OGC® DOCUMENT: 21-017R1

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t17-D008

Open Geospatial Consortium

OGC TESTBED-17: OGC FEATURES AND GEOMETRIES JSON ENGINEERING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2021-11-19
Approval Date: 2021-12-09
Publication Date: 2022-02-08
Editor: Clemens Portele

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.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Copyright notice

Note

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 standard set forth in this document, and to provide supporting documentation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

The OGC Testbed-17 Features and Geometries JSON task investigated proposals for how feature data could be encoded in JSON so that:

- Different Coordinate Reference Systems (CRS) are supported and
- Communities can build and formally specify profiles of the fully CRS-enabled JSON with limited sets of supported geometry types and with clear constraints for feature type definitions.

GeoJSON, a standard of the Internet Engineering Task Force (IETF), was used as a starting point.

This Engineering Report (ER) captures the results and discussions, including material that was submitted to the OGC Features and Geometries JSON Standards Working Group.

II EXECUTIVE SUMMARY

JavaScript Object Notation (JSON) is a popular encoding format for geospatial data. The lightweight, simple syntax, and clear human and machine readability of JSON appeals to developers. GeoJSON has become a very popular encoding and is supported in most deployments of APIs implementing OGC API Features. However, GeoJSON has limitations that prevent or limit its use in some cases, e.g., if other coordinates should be in a projected coordinate reference system.

To support additional use cases, in 2021 the OGC formed a new Standards Working Group (SWG) to develop an OGC Features and Geometries JSON standard (JSON-FG).

As identified in the SWG charter, the standard will:

- Include the ability to use Coordinate Reference Systems (CRSs) other than WGS84;
- Support the use of non-Euclidean metrics, in particular ellipsoidal metrics;
- Support solids and multi-solids as geometry types; and
- Provide guidance on how to represent feature properties, e.g., including temporal properties.

JSON Schema has been identified as the mechanism to formally specify the syntax of JSON-FG.

Given the popularity of GeoJSON, the participants determined that a design goal is that JSON-FG will be specified as a superset of GeoJSON. Valid JSON-FG instances are also valid GeoJSON instances.

In order to support the work of the SWG, Testbed-17 included a task to:

- Develop specification proposals for the SWG;
- Engage with the SWG in discussions; and
- Implement and test draft JSON-FG specifications in three server and three client implementations.

The general JSON-FG design pattern is that information that can be represented as GeoJSON is encoded as GeoJSON. Additional information is mainly encoded in additional top-level members of GeoJSON objects. The members use keys that do not conflict with GeoJSON including the obsolete version that pre-dates the IETF standard. GeoJSON clients will be able to parse and understand all aspects that are specified by GeoJSON, JSON-FG clients will also parse and understand the additional capabilities.

The following extensions were discussed and developed during the testbed:

- Identifying the feature type(s)
- Identifying the schema(s)
- Encoding the primary temporal extent
- Encoding the primary spatial geometry
- Encoding of reference systems
- Relationships and links
- Use in offline containers

Of these, the first five resulted in requirements and normative statements.

For the topic “relationships and links”, the participants identified and documented potential JSON-FG encoding patterns. However, the participants agreed that depending on the data and how the data is expected to be used, the preferences of data publishers for one or the other pattern will vary.

The work on “offline containers” is out-of-scope of the SWG charter and was a Testbed-only activity.

All Technology Integration Experiments (TIEs) were completed successfully. Problems identified during initial experiments were usually a result of simple errors or temporary unavailability of server deployments.

Most of the JSON-FG extensions to GeoJSON were tested and proved useful for clients processing the JSON feature data. The experiences are consistent with the feedback from the

Aviation task in Testbed-17 on the benefits of using JSON-FG as a feature encoding in the aviation domain [4].

The experiences support the current JSON-FG approach that:

- Extends GeoJSON (every JSON-FG document is a valid GeoJSON document);
- Focusses on minimal extensions to GeoJSON that are useful in many contexts relevant to OGC members and avoid edge cases;
- Specifies the extensions as additional top-level JSON members (do not add constraints on “properties” or any other GeoJSON member); and
- Specifies the extensions in a modular way, so that implementations can pick and choose the capabilities that they need.

Extending existing GeoJSON writers and readers with the additional JSON members was in general straightforward in the six software components implementing JSON-FG. However, more testing and developer feedback will be needed to mature the draft JSON-FG specification.

There are also open issues that should be discussed in the Features and Geometries JSON SWG and, if possible, future Innovation Program initiatives, including, but not limited to the following:

- Recommendations when to include the fallback GeoJSON “geometry” member or not and OGC API building blocks to control the behavior.
- Support for 3D geometries through polyhedron geometry objects or other encodings (base surface plus height, support for circles, more compact coordinate encodings).
- How to simplify the parsing of the JSON schemas describing the feature schemas.
- Continue to investigate the options for representing relationships with or links to other resources.
- Potential support for a “geometryDimension” member and a potential conformance class for homogeneous feature collections.

III KEYWORDS

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

ogcdoc, OGC document, JSON

IV SECURITY CONSIDERATIONS

No security considerations have been made for this document.

V SUBMITTING ORGANIZATIONS

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

- interactive instruments GmbH

VI SUBMITTERS

All questions regarding this document should be directed to the editor or the contributors:

NAME	ORGANIZATION	ROLE
Clemens Portele	interactive instruments	Editor
Ignacio Correas	Skymantics	Contributor
Patrick Dion	Ecere Corporation	Contributor
Jérôme Jacovella-St-Louis	Ecere Corporation	Contributor
Stefano Bovio	GeoSolutions	Contributor
Felipe Carrillo Romero	Hexagon	Contributor
Panagiotis (Peter) Vretanos	CubeWerx	Contributor

1 SCOPE

1 SCOPE

OGC Features and Geometries JSON (JSON-FG) extends GeoJSON to support a limited set of additional capabilities that are out-of-scope for GeoJSON, but that are essential or important for a variety of use cases involving feature data.

Information that can be represented as GeoJSON is encoded as GeoJSON. Additional information is mainly encoded in additional top-level members of GeoJSON objects. The members use keys that do not conflict with GeoJSON including the obsolete version that pre-dates the IETF standard. GeoJSON clients will be able to parse and understand all aspects that are specified by GeoJSON, JSON-FG clients will also parse and understand the additional capabilities.

JSON Schema is used to formally specify the JSON-FG syntax.

This document includes the current JSON-FG specification, describes implementations and technology experiments as well as recommendations.

2

TERMS, DEFINITIONS AND ABBREVIATED TERMS

2 TERMS, DEFINITIONS AND ABBREVIATED TERMS

This document uses the terms defined in OGC Policy Directive 49, 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 document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

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

2.1. Terms and definitions

2.1.1. **coordinate reference system**

coordinate system that is related to an object by a datum (source: OGC Topic 2, version 5.0)

2.1.2. **feature**

abstraction of real world phenomena [ISO 19101-1:2014]

Note 1 to entry: More details about the term ‘feature’ may be found in the W3C/OGC Spatial Data on the Web Best Practice in the section ‘Spatial Things, Features and Geometry’.

2.1.3. **feature collection**

a set of **features** from a dataset

2.2. Abbreviated terms

AIXM	Aeronautical Information Exchange Model
AMDB	Airport Mapping Database
AMXM	Aerodrome Mapping Exchange Model
API	Application Programming Interface
CNS	Communications, Navigation and Surveillance
CRS	Coordinate Reference System
DSS	Discovery and Synchronization Service
FAA	Federal Aviation Administration
JSON	JavaScript Object Notation
JSON-FG	OGC Features and Geometries JSON
JSON-LD	JSON for Linking Data
KMEM	Memphis airport
OGC	Open Geospatial Consortium
REM	Route Exchange Model
SFDPS	SWIM Flight Data Publication Service
STAC	SpatioTemporal Asset Catalog
SWG	Standards Working Group
SWIM	System Wide Information Management
TIE	Technology Integration Experiment
UAS	Unmanned Aircraft System
UI	User Interface
USS	UTM Service Suppliers
UTM	UAS Traffic Management

3

INTRODUCTION

3 INTRODUCTION

Clause 4 summarizes the reasons for and the scope of the development of JSON-FG in both the Standards Working Group (SWG) and in Testbed-17.

This discussion is followed by a brief chapter, Clause 5, that summarizes key user stories driving the design of JSON-FG.

The next chapter, Clause 6, specifies the extensions to GeoJSON and discusses design decisions and considerations. The OGC Testbed-17 Engineering Report “Features and Geometries JSON CRS Analysis of Alternatives” [3] has additional discussion about representing and referencing coordinate reference systems in a JSON-FG document.

The server and client implementations are described in Clause 7.

The final chapter, Clause 8, summarizes and discusses results from the Technology Integration Experiments (TIEs). The chapter identifies open issues and recommendations.

4

MOTIVATION AND REQUIREMENTS

4 MOTIVATION AND REQUIREMENTS

JavaScript Object Notation (JSON) is a popular encoding format for geospatial data. The lightweight, simple syntax, and clear human and machine readability of JSON appeals to developers. GeoJSON has become a very popular encoding and is supported in most deployments of APIs implementing OGC API Features. However, GeoJSON has limitations that prevent or limit its use in some cases, including:

- WGS 84 is the only coordinate reference system that is supported;
- Geometries are restricted to points, curves and surfaces with linear interpolation; and
- No general capability is available to specify the schema of a feature, its type or its temporal extent.

In 2021, the OGC Membership approved the formation of a OGC Features and Geometries JSON Standards Working Group (SWG). The goal of the SWG is to specify minimal extensions to the GeoJSON Standard to support additional uses cases that are important to many OGC members.

The main extensions identified in the SWG charter are to:

- Include the ability to use Coordinate Reference Systems (CRSs) other than WGS 84,
- Allow the use of non-Euclidean metrics, in particular ellipsoidal metrics,
- Support solids and multi-solids as geometry types, and
- Provide guidance on how to represent feature properties, e.g., including temporal properties.

Testbed-17 participants were tasked to support the work of the SWG and in addition to also look into the following topics:

- How to support profiles with reduced geometry options and value constraints for features properties?
- How to support usage of Features and Geometries JSON in offline containers?

5

SCENARIO

5 SCENARIO

The following general scenarios are useful in understanding the principles that have been used in the JSON-FG design:

1. In order to display features in a 3D scene, a client accesses a Web API that shares a building dataset and that implements OGC API Features. The scene uses a projected CRS. The API advertises support for both GeoJSON and OGC JSON-FG as feature representations and support for the projected CRS in addition to WGS 84.

2. A client accesses a JSON-FG file with building data that is stored locally or as a static file in the cloud (e.g., in an Object Storage or a GitHub repository). Again, the file is accessed to display features in a 3D scene and a time slider is used to suppress features outside of a user-specified time interval.

NOTE: The following description uses a hypothetical media type `application/fg+json` for OGC JSON-FG.

5.1. Using a GeoJSON client

The client supports only GeoJSON and not JSON-FG.

In the first scenario (API access), the client requests the GeoJSON representation of the feature using `Accept: application/geo+json`. This is because the client only supports a GeoJSON feature encoding. The response will in general not include the JSON-FG extensions and is provided in WGS 84 with axis order longitude/latitude as the spatial CRS. The client will transform the geometry to the projected CRS.

In the second scenario (file access), the client has no access to a media type and has to inspect the file to determine if the file is a GeoJSON document that it can process. While the document is a JSON-FG document it should also be a valid GeoJSON document, so that the client is still able to use and display the features. The client will, however, not understand the additional information introduced by JSON-FG. Any JSON-FG extensions must not conflict with existing GeoJSON members to avoid issues for GeoJSON clients parsing JSON-FG documents.

5.2. Using a JSON-FG client

The client supports GeoJSON and JSON-FG.

In the first scenario (API access), the client will typically request the JSON-FG representation of the feature using an HTTP like `Accept: application/fg+json, application/geo`

`+json;q=0.8` in the projected CRS. The header states that the client prefers JSON-FG, but will also accept GeoJSON, if JSON-FG is not available. The response will state `Content-Type: application/fg+json` and include the JSON-FG extensions which will include the spatial geometry in the requested projected CRS and the temporal extent.

In the second scenario (file access), the client is in the same position as the GeoJSON client — it has no access to a media type and has to inspect the file to determine — if the file is a GeoJSON, JSON-FG or some other kind of document that it does not understand. JSON-FG documents include explicit declarations that conform to both GeoJSON and JSON-FG. Therefore, the client can easily identify the document as a JSON-FG document and process the content. Since the JSON-FG document identifies the spatial and temporal extent of each feature, the scene extent and, for example, a time slider can be provided by the client so that the user can filter the features to display in the scene.

6

SPECIFICATION OF JSON-FG EXTENSIONS

6 SPECIFICATION OF JSON-FG EXTENSIONS

6.1. General Approach

OGC Features and Geometries JSON (JSON-FG) extends GeoJSON to support a limited set of additional capabilities that are out-of-scope for GeoJSON, but that are essential or important for a variety of use cases involving feature data.

Information that can be represented as GeoJSON is encoded as GeoJSON. Additional information is mainly encoded in additional top-level members of GeoJSON objects. The members use keys that do not conflict with GeoJSON including the obsolete version that pre-dates the IETF standard. GeoJSON clients will be able to parse and understand all aspects that are specified by GeoJSON, JSON-FG clients will also parse and understand the additional capabilities.

JSON Schema is used to formally specify the JSON-FG syntax.

6.2. Media types

A media type should eventually be registered for JSON-FG, for example, `application/fg+json`.

Until the JSON-FG extensions are stable, a different media type should be used to avoid issues with implementations of drafts of the extensions. The implementations in the testbed used `application/vnd.ogc.fg+json` as the media type.

Since a JSON-FG document also conforms to GeoJSON, both the GeoJSON and the JSON-FG media types can be used. APIs that provide feature data that conforms to both GeoJSON and JSON-FG should declare support for both media types in the API definition to support clients that know JSON-FG and also those that only support GeoJSON.

6.3. Extension overview

- Identifying the feature type(s)
- Identifying the schema(s)
- Encoding the primary temporal extent

- Encoding the primary spatial geometry
- Encoding of reference systems
- Relationships and links
- Use in offline containers

6.4. Requirements classes

The following is a proposal of how to aggregate the JSON-FG extensions into requirements classes:

- “Core”: Support for Clause 6.8, Clause 6.9 and Clause 6.10, except support for Clause 6.9.2.3 and Clause 6.9.2.4. Depends on GeoJSON.
- “3D”: Geometries are in a 3D CRS and may be a Clause 6.9.2.3 or Clause 6.9.2.4. Depends on “Core”.
- “Feature Types and Schemas”: Support for Clause 6.6 and Clause 6.7. No dependencies.

Implementations must support at least the Core requirements class to use the JSON-FG media type.

The Features API SWG or the Features and Geometries JSON SWG could consider defining mechanisms for how to add optional support for “Feature Types and Schemas” and for GeoJSON output as these capabilities can be useful for GeoJSON clients, too.

6.5. An example feature

The following feature instance is an example of a building feature that includes all new capabilities proposed for JSON-FG. The JSON-FG extensions are described in the following sections.

