Table of Contents

Open Geospatial Consortium

Submission Date: 2019-10-30

Approval Date: 2019-11-22

Publication Date: 2019-12-12

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t15-D012

Internal reference number of this OGC® document: 19-010r2

Category: OGC Public Engineering Report

Editor: Clemens Portele

OGC Testbed-15: Styles API Engineering Report

Warning

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Public Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type:	OGC Public Engineering Report
Document subtype:	Interface
Document stage:	Approved for public release
Document language:	English

License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize

you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

i. Abstract

This document is a proof of concept of a draft specification of the OGC Styles Application Programming Interface (API) that defines a Web API that enables map servers and clients as well as visual style editors to manage and fetch styles.

Web APIs are software interfaces that use an architectural style that is founded on the technologies of the Web. Styles consist of symbolizing instructions that are applied by a rendering engine on features and/or coverages.

The Styles API supports several types of consumers, mainly:

- Visual style editors that create, update and delete styles for datasets that are shared by other Web APIs implementing the OGC API - Features - Part 1: Core standard or the draft OGC API - Coverages or draft OGC API - Tiles specifications;
- Web APIs implementing the draft OGC API - Maps specification fetch styles and render spatial data (features or coverages) on the server;
- Map clients that fetch styles and render spatial data (features or coverages) on the client.

Feature data is either accessed directly or organized into spatial partitions such as a tiled data store (aka "vector tiles").

The Styles API is consistent with the emerging OGC API family of standards.

The Styles API implements the conceptual model for style encodings and style metadata as documented in chapter 6 of the "OGC Testbed-15: Encoding and Metadata Conceptual Model for Styles Engineering Report".

The model defines three main concepts:

1. The **style** is the main resource.
2. Each style is available in one or more **stylesheets** - the representation of a style in an encoding like OGC SLD 1.0 or Mapbox Style. Clients will use the stylesheet of a style that fits best based on the capabilities of available tools and their preferences.
3. For each style there is **style metadata** available, with general descriptive information about the style, structural information (e.g., layers and attributes), and so forth to allow users to discover and select existing styles for their data.

This model directly maps to the resources and documents in the Styles API, which supports the resources and operations listed in the Table below.

Table 1. Styles API - overview of resources and applicable HTTP methods

Resource	Path	HTTP method	Document reference
Landing page	`/`	GET	API landing page
Conformance declaration	`/conformance`	GET	Declaration of conformance classes

Resource	Path	HTTP method	Document reference
Styles	`/styles`	GET	Fetch styles
		POST	Create a new style
			Validate a style
Style	`/styles/{styleId}`	GET	Fetch style
		PUT	Update or create a style
			Validate a style
		DELETE	Delete a style
Style metadata	`/styles/{styleId}/metadata`	GET	Fetch style metadata
		PUT	Replace the metadata of a style
		PATCH	Update parts of the metadata of a style
Resources	`/resources`	GET	Fetch resources
Resource	`/resources/{resourceId}`	GET	Fetch resource
		PUT	Create or replace a resource
		DELETE	Delete a resource

In order to support styles, data APIs (for example, supporting OGC API Features and/or the draft OGC API Tiles) require additional capabilities, too. These are:

- List and manage the applicable styles per feature collection (path `/collections/{collectionId}`).
- Add a `queryables` resource (path `/collections/{collectionId}/queryables`) to support clients such as visual style editors to construct expressions for selection criteria in queries on features in the collection. "Queryable" means that the property may be used in styling rules or other filter expressions.

To support styling of coverage data, other additional capabilities in the data API may be required, but have not been investigated by Testbed 15.

This document uses OpenAPI 3.0 to specify the building blocks of the API.

> The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.
>
> An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.
>
> — OpenAPI Specification, Introduction

ii. Keywords

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, OpenAPI, OGC API, style, style encoding, style metadata, Styles API

iii. Preface

OGC is currently missing a robust conceptual model and APIs capable of supporting styles with multiple style encodings (for example OGC SLD and Mapbox Style). The Open Portrayal Framework (OPF) task in Testbed-15 investigated this issue, building on previous portrayal activities in the OGC. This document specifies building blocks for Web APIs consistent with the OGC API series to manage and fetch styles.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the draft specification set forth in this document, and to provide supporting documentation.

iv. Submitting organizations

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- Ecere Corporation
- GeoSolutions
- interactive instruments GmbH
- Leidos
- Reinventing Geospatial, Inc.
- US Army Geospatial Center (AGC)

v. Submitters

All questions regarding this submission should be directed to the editor or the submitters:

Name	**Affiliation**
Clemens Portele *(editor)*	interactive instruments GmbH
Andrea Aime	GeoSolutions
Jeff Harrison	US Army Geospatial Center (AGC)
Jérôme Jacovella-St-Louis	Ecere Corporation
Richard Kim	Reinventing Geospatial, Inc.

Chapter 1. Scope

The Styles Application Programming Interface (API) is a Web API that enables map servers and clients as well as visual style editors to manage and fetch styles.

The API is consistent with the emerging OGC API family of standards. The API complements the current and emerging OGC API specifications for features, maps and tiles and builds on the conceptual model for the encoding of styles and their metadata developed in OGC Testbed-15.

The building blocks of the API are specified using OpenAPI 3.0.

Chapter 2. Conformance

This draft specification defines five requirements/conformance classes for the Styles API:

- "core" provides access to styles and their metadata. JSON is a mandatory encoding in requests and responses where JSON schemas have been specified for the Styles API.
- "manage-styles" adds the capabilities for creating, updating and deleting styles and their metadata.
- "style-validation" adds the capability to validate a stylesheet.
- "resources" add the capabilities to provide access to resources referenced from stylesheets (symbols, sprites) or style metadata (thumbnails).
- "manage-resources" add the capabilities for creating, updating and deleting resources.

In addition, there are four requirements/conformance classes for additional encodings supported by resources of the API:

- "html" supports HTML in responses to GET requests for all requests to the Styles API.
- "mapbox-styles" supports Mapbox Styles as a style encoding.
- "sld-10" supports OGC SLD 1.0 as a style encoding.
- "sld-11" supports OGC SLD 1.1 as a style encoding.

Finally, there are two requirements/conformance classes extending the information about Collection resources specified in OGC API - Features - Part 1: Core:

- "style-info" adds information about available styles for each collection.
- "queryables" adds information about the feature properties that may be used in styling rules.

The standardization target for all classes is: Web API.

Conformance with this draft specification shall be checked using all the relevant tests specified in Annex A (normative) of this document. The framework, concepts, and methodology for testing, and the criteria to be achieved to claim conformance are specified in the OGC Compliance Testing Policies and Procedures and the OGC Compliance Testing web site.

In order to conform to this draft specification, a software implementation has to implement "core".

Chapter 3. References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this document. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies.

IETF: RFC 7396, JSON Merge Patch (2014)

OGC: OGC 02-070, Styled Layer Descriptor, Version 1.0 (2002)

OGC: OGC 05-078r4, Styled Layer Descriptor, Version 1.1 (2007)

OGC: OGC 17-069r3, OGC API - Features - Part 1: Core (2019)

NOTE	If "OGC API - Common" would be available and consistent with "OGC API - Features - Part 1: Core", "OGC API - Common" would be a normative reference instead of "OGC API - Features - Part 1: Core".

WhatWG: HTML (Living Standard)

Chapter 4. Terms and Definitions

This document uses the terms defined in Sub-clause 5.3 of [OGC 06-121r8], which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this draft specification.

For the purposes of this document, the following additional terms and definitions apply.

4.1. queryable

a property that can be queried

4.2. <portrayal> sprite

an image containing a collection of uniformly-sized symbols as sub-images

4.3. style

a sequence of rules of symbolizing instructions to be applied by a rendering engine on one or more features and/or coverages

4.4. style encoding

specification to express a style as one or more files

NOTE	In Testbed-15 Mapbox Styles, OGC SLD versions 1.0 and 1.1 are used.

4.5. stylesheet

representation of a style in a style encoding

4.6. style metadata

essential information about a style in order to support users in discovering and selecting styles for rendering their data and for visual style editors to create user interfaces for editing a style

4.7. Web API

API using an architectural style that is founded on the technologies of the Web [source: OGC API - Features - Part 1: Core]

NOTE	See Best Practice 24: Use Web Standards as the foundation of APIs (W3C Data on the Web Best Practices) for more detail.

Chapter 5. Conventions

This section provides details and examples for any conventions used in the document. Examples of conventions are symbols, abbreviations, use of XML schema, or special notes regarding how to read the document.

5.1. Identifiers

The normative provisions in this draft specification are denoted by the URI

http://www.opengis.net/t15/opf-styles-1/1.0

All requirements and conformance tests that appear in this document are denoted by partial URIs which are relative to this base.

5.2. Abbreviated terms

API

Application Programming Interface

NGA

US National Geospatial Intelligence Agency

OGC

Open Geospatial Consortium

SLD

OGC Styled Layer Descriptor

TDS

Topographic Data Store (an NGA specification)

Chapter 6. Introduction

6.1. Overview

This document specifies draft building blocks for Web APIs to manage and fetch styles supporting multiple style encodings and metadata to describe and discover styles.

The Styles API supports several types of consumers, mainly:

- Visual style editors that create, update and delete styles for datasets that are shared by other Web APIs implementing the OGC API - Features - Part 1: Core standard or the draft OGC API - Coverages or draft OGC API - Tiles specifications;
- Web APIs implementing the draft OGC API - Maps specification fetch styles and render spatial data (features or coverages) on the server;
- Map clients that fetch styles and render spatial data (features or coverages) on the client.

Feature data is either accessed directly or organized into spatial partitions such as a tiled data store (aka "vector tiles").

The Styles API is consistent with the emerging OGC API family of standards.

The remainder of this Clause illustrates use cases and workflows that the Styles API could support.

Clause 7 specifies the Styles API.

Clause 8 specifies extensions to OGC API - Features - Part 1: Core standard (or the emerging OGC API - Common specification) to support the use cases.

6.2. Use cases

This section describes expectations of how clients will interact with the Styles API.

