDG may follow the same convention. Other organisations requiring such a prefix (to remove any possibility of naming clashes) may make a request to DCSA. The prefixes for DCSA members are listed in the table below:
- For governed extensions, the extension header name will be assigned by DCSA when the request for the extension is approved and added to the registry. This is intentionally not associated with a particular API provider so that it MAY be reused by other API providers prior to formal standardisation.

DCSA Member / Partner	**Ungoverned Extension Header Name Prefix (based on SMDG Liner Code List v20240506)**
CMA CGM	CMA
Evergreen Line	EMC
Hapag Lloyd	HLC
HMM	HMM
Maersk	MSK
MSC	MSC
ONE	ONE
PIL	PIL

Yang Ming	YML
ZIM	ZIM



Governed extensions

DCSA maintains a registry of extensions that are candidates for future standardisation. These are known as governed extensions.

The conformance requirements for governed extensions are stricter than for ungoverned extensions.

A governed extension MUST NOT introduce a change that would require a new <MAJOR> version if it were introduced as a standard.

The registry provides full details on this topic. It is published separately by DCSA.

Multiple extensions on the same endpoint

An endpoint MAY support multiple extensions on the same endpoint. This will lead to complex behaviour that will be difficult to test so it is not recommended, but it is considered to be conformant.

Conformance Checklist : API Extensions			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Provider	To be conformant, an API endpoint MUST NOT expose any features (endpoints, methods, properties, values etc.) that are part of an extension unless the API consumer explicitly includes the appropriate custom header with a matching <MAJOR> version (of the extension) in the request. Summarising - “DCSA in means DCSA out”.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	For a series of releases within an extension, the DCSA	☐ Conformant ☐ Not conformant	

	backwards compatibility rules MUST be followed for the features that are part of the extension. The <MAJOR> version of the extension MUST be incremented if the extension is not backward compatible with the previous version of the extension.	☐ Not applicable ☐ Unknown	
API Provider	The custom header that identifies the extension MUST include a header name prefix that is specific for the API provider organisation, unless this is a “governed extension” that has been registered and approved by DCSA. The header name prefix MUST be taken from SMDG LCL v20230303 where possible.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	If features that are part of an extension are included in a response. the custom header associated with the extension MUST be added to the response and MUST include the <MAJOR>.<MINOR> version of the extension.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	Where a governed or ungoverned extension is needed, the appropriate custom header MUST be added to API requests.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

---

4. Secure

APIs need to be secure. Measures and practices are implemented to support the requirements for confidentiality, integrity and availability.

API providers and API consumers SHOULD consult their CISO or other security professional to determine appropriate measures. DCSA intentionally places this topic largely out of scope in order to prevent conflicts / duplication with corporate policy.

Some security concerns are particularly relevant from an interoperability perspective. These are included in this document.

For Internet-facing API services, HTTPS MUST be used. The TLS version in use MUST NOT have any known vulnerabilities. Mutual TLS MAY be used but it is not recommended because it is likely to be an impediment to (current and future) interoperability.

Authentication and authorization

All endpoints offered by API providers MUST be accessible only following authentication of API consumers.

All endpoints offered by API providers MUST be accessible only following authorization of API consumers.

OAuth2 SHOULD be implemented by API providers as an authorization solution.

Other approaches to authentication and authorization MAY be used.

Conformance Checklist : Security			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Provider	For APIs offered on the public Internet, HTTPS MUST be used. The version of TLS used MUST NOT contain any known vulnerabilities. Downgrading TLS during negotiation to a version that does contain vulnerabilities MUST NOT be allowed.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All endpoints MUST be accessible only following authentication of API consumers.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Provider	All endpoints MUST be accessible only following authorization of API consumers.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Consumer	For webhook or callback APIs offered on the public Internet, HTTPS MUST be used. The version of TLS used MUST NOT contain any known vulnerabilities. Downgrading TLS during negotiation to a version that does contain vulnerabilities MUST NOT be allowed.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

5. Performant

The performance consequences of API design need to be considered, monitored, and optimised for common interactions.

To achieve this objective, two recommendations are identified below:

- Circuit Breaker Pattern SHOULD be used for fast fail → bliki: Circuit Breaker
- Rate Limiting SHOULD be used to prevent too many requests.

Clarification and usage guidelines (not normative)
DCSA does not set conformance criteria around non-functional requirements such as performance and service levels. This is outside of the DCSA scope. This intentionally provides an area where API providers can differentiate.

6. Well-documented

Up-to-date, complete, and easy-to-read documentation SHOULD be available to facilitate adoption.

This documentation SHOULD be easy for engineers of adopters to understand and use.

OpenAPI Specification 3.0.3 and/or OpenAPI Specification 3.1 MUST be used for documenting API standards using YAML, as a minimum.

British English MUST be used in documentation.

DCSA will publish API standards documentation and make this clearly available either directly from the DCSA website, or from another repository that is linked from the DCSA website.

Conformance Checklist : Well-documented			
Defined by the API Design Principles Document		**To be completed for a particular standard / implementation**	
Conformance Role	**Conformance Criteria**	**Conformance Status**	**Justification for Conformance Status**
API Design	OpenAPI Specification 3.0.3 and/or OpenAPI Specification 3.1 MUST be used for documenting API standards using YAML.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	
API Design	British English MUST be used in documentation.	☐ Conformant ☐ Not conformant ☐ Not applicable ☐ Unknown	

Normative References

This document is based on a number of normative references, which are listed below.

Reference	**URL**
JSON	ECMA-404 - Ecma International
OpenAPI Specification 3.0.3	OpenAPI Specification v3.0.3 | Introduction, Definitions, & More

OpenAPI Specification 3.1	OpenAPI Specification v3.1.0 | Introduction, Definitions, & More
Unicode 15.0.0	Unicode 15.0.0
ISO 10646	Publicly Available Standards
application/json Media Type	https://www.ietf.org/rfc/rfc4627.txt
Content Type HTTP header	Content-Type fields in MIME
JSON Schema 2020-12	JSON Schema - Specification [#section]
JSON Merge Patch	RFC 7396: JSON Merge Patch
JSON Patch	RFC 6902: JavaScript Object Notation (JSON) Patch
Developing Restful APIs: A Comprehensive Set of Guidelines by Zalando	GitHub - zalando/restful-api-guidelines at 815821712a263a1afa0bd8c1e7785a4222b5efbe
Link header	RFC 5988: Web Linking
JSONPath	JSONPath - XPath for JSON
UUID	RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace
SMDG Liner Code List	SMDG Liner Code List (LCL) | SMDG e.V.