Example — Building with a polyhedron geometry and the polygon footprint:

```
{
   "type": "Feature",
   "id": "DENW19AL0000giv5BL",
   "featureType": "app:building",
   "when": {
      "interval": [ "2014-04-24T10:50:18Z", null ]
   },
   "coordRefSys": "http://www.opengis.net/def/crs/EPSG/0/5555",
   "where": {
      "type": "Polyhedron",
```

```
      "coordinates": [
         [
            [
               [ 479816.67, 5705861.672, 100 ],
               [ 479824.155, 5705853.684, 100 ],
               [ 479829.666, 5705858.785, 100 ],
               [ 479822.187, 5705866.783, 100 ],
               [ 479816.67, 5705861.672, 100 ]
            ]
         ],
         [
            [
               [ 479816.67, 5705861.672, 110 ],
               [ 479824.155, 5705853.684, 110 ],
               [ 479829.666, 5705858.785, 120 ],
               [ 479822.187, 5705866.783, 120 ],
               [ 479816.67, 5705861.672, 110 ]
            ]
         ],
         [
            [
               [ 479816.67, 5705861.672, 110 ],
               [ 479824.155, 5705853.684, 110 ],
               [ 479824.155, 5705853.684, 100 ],
               [ 479816.67, 5705861.672, 100 ],
               [ 479816.67, 5705861.672, 110 ]
            ]
         ],
         [
            [
               [ 479824.155, 5705853.684, 110 ],
               [ 479829.666, 5705858.785, 120 ],
               [ 479829.666, 5705858.785, 100 ],
               [ 479824.155, 5705853.684, 100 ],
               [ 479824.155, 5705853.684, 110 ]
            ]
         ],
         [
            [
               [ 479829.666, 5705858.785, 120 ],
               [ 479822.187, 5705866.783, 120 ],
               [ 479822.187, 5705866.783, 100 ],
               [ 479829.666, 5705858.785, 100 ],
               [ 479829.666, 5705858.785, 120 ]
            ]
         ],
         [
            [
               [ 479822.187, 5705866.783, 120 ],
               [ 479816.67, 5705861.672, 110 ],
               [ 479816.67, 5705861.672, 100 ],
               [ 479822.187, 5705866.783, 100 ],
               [ 479822.187, 5705866.783, 120 ]
            ]
         ]
      ]
   },
   "geometry": {
      "type": "Polygon",
      "coordinates": [
         [
            [ 8.709204563652449, 51.50352856284526, 100 ],
            [ 8.709312860802727, 51.503457005181794, 100 ],
```

```
            [ 8.709391968693081, 51.50350306810203, 100 ],
            [ 8.709283757429898, 51.503574715968284, 100 ],
            [ 8.709204563652449, 51.50352856284526, 100 ]
         ]
      ]
   },
   "links": [
      {
         "href": "https://ogc-api.nrw.de/lika/v1/collections/gebaeude_bauwerk/
items/DENW19AL0000giv5BL?f=json",
         "rel": "self",
         "type": "application/geo+json",
         "title": "This document"
      },
      {
         "href": "https://ogc-api.nrw.de/lika/v1/collections/gebaeude_bauwerk/
items/DENW19AL0000giv5BL?f=html",
         "rel": "alternate",
         "type": "text/html",
         "title": "This document as HTML"
      },
      {
         "href": "https://ogc-api.nrw.de/lika/v1/collections/gebaeude_bauwerk?
f=json",
         "rel": "collection",
         "type": "application/json",
         "title": "The collection the feature belongs to"
      },
      {
         "href" : "https://ogc-api.nrw.de/lika/v1/collections/flurstueck/
items/05297001600313_____",
         "rel" : "http://www.opengis.net/def/rel/ogc/1.0/within",
         "title" : "Cadastral parcel 313 in district Wünnenberg (016)"
      },
      {
         "href":"https://inspire.ec.europa.eu/featureconcept/Building",
         "rel":"type",
         "title":"This feature is of type 'building'"
      },
      {
         "href": "https://ogc-api.nrw.de/lika/v1/collections/gebaeude_bauwerk/
schema",
         "rel": "describedby",
         "type": "application/schema+json",
         "title": "JSON Schema of this document"
      },
      {
         "href":"http://schemas.opengis.net/tbd/Feature.json",
         "rel":"describedby",
         "type":"application/schema+json",
         "title":"This document is a JSON-FG Feature"
      },
      {
         "href":"https://geojson.org/schema/Feature.json",
         "rel":"describedby",
         "type":"application/schema+json",
         "title":"This document is a GeoJSON Feature"
      }
   ],
   "properties": {
      "lastChange": "2014-04-24T10:50:18Z",
      "built": "2012-03",
      "function": "Agricultural building",
```

```
      "height_m": 20.0,
      "owners": [
         {
            "href": "https://example.org/john-doe",
            "title": "John Doe"
         },
         {
            "href": "https://example.org/jane-doe",
            "title": "Jane Doe"
         }
      ]
   }
}
```

6.6. Identifying the feature type(s)

6.6.1. Overview

Features are often categorized by type. Typically, all features of the same type have the same schema and the same properties.

Many GIS clients depend on knowledge about the feature type when processing feature data. For example, associating a style to a feature in order to render that feature on a map.

GeoJSON is schema-less in the sense that it has no concept of feature types or feature schemas.

In most cases, a feature is an instance of a single feature type. The current draft revision of the Simple Features standard supports features that are instances of multiple types. JSON-FG, therefore, also supports multiple feature types.

The related element Identifying the schema specifies which elements of the JSON Schema documents are identified that the JSON-FG document conforms to. This element specifies how to represent feature type information in the JSON object that represents the feature.

6.6.2. Description

6.6.2.1. The “featureType” member

The feature types of a feature are declared in a top-level member with the key “featureType”. The value is either a string (in the standard case of a single feature type) or an array of strings (to support features that instantiate multiple feature types). Each string should be a code, convenient for the use in filter expressions.

6.6.2.2. Homogeneous feature collections

Some clients will process feature collections differently depending on whether the collection is homogenous with respect to the feature type or the geometry type. These clients will benefit from information that declares the feature and/or geometry type for all features in a collection.

If the JSON document is a feature collection and all features in the feature collection have the same “featureType” value, the “featureType” member can and should be added once for the feature collection. The “featureType” member can then be omitted in the feature objects. Declaring the feature type(s) once signals to clients that the feature collection is homogeneous with respect to the type, which clients can use to optimize their processing.

If the JSON document is a feature collection and all features in the feature collection have the same geometry type as their primary geometry (point, curve, surface, solid, including homogenous aggregates), a “geometryDimension” member can and should be added once for the feature collection with the dimension of the geometry (0 for points, 1 for curves, 2 for surfaces, 3 for solids, null/not set for mixed dimensions or unknown). Declaring the geometry dimension once signals to clients that the feature collection is homogeneous with respect to the dimension, which clients can use to optimize their processing.

6.6.2.3. Links to a semantic type

If a persistent resource exists, such as in a registry, that describes a feature type, a link to that resource with link relation type `type` should be added. In the case of multiple feature types per feature, multiple links are added.

OGC API Features already specifies a general “links” member with an array of link objects based on RFC 8288 (Web linking) and feature responses from APIs implementing OGC API Features will already include a “links” member. JSON-FG builds on this approach and includes a “type” link to a resource identifying the abstract semantic type of which the feature is considered to be an instance.

Table 1 — Link properties

PROPERTY	TYPE	DESCRIPTION
href	URI	**REQUIRED**. The URI of a persistent resource that describes a feature type that is instantiated by the feature that is the link context.
rel	String	**REQUIRED**. The link relation type, always “type”.
type	String	To indicate a hint about a specific media type in which the target of the link is available, set the value to that media type; for example, “text/html”.
title	String	Include this link attribute for a human readable label of the link, e.g. for use in a derived HTML representation.

Additional link attributes may be added to the Link object.

6.6.3. Discussion

The following aspects were discussed:

1. Initial experiments used JSON-LD `@type` member to identify the feature type(s) of the feature. This approach, however, was changed to the current approach for the following reasons:

    - In many cases, feature types are specified by communities or data publishers without a persistent URI.
    - There is demand for feature type tokens that can be processed and filtered with simple string comparisons, with URI expansions.
    - While JSON-FG can be used with additional JSON-LD annotations, it is a design goal to avoid normative dependencies on JSON-LD.

    If JSON-LD is used, the “featureType” member can be mapped to “@type”.

2. This approach is similar to the approach taken by OGC API Records to identify the type of a record.

3. In a JSON-FG extension to OGC API Features (or maybe in the new part “Schemas”), the Collection resources should be extended to include (optional) information about the feature types and geometry types included the collection. There could be a recommendation for “featureType” and “geometryDimension” as queryables for such collections.

6.6.4. Open questions

The following question remains open:

1. Should there be a capability to distinguish between feature types that “just” identify a concept, but have no associated or no well-defined schema, and feature types that have an associated schema (the schema would be linked using a “describedby” link relation type)?

6.7. Identifying the schema(s)

6.7.1. Overview

A schema is metadata about a JSON document that clients can use to validate the JSON document or to derive additional information about the content of the JSON document, such as a textual description of the feature properties or their value range.

NOTE: As of 2021, the OGC Features API Standards Working Group is working on a specification in the OGC API Features series for using JSON schemas to describe the schema of features.

The JSON-FG standard will provide guidance on how to include information about the schema of a JSON document that is a single feature or a feature collection.

6.7.2. Description

The JSON Schema specification [7] recommends to use a `describedby` link relation to the schema:

> It is RECOMMENDED that instances described by a schema provide a link to a downloadable JSON Schema using the link relation "describedby" [...].

OGC API Features already specifies a general "links" member with an array of link objects based on RFC 8288 (Web linking). Therefore, feature responses from APIs implementing OGC API Features will already include a "links" member. JSON-FG builds on this approach and includes a "describedby" link to a JSON Schema document, if schema information is important for the target users of the JSON feature documents.

Table 2 — Link properties

PROPERTY	TYPE	DESCRIPTION
href	URI	**REQUIRED**. The URI of a JSON Schema document that describes the JSON document that is the link context.
rel	String	**REQUIRED**. The link relation type, which is always "describedby" for the link to the JSON Schema document.
type	String	**REQUIRED**. To indicate that the target of the link is a JSON Schema document, set the value to "application/schema+json".
title	String	Include this link attribute for a human readable label of the link, e.g. for use in a derived HTML representation.

An example of a link object:

```
{
  "href": "https://demo.ldproxy.net/zoomstack/collections/airports/schema",
  "rel": "describedby",
  "type": "application/schema+json",
  "title": "JSON Schema of this document"
}
```

Additional link attributes may be added to the Link object.

Each JSON-FG document is either a feature or a feature collection.

A feature collection document must contain a link to the JSON-FG feature collection schema at `http://schemas.opengis.net/json-fg/1.0/FeatureCollection.json`. If the document is also a GeoJSON feature collection, it must contain a link to the GeoJSON feature collection schema at `https://geojson.org/schema/FeatureCollection.json`. The document should also contain another link to a schema document that specifies the properties of the features in the collection.

A feature document must contain a link to the JSON-FG feature schema at `http://schemas.opengis.net/json-fg/1.0/Feature.json`. If the document is also a GeoJSON feature, it must contain a link to the GeoJSON feature schema at `https://geojson.org/schema/Feature.json`. The document should also contain another link to a schema document that specifies the properties of the feature.

NOTE: These are canonical URIs. Clients can identify that a JSON document is a GeoJSON and JSON-FG feature collection or feature by string comparisons.

6.7.3. Discussion

The following aspects were discussed:

1. Some environments use a "$schema" member to reference the schema of a JSON document. However, that is not the approach recommended by the JSON Schema specification itself. In JSON Schema, the value of a "$schema" property is always the URI of a meta-schema.

2. The OGC API specification for feature schemas should support at least the following two types of schemas so that they can be referenced from JSON documents retrieved from an API:

   - The schema of a feature, and
   - The schema of a feature collection, which is basically a FeatureCollection object that references the feature schema.

   This issue will be addressed by the Features API SWG (see issue #612). This topic was a topic in the November 2021 Code Sprint on OGC API Features and link relation types, relative API paths to the schema resources and JSON schema profiles were discussed.

6.7.4. Open questions

The following questions remain open:

1. The current JSON Schema version is 2020-12, which is also used in OpenAPI 3.1, but that version is not yet widely implemented. The JSON-FG schemas have been created using version 2019-09. Should schemas be published in multiple versions?

2. The current approach to signal to clients that the instance conforms to GeoJSON and/or JSON-FG is to link to the respective schemas. See the discussion in issue 10. There are at least two potential aspects that should be investigated:

   - The GeoJSON schemas are in draft 07 of JSON Schema and not available in the more recent versions 2019-09 or 2020-12. Many parsers seem to have issues with mixed schema versions. However, that is less of an issue, if the GeoJSON schemas, the JSON-FG schemas and the API-specific or community schemas are referenced separately and do not reference each other. This will still be an issue, if the schema of a particular feature type wants to reference, for example, the GeoJSON Point schema to constrain the geometries to points.

   - The status and persistence of the GeoJSON schemas is unclear.

   The GeoJSON maintainers should be contacted by the SWG to discuss these topics.

3. The JSON-FG schema documents will only be published on `schemas.opengis.net` once the JSON-FG standard has been approved. Until then, the schema documents are available at `https://raw.githubusercontent.com/opengeospatial/ogc-feat-geo-json/main/proposals/Feature.json` and `https://raw.githubusercontent.com/opengeospatial/ogc-feat-geo-json/main/proposals/FeatureCollection.json`.

4. If features are accessed using building blocks from OGC API Features, a collection can be comprised of features with different feature types. The Features API SWG should include guidance in the Schema extension how to construct a feature schema for such a collection. Multiple options exist, including:

   - A schema using “oneOf” with one set of properties for each feature type;

   - A schema with a single properties object with the superset that all features conform to; and

   - A separate schema per feature type.

5. JSON Schema is a rich language and should be considered limiting the language constructs that should be used in describing the feature schema / profile. A

potential starting point is the current proposal for a JSON Schema profile for queryable feature properties.

6. The schema of a feature type will typically specify the details of the feature properties, but it can also profile the top-level members including the “geometry”, “where” and “when” members. A typical example is to restrict the list of allowed geometry types. To simplify parsing the feature schemas it could be discussed, if canonical schemas for well-known types should be used in “$ref” members. For example, if the spatial geometry is restricted to points, the “geometry” and “where” members could reference `https://geojson.org/schema/Point.json`.

6.8. Encoding the primary temporal extent

6.8.1. Overview

Many features have a spatial geometry that provides information about the location of the feature. In GeoJSON, this information is encoded in the top-level “geometry” member. Features are also often associated with temporal information. In most cases this is either an instant (e.g., an event) or an interval (e.g., an activity or a temporal validity). In OGC API Features this is reflected in the `datetime` parameter for temporal filtering of features.

JSON-FG adds support for the most common case: Associating a feature with a single temporal instant or interval in the Gregorian calendar.

More complex cases and other temporal coordinate reference systems are out-of-scope for JSON-FG for now and might be specified in future extensions.

6.8.2. Description

Features can have temporal properties. These will typically be included in the “properties” member.