The following use cases assume that:

- Some feature dataset that is structured according to a data specification, such as the NGA Topographic Data Store 6.1 (TDS), is available via an API that implements the OGC API - Features - Part 1: Core and draft OGC API - Tiles specifications;
- Roads are included in the data in a collection `transportationgroundcrv` as features with a property f_code with a value of AP030;
- The URI of the landing page is `http://example.org/data-api`;
- A style repository is available via an API that implements the Styles API specification;
- The URI of the landing page of the Styles API is `http://example.org/styles-api`.

NOTE	The URIs in the use case descriptions are examples and use the domain `example.org`, a reserved domain maintained by IANA for illustrative examples in documents without prior coordination with IANA.

6.2.1. A map client

A map client that wants to visualize data for features or tiled feature data for the collection http://example.org/data-api/collections/transportationgroundcrv will look for a styles member in the response. The client will probably select one of the styles from the list taking the media types of the supported stylesheets into account and provide a capability so that users can change the style. The stylesheet returned based on the href member of the link will be used to render the data.

In addition to feature data, the map client might also fetch a hillshade style to apply to an elevation coverage accessed from a Web API supporting the Testbed-15 Image API or OGC API Coverages.

6.2.2. A visual style editor creating a new style

A user wants to create a new style for TDS roads using a visual style editor. The user knows the dataset and the data access API.

A user creates the style in the visual style editor, selects the native stylesheet language for the style and identifies the transportationgroundcrv collection in the dataset as a sample data source. The visual style editor executes a request to the landing page (http://example.org/data-api) and the conformance declaration (http://example.org/data-api/conformance) of the data access API to determine the API capabilities. Note that alternatively the OpenAPI definition may be inspected, but for a client that supports the OGC API standards in general, using the API resources directly is often simpler and, therefore, used in this example.

If the visual style editor supports, for example, both the styling of GeoJSON features or Mapbox Vector Tile data, the editor would require support for at least one of the two following sets of conformance classes:

- http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/core and http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/geojson or
- http://www.opengis.net/spec/ogcapi-tiles-1/1.0/conf/core (with URI templates referencing tiles of media type application/vnd.mapbox-vector-tile).

The first option provides access to GeoJSON features via http://example.org/data-api/collections/transportationgroundcrv/items, the second one provides access to Mapbox Vector Tiles (MVT) encoded data via http://example.org/data-api/collections/transportationgroundcrv/tiles.

In addition, the visual style editor will look for the following conformance classes:

- http://www.opengis.net/t15/opf-styles-1/1.0/conf/queryables: If this conformance class is supported, the visual style editor can specify styling rules that make use of feature properties. Otherwise all styling rules will apply to all features in each collection.
- http://www.opengis.net/t15/opf-styles-1/1.0/conf/style-info: If this conformance class is supported, the visual style editor will be able to create a link from the collection to the newly created style.

The editor will also request information about the features in the collection via a request to http://example.org/data-api/collections/transportationgroundcrv.

If http://www.opengis.net/t15/opf-styles-1/1.0/conf/queryables is supported, the queryables are retrieved via a request to http://example.org/data-api/collections/transportationgroundcrv/queryables.

Based on this information, the visual style editor is able to configure its user interface and guide the user through the creation of the style for road features and visualize the draft style using the sample data. Once the user has finished the style, the style is published on a Style repository that supports the Styles API.

If the user requests the use of a Style repository that the editor interacts with for the first time, the editor will again inspect the capabilities of the repository by fetching the conformance declaration at http://example.org/styles-api/conformance.

At least the following conformance classes must be supported in order for sharing the new style via the repository.

- http://www.opengis.net/t15/opf-styles-1/1.0/conf/core
- http://www.opengis.net/t15/opf-styles-1/1.0/conf/manage-styles

In addition, if the style includes symbols or sprites, the repository also has to support the following conformance classes:

- http://www.opengis.net/t15/opf-styles-1/1.0/conf/resources
- http://www.opengis.net/t15/opf-styles-1/1.0/conf/manage-resources

Finally, the repository has to support the native stylesheet language that the user has selected for the style definition, i.e. one of:

- http://www.opengis.net/t15/opf-styles-1/1.0/conf/mapbox-styles
- http://www.opengis.net/t15/opf-styles-1/1.0/conf/sld-10
- http://www.opengis.net/t15/opf-styles-1/1.0/conf/sld-11

The visual style editor will ask the user for her credentials (username and password) in the style repository and use the credentials in any of the following POST/PUT/PATCH requests.

If http://www.opengis.net/t15/opf-styles-1/1.0/conf/style-validation is supported, the visual style editor can also offer validation of the draft style any time during the drafting process using POST requests with the draft stylesheet to http://example.org/styles-api/styles?validate=only.

To create the new style either a POST request with the stylesheet to http://example.org/styles-api/styles or a PUT request to http://example.org/styles-api/styles/{styleId} (where {styleId} is the identifier of the style specified by the user) is sent. ?validate=true may also be added to the request URI to trigger validation in this step if the style validation conformance class is supported. If PUT is used, the visual style editor should check that no existing style {styleId} exists.

After a successful creation of the style (in case of a POST request, the URI of the new style http://example.org/styles-api/styles/{styleId} is returned in an HTTP header Location), the visual style editor will update the style metadata using a PUT or PATCH request to http://example.org/styles-api/styles/{styleId}/metadata.

If the data access API supports the conformance class http://www.opengis.net/t15/ogcapi-features-

m/1.0/conf/style-links, the visual style editor will add a link to the new style using a PATCH request to http://example.org/data-api/collections/transportationgroundcrv.

6.2.3. A visual style editor updating an existing style

The process is quite similar to the previous example with the following changes:

- The user will start from an existing style, not with a new style. In other words, the user will open/load the style from the style repository and the editor will fetch a stylesheet of the style from http://example.org/styles-api/styles/{styleId} (in the style encoding of choice) and the styles metadata from http://example.org/styles-api/styles/{styleId}/metadata.
- If the style metadata includes links to sample data (e.g., http://example.org/data-api/collections/transportationgroundcrv), the editor may use that data for sample visualizations and perhaps to determine changes to queryables. The user may also select other data sources for these purposes.
- Since an existing style is updated, the style definition will always be updated with a PUT request to http://example.org/styles-api/styles/{styleId} (no POST request to http://example.org/styles-api/styles, which would create a new style).

6.2.4. A Web API implementing OGC API - Maps

A Web API that implements the conformance class "Map tile" of the OGC API Maps specification returns geo-referenced bitmap images showing maps. The URI template for the map tiles is /collections/{collectionId}/map/{styleId}/tiles/{tileMatrixSetId}/{tileMatrix}/{tileRow}/{tileCol} and includes a query parameter styleId. If a client requests a map tile for the collection transportationgroundcrv the API will use the requested style to render the map. The stylesheet may be fetched from the same Web API or another Web API that supports the Styles API.

Chapter 7. The Styles API

NOTE	This clause specifies the Styles API as designed and implemented in the Open Portrayal Framework task of OGC Testbed 15.

Stylesheets often reference external resources, especially symbols and fonts to be used in the rendering process. Symbols are either managed as a single file for each symbol or they are organized in a sprite. In a sprite, all symbols are combined into a single bitmap image to reduce memory and the number of http requests. Single symbols and sprites are both supported by the Styles API. Further, they may be stored in the Styles API. For example, this approach would avoid issues with cross-origin requests. Of course, existing external symbol libraries may also be referenced from stylesheets. The Styles API currently does not support font resources. If external fonts / glyphs are used in a stylesheet, an existing font library has to be referenced.

The API supports the resources and operations listed in the Table below with the associated conformance class and the link to the document section that specifies the requirements.

Table 2. Overview of resources and applicable HTTP methods

Resource	Path	HTTP method	Conformance class	Document reference
Landing page	`/`	GET	core	API landing page
Conformance declaration	`/conformance`	GET	core	Declaration of conformance classes
Styles	`/styles`	GET	core	Fetch styles
		POST	manage-styles	Create a new style
			style-validation	Validate a style
Style	`/styles/{styleId}`	GET	core	Fetch style
		PUT	manage-styles	Update or create a style
			style-validation	Validate a style
		DELETE	manage-styles	Delete a style
Style metadata	`/styles/{styleId}/metadata`	GET	core	Fetch style metadata
		PUT	manage-styles	Replace the metadata of a style
		PATCH	manage-styles	Update parts of the metadata of a style
Resources	`/resources`	GET	resources	Fetch resources
Resource	`/resources/{resourceId}`	GET	resources	Fetch resource
		PUT	manage-resources	Create or replace a resource
		DELETE	manage-resources	Delete a resource

The conceptual model and this draft specification support multiple style encodings (stylesheets) per style. For example, a Styles API may publish a "night" style in the style encodings OGC SLD 1.0, OGC SLD 1.1 and Mapbox Style. The client will select the stylesheet that fits best based on its capabilities and preferences.

This version of the Styles API was written with the following assumptions:

- When a new style is created using POST `/styles` or PUT `/styles/{styleId}`, the submitted stylesheet is the reference.
- A server may derive stylesheets in other style encodings from the reference stylesheet, but there is no requirement to support such a capability. If one or more stylesheets are derived, they will be automatically be added to the style metadata.
- When an existing style is updated using PUT `/styles/{styleId}`, the submitted stylesheet becomes the new reference and all other stylesheets for the style are removed. New stylesheets may be derived from the new reference stylesheet. The style metadata is updated.

7.1. Requirements Class "Core"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/core	
Target type	Web API
Dependency	OGC API - Common (Core)

An "OGC API - Common" specification is under development (October 1, 2019). The current draft of Common is based on the generic concepts of the OGC API - Features - Part 1: Core standard. However, the work is in the earliest stages of the standardization process. In order to avoid duplicating content, this document does not copy the basic requirements, recommendations and permissions from OGC API Common/Features. If "OGC API - Common" is not available as a normative reference, the alternative would be to remove the dependency and to specify the following normative statements are part of this requirements class:

- Landing page
  - Requirement /req/core/root-op
  - Requirement /req/core/root-success
    - Change: No `data` link to `/collections` is required, but a `styles` link has to point to the `/styles` resource
- API definition
  - Requirement /req/core/api-definition-op
  - Permission /per/core/api-definition-uri
  - Requirement /req/core/api-definition-success
  - Recommendation /rec/core/api-definition-oas
- Conformance declaration

NOTE

    - Requirement /req/core/conformance-op
    - Requirement /req/core/conformance-success
- Web API
    - Requirement /req/core/http
    - Recommendation /rec/core/head
    - Permission /per/core/additional-status-codes
    - Requirement /req/core/query-param-unknown
    - Requirement /req/core/query-param-invalid
    - Recommendation /rec/core/etag
    - Recommendation /rec/core/cross-origin
    - Recommendation /rec/core/link-header

The recommendation /rec/core/cross-origin is in particular relevant to support browser-based visual style editors. It is recommended to support CORS. It is important to declare all relevant headers in the response. For APIs that support the "manage-styles" conformance class especially the `Location` header needs to be declared (for example, "access-control-expose-headers: Location, Link") to allow clients access to the URI of a newly created style.