- In many datasets all temporal properties are instants (a date or a timestamp) and intervals will be described using two temporal instants, one for the start and one for the end.
- Multiple temporal properties are sometimes used to describe different temporal characteristics of a feature. For example, the time instant or interval when the information in the feature is valid (sometimes called “valid time”) and the time when the feature was recorded in the dataset (sometimes called “transaction time”). Another example is the Observations & Measurements standard, where an observation has multiple temporal properties including “phenomenon time”, “result time” and “valid time”.

Like GeoJSON, JSON-FG does not place constraints on the information in the “properties” member. JSON-FG specifies a new top-level JSON member in a feature object (key: “when”). The member describes a default temporal extent (an instant or an interval) that can be used by clients without a need to inspect the “properties” member or to understand the schema of the feature. Clients that are familiar with a dataset can, of course, inspect the information in the “properties” member instead of inspecting the “when” member.

The publisher of the data needs to decide which temporal feature properties are used in the “when” member.

The value of “when” is an object.

Table 3 — Properties of the “when” object

PROPERTY	TYPE	DESCRIPTION
instant	string	The temporal extent as an instant. See below for more details about instants.
interval	[string]	The temporal extent as an interval, an array of two instants. See below for more details about intervals.

If both values intersect it is valid to include both an instant and an interval. Clients should use the interval and may use the instant to determine the temporal extent of the feature.

The “when” object may be extended with additional members. Clients processing a “when” object must be prepared to parse additional members. Clients may ignore members that they do not understand. For example, in cases where the “when” member neither includes an “instant” or “interval”, a client may process the feature as a feature without a temporal extent.

NOTE: The data publisher decides how temporal properties inside the “properties” member are encoded. The schema for the “when” member does not imply a recommendation that temporal feature properties reuse the same schema. For example, it is expected that a date-valued feature attribute will in most cases be represented as string with an RFC 3339 date value.

6.8.3. Instants

An instant is a value that conforms to RFC 3339 (Date and Time on the Internet: Timestamps) and is consistent with one of the following production rules of the ISO 8601 profile specified in the RFC:

- `full-date` (e.g., `"1969-07-20"`)
- `date-time` (e.g., `"1969-07-20T20:17:40Z"`)

Note that all timestamps have to include a time zone. The use of UTC is recommended (“Z”).

The JSON schema for an instant:

```
oneOf:
- type: string
```

```
  format: date
- type: string
  format: date-time
```

This describes the initial range of instant values. This range may be extended in the future to support additional use cases. Clients processing instant values must be prepared to receive other values. Clients may ignore values that they do not understand.

6.8.4. Intervals

An interval is described by the start and end instants. Both start and end instants are included in the interval.

Open ranges at the start or end are represented by a `null` value for the start/end.

```
type: array
minItems: 2
maxItems: 2
items:
  oneOf:
  - oneOf:
    - type: string
      format: date
    - type: string
      format: date-time
  - null
```

This describes the initial range of interval values. This range may be extended in the future to support additional use cases. Clients processing interval values must be prepared to receive other values. Clients may ignore values that they do not understand.

6.8.5. Discussion

The following aspects were discussed:

1. The current specification of the “when” member could be extended with minimal extensions beyond the RFC 3339 timestamps to support additional use cases, but this would be left to a future extension:
    - Supporting instant values of a year (e.g., “1969”) or a month (e.g., “1967-07”) in addition to dates and timestamps could be useful for extents that cover a complete month or year.
    - Supporting instant values of the proleptic Gregorian calendar (i.e., dates before 1582 including negative years) could be useful for historic information.

2. An alternative interval encoding would be to use ISO 8601-1/8601-2 and represent intervals as a string, too, instead of an array. In this case, the value of "interval" would be encoded as:

   - `"1969-07-16/1969-07-24"`
   - `"1969-07-16T05:32:00Z/1969-07-24T16:50:35Z"`
   - `"2019-10-14/.."`

   The main reason for the current design is that this is easier to parse. In a structured language like JSON it is natural to be explicit and use the structures. For example, GeoJSON does not represent the geometry as a WKT string, but as a JSON structure. The JSON encoding of the candidate Common Query Language (CQL2) standard also follows that approach for spatial and temporal geometries. Strings are used for instants (date and timestamp values according to RFC 3339), which are supported in most programming environments. However, any temporal type that is constructed from multiple instants uses JSON structures.

   Another reason for the current design is that it can support additional temporal types in the future besides instants and intervals. An example is a time series with an array of timestamps.

3. The current proposal only specifies the use of "when" in the context of a feature. Other contexts could also be supported in the future. For example, a temporal extent for each geometry in a geometry collection or for properties where the value changes over time.

4. SpatioTemporal Asset Catalog (STAC) specifies a property datetime that is included in the "properties" member. This approach was not followed for two reasons:

   - Like the feature identifier and the default spatial geometry of a feature, the default temporal extent of a feature should be a "first-class" citizen in the JSON encoding of the feature.
   - The use of reserved property names might conflict with existing features that include a "datetime" property with a different specification.

5. Just as the STAC "datetime" property, "when" does not provide an indication of the semantics associated with the temporal information. One potential approach to provide hints to clients could be the use of a JSON-LD context to associate the "when" member with a property definition in some vocabulary.

6. Currently there is no information about the semantics of the temporal extent. Is it a temporal extent of a feature that is a version of a real-world entity and there are potential predecessor/successors features of the same entity? Or, in case of a bitemporal dataset, is it the valid time or the transaction/record time? Etc. It was decided to keep the JSON-FG simple and leave support for more complex

requirements to extensions. Support for versioned feature data is one of the planned future work items of the Features API SWG.

6.8.6. Open questions

The following questions remain open:

1. For timestamps, the current proposal recommends the use of UTC. GeoPackage, the next versions of CDB and the Common Query Language will only support UTC as a time zone in literal values. Should JSON-FG follow this approach?

2. ISO 8601 also supports intervals by a duration (a start instant and the duration or the duration and an end instant). Should this also be supported or does that make parsing just more complex for clients?

3. There is an existing initiative for a temporal GeoJSON extension ("GeoJSON-T"). The proposal also uses "when" as a key, but with a different schema for the "when" object. The GeoJSON-T design supports more complex use cases that go beyond the scope of the proposal. To avoid confusion, the SWG should either use a different key than "when" or agree on a joint approach with the GeoJSON-T author (there should be support for simple instants/intervals as a minimal profile, additional capabilities would then extend that minimal profile).

6.9. Encoding the primary spatial geometry

6.9.1. Overview

Features typically have a spatial geometry that provides information about the location of the feature.

In GeoJSON, this information is encoded in the top-level "geometry" member. Geometries are according to the Simple Features Standard (2D or 2.5D points, line strings, polygons or aggregations of them) in WGS 84 as the coordinate reference system (OGC:CRS84 or OGC:CRS84h).

A key motivation for JSON-FG is to support additional requirements, including other CRSs as well as solids, where the boundary is specified using polygons.

To avoid confusing existing GeoJSON readers, such geometries will be provided in a new top-level member with the key "where" (or another key).

6.9.2. Description

The main spatial location of a feature is provided in the “geometry” and “where” members of the feature object. The value of both keys is an object representing a spatial geometry — or `null`.

The value of the “geometry” member is specified in the GeoJSON standard.

The value range of the “where” member is an extended and extensible version of the value range of the value range of the “geometry” member:

- Is extended by the value options (additional JSON-FG geometry types Clause 6.9.2.3 and Clause 6.9.2.4) as well as by the capabilities to declare the coordinate reference system of the coordinates of the geometry.
- Is extensible and future parts of Features and Geometries JSON or community extensions may specify additional members or additional geometry types. JSON-FG readers should be prepared to parse values of “where” that go beyond the schema that is implemented by the reader. Unknown members should be ignored and geometries that include an unknown geometry type should be mapped to `null`.

6.9.2.1. Use of “geometry” and/or “where”

If the geometry that describes the main geometry of the feature can be represented as a valid GeoJSON geometry (one of the GeoJSON geometry types, in WGS84), it is encoded as the value of the “geometry” member. The “where” member has the value `null`.

If the geometry cannot be represented as a valid GeoJSON geometry, it is encoded as the value of the “where” member. In addition, a valid GeoJSON geometry may be provided in the “geometry” member in the coordinate reference system WGS84 as specified in the GeoJSON standard (otherwise “geometry” is set to `null`). The geometry in “geometry” is a fallback for readers that support GeoJSON, but not JSON-FG. This could be a simplified geometry, like the building footprint in the example “building with a polyhedron geometry and the polygon footprint” instead of the solid geometry or the same point/line string/polygon geometry, but in WGS 84 (potentially with fewer vertices to reduce the file size).

NOTE: The OGC Code Sprint in November 2021 discussed the API building blocks to request a fallback geometry in the “geometry” member or not. It was agreed that the default behavior is to omit the “geometry” member, if a “where” member is included. If a JSON-FG client wants to include the fallback geometry — typically, because the data will be published and shared with other users and tools — the requested media type should include a parameter `compatibility=geojson` (that is, the media type would be `application/fg+json;compatibility=geojson`). This approach was validated with two implementations.

6.9.2.2. Metrics

If the CRS uses axis order longitude and latitude, clients should perform geometrical computations — including computation of length or area on the curved surface that approximates the earth's surface. Details are provided in the drafts of "Features and Geometry — Part 2: Metrics" [5].

Note that this differs from GeoJSON which states:

> A line between two positions is a straight Cartesian line, the shortest line between those two points in the coordinate reference system. In other words, every point on a line that does not cross the antimeridian between a point (lon0, lat0) and (lon1, lat1) can be calculated as `F(lon, lat) = (lon0 + (lon1 - lon0) * t, lat0 + (lat1 - lat0) * t)` with t being a real number greater than or equal to 0 and smaller than or equal to 1. Note that this line may markedly differ from the geodesic path along the curved surface of the reference ellipsoid.
>
> — GeoJSON (RFC 7946)

6.9.2.3. Polyhedron

A *polyhedron* is a non-empty array of *multi-polygon* arrays. Each *multi-polygon* array is a shell and must be closed. The first shell is the exterior boundary, all other shells are holes.

Example — JSON Schema for a polyhedron geometry:

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://schemas.opengis.net/tbd/Polyhedron.json",
  "title": "A polyhedron geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Polyhedron"
      ]
    },
    "coordinates": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
```

```
            "minItems": 4,
            "items": {
              "type": "array",
              "minItems": 3,
              "maxItems": 3,
              "items": {
                "type": "number"
              }
            }
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 6,
      "maxItems": 6,
      "items": {
        "type": "number"
      }
    }
  }
}
```

6.9.2.4. MultiPolyhedron

A *multi-polyhedron* is an array of *polyhedron* objects. The order of the polyhedron geometry objects in the array is not significant.

Example — JSON Schema for a multi-polyhedron geometry:

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://schemas.opengis.net/tbd/MultiPolyhedron.json",
  "title": "A multi-polyhedron geometry",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiPolyhedron"
      ]
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
            "minItems": 1,
            "items": {
              "type": "array",
```

```
              "minItems": 4,
              "items": {
                "type": "array",
                "minItems": 3,
                "maxItems": 3,
                "items": {
                  "type": "number"
                }
              }
            }
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 6,
      "maxItems": 6,
      "items": {
        "type": "number"
      }
    }
  }
}
```

6.9.3. Discussion

The following aspect was discussed:

1. As of the writing of this ER, JSON-FG will not support multiple geometries per feature (note that multiple geometries can be included inside the “properties” member) or any hints about the semantics of the geometry. The latter could potentially be addressed using JSON-LD contexts where this is important.

2. The OGC Code Sprint in November 2021 also tested JSON-FG data with draft OGC API Features building blocks to access feature data suitable for display at a certain map scale / zoom level. This included API building blocks for simplifying geometries that are not points as well as filtering features that are typically not shown in maps at the scale or zoom level. This was validated with multiple implementations (a JSON-FG client and three OGC API Features servers).

6.9.4. Open questions

The following questions remain open:

1. Including information in some kind of ‘header’ would be helpful. This would support how a client should parse the content (as JSON-FG with its extensions, as standard GeoJSON, etc.), in particular, if the data is parsed as a file, not as an API response with a content type header.

2. Should a JSON Pointer be allowed in “where”, if “where” and “geometry” are the same geometry to reduce duplication?

3. Should a 3D geometry that represents a simple solid constructed using an extruded polygon also be supported? This would consist of a (horizontal) 2D polygon and separate attributes for the lower and upper limits. How often are such geometries used? With respect to “extruded polygons” it seems like they could be useful, but it is unclear if the added complexity of an additional geometry type is valuable enough. This is a broader topic as to how to handle geometries that are constructed using “regular” feature properties.

6.10. Encoding of reference systems

6.10.1. Overview

Without any other information, the following defaults apply in a JSON-FG file:

- spatial coordinate reference system: OGC:CRS84 (2D) or OGC:CRS84h (3D)
- temporal coordinate reference system: Gregorian

For asserting CRS information in a JSON-FG file:

- The key `coordRefSys` is defined and can be used to assert a CRS at the collection, feature or value levels.
- The value of the `coordRefSys` key can be:
  - a simple URI reference,
  - a URI reference with an epoch,
  - or as an array of CRS references (with or without epoch) for an ad hoc compound CRS.

It is anticipated that if a CRS is asserted for a JSON-FG file, that assertion will be made at the top level of the document, either at the collection level or the feature level depending on the contents of the document.

NOTE: The key was originally `coord-ref-sys`, but the naming style of keys was harmonized to consistently use lowerCamelCase. The implementations described in chapter Clause 7 have used the original key.

6.10.2. Description

Spatio-temporal objects are specified relative to some reference system.

GeoJSON (both the current RFC and the legacy version) fixed the reference system for geometric values to the "WGS84 datum, and with longitude and latitude units of decimal degrees". The legacy version included a "prior arrangement" provision to allow other reference systems to be used and also defined the `crs` key for specifying the reference system. This *prior arrangement* mechanism survived into the RFC but the accompanying `crs` key did not. The result is that there is no interoperable way to unambiguously specify a different CRS if GeoJSON files and the safest thing to do is to stick with CRS84(h) for GeoJSON members and ignore the *prior arrangement* provision and the old `crs` key.

JSON-FG is not bound by these restrictions and so this document outlines a proposal for handling reference systems in JSON-FG documents that does not interfere with anything, past or present, defined in any of the GeoJSON specifications. The GeoJSON elements can continue to operate as always but JSON-FG elements can avail themselves of enhanced CRS support.

6.10.2.1. Reference system values

A reference system can be specified in a JSON-FG document using a `coordRefSys` member in one of three ways:

- As a simple reference using a URI;
- As a simple reference using a URI accompanied by an epoch value;
- As an array of reference system values denoting an ad hoc compound reference system.

Used at the collection level, the `coordRefSys` key asserts the coordinate reference system for JSON-FG spatiotemporal values found anywhere in the document that are not otherwise tagged with closer-to-scope coordinate reference system information.

Used at the feature level, the `coordRefSys` key asserts the coordinate reference system for geometric JSON-FG value found anywhere in the feature that are not otherwise tagged with closer-to-scope coordinate reference system information.

Used at the geometry level, the `coordRefSys` key asserts the coordinate reference system for the geometry JSON-FG value within which the key is contained.

6.10.2.2. Value schema

The following JSON Schema fragment defines a reference system value:

The schema of a reference system value:

```
{
```

```
      "$defs": {
         "refsysSimpleref": {
            "type": "string",
            "format": "uri"
         },
         "refsysByref": {
            "type": "object",
            "required": [ "href" ],
            "properties": {
               "href": {
                  "type": "string",
                  "format": "uri"
               },
               "epoch": {
                  "type": "string"
               }
            }
         },
         "refsys": {
            "oneOf": [
               { "$ref": "#/$defs/refsysSimpleref" },
               { "$ref": "#/$defs/refsysByref" },
               {
                  "type": "array",
                  "items": {
                     "oneOf": [
                        { "$ref": "#/$defs/refsysSimpleref" },
                        { "$ref": "#/$defs/refsysByref" },
                     ]
                  }
               }
            ]
         }
      },
      "$ref": "#/$defs/refsys"
   }
```

A simple reference system value by reference.:

```
"http://www.opengis.net/def/crs/EPSG/0/3857"
```

A reference system value by reference and with an epoch.:

```
{
  "href": "http://www.opengis.net/def/crs/EPSG/0/4979",
  "epoch": "2016.47"
}
```

A ad hoc compound reference system value:

```
[
  {
    "href": "http://www.opengis.net/def/crs/EPSG/0/4258",
    "epoch": "2016.47"
  },
  "http://www.opengis.net/def/crs/EPSG/0/7837"
]
```

6.10.3. Discussion

This ER only includes a summary of the CRS topic. The detailed description and discussion is part of the Features and Geometries JSON CRS Analysis of Alternatives Engineering Report [3].

6.11. Relationships and links

Features can have properties that are relationships with other features or resources like codelists, etc. There are multiple options for how to encode such relationships and JSON-FG could provide guidance for how to represent such relationships.

Relationships will often be direct properties of the feature, but they may also occur in embedded JSON objects.

Like all other properties, properties that are relationships may have a maximum multiplicity greater than one. That is, the JSON representation may be an array of relationships.

JSON-FG will not mandate a specific way to represent relationships and links in the feature object. There are, however, three basic patterns of how to represent relationships and links. Each pattern may or may not be applicable to the intended use of the data.

Pattern 1 seems best suited: If the intended use of the data benefits from a consistent place where links are included in the JSON document.

Pattern 2 seems best suited: If the JSON features should closely reflect the application schema of the features (in case a schema is available).

The same also applies to pattern 3, but this option seems mainly useful in combination with JSON-LD. Also, the information is not sufficient to render a useful HTML representation from the JSON representation without fetching the linked resources.

Depending on the data and how the data is expected to be used, the preferences of data publishers for one or the other pattern will vary.

6.11.1. Pattern 1: Add all relationships to a “links” member of the JSON object that is the link anchor

This option is consistent with the general approach used in the current OGC API standards. This option uses Web linking and a consistently named JSON member with an array of OGC API Link objects.

The semantics of the relationship / association role is expressed via the link relation type. Where possible, link relation types registered with IANA or OGC should be used, but the data publisher can also define their own link relation types. Note that this introduces overhead, because it

requires minting persistent URIs for the link relation types. Where this is too much, an existing link relation type should be used and “related” could be used as a fallback.

6.11.2. Pattern 2: Encode links like other feature properties — using a simplified link object

This option treats the relationships like other properties and uses a simplified OGC API Link object without a “rel” attribute, since the semantics of the link is already expressed by the property.

A variation could be to require the use of a valid link relation type as the key of the JSON member, which would basically move the link relation type to a key to group all links with the same link relation type.

Another variation of this option would be to flatten the link objects.

Note that instead of using a Link object to reference the related resource, the related resource could also be embedded — at least as long as the referenced resource can be represented as a JSON object.

6.11.3. Pattern 3: Only use the URI

This option is similar to option 2, but the link objects are reduced to the href value. As a result, this option is more concise, but it lacks information that would be useful for the human (unless the URIs are dereferenced to fetch a label/title). In addition, since this approach does not use web linking according to RFC 8288, no link relation types for the links are available.

6.11.4. Discussion

The following aspects were discussed:

1. A GeoJSON feature that is encoded by a Web API implementing OGC API standards will often already include a “links” member with an array of OGC API Link objects. The OGC API Link object is a JSON implementation of web links according to RFC 8288. The OGC API Features standard currently only specifies requirements for links to resources in the API to support clients navigating the API.

2. Are links part of the resource or metainformation? There is a general, somewhat philosophical discussion topic related to links between resources.

   In general, the links of a web resource are considered metainformation, and strictly the links do not have to be part of the cachable representations of the resource. RFC 8288 (Web linking) supports this by supporting that links are only

represented as HTTP headers, outside of the representations. Changes to a link would not impact the ETag or Last-Modified headers of the resource.

The links in feature (collection) resources specified in OGC API Features ("self", "alternate", "next", "collection") or in JSON Schema ("describedby") are a good example. A change in any of those links does not indicate a change in the resource itself, but it indicates a technical change in the implementation. For example, another alternate representation has been added or the schema has moved to a different URI.

However, because the OGC API standards include the links in the JSON representation — like most of the existing approaches to JSON-based Web APIs, a change in the links will also invalidate cached representations of the resource (and update the ETag and Last-Modified headers). A conscious decision is to include the links in the JSON representation. This approach seems to meet the expectations of developers today.

The same applies to many of the explicit or implicit relationships that are expressed in geospatial datasets today. Whether a second building is erected on the parcel or not does not really change the parcel. It could be argued that the relationship between the parcel and the building is metainformation and a change to a relation does not change the parcel — and should not invalidate any cached representations. Links between the resources could be managed — and accessed — as separate resources (e.g. linksets).

Nevertheless, many users and developers will prefer a more "traditional" way of sharing geospatial features with relationships included in the resource representation and the discussion below is based on this assumption.

3. An extension to CQL2 to properly support filtering links should be considered by the Features API SWG, too.

6.11.5. Open questions

The following question remains open:

1. Link relation types for the topological interval relationships as specified in OWL Time are already registered with IANA. The registration of the named spatial topological relationships specified in the Simple Feature Access standard [6] should be considered, either in the IANA or the OGC register. Such link relation types are useful for cases where relationships are pre-computed or are computed on the fly but are presented, because the clients cannot easily compute them.

6.12. Use in offline containers

6.12.1. Overview

This clause describes the work done in the OGC Feature and Geometries JSON thread of Testbed-17 that investigated the use of JSON-FG in *offline containers*.

For situations with intermittent or no network connectivity it can be necessary to create an offline container of the data of interest. Such a container allows the data to be pre-loaded onto a device and then used locally when the device is disconnected and thus has no network access.

Brief consideration was given to using a JSON-FG file itself as the offline container. However, the testbed participants felt that too many additional extensions beyond what is described in this ER would be required to accommodate all the information that typically accompanies feature data. Such accompanying information includes descriptive metadata, styling information, CRS definitions, etc. Unambiguously accommodating all this information in a single file would present a unique access pattern that would require specifically designed tools and would thus limit the use of existing OGC-aware tools to access the information in the container.

Geospatial data is often associated with coordinate system references and there often exist relationships between data items manifested as links in the data. Such CRS references and links to other resources require network connectivity and without it, other provisions need to be made in an offline container to satisfy such linking requirements.

6.12.2. File structure of an offline container

The original approach for creating an offline container was to reuse work previously done in the OWS-8 Testbed and described in the "OGC OWS-8 Bulk Geodata Transfer Using GML Engineering Report" (OGC 11-085r1). OGC 11-085r1 describes an offline container called a ***GBT*** file.

A GBT file is a ZIP archive that contains feature data encoded as GML. Accompanying the GML data are files describing the application schema of each GML file in the GBT and SLD files that contain styling information. The result of the OWS-8 work was two utilities named `gbtexport` and `gbtimport`. The `gbtexport` utility would read a source datastore and generate a GBT file. The `gbtimport` utility would read a GBT file and load it into a target datastore.

In the work done for Testbed 17, only the `gbtexport` utility was considered since the goal was to create an offline container and not necessarily reload the contents of that container into some other datastore; although that is certainly possible.

The following is a listing of the contents of the GBT file from OWS-8 created using the `gbtexport` utility:

```
Archive:  foundation.gbt
Geodata Bulk Transfer (GBT) file of "foundation" data store, packaged on 2008-
10-27.
```

```
  Length      Date    Time    Name
---------  ---------- -----   ----
  3672595  10-27-2008 07:53   aerofacp_1m.gml
     4690  10-27-2008 07:53   aerofacp_1m.xsd
     1347  10-27-2008 07:53   aerofacp_1m.sld
 37749241  10-27-2008 07:53   coastl_1m.gml
     3389  10-27-2008 07:53   coastl_1m.xsd
     1332  10-27-2008 07:53   coastl_1m.sld
     1958  10-27-2008 07:53   ows-manifest.xml
---------                     -------
 41431873                     5 files
```

The file structure is flat and the various files are associated using a manifest file, `ows-manifest.xml`. The manifest file was created according to the OGC Web Service Common Implementation Specification. Although this structure is suitable for the offline container use case, in 2021 it makes more sense to use a file structure that is more likely to be understood by current tools that access OGC feature data.

For Testbed 17, the code for the `gbtexport` utility was resurrected and modified to generate an offline container using JSON-FG as the primary means of encoding feature data. Like GBT, the offline container is a ZIP archive. The following is a sample listing of the contents of an offline container (i.e. ZIP archive) created using the updated `gtbexport` utility:

```
Archive:  foundation.gbt
Geodata Bulk Transfer (GBT) file of "foundation" data store, packaged on 2021-
10-13.
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  10-13-2021 20:07   collections/
        0  10-13-2021 21:40   collections/coastl_1m/
 44915138  10-13-2021 07:45   collections/coastl_1m/items.json
        0  10-13-2021 19:18   collections/coastl_1m/schemas/
     3693  10-13-2021 22:31   collections/coastl_1m/schemas/collection.json
     2392  10-13-2021 19:17   collections/coastl_1m/schemas/feature.json
        0  10-13-2021 19:18   collections/coastl_1m/metadata/
        0  10-13-2021 19:18   collections/coastl_1m/metadata/iso19115.xml
        0  10-13-2021 21:40   collections/coastl_1m/styles/
        0  10-13-2021 21:43   collections/aerofacp_1m/
  5019561  10-13-2021 20:06   collections/aerofacp_1m/items.json
        0  10-13-2021 19:20   collections/aerofacp_1m/schemas/
  3031044  10-13-2021 07:51   collections/aerofacp_1m/schemas/collection.json
     1576  10-13-2021 19:20   collections/aerofacp_1m/schemas/feature.json
        0  10-13-2021 19:15   collections/aerofacp_1m/metadata/
        0  10-13-2021 06:42   collections/aerofacp_1m/metadata/iso19115.xml
        0  10-13-2021 21:43   collections/aerofacp_1m/styles/
     4098  10-13-2021 20:09   collections.json
---------                     -------
 52977502                     18 files
```

NOTE 1: The ISO metadata files in this example are just placeholders to illustrate how associated metadata is included in the offline container. As such, their size is 0 bytes.

Anyone familiar with the OGC API — Features — Part 1: Core specification will recognize the file structure presented above; it mirrors the path structure of the OGC resource tree with some extensions to handle other associated resources such as metadata and/or styles.

NOTE 2: The original intent was to absorb the capabilities of the `gbtexport` utility into the CubeWerx OGC API Features server. The offline container would simply be another output format supported by the server. However, the current inability to simultaneously request

features from multiple collections through the API and constraints on time and resources limited the scope of work to updating the existing code for the `gbtexport` utility.

6.12.3. The manifest

The original GBT file from OWS-8 contained an `ows-manifest.xml` file that listed all the files in the archive. In TB17 the role of manifest is now played by the `collections.json` file. The following JSON document illustrates the contents of the `collections.json` file:

Example — Sample collection.json file from an offline container:

```
{
  "title": "foundation GBT",
  "description": "Geodata Bulk Transfer (GBT) file of the \"foundation\" data
 store, packaged on 2021-10-13.",
  "refSystems": {
    "CRS84": {
      "valueType": "ogcwkt",
      "value": "GEOGCS[\"WGS 84\",DATUM[\"WGS_1984\",SPHEROID[\"WGS 84\",
 6378137, 298.257223563]],PRIMEM[\"Greenwich\", 0],UNIT[\"degree\", 0.
0174532925199433],AUTHORITY[\"EPSG\", \"4326\"]]"
    }
  },
  "collections": [
    {
      "itemType": "feature",
      "id": "aerofacp_1m",
      "title": "Airport Facilities Points",
      "description": "VPF Narrative Table for \"Airport Facilities Points\":\n
\nInformation on airports/airfields (GB005) was derived from the DAFIF (Digital
 Aeronautical Flight Information File) and TINT (Target Intelligence) in areas
 where such data was available.  Each airfield's DAFIF reference number was
 placed in the 'na3' (classification name) attribute field.  Only airfields
 which had at least one hard surface runway longer that 3,000 feet (910 meters)
 were collected.\n",
      "nFeatures": 9335,
      "links": [
        {
          "href": "collections/aerofacp_1m/items.json",
          "rel": "items",
          "type": "application/vnd.ogc.fg+json",
          "title": "the vector features of this collection as OGC Features &
 Geometries JSON"
        },
        {
          "href": "collections/aerofacp_1m/schemas/collection.json",
          "rel": "http://www.opengis.net/def/rel/ogc/0.0/schema-collection",
          "title": "the schema of this collection",
          "type": "application/schema+json"
        },
        {
          "href": "collections/aerofacp_1m/schemas/feature.json",
          "rel": "http://www.opengis.net/def/rel/ogc/0.0/schema-item",
          "title": "the schema of feature instances of this collection",
          "type": "application/schema+json"
        },
        {
          "href": "collections/aerofacp_1m/metadata/iso19115.json",
          "rel": "describedby",
          "title": "ISO19115 metadata describing this collection",
```

```
          "type": "application/xml"
        }
      ],
      "extent": {
        "spatial": {
          "bbox": [
            [
              -179.878326416016,
              -54.9311103820801,
              179.339859008789,
              79.52944183349609
            ]
          ],
          "crs": "#/refSystems/CRS84"
        }
      },
      "crs": [
        "#/refSystems/CRS84"
      ]
    },
    {
      "itemType": "feature",
      "id": "coastl_1m",
      "title": "Coastlines",
      "description": "VPF Narrative Table for \"Coastlines\":\n\nCoastline/
shorelines (BA010) have been portrayed for coastal islands but not inland
 islands.\n",
      "nFeatures": 36371,
      "links": [
        {
          "href": "collections/coastl_1m/items.json",
          "rel": "items",
          "type": "application/vnd.ogc.fg+json",
          "title": "the vector features of this collection as OGC Features &
 Geometries JSON"
        },
        {
          "href": "collections/coastl_1m/schemas/collection.json",
          "rel": "http://www.opengis.net/def/rel/ogc/0.0/schema-collection",
          "title": "the schema of this collection"
        },
        {
          "href": "collections/coastl_1m/schemas/feature.json",
          "rel": "http://www.opengis.net/def/rel/ogc/0.0/schema-item",
          "title": "the schema of feature instances of this collection"
        },
        {
          "href": "collections/coastl_1m/metadata/iso19115.json",
          "rel": "describedby",
          "title": "ISO19115 metadata describing this collection",
          "type": "application/xml"
        }
      ],
      "extent": {
        "spatial": {
          "bbox": [
            [
              -179.999420166016,
              -85.582763671875,
              179.9999,
              83.62741851806639
            ]
          ],
```

```
          "crs": "#/refSystems/CRS84"
        }
      },
      "crs": [
        "#/refSystems/CRS84"
      ]
    }
  ],
  "links": [
    {
      "href": "collections.json",
      "rel": "self",
      "type": "application/json",
      "title": "this page of collections as JSON"
    }
  ]
}
```

The `collections.json` file is based on the structure of the collections object from OGC API — Features with some additions. Specifically, the `nFeatures` member is added to include a count of the number of features in the collection and the `refSystems` member is added as a dictionary of CRSs used in the file. All CRS references are to this local dictionary.