The recommendation /rec/core/string-i18n is mainly implemented by the Content-Language header in the response to requests, in particular to those returning a stylesheet or style metadata.

The requirement /req/core/crs84 is not applicable to the Styles API since no geometries are used in the API.

7.1.1. API landing page

The following is an example of the landing page of a Styles API. This implementation supports the "json" conformance class, but not the "html" conformance class.

Example 1. Landing page in JSON


```
{
  "links": [
    {
      "href": "https://example.org/api/v1",
      "rel": "self",
      "type": "application/json",
      "title": "this document"
    },
    {
      "href": "https://example.org/api/v1/api",
      "rel": "service-desc",
      "type": "application/vnd.oai.openapi+json;version=3.0",
      "title": "the API definition in OpenAPI JSON"
    },
    {
      "href": "https://example.org/api/v1/api.html",
      "rel": "service-doc",
      "type": "text/html",
      "title": "the API documentation in HTML"
    },
    {
      "href": "https://example.org/api/v1/conformance",
      "rel": "conformance",
      "type": "application/json",
      "title": "list of conformance classes implemented by this API"
    },
    {
      "href": "https://example.org/api/v1/styles",
      "rel": "styles",
      "type": "application/json",
      "title": "the styles shared via this API"
    }
  ]
}
```


7.1.2. Declaration of conformance classes

The following is an example of the conformance declaration of a Styles API that implements all requirements classes except "html".

Example 2. Conformance declaration in JSON

```
{
  "conformsTo": [
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/core",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/manage-styles",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/style-validation",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/resources",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/manage-resources",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/mapbox-styles",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/sld-10",
      "http://www.opengis.net/t15/opf-styles-1/1.0/conf/sld-11"
    ]
}
```

7.1.3. Fetch styles

This operation returns a list of styles that are currently available.

Requirement 1	**/req/core/styles-op**
A	The server SHALL support the HTTP GET operation at the path `/styles`.

Requirement 2	**/req/core/styles-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `200`.

B	The content of that response SHALL be based upon the following OpenAPI 3.0 schema:
	
	`type: object`
	`required:`
	`  - styles`
	`properties:`
	`  styles:`
	`  type: array`
	`  nullable: true`
	`  items:`
	`  type: object`
	`  nullable: true`
	`  required:`
	`  - id`
	`  - links`
	`  properties:`
	`  id:`
	`  type: string`
	`  nullable: true`
	`  title:`
	`  type: string`
	`  nullable: true`
	`  links:`
	`  type: array`
	`  nullable: true`
	`  minItems: 1`
	`  items:`
	`  $ref:`
	`'http://schemas.opengis.net/ogcapi/features/part1/1.0/op`
	`enapi/ogcapi-features-1.yaml#/components/schemas/link'`

C	The `styles` member SHALL include one item for each style currently on the server.
D	The `id` member of each style SHALL be unique.
E	Each style SHALL have at least one link to a style encoding supported for the style (link relation: `stylesheet`) with the `type` attribute stating the media type of the style encoding.
F	Each style SHALL have a link to the style metadata (link relation: `describedBy`) with the `type` attribute stating the media type of the metadata encoding.

NOTE	Currently the links to the thumbnails of a style are available only as part of the style metadata (see recommendation "/rec/core/style-md-preview"). To display an overview of the styles with a thumbnail image, a client needs to send multiple requests, the first one for the list of styles and then a request for each style metadata to get the thumbnail links. Whether the preview should also be included for each style in the Styles resource should be discussed.

Recommendation 1	**/rec/core/style-title**
A	If a style has a title, it SHOULD be included in the `title` member of the style.

Example 3. JSON encoding of styles


```
{
  "styles": [
    {
      "id": "night",
      "title": "Topographic night style",
      "links": [
        {
          "href": "https://example.com/api/v1/styles/night?f=mapbox",
          "type": "application/vnd.mapbox.style+json",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/v1/styles/night?f=sld10",
          "type": "application/vnd.ogc.sld+xml;version=1.0",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/v1/styles/night/metadata?f=json",
          "type": "application/json",
          "rel": "describedBy"
        }
      ]
    },
    {
      "id": "topographic",
      "title": "Regular topographic style",
      "links": [
        {
          "href": "https://example.com/api/v1/styles/topographic?f=mapbox",
          "type": "application/vnd.mapbox.style+json",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/v1/styles/topographic?f=sld10",
          "type": "application/vnd.ogc.sld+xml;version=1.0",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/v1/styles/topographic/metadata?f=json",
          "type": "application/json",
          "rel": "describedBy"
        }
      ]
    }
  ]
}
```

7.1.4. Fetch style

This operation returns the stylesheet of a style.

Requirement 3	**/req/core/style-op**
A	The server SHALL support the HTTP GET operation at the path `/style/{styleId}` for each style referenced from the Styles resource at `/styles`.

Requirement 4	**/req/core/style-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `200`.
B	The content of that response SHALL conform to the media type stated in the `Content-Type` header.
C	The language used in linguistic text in the response SHALL be consistent with the language stated in the `Content-Language` header.

NOTE	The `Content-Language` header in a HTTP response is used to describe the language(s) intended for the audience. If no Content-Language is specified, the default is that the content is intended for all language audiences.

7.1.5. Fetch style metadata

This operation returns the metadata of a style.

Requirement 5	**/req/core/style-md-op**
A	The server SHALL support the HTTP GET operation at the path `/style/{styleId}/metadata` for each style metadata referenced from the Styles resource at `/styles`.

Requirement 6	**/req/core/style-md-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `200`.

B	The content of that response SHALL be based upon the following OpenAPI 3.0 schema:
	
	`type: object`
	`required:`
	`  - id`
	`properties:`
	`  id:`
	`  type: string`
	`  title:`
	`  type: string`
	`  nullable: true`
	`  description:`
	`  type: string`
	`  nullable: true`
	`  keywords:`
	`  type: array`
	`  nullable: true`
	`  items:`
	`  type: string`
	`  pointOfContact:`
	`  type: string`
	`  nullable: true`
	`  accessConstraints:`
	`  type: string`
	`  nullable: true`
	`  enum:`
	`  - unclassified`
	`  - confidential`
	`  - restricted`
	`  - secret`
	`  - topSecret`
	`  dates:`
	`  type: object`
	`  nullable: true`
	`  properties:`
	`  creation:`
	`  type: string`
	`  format: date-time`
	`  nullable: true`
	`  publication:`
	`  type: string`
	`  format: date-time`
	`  nullable: true`
	`  revision:`
	`  type: string`
	`  format: date-time`
	`  nullable: true`
	`  validTill:`
	`  type: string`

C	The language used in linguistic text in the response SHALL be consistent with the language stated in the `Content-Language` header.

The elements of the schema are defined in "OGC Testbed-15: Encoding and Metadata Conceptual Model for Styles Engineering Report".

Recommendation 2	**/rec/core/style-md-sample-data**
A	Sample data that can be used to illustrate the style SHOULD be represented as links with the following link relation types: • `enclosure` for links to sample data that may be downloaded (e.g. a GeoPackage); • `collection` for links to a Collection resource according to OGC API Common (e.g. `/collections/{collectionId}`; the collection may be available as features (tiled or not) or as gridded data); • `start` for links to a Features resource according to OGC API Features (e.g. `/collections/{collectionId}/items`; the response may contain a `next` link to additional features); • `tiles` for a link to a Tile Collection resource (e.g. `/collections/{collectionId}/tiles`).

NOTE Additional rules may be needed for links to coverage data.

Recommendation 3	**/rec/core/style-md-preview**
A	A link to a thumbnail SHOULD be included with link relation `preview` (specified by RFC 6903) and the appropriate media type in the `type` parameter.

The thumbnail may be an image that is published as a resource in the API. The thumbnail can reference an appropriate raster tile, a map request, etc.

Example 4. Style metadata in JSON

```
{
  "id": "night",
  "title": "Topographic night style",
  "description": "This topographic basemap style is designed to be used in
situations with low ambient light. The style supports datasets based on the TDS
6.1 specification.",
  "keywords": [
    "basemap",
```

```
    "TDS",
    "TDS 6.1",
    "OGC API"
  ],
  "pointOfContact": "John Doe",
  "accessConstraints": "unclassified",
  "dates": {
    "creation": "2019-01-01T10:05:00Z",
    "publication": "2019-01-01T11:05:00Z",
    "revision": "2019-02-01T11:05:00Z",
    "validTill": "2019-02-01T11:05:00Z",
    "receivedOn": "2019-02-01T11:05:00Z"
  },
  "scope": "style",
  "version": "1.0.0",
  "stylesheets": [
    {
      "title": "Mapbox Style",
      "version": "8",
      "specification": "https://docs.mapbox.com/mapbox-gl-js/style-spec/",
      "native": true,
      "tilingScheme": "GoogleMapsCompatible",
      "link": {
        "href": "https://example.org/api/v1/styles/night?f=mapbox",
        "rel": "stylesheet",
        "type": "application/vnd.mapbox.style+json"
      }
    },
    {
      "title": "OGC SLD",
      "version": "1.0",
      "native": false,
      "link": {
        "href": "https://example.org/api/v1/styles/night?f=sld10",
        "rel": "stylesheet",
        "type": "application/vnd.ogc.sld+xml;version=1.0"
      }
    }
  ],
  "layers": [
    {
      "id": "vegetationsrf",
      "type": "polygon",
      "sampleData": {
        "href": "https://services.interactive-
instruments.de/vtp/daraa/collections/vegetationsrf/items?f=json&limit=100",
        "rel": "data",
        "type": "application/geo+json"
      }
    },
    {
```

```
      "id": "hydrographycrv",
      "type": "line",
      "sampleData": {
        "href": "https://services.interactive-
instruments.de/vtp/daraa/collections/hydrographycrv/items?f=json&limit=100",
        "rel": "data",
        "type": "application/geo+json"
      },
      "attributes": [
        {
          "id": "f_code",
          "type": "string"
        }
      ]
    }
  ],
  "links": [
    {
      "href": "https://example.org/api/v1/resources/night-thumbnail.png",
      "rel": "preview",
      "type": "image/png",
      "title": "thumbnail of the night style applied to OSM data from Daraa,
Syria"
    }
  ]
}
```

7.2. Requirements Class "Manage styles"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/manage-styles	
Target type	Web API
Dependency	Requirements Class "Core"
Dependency	RFC 7396 (JSON Merge Patch)

7.2.1. Create a new style

This operation creates a new style. The payload of the request is a stylesheet of the style in one of the supported style encodings.