NOTE: See CRS Consideration.

6.12.4. Feature data

Each feature collection included in an offline container is encoded as a JSON-FG file with the fixed name `items.json`. The `items.json` file is located in the `collections/{collectionId}` directory (e.g.: `collections/coastl_1m/items.json`) of the offline container.

Each feature collection file is created as specified in this document with modifications to handle CRS references as described in the OGC Testbed-17 Features and Geometries JSON CRS Analysis of Alternatives ER [3].

```
{
  "type": "FeatureCollection",
  "timeStamp": "2021-10-25T22:06:27-04:00",
  "numberMatched": 36372,
  "numberReturned": 36372,
  "refSystems": {
    "CRS84": {
      "valueType": "ogcwkt",
      "value": "GEOGCS[\"WGS 84\",DATUM[\"WGS_1984\",SPHEROID[\"WGS 84\",
 637813
7, 298.257223563]],PRIMEM[\"Greenwich\", 0],UNIT[\"degree\", 0.
0174532925199433]
,AUTHORITY[\"EPSG\", \"4326\"]]"
    }
  },
  "coordRefSys": "#/refSystems/CRS84",
  "bbox": [
    -179.999420166016,
    -85.582763671875,
    179.9999,
    83.62741851806639
  ],
```

```
  "links": [
    {
      "href": "items.json",
      "rel": "self",
      "type": "application/vnd.ogc.fg+json; charset=utf-8"
    },
    {
      "href": "schemas/collection.json",
      "rel": "describedby",
      "type": "application/schema+json; charset=utf-8"
    }
  ],
  "features": [
    {
      "type": "Feature",
      "featureType": "coastl_1m",
      "id": "CWFID.COASTL_1M.0.0",
      "geometry": {
        "type": "LineString",
        "coordinates": [ ... ]
      },
      "where": null,
      "properties": {
        "id": 4671,
        "f_code": "BA010",
        "acc": 2,
        "exs": 44,
        "tile_id": 730,
        "edg_id": 9
      }
    },
    .
    .
    .
  ]
}
```

Sample items.json file from an offline container

By default the `gbtexport` utility will copy all features from a source collection into the offline container. However, the `gbtexport` utility accepts parameters that allow spatial, temporal and/or scalar predicates to be specified to define a subset of features to be copied to the offline container.

6.12.5. Coordinate reference system references

As specified in the Encoding of reference systems clause, CRS information is asserted in a JSON-FG file using the `coordRefSys` key. The value of the `coordRefSys` key is a URI. Usually, this URI points to the definition of the CRS. In an intermediate or no connectivity environment such CRS URIs cannot be resolved.

In order to make CRS URIs resolvable, the `gbtexport` utility includes a dictionary of CRSs at the head of each JSON-FG file using the `refSystems` member discussed in the OGC Testbed-17 Features and Geometries JSON CRS Analysis of Alternatives ER [3] and then rewrites all remote CRS references in the JSON-FG file to be local reference to the corresponding entry in the dictionary. The following JSON fragment illustrates the use of the `refSystems` member:

```
{
  "type": "FeatureCollection",
  "timeStamp": "2021-10-25T22:06:27-04:00",
  "numberMatched": 36372,
  "numberReturned": 36372,
  "refSystems": {
    "CRS84": {
      "valueType": "ogcwkt",
      "value": "GEOGCS[\"WGS 84\",DATUM[\"WGS_1984\",SPHEROID[\"WGS 84\",
 637813
7, 298.257223563]],PRIMEM[\"Greenwich\", 0],UNIT[\"degree\", 0.
0174532925199433]
,AUTHORITY[\"EPSG\", \"4326\"]]"
    }
  },
  "coordRefSys": "#/refSystems/CRS84",
  .
  .
  .
}
```


Consideration: Because many clients have built-in knowledge about common CRSs, such as CRS84, an open question is whether or not this `refSystems` mechanism for converting remote CRS references to local CRS references is of value.

6.12.6. Schemas

For the purpose of validation each feature collection included in the offline container may be accompanied by a JSON Schema file that declares the schema of the collection and the schema for a feature instance. The schema file for the collection is named `collection.json`. The schema file for feature instances is named `feature.json`. Both files are located in the `collections/{collectionsId}/schemas` directory. The Identifying the schema(s) clause describes how to reference these schemas within a JSON-FG file.

6.12.7. Relationships and links

The Relationships and links clause describes patterns for linking features to other features and other resources. Since an offline container is meant to be used in an intermittent or no connectivity environment, such references will break. In order to make such links resolvable, the resources being references need to be copied into the offline container and all remote references need to be rewritten into local references.

Consider the case where collection A includes features that reference features from collection B. If collection A is copied into an offline container then collections B also needs to be copied into the offline container. However, only instances of B necessary to resolve references from instances of A need to be included in the offline container. Furthermore, all link in instances of A that references instances of B need to be rewritten to reference the local copies of B.

NOTE: The `gdbexport` utility was updated to handle the resolution of feature references but not references to other resources (e.g. code lists). Also, because of time, resource and data constrains this capability could not be tested during the testbed.

6.12.8. `gbtexport` utility

The `gbtexport` utility was created for OWS-8 and was modified for Testbed 17. The utility generates an offline container as a ZIP archive with a file structure that mirrors the OGC API resource tree structure. The following is the argument list for the `gbtexport` utility:

```
BT (Geodata Bulk Transfer) exporter, version 9.3.63

Copyright © 1997-2021 CubeWerx Inc.

Usage: gbtexport keyword=value ... (or) -keyword value ... -boolflag

KEYWORD    ABB  DESCRIPTION                                          DEFAULT
---------  ---  ---------------------------------------------------  -------
PATH       U    address of data store to export (REQUIRED)           (none)
DRIVER     D    data-store format type                               (auto)
FSET       F    feature set(s) of data store to export               (all)
SQLFILTER  SQL  SQL filter(s), parallel to FSET(s)                   (none)
WINDOW          bounding box to export (minX,minY,maxX,maxY)         (all)
WINDOWCS        coordinate system of bounding box                    (auto)
FROMDATE        from date/time (YYYY-MM-DD HH:mm:ss)                 -INF
TODATE          to date/time (YYYY-MM-DD HH:mm:ss)                   INF
LAYERS          boolean - export corresponding layers as well?       TRUE
SLDVERSION      version of SLD to use if LAYERS=TRUE                 (auto)
PACK            boolean - generate space-efficient XML or JSON?      TRUE
TITLE      T    title of GBT file                                    (auto)
ABSTRACT   A    abstract of GBT file                                 (auto)
OUTFILE    O    path of GBT file to generate                         (auto)
RESOLVEREF R    resolve references to other features                 (FALSE)
REFLEVEL   RL   number of level of indirection                       (1)
ZIPCOMMAND      full path to the zip command                         (auto)
ZIPLEVEL        compression level (0=none ... 9=max)                 6
VERBOSITY  VB   verbosity level: 0=quiet, 1=normal, 2+=more detail   1
CHECKONLY       boolean - don't export; merely do integrity checks   FALSE
LOGFILE    LOG  file to write logging information to                 (none)
(also: convert-library hints and XML-generation hints)
```

The following table defines the parameters relevant to the work done in Testbed 17:

Table 4

PARAMETER NAME	DESCRIPTION
FSET	a comma-separated list of feature collections to copy to the offline container
SQLFILTER	a scalar SQL filter expression for defining a subset of features to copy to the offline container
WINDOW	equivalent to the `bbox` parameter
WINDOWCS	equivalent to the `bbox-crs` parameter
FROMDATE/ TODATE	equivalent to the `datetime` parameter

PARAMETER NAME	DESCRIPTION
OUTFILE	name of the offline container

```
% gbtexport foundation/foundation fset=aerofacp_1m,coastl_1m
```

Simple invocation of the `gbtexport` utility

6.12.9. Future work

The following topics could be addressed in future initiatives:

- The work in Testbed 17 was limited to features. However, there is no reason why other OGC resource types (e.g. map or vector tiles) could not be included in the offline container as well.
- Link resolution is limited to relationships between features. Links to other resources (e.g. code lists) could also be handled.
- Testing on a mobile device.

7

IMPLEMENTATIONS

7 IMPLEMENTATIONS

This chapter describes the server and client implementations.

7.1. Overview

Three server and three client implementations with support for JSON-FG extensions were implemented as deliverables in Testbed-17.

Servers:

- D100 Features and Geometries JSON Server for Aviation (interactive instruments)
- D101 Features and Geometries JSON Server for Aviation (Skymantics)
- D115 Features and Geometries JSON Server (Cubewerx)

Clients:

- D102 Features and Geometries JSON Client for Aviation (Hexagon)
- D103 Features and Geometries JSON Client for Aviation (Ecere)
- D116 Features and Geometries JSON Client (GeoSolutions)

Ecere also implemented support for JSON-FG in its server component as an in-kind contribution.

Each component implementation is described in a section in this chapter. In addition to the description of the component and any issues encountered or lessons learned during the implementation, each server section also describes the JSON-FG options that are supported with examples. Each client section also describes the JSON-FG options that are supported and used by the client as well as the TIEs executed with the server components.

7.2. Technology Integration Experiments (TIEs)

The following eight functional tests were executed between each client-server pair. Two of the tests were optional, depending on the capabilities of the clients and servers.

Table 5

#	FUNCTION	DESCRIPTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
1	Landing Page	Request, receive, parse landing page	Form landing page request, parse response, display and/or act on it	Receive landing page request, response with accepted / requested format (f= json, html)	Client displays API "data" link and/or navigates to it
2	Collections	Request, receive, parse array of collection objects in Collections document	Follow "data" link in landing page and receive / parse / display Collections document	Receive URL dereference from client, return Collections document	Client displays in menu and/ or map form the Collections document and constituent collections
3	Detect JSON-FG	Negotiate JSON-FG media type for the items resource ("rel" : "items"), either by selecting a link with "type" set to `application/vnd.ogc.fg+json`, or using HTTP content negotiation mechanism.	Follow/detect JSON-FG "items" link on the Collections page	Provide the link for JSON-FG items in the Collections endpoint	Client is able to select a JSON-FG representation
4	CoordRefSys	Client is able to parse the new property `coord-ref-sys` inside the JSON-FG collection	The property `coord-ref-sys` should be taken into account in the rendering/ visualization process once the JSON-FG collection is imported in the client	Add the `coord-ref-sys` property inside the collection when required	Client is able to parse the coord-ref-sys property
5	Where	Client is able to parse the new property `where` inside the JSON-FG feature, or the equivalent GeoJSON `geometry` when a `where` property is not useful (i.e.	The property `where` (or `geometry`) should be taken into account in the rendering/ visualization process once each feature of the JSON-FG collection is imported in the client	Add the `where` property inside the feature when applicable	Client is able to decide how to use the `where` (or `geometry`) property based on its own rendering environment

#	FUNCTION	DESCRIPTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
		geometry types supported by Geo JSON in CRS84)			
6	When	Client is able to parse the new property `when` inside the JSON-FG feature	The property `when` should be taken into account in the rendering/visualization process once each feature of the JSON-FG collection is imported in the client	Add the `when` property inside the feature if the feature has information related to date/time	Client is able to read the `when` and render it (some examples: filter client side, filter server side, display when values in a chart, display the when values in a json, ...)
7	Schemas / Metadata (optional)	Client uses one of the endpoints that provides information about the attributes and use them in the rendering phase (queryables, schema-item, schema-collection)	Client reads the attributes information from one of the provided endpoints	Server exposes endpoints related to attributes	Client is able to use the attributes value in the rendering phase (some examples: use attribute as upper lower limit of the 3d volume, use the attributes to apply a style, ...)
8	CRS (optional)	Collect all the available CRSs related to a selected collection	Parse all the available CRSs of the collection	Provide available CRSs for the collections	Client is able to parse/collect all the available CRSs

All mandatory TIEs were successfully completed. The following table provides an overview of the executed and successful TIEs:

Table 6

SERVER / CLIENT	D116 GEOSOLUTIONS	D103 ECERE	D102 HEXAGON
D100 interactive instruments	8/8	8/8	6/8
D101 Skymantics	8/8	8/8	6/8
D115 CubeWerx	8/8	8/8	6/8

GeoSolutions also demonstrated a successful TIE visualizing JSON-FG provided by the Ecere server.

More details about the TIE results for each client are included in the description of each client.

7.3. D100 Features and Geometries JSON Server for Aviation (interactive instruments)

Deliverable D100 was implemented by interactive instruments. The component also represents deliverable D104 from the Testbed-17 Aviation API task. The component and its four APIs are described in detail in the Aviation API Engineering Report [4]. This section only describes the JSON-FG extensions implemented by the server.

A JSON-FG document can be requested from the server using the media type `application/vnd.ogc.fg+json` or the `f` query parameter with the value “jsonfg”.

The following figure shows the HTML representation of a Notice to Airmen (NOTAM) and the corresponding JSON-FG representation.

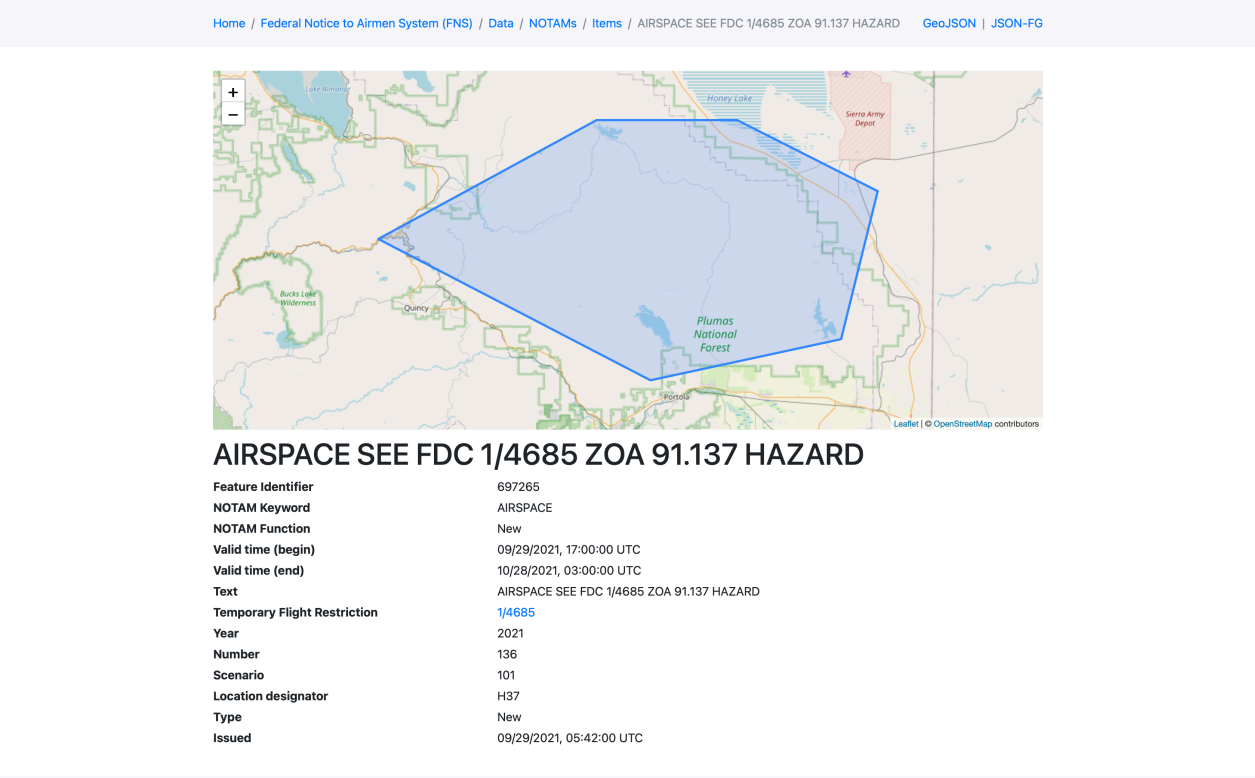

Figure 1 — D100 NOTAM API — HTML output of a NOTAM

Example — JSON-FG representation of the NOTAM:

```
{
   "type":"Feature",
   "featureType":"aixm:Event",
   "id":697265,
   "when":{
      "interval":[
```

```
      "2021-09-29T17:00:00Z",
      "2021-10-28T03:00:00Z"
   ]
},
"coord-ref-sys":"http://www.opengis.net/def/crs/EPSG/0/3857",
"where":{
   "type":"Polygon",
   "coordinates":[
      [
         [
            -13398228.38,
            4895048.44
         ],
         [
            -13426985.91,
            4895048.44
         ],
         [
            -13471513.71,
            4870787.37
         ],
         [
            -13415853.97,
            4841752.21
         ],
         [
            -13376892.14,
            4850212.04
         ],
         [
            -13369470.84,
            4880484.67
         ],
         [
            -13398228.38,
            4895048.44
         ]
      ]
   ]
},
"geometry":{
   "type":"Polygon",
   "coordinates":[
      [
         [
            -120.3583333,
            40.2
         ],
         [
            -120.6166667,
            40.2
         ],
         [
            -121.0166667,
            40.0333333
         ],
         [
            -120.5166667,
            39.8333333
         ],
         [
            -120.1666667,
            39.8916667
```

```
            ],
            [
               -120.1,
               40.1
            ],
            [
               -120.3583333,
               40.2
            ]
         ]
      ]
   },
   "properties":{
      "notam_keyword":"AIRSPACE",
      "notam_function":"NOTAMN",
      "valid_time_begin":"2021-09-29T17:00:00Z",
      "valid_time_end":"2021-10-28T03:00:00Z",
      "text":"AIRSPACE SEE FDC 1/4685 ZOA 91.137 HAZARD",
      "tfr":"1/4685",
      "year":"2021",
      "number":136,
      "scenario":"101",
      "location":"H37",
      "type":"N",
      "issued":"2021-09-29T05:42:00Z"
   },
   "links":[
      {
         "href":"https://t17.ldproxy.net/fns/collections/notam/items/697265?
crs=http%3A%2F%2Fwww.opengis.net%2Fdef%2Fcrs%2FEPSG%2F0%2F3857&f=jsonfg",
         "rel":"self",
         "type":"application/vnd.ogc.fg+json",
         "title":"This document"
      },
      {
         "href":"https://t17.ldproxy.net/fns/collections/notam/items/697265?
crs=http%3A%2F%2Fwww.opengis.net%2Fdef%2Fcrs%2FEPSG%2F0%2F3857&f=json",
         "rel":"alternate",
         "type":"application/geo+json",
         "title":"This document as GeoJSON"
      },
      {
         "href":"https://t17.ldproxy.net/fns/collections/notam/items/697265?
crs=http%3A%2F%2Fwww.opengis.net%2Fdef%2Fcrs%2FEPSG%2F0%2F3857&f=html",
         "rel":"alternate",
         "type":"text/html",
         "title":"This document as HTML"
      },
      {
         "href":"https://t17.ldproxy.net/fns/collections/notam?f=json",
         "rel":"collection",
         "type":"application/json",
         "title":"The collection the feature belongs to"
      },
      {
         "href":"https://t17.ldproxy.net/fns/collections/notam/schemas/
feature",
         "rel":"describedby",
         "type":"application/schema+json",
         "title":"Schema of features in 'NOTAMs'"
      },
      {
         "href":"https://geojson.org/schema/Feature.json",
```

```
            "rel":"describedby",
            "type":"application/schema+json",
            "title":"This document is a GeoJSON Feature"
         }
      ]
}
```

The following sections describe to what extent the implementation supports the JSON-FG capabilities described in the previous chapter.

7.3.1. Identifying the feature type

The source data is Aeronautical Information Exchange Model (AIXM) 5.1 and in the AIXM data model the feature is of type Event. This is expressed using the member “featureType”.

```
...,
"featureType":"aixm:Event",
...
```

AIXM does not provide URIs for its feature types. Therefore, no link with link relation “type” has been added to the links array.

7.3.2. Identifying the schema

The schema links identify the feature as a GeoJSON feature and link to the schema that describes the feature properties.

NOTE: At the moment, ldproxy will generate a schema for the GeoJSON representation of the feature or collection. The server still needs to be updated to generate a schema for the JSON-FG representation.

```
...,
"links":[
   ...,
   {
      "href":"https://t17.ldproxy.net/fns/collections/notam/schemas/feature",
      "rel":"describedby",
      "type":"application/schema+json",
      "title":"Schema of features in 'NOTAMs'"
   },
   {
      "href":"https://geojson.org/schema/Feature.json",
      "rel":"describedby",
      "type":"application/schema+json",
      "title":"This document is a GeoJSON Feature"
   }
]
```

7.3.3. Encoding the primary temporal extent

The temporal validity of the NOTAM is included in the top-level “where” member. Clients do not have to understand the feature schema to determine the main temporal information about the NOTAM.

```
...,
"when":{
   "interval":[
      "2021-09-29T17:00:00Z",
      "2021-10-28T03:00:00Z"
   ]
},
...
```

7.3.4. Encoding the primary spatial geometry including the coordinate reference system

Since the geometry was requested in Web Mercator, the “coord-ref-sys” member is set to the canonical URI for the CRS and the geometry is in the “where” member with the coordinates in Web Mercator.

NOTE: The key of the JSON member to indicate the coordinate reference system was changed late in the testbed from “coord-ref-sys” to “coordRefSys”. To avoid breaking clients, the original key was kept for the D100 server. The software ldproxy has been updated to also provide the new key in other deployments. The support for the “coord-ref-sys” key will be removed in a future version.

To support GeoJSON clients that expect WGS 84 coordinates, the geometry is also included in the “geometry” member from GeoJSON with coordinates in WGS 84.

```
...,
"coord-ref-sys":"http://www.opengis.net/def/crs/EPSG/0/3857",
"where":{
   "type":"Polygon",
   "coordinates":[
      [
         [
            -13398228.38,
            4895048.44
         ],
         [
            -13426985.91,
            4895048.44
         ],
         [
            -13471513.71,
            4870787.37
         ],
         [
            -13415853.97,
            4841752.21
         ],
         [
```

```
            -13376892.14,
            4850212.04
         ],
         [
            -13369470.84,
            4880484.67
         ],
         [
            -13398228.38,
            4895048.44
         ]
      ]
   ]
},
"geometry":{
   "type":"Polygon",
   "coordinates":[
      [
         [
            -120.3583333,
            40.2
         ],
         [
            -120.6166667,
            40.2
         ],
         [
            -121.0166667,
            40.0333333
         ],
         [
            -120.5166667,
            39.8333333
         ],
         [
            -120.1666667,
            39.8916667
         ],
         [
            -120.1,
            40.1
         ],
         [
            -120.3583333,
            40.2
         ]
      ]
   ]
},
...
```

7.3.5. Issues encountered

No real issues were encountered when implementing the extensions. The TIEs with clients helped to quickly identify and correct bugs or deviations from the specification.

7.3.6. Lessons learned

Adding the additional capabilities to the code generating the JSON feature output was straightforward.

However, it is worth pointing out the iterations with respect to how the type of a feature type is identified. Identifying the feature type of a feature was a capability that was requested early in the development to support clients to render the features on a map with the styling rules appropriate for the feature.

The first approach was to leverage JSON-LD and the https://semantics.aero vocabularies:

```
{
  "type": "Feature",
  "@context": "https://t17.ldproxy.net/airspace/collections/class_b/context",
  "@type": [
    "geojson:Feature",
    "ac:class-b"
  ],
  "@id": "https://t17.ldproxy.net/airspace/collections/class_b/items/9",
  "id": 9,
  "geometry": {},
  "properties": {}
}
```

This approach was considered insufficient for the client requirements. This is because the approach only identifies the general semantic type of the feature in some vocabulary, but not the feature type in the AIXM application schema that the data conforms to and that specifies the feature properties.

To address the requirement, “aixm:Airspace” was added to the “@type” array. One issue with this approach is that there is no URI for the AIXM feature types and, therefore, no valid value for the “aixm” base URI. As a stopgap it was set to `"aixm":"http://www.aixm.aero/schema/5.1#"` using the XML schema namespace, but the resulting URIs are not de-referenceable and the definition of “aixm:Airspace” cannot be determined by clients.

As a result of these insufficiencies and the design goal to avoid normative dependencies to JSON-LD, the approach specified in section Clause 6.6 was used and implemented.

7.4. D101 Features and Geometries JSON Server for Aviation (Skymantics)

Skymantics deployed three different use cases using JSON-FG and OGC API — Features that are described in detail below:

1. Unmanned Aircraft System (UAS) geofencing at airports

2. Air routes

3. Event agenda in Madrid

7.4.1. UAS geofencing at airports

7.4.1.1. Problem statement

UAS Traffic Management (UTM) and UTM Service Suppliers (USS) interoperate to coordinate drone operations through the Discovery and Synchronization Service (DSS). UTM geospatial data is currently encoded in non-standard JSON format. GeoJSON was evaluated but discarded for not adapting well enough to requirements, mostly because geometry options in GeoJSON are too restricted. However, the current JSON format is not fully satisfactory either, as it does not provide flexibility to combine area-based and trajectory-based operational intents.

Why an airport environment?

- Airports have an urgent need for safe UAS operations, and are probably the most complex type of airspace.
- Missions in airport environments are subject to communication link loss due to channel congestion and interference with Communications, navigation and surveillance (CNS) systems.

This use case is defined as a set of real-world scenarios validated by sponsor. A top-down approach is followed to go from operational to technical.

NOTE: UTM is not a “free flight” environment where drones deconflict directly between them. Deconfliction is instead applied strategically by USS.

7.4.1.2. Geofencing concept

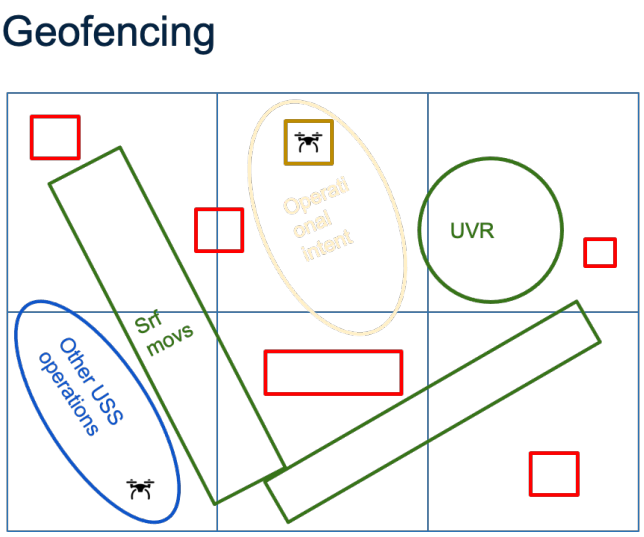

Figure 2 — The concept of geofencing

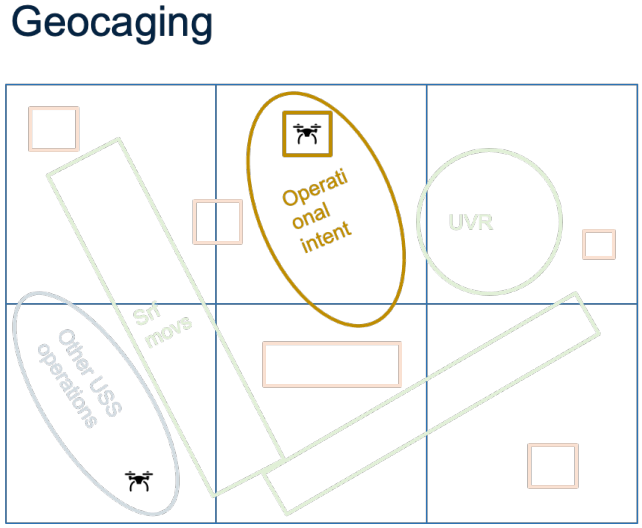

Figure 3 — The concept of geocaging

This use case implements the concept of geofencing, in which the UAS operator receives information on the areas or volumes that should not be entered (obstacles, restrictions, moving objects) within a wider operational area. This is a different concept from geocaging, in which the UAS operator receives information on just the area or volume where it can freely operate.

7.4.1.3. Baseline scenario

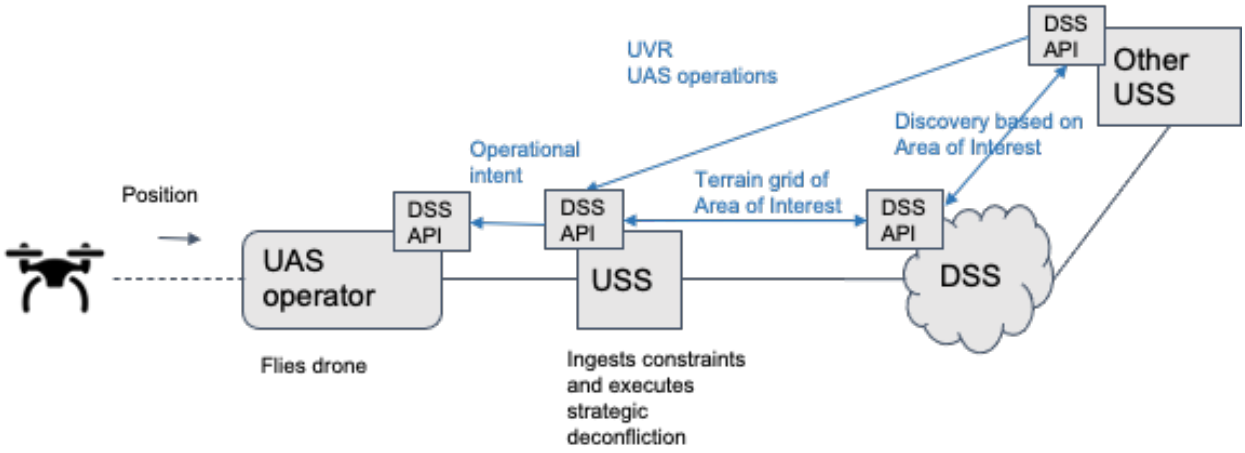

Figure 4 — UAS geofencing baseline scenario

How UTM works today for constraint management and strategic deconfliction:

- According to the Area of Interest of the UAS operator, USS makes a request through DSS for operational intents already approved by other USSs in that Area of Interest. DSS provides endpoints for relevant USSs.
- USSs provide already approved operational intents, off-nominal UAS operations, and constraints. Constraints are provided in the form of a UAS Volume Restriction (UVR). With the received ensemble of operations and constraints from other USSs, the USS generates the approved operational intent for the UAS operator. Operator will be required to fly within these boundaries.