If the style submitted in the request body includes an identifier (this depends on the style encoding), that identifier will be used. If a style with that identifier already exists, an error is returned.

EXAMPLE

For Mapbox Styles use the value of the `name` member and for OGC SLD use the value of the `Name` child element, if these are provided.

Note that such identifiers may result in URIs that include encoded characters. To avoid this, use PUT `/styles/{styleId}` instead and specify the desired `styleId` explicitly.

If no identifier can be determined from the submitted style, the server will assign a new identifier to the style.

The URI of the new style is returned in the header `Location`.

Requirement 7	**/req/manage-styles/create-style-op**
A	The server SHALL support the HTTP POST operation at the path `/styles`.
B	The server SHALL accept a stylesheet in one of the style encodings supported by the API.

Requirement 8	**/req/manage-styles/create-style-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `201`.
B	The response SHALL include a header `Location` with the URI of the new style.
C	A minimal style metadata resource SHALL be created at `/styles/{styleId}/metadata`.

Note that the metadata will be incomplete and should be updated by the client to keep the style metadata consistent with the style definition.

Requirement 9	**/req/manage-styles/create-style-error**
A	If the request does not conform to the requirements (e.g., the stylesheet is invalid) a response with status code `400` SHALL be returned.

Recommendation 4	**/rec/manage-styles/id-exists**
A	If the request is valid, but the server already has a style with the identifier stated in the stylesheet, a response with status code `409` SHOULD be returned.

Example 5. New style response

The URI of the new style is `https://example.org/api/v1/styles/night`.

```
HTTP/1.1 201 Created
Date: Sun, 28 Jul 2019 12:32:34 GMT
Location: https://example.org/api/v1/styles/night
```

7.2.2. Update or create a style

This operation updates the style with the id `styleId`. If no such style exists, a new style with that id is added.

For updated styles, the style metadata resource at `/styles/{styleId}/metadata` is not updated. For new styles a minimal style metadata resource is created, too. Please update the metadata using a PUT request to keep the style metadata consistent with the style definition.

Requirement 10	**/req/manage-styles/update-style-op**
A	The server SHALL support the HTTP PUT operation at the path `/styles/{styleId}`.
B	The server SHALL accept a stylesheet in one of the style encodings supported by the API.

Requirement 11	**/req/manage-styles/update-style-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.
B	If a new style is created, a minimal style metadata resource SHALL be created at `/styles/{styleId}/metadata`.

Note that the metadata should be updated by the client, too, to keep the style metadata consistent with the style definition.

Requirement 12	**/req/manage-styles/update-style-error**
A	If the request does not conform to the requirements (e.g., the stylesheet is invalid) a response with status code `400` SHALL be returned.

7.2.3. Delete a style

This operation deletes the style with the id `styleId`. If no such style exists, an error is returned.

Deleting a style also deletes the subordinate resources, i.e., the style metadata.

Requirement 13	**/req/manage-styles/delete-style-op**
A	The server SHALL support the HTTP DELETE operation at the path `/styles/{styleId}`.

Requirement 14	**/req/manage-styles/delete-style-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.
B	All subordinate resources including the style metadata at `/styles/{styleId}/metadata` SHALL be deleted, too.

Requirement 15	**/req/manage-styles/delete-style-error**
A	If the style does not exist, a response with status code `404` SHALL be returned.

7.2.4. Replace the metadata of a style

This operation replaces the metadata of the style with the id `styleId`. If no such style exists, an error is returned.

Requirement 16	**/req/manage-styles/update-style-md-op**
A	The server SHALL support the HTTP PUT operation at path `/styles/{styleId}/metadata`.
B	The server SHALL accept style metadata based on the schema requirement /req/core/style-md-success, item B in all encodings supported by the API.

Requirement 17	**/req/manage-styles/update-style-md-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.

B	The style metadata SHALL be replaced by the content submitted in the request.

Requirement 18	**/req/manage-styles/update-style-md-error**
A	If the style does not exist, a response with status code `404` SHALL be returned.

7.2.5. Update parts of the metadata of a style

This operation updates the metadata of the style with the id `styleId`. If no such style exists, an error is returned.

Requirement 19	**/req/manage-styles/patch-style-md-op**
A	The server SHALL support the HTTP PATCH operation at path `/styles/{styleId}/metadata`.
B	The server SHALL accept style metadata based on the schema requirement /req/core/style-md-success, item B in all encodings supported by the API.

Requirement 20	**/req/manage-styles/patch-style-md-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.
B	The style metadata SHALL be updated by the content submitted in the request as specified by RFC 7396 (JSON Merge Patch).

From the RFC 7396 (JSON Merge Patch) specification:

> A JSON merge patch document describes changes to be made to a target JSON document using a syntax that closely mimics the document being modified. Recipients of a merge patch document determine the exact set of changes being requested by comparing the content of the provided patch against the current content of the target document. If the provided merge patch contains members that do not appear within the target, those members are added. If the target does contain the member, the value is replaced. Null values in the merge patch are given special meaning to indicate the removal of existing values in the target.

NOTE	A more flexible, but more complex option for JSON-based PATCH operations is specified by RFC 6902. JSON Merge Patch is used because of its simpler and more intuitive design. An XML-based PATCH operation is specified by RFC 5261.

Some examples using JSON Merge Patch include:

To add or update the point of contact, the access constraint and the revision date, just send:

```
{
  "pointOfContact": "Jane Doe",
  "accessConstraints": "restricted",
  "dates": {
    "revision": "2019-05-17T11:46:12Z"
  }
}
```

To remove the point of contact, the access constraint and the revision date, send:

```
{
  "pointOfContact": null,
  "accessConstraints": null,
  "dates": {
    "revision": null
  }
}
```

For arrays the complete array needs to be sent. To add a keyword to the example style metadata object, send:

```
{
  "keywords": [ "basemap", "TDS", "TDS 6.1", "OGC API", "new keyword" ]
}
```

To remove the "TDS" keyword, send:

```
{
  "keywords": [ "basemap", "TDS 6.1", "OGC API", "new keyword" ]
}
```

To remove the keywords, send:

```
{
  "keywords": null
}
```

The same applies to `stylesheets`, `layers` and `links`. To update these members, the complete new array value has to be sent.

Requirement 21	**/req/manage-styles/patch-style-md-error**
A	If the request does not conform to the requirements (e.g., the patch document is invalid) a response with status code `400` SHALL be returned.
B	If the style does not exist, a response with status code `404` SHALL be returned.
C	If the patch document appears to be valid, but the server is incapable of processing the request, a response with status code `422` SHALL be returned.
D	If the media type of the patch document is not supported by the API, a response with status code `415` and an `Accept-Patch` header with the supported media types SHALL be returned.

7.3. Requirements Class "Validation of styles"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/style-validation	
Target type	Web API
Dependency	Requirements Class "Manage styles"

7.3.1. Validate a style

Requirement 22	**/req/style-validation/input**

A	The server SHALL support a parameter with the name "validate" in POST requests to the path `/styles` and in PUT requests to the path `/styles/{styleId}` with the following schema:
	
	`name: validate`
	`in: query`
	`required: false`
	`style: form`
	`explode: false`
	`schema:`
	`  type: string`
	`  enum:`
	`  - yes`
	`  - no`
	`  - only`
	`  default: no`

Requirement 23	**/req/style-validation/output**
A	If the `validate` parameter has been provided in the request with the value 'yes', the server SHALL validate the submitted stylesheet for conformance with the style encoding. If an error is identified, a response with status code `400` shall be returned.
A	If the `validate` parameter has been provided in the request with the value 'only', the server SHALL validate the submitted stylesheet for conformance with the style encoding. If an error is identified, a response with status code `400` SHALL be returned. If no error is identified, a response with status code `204` SHALL be returned and no style SHALL be created or updated.

If no parameter `validate` is provided or the parameter has the value 'no', the standard response is returned (for a POST on `/styles` a `201` response with the `Location` header pointing to the new Style resource, for a PUT request on `/styles/{styleId}` a `204` response).

7.4. Requirements Class "Resources"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/resources	
Target type	Web API
Dependency	Requirements Class "Core"

7.4.1. Fetch resources

A GET request returns a list of resources that are currently available. The resources can be referenced from stylesheets. Resources in the Styles API are symbols, sprites and thumbnails.

For each resource the id and a link to the resource is provided.

NOTE	Testbed-15 required only support for a limited number of the resources. Therefore, the currently simple approach is sufficient, but in general the operation could support paging (using a parameter `limit` and links to the `next` page in responses).

Requirement 24	**/req/resources/resources-op**
A	The server SHALL support the HTTP GET operation at the path `/resources`.

Requirement 25	**/req/resources/resources-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `200`.

B	The content of that response SHALL be based upon the following OpenAPI 3.0 schema:
	`type: object`
	`required:`
	`  - resources`
	`properties:`
	`  resources:`
	`  type: array`
	`  items:`
	`  type: object`
	`  required:`
	`  - id`
	`  properties:`
	`  id:`
	`  type: string`
	`  link:`
	`  $ref:`
	`'http://schemas.opengis.net/ogcapi/features/part1/1.0/openapi/ogcapi-features-1.yaml#/components/schemas/link'`

C	The `resources` member SHALL include one item for each resource currently on the server.

D	The `id` member of each resource SHALL be unique.
E	Each resource SHALL have a link to the resource (link relation: `item`) with the `type` attribute stating the media type of the resource.

Example 6. JSON encoding of resources

```
{
  "resources": [
    {
      "id": "sprite.json",
      "link": {
        "href": "https://example.com/api/v1/resources/sprite.json",
        "type": "application/json",
        "rel": "item"
      }
    },
    {
      "id": "sprite.png",
      "link": {
        "href": "https://example.com/api/v1/resources/sprite.png",
        "type": "image/png",
        "rel": "item"
      }
    },
    {
      "id": "sprite.@2x.png",
      "link": {
        "href": "https://example.com/api/v1/resources/sprite.@2x.png",
        "type": "image/png",
        "rel": "item"
      }
    },
    {
      "id": "building.svg",
      "link": {
        "href": "https://example.com/api/v1/resources/building.svg",
        "type": "image/svg+xml",
        "rel": "item"
      }
    }
  ]
}
```

7.4.2. Fetch resource

A GET request returns a single resource.

Requirement 26	**/req/resources/resource-op**
A	The server SHALL support the HTTP GET operation at the path `/resources/{resourceId}` for each resource referenced from `/resources`.

Requirement 27	**/req/resources/resource-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `200`.
B	The content of that response SHALL conform to the media type stated in the `Content-Type` header.

7.5. Requirements Class "Manage resources"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/manage-resources	
Target type	Web API
Dependency	Requirements Class "Resources"
Dependency	RFC 7396 (JSON Merge Patch)

7.5.1. Create or replace a resource

This operation creates or replaces the resource with id `resourceId`.

Requirement 28	**/req/manage-resources/update-resource-op**
A	The server SHALL support the HTTP PUT operation at path `/resources/{resourceId}`.

Requirement 29	**/req/manage-resources/update-resource-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.
B	The resource SHALL be the content submitted in the request.

7.5.2. Delete a resource

This operation deletes the resource with the id `resourceId`. If no such resource exists, an error is returned.

Requirement 30	**/req/manage-resources/delete-resource-op**
A	The server SHALL support the HTTP DELETE operation at the path `/resources/{resourceId}`.

Requirement 31	**/req/manage-resources/delete-resource-success**
A	A successful execution of the operation SHALL be reported as a response with an HTTP status code `204`.

Requirement 32	**/req/manage-resources/delete-resource-error**
A	If the style does not exist, a response with status code `404` SHALL be returned.

7.6. Requirements Class "HTML"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/html	
Target type	Web API
Dependency	Requirements Class "Core"
Dependency	HTML

Requirement 33	**/req/html/get**
A	Every `200`-response of a GET operation of the server that supports the media type `application/json` SHALL support the media type `text/html`.

That is, all resources are expected to have a HTML representation except the stylesheets and the resources (symbols, etc.).

Requirement 34	**/req/html/content**

A	Every `200`-response of the server with the media type `text/html` SHALL be a HTML 5 document that includes the following information in the HTML body: • all information identified in the schemas of the Response Object in the HTML `<body>`, and • all links in HTML `<a>` elements in the HTML `<body>`.

7.7. Requirements Class "OGC SLD 1.0"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/sld-10	
Target type	Web API
Dependency	Requirements Class "Core"
Dependency	OGC 02-070, Styled Layer Descriptor, Version 1.0

Requirement 35	**/req/sld-10/media-type**
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL support the media type `application/vnd.ogc.sld+xml;version=1.0`.

Requirement 36	**/req/sld-10/content**
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL accept valid OGC SLD 1.0 documents without errors.

The list of operations in a server implementing all conformance classes of this draft specification is:

- POST `/styles`
- PUT `/styles/{styleId}`

7.8. Requirements Class "OGC SLD 1.1"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/sld-11	
Target type	Web API
Dependency	Requirements Class "Core"
Dependency	OGC 05-078r4, Styled Layer Descriptor, Version 1.1

Requirement 37	/req/sld-11/media-type
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL support the media type `application/vnd.ogc.sld+xml;version=1.1`.

Requirement 38	/req/sld-11/content
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL accept valid OGC SLD 1.1 documents without errors.

The list of operations in a server implementing all conformance classes of this draft specification is:

- POST `/styles`
- PUT `/styles/{styleId}`

7.9. Requirements Class "Mapbox Style"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/mapbox-style	
Target type	Web API
Dependency	Requirements Class "Core"
Dependency	Mapbox Style Specification, Version 8

Requirement 39	/req/mapbox-style/media-type
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL support the media type `application/vnd.mapbox.style+json`.

Requirement 40	/req/mapbox-style/content
A	Every POST or PUT operation of the server that accepts a stylesheet document as content SHALL accept valid Mapbox Style documents (version 8) without errors.

The list of operations in a server implementing all conformance classes of this draft specification is:

- POST `/styles`
- PUT `/styles/{styleId}`

Chapter 8. Extensions to the Collection resource

The previous clause specifies the Styles API. In order to support portrayal workflows, data APIs (supporting OGC API Features and/or Tiles) should provide additional information about the data to support styling.

This clause specifies the extensions to the Collection as additional requirements/conformance classes to OGC API Features.

NOTE	In the future, these classes could extend an OGC API Common requirements/conformance class that supports feature collections, but as no mature draft for OGC API Common exists, this document extends OGC API Features.

The extensions are the following:

- The feature collection (path `/collections/{collectionId}`) is extended by the set of applicable styles (member `styles`, same value as in `/styles` in the Styles API) and a default style (member `defaultStyle`, the style id).
- The PATCH operation on the same resource (path `/collections/{collectionId}`) is added. Only `styles` and `defaultStyle` may be updated.
- The `queryables` resource (path `/collections/{collectionId}/queryables`) has been added to support clients like visual style editors to construct expressions for selection criteria in queries on features in the collection.

NOTE	There is planned work on an extension for queryables for the OGC API Features standard and the draft OGC API Catalogues specification. The requirements for stating the queryables for the use by a visual style editor should be brought into this work activity. Once that extension is available, the requirements class for queryables can be dropped from this document.

This resulting Features API has the resources listed in the Table below.

Table 3. Overview of resources, applicable HTTP methods

Resource	Path	HTTP method	Changes
Landing page	`/`	GET	unchanged
Conformance declaration	`/conformance`	GET	unchanged, except for returning additional conformance classes
Feature collections	`/collections`	GET	unchanged

Resource	Path	HTTP method	Changes
Feature collection	`/collections/{collectionId}`	GET	include links to styles in the response, see Fetch styles associated with a collection
		PATCH	new, see Update styles associated with a collection
Queryables	`/collections/{collectionId}/queryables`	GET	new, see Fetch the queryable properties of the features in a collection
Features	`/collections/{collectionId}/items`	GET	unchanged
Feature	`/collections/{collectionId}/items/{featureId}`	GET	unchanged

The following is an example of the conformance declaration of a Styles API that implements all requirements classes except "html".

Example 7. Updated conformance declaration

```
{
  "conformsTo": [
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/core",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/oas30",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/html",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/geojson",
    "http://www.opengis.net/t15/opf-styles-1/1.0/conf/style-info",
    "http://www.opengis.net/t15/opf-styles-1/1.0/conf/queryables"
  ]
}
```

NOTE To support styling of coverage data, other additional capabilities in the relevant OGC API specifications supporting coverage data may be required, but have not been investigated by Testbed 15.

8.1. Requirements Class "Style information"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/style-info	
Target type	Web API
Dependency	OGC API - Features - Part 1: Core, conformance class "Core"
Dependency	OGC API - Features - Part 1: Core, conformance class "GeoJSON"
Dependency	RFC 7396 (JSON Merge Patch)

8.1.1. Fetch styles associated with a collection

The description of the collection includes additional information related to styles:

- The `styles` array lists styles that can be used to render features in this collection.
- The `defaultStyle` is the id of a recommended style to use for this collection.

Requirement 41	**/req/style-info/success**
A	A successful execution of the operation GET on `/collections/{collectionId}` SHALL include two members (with name `styles` and `defaultStyle`), if style information is available for the collection (for example, the style information may be set using a PATCH operation as described below).

B	The `styles` member SHALL be based on the following OpenAPI 3.0 schema:
	`type: array`
	`items:`
	`  type: object`
	`  nullable: true`
	`  required:`
	`  - id`
	`  - links`
	`  properties:`
	`  id:`
	`  type: string`
	`  nullable: true`
	`  title:`
	`  type: string`
	`  nullable: true`
	`  links:`
	`  type: array`
	`  nullable: true`
	`  minItems: 1`
	`  items:`
	`  $ref:`
	`'http://schemas.opengis.net/ogcapi/features/part1/1.0/openapi/ogcapi-features-1.yaml#/components/schemas/link'`

C	The `defaultStyle` member SHALL be based on the following OpenAPI 3.0 schema: `type: string`

Recommendation 5	/rec/style-info/consistency
A	Each style SHOULD have a link for each style encoding supported for the style (link relation: `stylesheet`) with the `type` attribute stating the media type of the style encoding.
B	Each style SHOULD have a link to the style metadata (link relation: `describedBy`) with the `type` attribute stating the media type of the metadata encoding.
C	The value of the `defaultStyle` member SHOULD be an `id` included in the `styles` array.

Example 8. JSON encoding of style information

This example links to two styles, "night" and "topographic". Each is available in two style encodings. The "topographic" style is identified as the recommended default style for rendering the features on a map.

```
{
  "id": "address",
  "title": "address",
  ...
  },
  "itemType": "feature",
  "styles": [
    {
      "id": "night",
      "title": "Topographic night style",
      "links": [
        {
          "href": "https://example.com/api/1.0/styles/night?f=mapbox",
          "type": "application/vnd.mapbox.style+json",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/1.0/styles/night?f=sld10",
          "type": "application/vnd.ogc.sld+xml;version=1.0",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/1.0/styles/night/metadata?f=json",
          "type": "application/json",
          "rel": "describedBy"
        }
      ]
    },
    {
```

```
      "id": "topographic",
      "title": "Regular topographic style",
      "links": [
        {
          "href": "https://example.com/api/1.0/styles/topographic?f=mapbox",
          "type": "application/vnd.mapbox.style+json",
          "rel": "stylesheet"
        },
        {
          "href": "https://example.com/api/1.0/styles/topographic?f=sld10",
          "type": "application/vnd.ogc.sld+xml;version=1.0",
          "rel": "stylesheet"
        },
        {
          "href":
"https://example.com/api/1.0/styles/topographic/metadata?f=json",
          "type": "application/json",
          "rel": "describedBy"
        }
      ]
    }
  ],
  "defaultStyle": "topographic"
}
```

8.1.2. Update styles associated with a collection

In the previous section the additional style information for each feature collection is described. This operation can be used to update the style information.