7.4.1.4. Additional datasets

Other than data through DSS, other entities provide other constraints:

- Using an Airport Mapping Database (AMDB) dataset with information on buildings can act as additional flight restrictions to UAS.
- Through D107's FAA System Wide Information Management (SWIM) façade, when a flight is scheduled for landing or takeoff, a restriction is created around the airport's runways, to avoid UAS interfering in the aircraft trajectory.

The airport selected to implement the use case is the Memphis International Airport (KMEM), with synthetic DSS operations and restrictions, airport buildings and flight constraints generated based on real-time flight plans from SWIM Flight Data Publication Service (SFDPS).

7.4.1.5. DSS operations and restrictions dataset

These features can be accessed in KMEM dss collection.

The original data's schema is described in detail in the ASTM UTM protocol github page.

The following operations and restrictions were created following the DSS specification:

- Aircraft maintenance operations at the gates (defined in JSON as circle+radius);
- Forest Hill natural area surveillance (defined in JSON as polygon);
- Cargo delivery between out-of-airport industry park and in-airport cargo area (defined in JSON as combination of two circle+radius and a polygon); and
- Urgent delivery between airport and Methodist University Hospital in downtown Memphis (defined in JSON as combination of two circle+radius and two polygon).

These operations and restrictions were transformed first to GeoJSON and then to JSON-FG, and published using a pygeoapi instance's OGC API- Features. In order to keep temporal extents updated and comparable with other datasets, a script is executed daily to push the date to the following day.

Here is an example of the original DSS data:


```
{
  "operation": {
    "reference": {
      "id": "gate_operation1",
      "owner": "uss1",
      "uss_availability": "Unknown",
      "version": 1,
      "state": "Accepted",
      "ovn": "9d158f59-80b7-4c11-9c0c-8a2b4d936b2d",
      "time_start": {
        "value": "2021-06-01T18:00:00.00Z",
        "format": "RFC3339"
      },
      "time_end": {
        "value": "2021-06-01T19:00:00.00Z",
        "format": "RFC3339"
      },
      "uss_base_url": "https://utm_uss.com/utm",
      "subscription_id": "sid1"
    },
    "details": {
      "volumes": [
        {
          "volume": {
            "outline_circle": {
              "center": {
                "lng": -89.979,
                "lat": 35.0448
              },
              "radius": {
```

```
                "value": 150,
                "units": "M"
              }
            },
            "altitude_lower": {
              "value": 104,
              "reference": "W84",
              "units": "M"
            },
            "altitude_upper": {
              "value": 204,
              "reference": "W84",
              "units": "M"
            }
          },
          "time_start": {
            "value": "2021-06-01T18:00:00.00Z",
            "format": "RFC3339"
          },
          "time_end": {
            "value": "2021-06-01T19:00:00.00Z",
            "format": "RFC3339"
          }
        }
      ],
      "priority": 0
    }
  }
}
```

The transformation to JSON-FG required the following changes:

1. Geometry changed from being defined as a circle with center and radius to a multi-sided polygon. Although the volume of data required to express this geometry was much longer, the feature could immediately be rendered in any GeoJSON client.

2. Volume was encoded as a polyhedron in the `where` member and could be assessed and rendered as such by compatible clients. This was at the cost of adding even more data to the geometry definition.

3. Operations and restrictions time extents could be encoded in a standard feature member (`when`). Other non-DSS restrictions could use the same member, making it easier to compare volumes and their time validity. Besides, time format is not required any longer as it is specified by JSON-FG.

4. CRS does not need to be specified any longer as long as it is WGS84 (as in our case).

5. Altitude boundaries are embedded in the geometry (`where`) and do not need to be specified in the properties.

6. Type specification encoded in the attribute name (`operation` or `restriction`) can be specified in the `featureType` member. By mapping the `featureType` member to `@type` the feature definition can be linked to a dictionary.

7. Definition of a JSON schema for DSS operations and restrictions, describing in detail the structure and semantics of the JSON-FG features in the collection.

This JSON schema is provided as a link together with the more generic GeoJSON feature schema, as each feature complies with both schemas.

The previous DSS example would have the following encoding in JSON-FG:

```
{
    "@context": {
        "geojson": "https://purl.org/geojson/vocab#",
        "dss": "https://redocly.github.io/redoc/?url=https://raw.
githubusercontent.com/astm-utm/Protocol/master/utm.yaml",
        "featureType": "@type"
    },
    "type": "Feature",
    "coord-ref-sys": "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
    "geometry": {
        "type": "Polygon",
        "coordinates": [ ... ]
    },
    "id": "gate_operation1",
    "featureType": "dss:operation"
    "properties": {
        "owner": "uss1",
        "uss_availability": "Unknown",
        "version": 1,
        "state": "Accepted",
        "ovn": "9d158f59-80b7-4c11-9c0c-8a2b4d936b2d",
        "uss_base_url": "https://utm_uss.com/utm",
        "subscription_id": "sid1",
        "altitude_lower": 104,
        "altitude_upper": 204,
        "priority": 0
    },
    "when": {
        "interval": [
            "2021-10-06T18:00:00.000000Z",
            "2021-10-06T19:00:00.000000Z"
        ]
    },
    "where": {
        "type": "Polyhedron",
        "coordinates": [ ... ]
    },
    "links": [
        {
            "rel": "alternate",
            "type": "application/geo+json",
            "title": "This document as GeoJSON",
            "href": "https://fgjson.skymantics.com/collections/KMEM_dss/items/
gate_operation1?f=json"
        },
        {
            "rel": "self",
            "type": "application/vnd.ogc.fg+json",
            "title": "This document as JSON-FG",
            "href": "https://fgjson.skymantics.com/collections/KMEM_dss/items/
gate_operation1?f=jsonfg"
        },
        {
            "rel": "alternate",
            "type": "application/ld+json",
            "title": "This document as RDF (JSON-LD)",
```

```
            "href": "https://fgjson.skymantics.com/collections/KMEM_dss/items/
gate_operation1?f=jsonld"
        },
        {
            "rel": "alternate",
            "type": "text/html",
            "title": "This document as HTML",
            "href": "https://fgjson.skymantics.com/collections/KMEM_dss/items/
gate_operation1?f=html"
        },
        {
            "rel": "collection",
            "type": "application/json",
            "title": "KMEM DSS operations and restrictions",
            "href": "https://fgjson.skymantics.com/collections/KMEM_dss"
        },
        {
            "type": "application/schema+json",
            "title": "Schema of features in KMEM_dss",
            "rel": "describedby",
            "href": "https://fgjson.skymantics.com/schemas/feature/KMEM_dss.
json"
        },
        {
            "type": "application/schema+json",
            "title": "This document is a GeoJSON Feature",
            "rel": "describedby",
            "href": "https://geojson.org/schema/Feature.json"
        }
    ]
}
```


7.4.1.6. KMEM buildings dataset

Building features can be accessed from the KMEM buildings collection.

The original data were provided by Performance Software and initially extracted from Aerodrome Mapping Exchange Model (AMXM)-formatted datasets and transformed to GeoJSON. They include the blueprints of all the buildings at the Memphis airport with height and elevation information. Elevation was described by the data provider as the orthometric height at the top and at the base of the object, respectively, in meters and formatted in the NAVD88 datum. The dataset is static, with no variation in time and all coordinate data are in WGS84.

The GeoJSON features do not provide a schema or a definition, which needs to be checked from an XML schema describing the AMXM format. Information on how the altitudes and elevations were calculated is not provided, and it needs to be queried directly from the provider.

The transformation to JSON-FG implies the following changes:

1. Geometry is extended to model the volume occupied by the buildings. The original `geometry` member is kept untouched in order to maintain backward compatibility with GeoJSON clients, and a new `where` member is added with the volume encoded as a polyhedron. This volume is generated by extruding the buildings blueprints between the buildings' elevations and their altitudes.

2. `featureType` is used to define the type of building is being modelled, such as Terminal, Hangar or Control Tower for example. The value of featureType includes the context (for example, `amxm:hangar`), which together with the aliasing of featureType as `@type`, allows for the linking to a AMXM dictionary describing the semantics of the feature (although in XSD format). This capability facilitates the rendering of the building in clients.

3. A JSON schema is provided as a link in all features to describe their structure and the semantics of properties and members. A link to a generic GeoJSON schema is also provided.

4. An additional nested property called `containedInPlace` is added following schema.org standards and providing a link to the airport feature stored in a separate collection.

Here is an example of the control tower feature in KMEM airport encoded as JSON-FG:

```
{
    "@context": {
        "schema": "https://schema.org/",
        "geojson": "https://purl.org/geojson/vocab#",
        "amxm": "https://www.amxm.aero/schema/2.0.1/amxm.xsd",
        "featureType": "@type"
    },
    "type": "Feature",
    "id": "3",
    "featureType": "amxm:ControlTower",
    "coord-ref-sys": "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
    "geometry": {
        "type": "Polygon",
        "coordinates": [ ... ]

    },
    "properties": {
        "icaoCode": "KMEM",
        "name": "TWR",
        "height": 189,
        "elevation": 85.5,
        "horizontalAccuracy": 0.99,
        "revisionDate": 20210708,
        "containedInPlace": {
            "type": "schema:Airport",
            "id": "KMEM",
            "url": "https://aviationapi.skymantics.com/collections/airports/
items/KMEM"
        }
    },
    "where": {
        "type": "Polyhedron",
        "coordinates": [ ... ]
    },
    "links": [
        {
            "rel": "alternate",
            "type": "application/geo+json",
            "title": "This document as GeoJSON",
            "href": "https://fgjson.skymantics.com/collections/KMEM_buildings/
items/3?f=json"
        },
```

```
        {
            "rel": "self",
            "type": "application/vnd.ogc.fg+json",
            "title": "This document as JSON-FG",
            "href": "https://fgjson.skymantics.com/collections/KMEM_buildings/
items/3?f=jsonfg"
        },
        {
            "rel": "alternate",
            "type": "application/ld+json",
            "title": "This document as RDF (JSON-LD)",
            "href": "https://fgjson.skymantics.com/collections/KMEM_buildings/
items/3?f=jsonld"
        },
        {
            "rel": "alternate",
            "type": "text/html",
            "title": "This document as HTML",
            "href": "https://fgjson.skymantics.com/collections/KMEM_buildings/
items/3?f=html"
        },
        {
            "rel": "collection",
            "type": "application/json",
            "title": "KMEM buildings with heights",
            "href": "https://fgjson.skymantics.com/collections/KMEM_buildings"
        },
        {
            "type": "application/schema+json",
            "title": "Schema of features in KMEM_buildings",
            "rel": "describedby",
            "href": "https://fgjson.skymantics.com/schemas/feature/KMEM_
buildings.json"
        },
        {
            "type": "application/schema+json",
            "title": "This document is a GeoJSON Feature",
            "rel": "describedby",
            "href": "https://geojson.org/schema/Feature.json"
        }
    ]
}
```


7.4.1.7. KMEM flight restrictions dataset

These features can be accessed in the KMEM constraints collection.

SWIM flight plans are recorded in the FAA façade developed in the Testbed-17 Aviation API task. When a new flight plan leaving from or arriving to KMEM airport is received, a constraint is created around the airport runways during the takeoff or landing to avoid collisions with UAS.

These restrictions are encoded in the same way as those from the DSS operations and restrictions dataset, reusing the same JSON schema, and make the most of the `when` and `where` members to accurately define the space and time boundaries of the restriction.

7.4.2. Air routes

Thanks to the work being developed in the Testbed-17 Aviation API task, Skymantics had access to FAA and EUROCONTROL flight plans. Among the different attributes provided by EUROCONTROL services, the Filed Tactical Flight Model point profile describes in detail the planned trajectory of the flight and it has interesting similarities with the draft Route Exchange Model (REM) specification being defined in the OGC Routing SWG.

The Filed Tactical Flight Model point profile is the product of a fusion service that reflects the latest filed flight plan but updated with the latest information provided by Collaborative Decision Making systems and READY messages or amended by NM OPS room. The structure of the Filed Tactical Flight Model point profile is similar to the REM, with start and end points, and a series of intermediate route segments describing location, time, altitude, trend and covered distance. The REM is being developed with road transportation in mind, but it is expected to be adapted support encoding to other transportation modes in the long term.

Three main questions naturally raised in this situation:

1. How well does REM adapt to air routes?
2. Does JSON-FG provide valuable capabilities to REM? Should REM consider embracing JSON-FG?
3. Can API — Routes and REM be used in the aviation domain?