The PATCH request updates the metadata of the style with the id `styleId`. If no such style exists, an error is returned.

Requirement 42	**/req/style-info/patch-style-info-op**
A	The server SHALL support the HTTP PATCH operation at path `/collection/{collectionId}`.

B	The server SHALL accept content based on the following OpenAPI 3.0 schema:
	
	`type: object`
	`properties:`
	`  styles:`
	`  type: array`
	`  nullable: true`
	`  items:`
	`  type: object`
	`  nullable: true`
	`  required:`
	`  - id`
	`  - links`
	`  properties:`
	`  id:`
	`  type: string`
	`  nullable: true`
	`  title:`
	`  type: string`
	`  nullable: true`
	`  links:`
	`  type: array`
	`  nullable: true`
	`  minItems: 1`
	`  items:`
	`  $ref:`
	`'http://schemas.opengis.net/ogcapi/features/part1/1.0/openapi/ogcapi-features-1.yaml#/components/schemas/link'`
	`  defaultStyle:`
	`  type: string`
	`  nullable: true`

The only members that may be updated at this time are `styles` and `defaultStyle`. This specification does not specify how servers have to respond to additional content in the request content.

Requirement 43	**/req/style-info/patch-style-info-success**
A	A successful execution of the operation SHALL be reported as a response with a HTTP status code `204`.
B	The style information SHALL be updated by the content submitted in the request as specified by RFC 7396 (JSON Merge Patch).

In other words, a GET request to `/collections/{collectionId}` after the PATCH operation has to

reflect the updated style information.

See the explanations in the operation to update style metadata for more information about about RFC 7396 (JSON Merge Patch).

Some examples requests and their effect on the Collection resource:

To add or update the default style, just send:

```
{
  "defaultStyle": "night"
}
```

To remove the default style, send:

```
{
  "defaultStyle": null
}
```

For arrays the complete array needs to be sent. I.e., to update the list of styles, send the complete new array value.

To remove all styles, send:

```
{
  "styles": null
}
```

Requirement 44	**/req/style-info/patch-style-info-error**
A	If the request does not conform to the requirements (e.g., the patch document is invalid or includes additional members) a response with status code `400` SHALL be returned.
B	If the collection does not exist, a response with status code `404` SHALL be returned.
C	If the patch document appears to be valid, but the server is incapable of processing the request, a response with status code `422` SHALL be returned.
D	If the media type of the patch document is not supported by the API, a response with status code `415` and an `Accept-Patch` header with the supported media types SHALL be returned.

8.2. Requirements Class "Queryables"

Requirements Class	
http://www.opengis.net/t15/opf-styles-1/1.0/req/queryables	
Target type	Web API
Dependency	OGC API - Features - Part 1: Core, conformance class "Core"

8.2.1. Fetch the queryable properties of the features in a collection

This operation returns the list of queryable properties that can be used to filter features in a collection and supports clients in constructing expressions for selection criteria in queries on features in the collection.

The response is an object with a member `queryables`, which contains an array with a description of the queryable properties of the feature collection. "Queryable" means that the property may be used in query expressions, such as in a query extension to OGC API - Features or as part of a selection criteria in an OGC SLD/SE or Mapbox styling rule.

Often the list of queryables for a collection will be a subset of all available properties in the features and be restricted to those properties that are, for example, indexed in the backend datastore to support performant queries.

For each queryable property the following information is or may be provided:

- `id` (required) - the property name for use in expressions.
- `type` (required) - the data type of the property, one of
  - `string`
  - `uri`
  - `enum`
  - `number`
  - `integer`
  - `date`
  - `dateTime`
  - `boolean`
- `description` (optional) - a description of the property.
- `required` (optional) - indicator whether the property is always present in features.
- `mediaTypes` (optional) - in general, the representation of the queryables is meant to be independent of the feature encoding. However, this is not always the case. For example, length restrictions or namespace prefixes may result in different property identifiers for the same property. To support this, the definition of a queryable may be restricted to one or more feature encodings (media types).
- `pattern` (optional, only for "string" and "uri") - a regular expression to validate the values of the property.
- `values` (required, only for "enum") - an array of valid values of the property.

- `range` (optional, only for "number", "integer", "date" and "dateTime") - the range of valid values expressed as an array with two items. Open ranges can be expressed using null for the minimum or maximum value.

Note that this is not about providing a schema for the features in the collection. A schema provides a complete syntactic definition of a specific feature encoding, typically for validation purposes. Schema languages like XML Schema or JSON Schema are much richer and support more complex syntactic rules, but are also more complex to parse.

Requirement 45	**/req/queryables/op**
A	The server SHALL support the HTTP GET operation at the path `/collection/{collectionId}/queryables` for each collection.

Requirement 46	**/req/queryables/success**
A	A successful execution of the operation SHALL be reported as a response with a HTTP status code `200`.

B	The content of that response SHALL be based upon the OpenAPI 3.0 schema component "queryables", if the `itemType` of the collection is `feature`:
	
	queryables
	
	`type: object`
	`required:`
	`  - queryables`
	`properties:`
	`  queryables:`
	`  type: array`
	`  nullable: true`
	`  items:`
	`  oneOf:`
	`  - $ref: 'queryable-string'`
	`  - $ref: 'queryable-enum'`
	`  - $ref: 'queryable-number'`
	`  - $ref: 'queryable-boolean'`
	`  - $ref: 'queryable-date'`
	`  - $ref: 'queryable-dateTime'`
	
	queryable
	
	`type: object`
	`nullable: true`
	`required:`
	`  - id`
	`  - type`
	`properties:`
	`  id:`
	`  type: string`
	`  nullable: true`
	`  description: |-`
	`  the property name for use in expressions`
	`  title:`
	`  type: string`
	`  nullable: true`
	`  description: |-`
	`  the title of the property for presentation to a`
	`  human user`
	`  description:`
	`  type: string`
	`  nullable: true`
	`  description: |-`
	`  a description of the property`
	`  required:`
	`  type: boolean`
	`  nullable: true`
	`  default: false`

C	The `id` member of each queryable SHALL be unique.

Note that this requirements class does not specify any requirements on collections that are not feature collections.

Example 9. JSON encoding of queryables

```
{
  "queryables": [
    {
      "id": "name",
      "description": "the name of the vegetation area",
      "required": true,
      "type": "string",
      "example": "[A-Z0-9]{5}"
    },
    {
      "id": "type",
      "description": "the dominant characteristic of the vegetation area",
      "type": "enum",
      "values": [
        "grassland",
        "forest",
        "farmland"
      ]
    },
    {
      "id": "count",
      "description": "the number of cattle",
      "type": "integer",
      "range": [
        0,
        null
      ]
    },
    {
      "id": "fenced",
      "description": "indicator whether the area is walled or fenced",
      "type": "boolean"
    },
    {
      "id": "inspectionDate",
      "description": "the date of the last inspection",
      "type": "date",
      "range": [
        "2010-01-01",
        null
      ]
    },
```

```
    {
      "id": "lastUpdate",
      "description": "the date of the last update of the feature",
      "type": "dateTime",
      "range": [
        "2018-01-01T00:00:00Z",
        null
      ]
    }
  ]
}
```

```
        a regular expression to validate the values
        of the property
```

queryable-enum

```
allOf:
- $ref: 'queryable'
- type: object
  nullable: true
  required:
    - values
  properties:
    values:
      type: array
      nullable: true
      description: |-
        the list of values of the property
      items:
        type: string
```

queryable-number

```
allOf:
- $ref: 'queryable'
- type: object
  nullable: true
  properties:
    range:
      type: array
      nullable: true
      minItems: 2
      maxItems: 2
      items:
        type: number
        nullable: true
      description: |-
        a range of valid values; open range can be
        expressed using `null`
```

Chapter 9. Media Types

`application/json` is the JSON media type used for all content except the stylesheets and the symbol resources.

`text/html` is the HTML media type for all "web pages" provided by the API.

No media types have been formally registered with IANA for the style encodings (Mapbox Styles and OGC SLD). Temporary media types in the vnd-branch as specified below are used for now.

9.1. application/vnd.mapbox.style+json

- Type name: application
- Subtype name: vnd.mapbox.style+json
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: See RFC 8259, The JavaScript Object Notation (JSON) Data Interchange Format
- Security considerations: See Section 12 of RFC 8259
- Interoperability considerations: n/a
- Published specification: Mapbox Style Specification, Version 8
- Applications that use this media type: Geographic information systems (GIS)
- Additional information:
  - Deprecated alias names for this type: n/a
  - Magic number(s): n/a
  - File extension(s): .json
  - Macintosh file type code(s): n/a
- Person to contact for further information: n/a
- Intended usage: COMMON
- Restrictions on usage: none
- Author: n/a
- Change controller: Mapbox

9.2. application/vnd.ogc.sld+xml

- Type name: application
- Subtype name: vnd.ogc.sld+xml
- Required parameters: n/a
- Optional parameters:

- "charset": See Section 3 of RFC 7303.
- "version": If provided, this parameter indicates the major and the first minor version number of the SLD version used in the document, e.g. "1.1".
  - Syntax: `version = 1*DIGIT "." 1*DIGIT`. The first group of digits is the major version number, the second group is the minor version number.
  - The parameter can be used to provide protocol-specific operations, such as version-based content negotiation in HTTP. The parameter is a hint, if used in HTTP content negotiation. I.e., client implementations should be prepared to receive content in a different version than requested and server implementations should honor the version parameter during content negotiation, if possible.

- Encoding considerations: Same as application/xml - see section 9.1 of RFC 7303.
- Security considerations: OGC SLD is a generic XML grammar, but application designers must not assume that it provides generic protection against security threats. RFC 7303, Section 10, discusses security concerns for generic XML, which are also applicable to SLD. Xlink references inSLD documents may cause arbitrary URIs to be dereferenced. In this case, the security issues of RFC 3986, section 7, should be considered. SLD documents do not contain active or executable content.
- Interoperability considerations: n/a
- Published specification: OGC: OGC 02-070, Styled Layer Descriptor, Version 1.0 (2002) and OGC: OGC 05-078r4, Styled Layer Descriptor, Version 1.1 (2007)
- Applications that use this media type: Geographic information systems (GIS)
- Additional information:
  - Deprecated alias names for this type: n/a
  - Magic number(s): n/a
  - File extension(s): .sld, .xml
  - Macintosh file type code(s): n/a
- Person to contact for further information: n/a
- Intended usage: COMMON
- Restrictions on usage: none
- Author: n/a
- Change controller: OGC

queryable-boolean

```
allOf:
  - $ref: '#/components/schemas/queryable'
```

queryable-date

```
allOf:
  - $ref: 'queryable'
  - type: object
    nullable: true
    properties:
      range:
        type: array
        nullable: true
        minItems: 2
        maxItems: 2
        items:
          type: string
          format: date
          nullable: true
        description: |-
          a range of valid values; open range can be
          expressed using `null`
```

queryable-dateTime

```
queryable-dateTime:
  allOf:
    - $ref: 'queryable'
    - type: object
      nullable: true
      properties:
        range:
          type: array
          nullable: true
          minItems: 2
          maxItems: 2
          items:
            type: string
            format: date-time
            nullable: true
          description: |-
            a range of valid values; open range can be
            expressed using `null`
```

Annex A: Conformance Class Abstract Test Suite (Normative)

A.1. Conformance Class "Core"

A.1.1. Test Case 1

Test id:	/conf/core/1
Requirement(s):	/req/core/styles-op, /req/core/styles-success, /req/core/style-op, /req/core/style-success, /req/core/style-md-op, /req/core/style-md-success
Test purpose:	Verify that the style resources can be fetched.

Test method:	1. Issue an HTTP GET request to the path `/styles` with header `Accept: application/json`. 2. Validate that the response has a status code 200. 3. Validate the contents of the returned document against the schema in /req/core/styles-success, item B. 4. Verify that each style id `#/styles/{i}/id` (where `{i}` is the index of the style in the array) is unique. 5. Verify that each style has at least one link with `rel=stylesheet`. 6. Verify that for each link with `rel=stylesheet` that the `href` value links to a resource at the path `/styles/{styleId}` where `{styleId}` is the `id` member of the style. 7. For each link with `rel=stylesheet` send a GET request to the URI in `href` using the value of `type` in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). If the response has as `Content-Language` header, try to verify that linguistic text in the response in the stated language. 8. Verify that each style has at least one link with `rel=describedBy`. 9. Verify that for each link with `rel=describedBy` that the `href` value links to a resource at the path `/styles/{styleId}/metadata` where `{styleId}` is the `id` member of the style. 10. For each link with `rel=describedBy` send a GET request to the URI in `href` using the value of `type` in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). If the response has as `Content-Language` header, try to verify that linguistic text in the response in the stated language. Validate the contents of the returned document against the schema in /req/core/style-md-success, item B.

A.1.2. Test Case 2

Test id:	/conf/core/2
Requirement(s):	/req/core/styles-success
Test purpose:	Verify that `/styles` list all styles on the server.
Test method:	Use `manage-styles` operations or some other way to add and delete styles. Issue an HTTP GET request to the path `/styles` with header `Accept: application/json` before and after each change and verify that added styles are included and deleted styles have been removed. If no mechanism for adding/deleting styles is available, skip the test.

A.2. Conformance Class "Manage styles"

A.2.1. Test Case 1

Test id:	/conf/manage-styles/1
Requirement(s):	/req/manage-styles/create-style-op, /req/manage-styles/create-style-success
Test purpose:	Verify that styles can be created using POST requests
Test method:	1. Send a POST request to `/styles` with a valid stylesheet in one of the supported style encodings (inspect the API definition of the path) with the `Content-Type` header sent to the media type of the style encoding. 2. Validate that the response has an HTTP status code 201 and a header `Location` with a URI to path `/styles/{styleId}`. 3. Send a GET request to the URI in `Location` using the media type of the submitted stylesheet in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). 4. Send a GET request to the URI in `Location` with `/metadata` appended to the path. Use `application/json` in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). Validate the contents of the returned document against the schema in /req/core/style-md-success, item B.

A.2.2. Test Case 2

Test id:	/conf/manage-styles/2
Requirement(s):	/req/manage-styles/create-style-error
Test purpose:	Verify that POSTing invalid requests returns an error
Test method:	1. Send a POST request to `/styles` with empty payload and verify that the response has an HTTP status code 400. 2. Send a POST request to `/styles` with payload in an unsupported media type in the header `Content-Type` (inspect the API definition of the path) and verify that the response has an HTTP status code 400.

A.2.3. Test Case 3

Test id:	/conf/manage-styles/3

Requirement(s):	/req/manage-styles/update-style-op, /req/manage-styles/update-style-success
Test purpose:	Verify that styles can be created or updated using PUT requests
Test method:	1. Send a PUT request to `/styles/{styleId}` with a valid stylesheet in one of the supported style encodings (inspect the API definition of the path) with the `Content-Type` header sent to the media type of the style encoding. 2. Validate that the response has an HTTP status code 204. 3. Send a GET request to `/styles/{styleId}` using the media type of the submitted stylesheet in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). 4. Send a GET request to the URI in `Location` with `/metadata` appended to the path. Use `application/json` in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). Validate the contents of the returned document against the schema in /req/core/style-md-success, item B.

A.2.4. Test Case 4

Test id:	/conf/manage-styles/4
Requirement(s):	/req/manage-styles/update-style-error
Test purpose:	Verify that PUTting invalid requests returns an error
Test method:	1. Send a PUT request to `/styles/{styleId}` with empty payload and verify that the response has an HTTP status code 400. 2. Send a POST request to `/styles/{styleId}` with payload in an unsupported media type in the header `Content-Type` (inspect the API definition of the path) and verify that the response has an HTTP status code 400.

A.2.5. Test Case 5

Test id:	/conf/manage-styles/5
Requirement(s):	/req/manage-styles/delete-style-op, /req/manage-styles/delete-style-success
Test purpose:	Verify that styles can be deleted using DELETE requests

Test method:	1. Send a DELETE request to `/styles/{styleId}` where `{styleId}` is one of the style identifiers in the Styles resource. 2. Validate that the response has an HTTP status code 204. 3. Send a GET request to `/styles/{styleId}`. Verify that the response has a status code 404. 4. Send a GET request to `/styles/{styleId}/metadata`. Verify that the response has a status code 404.

A.2.6. Test Case 6

Test id:	/conf/manage-styles/6
Requirement(s):	/req/manage-styles/delete-style-error
Test purpose:	Verify that deleting a non-existent style returns an error
Test method:	1. Send a DELETE request to `/styles/{styleId}` where `{styleId}` is NOT one of the style identifiers in the Styles resource. 2. Validate that the response has an HTTP status code 404.

A.2.7. Test Case 7

Test id:	/conf/manage-styles/7
Requirement(s):	/req/manage-styles/update-style-md-op, /req/manage-styles/update-style-md-success
Test purpose:	Verify that style metadata can be updated using PUT requests
Test method:	1. Send a PUT request to `/styles/{styleId}/metadata` with a valid style metadata document (validate the metadata document against the schema in /req/core/style-md-success, item B) with the `Content-Type` header set to `application/json`. 2. Validate that the response has an HTTP status code 204. 3. Send a GET request to `/styles/{styleId}/metadata` with an `Accept: application/json` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). Verify that the retrieved document has the same content as the submitted document (formatting changes are allowed).

A.2.8. Test Case 8

Test id:	/conf/manage-styles/8
Requirement(s):	/req/manage-styles/update-style-md-error
Test purpose:	Verify that sending a metadata PUT request to a non-existing style returns an error
Test method:	1. Send a PUT request to `/styles/{styleId}` where `{styleId}` is NOT one of the style identifiers in the Styles resource. 2. Validate that the response has an HTTP status code 404.

A.2.9. Test Case 9

Test id:	/conf/manage-styles/9
Requirement(s):	/req/manage-styles/patch-style-md-op, /req/manage-styles/patch-style-md-success, /req/manage-styles/patch-style-md-error
Test purpose:	Verify that style metadata can be updated using PATCH requests
Test method:	1. Send a PATCH request to `/styles/{styleId}/metadata` with a valid style metadata document (validate the metadata document against the schema in /req/core/style-md-success, item B) with the `Content-Type` header set to `application/json`. 2. Validate that the response has an HTTP status code 204 or 422. 3. If the status code is 204, send a GET request to `/styles/{styleId}/metadata` with an `Accept: application/json` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). Verify that the retrieved document includes all the changes in the patch document (formatting changes are allowed). For example, retrieve the metadata document before the PATCH request and execute the patch locally and then compare the document with the API response after the PATCH.

A.2.10. Test Case 10

Test id:	/conf/manage-styles/10
Requirement(s):	/req/manage-styles/patch-style-md-error

Test purpose:	Verify that sending invalid PATCH requests returns an error
Test method:	1. Send a PATCH request to `/styles/{styleId}/metadata` where `{styleId}` is NOT one of the style identifiers in the Styles resource. Validate that the response has an HTTP status code 404. 2. Send a PATCH request to `/styles/{styleId}/metadata` with an invalid style metadata document (validating the metadata document against the schema in /req/core/style-md-success, item B, returns an error) with the `Content-Type` header set to `application/json`. Validate that the response has an HTTP status code 400. 3. Send a PATCH request to `/styles/{styleId}/metadata` with empty payload and verify that the response has an HTTP status code 400. 4. Send a PATCH request to `/styles/{styleId}/metadata` with payload in an unsupported media type in the header `Content-Type` (inspect the API definition of the path) and verify that the response has an HTTP status code 415 and an `Accept-Patch` header with the supported media types as stated in the API definition.

A.3. Conformance Class "Style validation"

A.3.1. Test Case 1

Test id:	/conf/style-validation/1
Requirement(s):	/req/style-validation/input, /req/style-validation/output
Test purpose:	Verify that styles are properly validated, if requested

Test method:	1. Repeat test case /conf/manage-styles/1, but with a query parameter `validate=true` in the POST request URI. 2. Repeat test case /conf/manage-styles/1, but with a query parameter `validate=no` in the POST request URI. 3. Send a POST request to `/styles?validate=true` with an invalid stylesheet and verify that the response has an HTTP status code 400. 4. Send a POST request to `/styles?validate=only` with the same stylesheet and verify that the response has an HTTP status code 400. 5. Send a POST request to `/styles?validate=only` with a valid stylesheet and verify that the response has an HTTP status code 204. 6. Repeat test case /conf/manage-styles/3, but with a query parameter `validate=true` in the PUT request URI. 7. Repeat test case /conf/manage-styles/3, but with a query parameter `validate=no` in the PUT request URI. 8. Send a PUT request to `/styles/{styleId}?validate=true` with an invalid stylesheet and verify that the response has an HTTP status code 400. 9. Send a PUT request to `/styles/{styleId}?validate=only` with the same stylesheet and verify that the response has an HTTP status code 400. 10. Send a PUT request to `/styles/{styleId}?validate=only` with a valid stylesheet and verify that the response has an HTTP status code 204.

A.4. Conformance Class "Resources"

A.4.1. Test Case 1

Test id:	/conf/resources/1
Requirement(s):	/req/resources/resources-op, /req/resources/resources-success, /req/resources/resource-op, /req/resources/resource-success
Test purpose:	Verify that the resources can be fetched.

Test method:	1. Issue an HTTP GET request to the path `/resources` with header `Accept: application/json`. 2. Validate that the response has a status code 200. 3. Validate the contents of the returned document against the schema in /req/core/resources-success, item B. 4. Verify that each resources id `#/resources/{i}/id` (where `{i}` is the index of the resources in the array) is unique. 5. Verify that each resource has a link with `rel=item`. 6. Verify that for each link with `rel=item` that the `href` value links to a resource at the path `/resources/{resourceId}` where `{resourceId}` is the `id` member of the resource. 7. For each link with `rel=item` send a GET request to the URI in `href` using the value of `type` in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`).

A.4.2. Test Case 2

Test id:	/conf/resources/2
Requirement(s):	/req/resources/resources-success
Test purpose:	Verify that `/resources` list all resources on the server.
Test method:	Use `manage-resources` operations or some other way to add and delete resources. Issue an HTTP GET request to the path `/resources` with header `Accept: application/json` before and after each change and verify that added resources are included and deleted resources have been removed. If no mechanism for adding/deleting resources is available, skip the test.

A.5. Conformance Class "Manage Resources"

A.5.1. Test Case 1

Test id:	/conf/manage-resources/1
Requirement(s):	/req/manage-resources/update-resources-op, /req/manage-resources/update-resources-success
Test purpose:	Verify that resources can be created or updated using PUT requests

Test method:	1. Send a PUT request to `/resources/{resourceId}`. 2. Validate that the response has an HTTP status code 204. 3. Send a GET request to `/resources/{resourceId}` using the media type of the submitted resource in the `Accept` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`).

A.5.2. Test Case 2

Test id:	/conf/manage-styles/2
Requirement(s):	/req/manage-resources/delete-resource-op, /req/manage-resources/delete-resource-success
Test purpose:	Verify that resources can be deleted using DELETE requests
Test method:	1. Send a DELETE request to `/resources/{resourceId}` where `{resourceId}` is one of the resource identifiers in the Resources resource. 2. Validate that the response has an HTTP status code 204. 3. Send a GET request to `/resources/{resourceId}`. Verify that the response has a status code 404.

A.5.3. Test Case 3

Test id:	/conf/manage-styles/6
Requirement(s):	/req/manage-resources/delete-resource-error
Test purpose:	Verify that deleting a non-existent resource returns an error
Test method:	1. Send a DELETE request to `/resources/{resourceId}` where `{resourceId}` is NOT one of the resource identifiers in the Resources resource. 2. Validate that the response has an HTTP status code 404.

A.6. Conformance Class "HTML"

A.6.1. Test Case 1

Test id:	/conf/html/1
Requirement(s):	/req/html/get, /req/html/content

Test purpose:	Verify that all resources support HTML
Test method:	1. Issue HTTP GET requests to the path `/styles` once with header `Accept: application/json` and once with `Accept: text/html`. Verify that both responses have a status code 200 and the requested content type (header `Content-Type`). Verify to the extent possible that the HTML response document is a HTML 5 document where all information identified in the JSON response is included in the HTML <body>, and all links are included in HTML <a> elements in the HTML <body>. 2. For each link with `rel=describedBy` in the JSON response document send again two GET requests to the URI in `href` using the headers `Accept: application/json` and `Accept: text/html` respectively. Verify that both responses have a status code 200 and the requested content type (header `Content-Type`). Verify to the extent possible that the HTML response document is a HTML 5 document where all information identified in the JSON response is included in the HTML <body>, and all links are included in HTML <a> elements in the HTML <body>.

A.7. Conformance Class "Mapbox Style"

A.7.1. Test Case 1

Test id:	/conf/mapbox-style/1
Requirement(s):	/req/mapbox-style/media-type, /req/mapbox-style/content
Test purpose:	Verify that Mapbox style is supported as a style encoding
Test method:	If the API supports the conformance classes "Manage styles" or "Style validation", execute all test cases of the supported conformance classes using stylesheets that are Mapbox Style documents (version 8) using the media type `application/vnd.mapbox.style+json`. Otherwise skip the test.

A.8. Conformance Class "SLD 1.0"

A.8.1. Test Case 1

Test id:	/conf/sld-10/1
Requirement(s):	/req/sld-10/media-type, /req/sld-10/content

Test purpose:	Verify that SLD 1.0 is supported as a style encoding
Test method:	If the API supports the conformance classes "Manage styles" or "Style validation", execute all test cases of the supported conformance classes using stylesheets that are OGC SLD 1.0 documents using the media type `application/vnd.ogc.sld+xml;version=1.0`. Otherwise skip the test.

A.9. Conformance Class "SLD 1.1"

A.9.1. Test Case 1

Test id:	/conf/sld-11/1
Requirement(s):	/req/sld-11/media-type, /req/sld-11/content
Test purpose:	Verify that SLD 1.1 is supported as a style encoding
Test method:	If the API supports the conformance classes "Manage styles" or "Style validation", execute all test cases of the supported conformance classes using stylesheets that are OGC SLD 1.1 documents using the media type `application/vnd.ogc.sld+xml;version=1.0`. Otherwise skip the test.

A.10. Conformance Class "Style information"

A.10.1. Test Case 1

Test id:	/conf/style-info/9
Requirement(s):	/req/style-info/patch-style-info-op, /req/style-info/patch-style-info-success, /req/style-info/success, /req/style-info/patch-style-info-error
Test purpose:	Verify that style information can be updated using PATCH requests

Test method:	1. Send a PATCH request to `/collection/{collectionId}` with a valid document (validate the document against the schema in /req/style-info/patch-style-info-op, item B) with the `Content-Type` header set to `application/json` for each collection listed in `/collections`. 2. Validate that the response has an HTTP status code 204 o 422. 3. If the status code is 204, send a GET request to `/collection/{collectionId}` with an `Accept: application/json` header. Verify that the response has a status code 200 and the requested content type (header `Content-Type`). Verify that the retrieved document includes all the changes in the patch document (formatting changes are allowed). For example, retrieve the collection document before the PATCH request and execute the patch locally and then compare the document with the API response after the PATCH.

A.10.2. Test Case 2

Test id:	/conf/style-info/2
Requirement(s):	/req/style-info/patch-style-info-error
Test purpose:	Verify that sending invalid PATCH requests returns an error
Test method:	1. Send a PATCH request to `/collection/{collectionId}` where `{collectionId}` is NOT one of the collection identifiers in the Collections resource. Validate that the response has an HTTP status code 404. 2. Send a PATCH request to `/collection/{collectionId}` with an invalid patch document (validating the metadata document against the schema in /req/style-info/patch-style-info-op, item B, returns an error) with the `Content-Type` header set to `application/json`. Validate that the response has an HTTP status code 400. 3. Send a PATCH request to `/collection/{collectionId}` with empty payload and verify that the response has an HTTP status code 400. 4. Send a PATCH request to `/collection/{collectionId}` with payload in an unsupported media type in the header `Content-Type` (inspect the API definition of the path) and verify that the response has an HTTP status code 415 and an `Accept-Patch` header with the supported media types as stated in the API definition.

A.11. Conformance Class "Queryables"

A.11.1. Test Case 1

Test id:	/conf/queryables/1
Requirement(s):	/req/queryables/op, /req/queryables/success
Test purpose:	Verify that the queryables can be fetched.
Test method:	1. Issue an HTTP GET request to the path `/collection/{collectionId}/queryables` with header `Accept: application/json` for each collection listed in `/collections`. 2. Validate that the response has a status code 200. 3. Validate the contents of the returned document against the schema in /req/queryables/success, item B, if the `itemType` is `feature`. 4. Verify that each queryable id `#/queryables/{i}/id` (where `{i}` is the index of the queryable in the array) is unique.

Annex B: Revision History

Date	Editor	Release	Primary clauses modified	Description
April 11, 2019	C. Portele	0.1	all	initial version
July 28, 2019	C. Portele	0.2	all	transfer content from SwaggerHub to ER
August 6, 2019	C. Portele	0.3	all	transfer content from SwaggerHub to ER, part 2
August 28, 2019	C. Portele	0.4	all	added link to GeoPackage content, update summary
September 24, 2019	C. Portele	0.5	all	changed from ER to Draft Specification, edits based on the review by Gobe Hobana
October 1, 2019	C. Portele	0.6 (19-010)	all	edits based on the review by Carl Reed
October 3, 2019	C. Portele	0.7	all	document type changed back to ER (using the document template for standards)
October 24, 2019	C. Portele	0.8 (19-010r1)	all	submission to TC
October 30, 2019	C. Portele	0.9 (19-010r2)	all	comments by Matt Sorensen, other discussions from TIEs, re-submit to TC
December 4, 2019	C. Reed	0.95	Various	FInal small edits prior to publication.

Annex C: Bibliography

1. IETF: RFC 7946 - The GeoJSON Format
2. OGC: OGC Testbed-15: Encoding and Metadata Conceptual Model for Styles Engineering Report. OGC 19-023, Open Geospatial Consortium (2019)
3. Mapbox: Mapbox Style Specification, Version 8