One EUROCONTROL flight (AT03477905, from Helsinki airport to New York's JFK airport) was selected to implement its trajectory using REM with JSON-FG capabilities and evaluate the process and results. This air route can be accessed in KMEM constraints, published in an OGC API — Features instance. The following JSON-FG code is a summary of the encoded flight air route:

```
{
    "type": "FeatureCollection",
    "name": "Flight AT02776877",
    "status": "successful",
    "links": [ ... ],
    "features": [
        {
            "type": "Feature",
            "id": 1,
            "geometry": {
                "type": "LineString",
                "coordinates": [ ... ]
            },
            "when": {
                "interval": [
                    "2021-09-06T14:39:00Z",
                    "2021-09-06T22:51:05Z"
                ]
            },
            "properties": {
                "length_m": 3617000,
                "duration_s": 29525.0
```

```
        },
        "featureType": "airroute:route overview"
    },
    {
        "type": "Feature",
        "id": 2,
        "geometry": {
            "type": "Point",
            "coordinates": [ ... ]
        },
        "when": {
            "instant": "2021-09-06T14:39:00Z"
        },
        "properties": {
            "aerodrome": "EFHK",
            "procedure": {
                "SID": {
                    "id": "NEPEK3N",
                    "aerodromeId": "EFHK"
                }
            }
        },
        "featureType": "airroute:start"
    },
    {
        "type": "Feature",
        "id": 3,
        "geometry": {
            "type": "Point",
            "coordinates": [ ... ]
        },
        "when": {
            "instant": "2021-09-06T14:46:32Z"
        },
        "properties": {
            "length_m": 35000,
            "duration_s": 452.0,
            "flight_level": "F158",
            "entry_trend": "CLIMB",
            "exit_trend": "CLIMB",
            "route": "Y357"
        },
        "featureType": "airroute:segment"
    },
    {
        "type": "Feature",
        "id": 4,
        "geometry": {
            "type": "Point",
            "coordinates": [ ... ]
        },
        "when": {
            "instant": "2021-09-06T14:53:08Z"
        },
        "properties": {
            "length_m": 44000,
            "duration_s": 396.0,
            "flight_level": "F256",
            "entry_trend": "CLIMB",
            "exit_trend": "CLIMB",
            "procedure": {
                "DCT": null
            }
```

```
            },
            "featureType": "airroute:segment"
        },
...
        {
            "type": "Feature",
            "id": 77,
            "geometry": {
                "type": "Point",
                "coordinates": [ ... ]
            },
            "when": {
                "instant": "2021-09-06T19:34:06Z"
            },
            "properties": {
                "length_m": 41000,
                "duration_s": 332.0,
                "flight_level": "F400",
                "entry_trend": "CRUISE",
                "exit_trend": "CRUISE",
                "procedure": {
                    "DCT": null
                }
            },
            "featureType": "airroute:segment"
        },
...
        {
            "type": "Feature",
            "id": 97,
            "geometry": {
                "type": "Point",
                "coordinates": [ ... ]
            },
            "when": {
                "instant": "2021-09-06T22:38:00Z"
            },
            "properties": {
                "length_m": 55000,
                "duration_s": 525.0,
                "flight_level": "F156",
                "entry_trend": "DESCENT",
                "exit_trend": "DESCENT",
                "procedure": {
                    "DCT": null
                }
            },
            "featureType": "airroute:segment"
        },
        {
            "type": "Feature",
            "id": 98,
            "geometry": {
                "type": "Point",
                "coordinates": [ ... ]
            },
            "when": {
                "instant": "2021-09-06T22:51:05Z"
            },
            "properties": {
                "aerodrome": "KJFK",
                "procedure": {
                    "DCT": null
```

```
                }
            },
            "featureType": "airroute:end"
        }
    ],
    "coord-ref-sys": "http://www.opengis.net/def/crs/OGC/1.3/CRS84",
    "timeStamp": "2021-10-12T17:35:56.835711Z"
}
```

The structure of the air route follows the same structure as the REM, with a route overview, a start point (departure airport), a list of route segments and an end point (arrival airport). The geometries used in the REM fit naturally with the needs of encoding an air route, with a LineString depicting the trajectory, and a series of Points indicating the pivoting moments of the route. Some properties such as `length_m` and `duration_s` are consistent with the draft REM specification and can be used in an air route as well. However, other properties such as `maxHeight_m`, `roadName`, `speedLimit` or `instruction` are specific for road transportation and need to be replaced by others required in air transportation, such as `flight_level`, `entry_trend` or `procedure`. Start and end points are usually specific aerodromes instead of just coordinates in a map. These start and end points could provide a link to airport features stored in external collections, for example with https://aviationapi.skymantics.com/eurocontrol/collections/airports/items/EFHK, allowing for additional information on these aerodromes.

The JSON-FG `when` member becomes very useful for encoding time-related properties in routes in a standard way. For starters, it specifies the estimated duration (interval) of the route in the route overview. It also specifies departure and arrival times (instants) in the start and end elements. And finally, at the end of each segment, it specifies the moment the flight will cross that point. These encodings can become useful for road (or any other type of) transportation.

The JSON-FG `featureType` member is also very useful, as the list of properties is different for different types of routes, such as in road or air transportation. Having a standard way of defining which type of route is being encoded is useful to specify the types of properties that might be included, while reusing the rest of the REM format. Besides, `featureType` can be complemented by contexts and dictionaries or by JSON schemas describing the format in detail.

7.4.3. Event agenda in Madrid

In order to test the case of non-WGS84 CRSs, the event agenda of Madrid was encoded in JSON-FG. This collection can be accessed in Madrid Agenda.

The original data provides locations in both WGS84 and ETRS89 / UTM zone 30N (EPSG:25830). ETRS89 is the EU-recommended frame of reference for geodata for Europe and is typically found in data provided by official institutions, such as the Madrid municipality in this case. The JSON-FG implementation of these features provides the coordinates in WGS84 in the `geometry` member and in ETRS89 in the `where` member. Events provide a rich variety for temporal extents, encoded partially in the `when` member. Properties have been structured following schema.org standards.

Here is an example of a JSON-FG encoding of an event in Madrid:

```
{
    "@context": {
        "schema": "https://schema.org"
    },
```

```
    "type": "Feature",
    "id": 11216477,
    "featureType": "schema:event",
    "when": {
        "interval": [
            "2021-09-21T00:00:00Z",
            "2021-09-28T23:59:00Z"
        ]
    },
    "coord-ref-sys": "http://www.opengis.net/def/crs/EPSG/9.9.1/25830",
    "where": {
        "type": "Point",
        "coordinates": [ 443307, 4472652 ]
    },
    "geometry": {
        "type": "Point",
        "coordinates": [ -3.66809232, 40.402549314 ]
    },
    "properties": {
        "itemid": 11216477,
        "name": "Los martes de Juntas Emprendemos",
        "offers": {
            "type": "schema:Offers",
            "price": "0",
            "priceCurrency": "EUR"
        },
        "eventtime": "11:00",
        "dayofweek": "M",
        "excludeddays": "8/12/2020;5/1/2021;18/5/2021;",
        "url": "http://www.madrid.es/sites/v/index.jsp?
vgnextchannel=ca9671ee4a9eb410VgnVCM100000171f5a0aRCRD&vgnextoid=
74abfe953fd44710VgnVCM2000001f4a900aRCRD",
        "location": {
            "type": "schema:Place",
            "name": "Espacio de Igualdad Elena Arnedo Soriano. Retiro",
            "address": {
                "type": "schema:PostalAddress",
                "streetAddress": "CALLE ARREGUI Y ARUEJ, 31, RETIRO",
                "postalCode": "28007"
            },
            "url": "http://www.madrid.es/sites/v/index.jsp?
vgnextchannel=bfa48ab43d6bb410VgnVCM100000171f5a0aRCRD&vgnextoid=
cb3655d1b6742610VgnVCM1000001d4a900aRCRD"
        },
        "audience": "Mujeres"
    },
    "links": [ ... ]
}
```


The use of the `where`, `crs`, and `geometry` members to provide the event location in two different CRSs was simple and straightforward. They allowed for native compatibility with GeoJSON clients while providing the location in the official CRS.

The variety of types of events could not be reflected using only the `when` member, requiring the use of additional properties. Only one-time events could be completely defined using an instant. However, recurring events could only define the interval of days since the first occurrence until the last one. These events required the following additional properties:

- Event time

- Day of week
- Excluded days

Several schema.org definitions were used to encode the properties: `event` (specified in the `featureType` member), `offer` (to specify the price for the event), `place` and `postalAddress` (for the event location). Some of these, such as event or place, fit naturally and were easily adopted. On the other hand, the `offer` object was unnecessarily complex and `postalAddress` did not have enough fields and some information was lost.

7.4.4. Issues encountered

pygeoapi is a stable, feature-rich server API but during the implementation of this task, three major issues were found that needed to be addressed:

1. OGC API — Features — Part 2: Coordinate Reference Systems by Reference is not implemented. This affected the lack of means to describe the list of CRSs provided by a collection or to accept the `crs` parameter in queries. Adding these capabilities was not particularly difficult, but it was surprising to find this issue.
2. Lack of support for nested properties. This is a relatively common issue in most implementations, probably due to a perception of being an uncommon need. A conversation with the developer community confirmed this hypothesis. Implementing support for nested properties is not difficult (in this task, it was implemented for datasets stored in GeoJSON files and PostgreSQL databases), although the rendering of features in HTML could need improvement. The main challenge is the filtering of nested properties and their declaration as queryables, in a way that is easily set in a configuration file. The results tested in this task were either complex implementations or too simplistic solutions. Additional discussion within the community is needed.
3. Lack of support for datetime filtering. This is a somewhat expected issue, as features do not have a predefined member to specify time constraints and thus cannot be filtered by datetime. Adding this capability in pygeoapi was not difficult as most of the structure was already in place.

Apart from these issues, adding support for JSON-FG was relatively painless, even as an additional format in parallel to GeoJSON, HTML and JSON-LD.

The implementation of the `where` geometries had two main use cases: Specifying coordinates in a non-WGS84 CRS or specifying new geometries not allowed in GeoJSON, in particular polyhedrons. This second use case raised two issues.

1. Adding polyhedron geometries could increase the size of the features considerably. For example, in the KMEM buildings collection, the use of Polyhedron geometries more than tripled the size of each feature. A similar proportional increase occurred in the KMEM DSS collection.

2. There could be legal compliance issues. One of the reasons why DSS format was not originally encoded in GeoJSON is because UAS operations and restrictions are often defined by a circumference or a circular area. Being forced to transform a circle to a polygon, practical as it might be, makes it non conformant with the norm. Extruding that polygon to a polyhedron does not fix the issue either.

The implementation of the `when` member was found particularly useful as a way to standardize time constraints among different collections. However, in the case of Madrid events several limitations were found. The cases of recurring events could only encode the interval for days when an event would occur, but not the event time, the day of week or the excluded days. Similar limitations would probably be found in other collections with complex time constraints, such as transit timetables.

Several types in schema.org were used to format properties in a standard way without the need of providing additional schemas. Although in collections addressing generic use cases, such as Madrid events, this schema fits relatively well. In many others focused on more particular use cases, such as DSS constraints or airport buildings, schema.org is too limited and needs additional schemas and definitions. The model defined by schema.org is not easily extendable and thus the solution would be to provide several contexts, making the semantics more complex. Even for those collections where schema.org is a natural fit, some properties are unnecessarily complex (for example, the price of an event is a nested property) or too simple (for example, `postalAddress` does not have fields to store the district, suburb or type of street). JSON schemas are a good alternative to JSON-LD contexts and schema.org types.

7.4.5. Lessons learned

Making the conscious decision of ensuring GeoJSON client compatibility with JSON-FG seems the right one. This not only ensures an already working ecosystem for the new format but also that the “upgrade” path for server and client implementations is relatively soft, as proven by the low effort required to add JSON-FG compatibility into `pygeoapi`.

The use of the `when` member can help in scenarios where time-referenced features need to be compared or fused, such as the UAS geofencing use case tested in this task. Having a common placeholder for the time values makes comparisons straightforward for features of different types, and developers do not need to dive into schemas or documentations to find the right properties and formats to make these comparisons. For this reason, the `when` member can be a very interesting addition for routes in the REM, as it can be a simple way to specify trajectories in the fourth dimension. Although this can be useful in any transportation scenario, temporal information can be particularly important for air, railway or urban mass transit, where route segments occupation need to be monitored at every moment and risks of collisions or congestions must be minimized.

As demonstrated in the Madrid events scenario, the `when` member does not cover all the cases that require temporal specification. However, adding additional capabilities might not be a desirable approach for the initial version of JSON-FG. Simplicity is key to ease of adoption, and trying to cover all cases is usually associated with more complex solutions. Further testing in different scenarios could help fine-tune the scope of the `when` member. More detailed temporal constraints, such as those required in the publication of events or bus timetables, could be

added in an extension to the standard, or just defined in a JSON schema linked from the JSON-FG collection.

The `where` member covers two use cases: A GeoJSON geometry published in a CRS different from WGS84 or a geometry not covered by GeoJSON. The first case was simple to implement and clients could easily render the geometries. The second case has the potential of bridging the gap between 3D models and JSON-encoded features. For example, for drone operational use cases such as the one covered in this Testbed-17 task. However, the second case raised several issues, partly because the only defined non-GeoJSON geometry is `Polyhedron`. This geometry considerably increases the size of the JSON-FG data and clients seem to need a non-negligible work to render the content. The issue becomes very relevant in the case of DSS constraints, in which a simple geometry defined by a center, a diameter, and two heights in its original JSON format is transformed to a multi-faced polyhedron, additionally breaking the legal conformance with the norm. This issue could be solved by defining new geometries, or allowing for the definition of new geometries, such as extruded polygons or circles.

The `featureType` member is an elegant way to add some basic semantic meaning to a feature and it can help linking to dictionaries providing additional context. It has proven to be one of the most used JSON-FG additions because of its simplicity and benefits.

JSON schemas might be the JSON-FG addition with the highest potential to trigger adoption in verticals. They are easily understandable by developers used to work with JSON formats and they can facilitate semantic definitions for non-geometry properties. They can help to define specific application formats and, by adding links to more detailed schemas. They can be concatenated in an increasingly detailed manner, in a similar way as a schema.org Place inherits from a Thing type, for example. JSON schemas can potentially turn a generic standard such as JSON-FG into specific vertical standards. JSON schemas can offer a natural and native replacement for adding semantic meaning to features. Further efforts in easing the definition of schemas, providing JSON schema repositories or promoting their potential will be beneficial for the adoption of JSON-FG.

7.5. D115 Features and Geometries JSON Server (Cubewerx)

7.5.1. Deployment environment

Deliverable D115 was implemented by CubeWerx Inc. This OGC API — Features server is a component of CubeWerx Suite 9.3.62. CubeWerx Suite is deployed on a bare-metal server running Fedora 31 and using Oracle 11.2 and MariaDB 10.3 as the backend databases to store feature data.

7.5.2. Datasets

The following datasets were deployed onto the CubeWerx server for use by thread participants:

- Air Spaces
  - FAA controlled airspaces covering different classifications of airspaces.
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/Air%20Spaces
- Arctic SDI
  - Database from the OGC Arctic Spatial Data Pilot.
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/ArcticSDI
- Daraa
  - Daraa dataset from TB16.
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/Daraa
- Foundation
  - VMAP Level 0 base maps.
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/Foundation
- NOTAMs
  - FAA “Notices to Airmen”.
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS
- US Building
  - Building footprints dataset for the United States released by Microsoft (https://github.com/microsoft/USBuildingFootprints).
  - Landing page: https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/US%20Buildings

7.5.3. Service characteristics

The `cubeserv` module implements the following parts of the OGC API Features suite of standards:

- OGC API — Features — Part 1: Core (http://docs.opengeospatial.org/is/17-069r3/17-069r3.html)
- OGC API — Features — Part 2: Coordinate Reference Systems by Reference (http://docs.opengeospatial.org/is/18-058/18-058.html)

NOTE: `cubeserv` implements other parts too but these are the ones that are relevant to this thread.

This baseline implementation was extended during the duration of the thread to support JSON-FG. The details of the implementation are described in the following sections.

7.5.4. Content negotiation

A JSON-FG document can be requested from `cubeserv` using the media type `application/vnd.ogc.fg+json` or the token `jsonfg`. The server supports content negotiation using the `Accept` header as per the HTTP specification. The server also supports the use of the vendor-specific query parameter `f` for specifying the desired output format in embedded URLs. The values for `f` can be the aforementioned media type or token.

```
      |     "number": 80,                                                  |
      |     "scenario": "401",                                             |
      |     "location": "FHAW",                                            |
      |     "icao_location": "FHAW",                                       |
      |     "series": "M",                                                 |
      |     "type": "C",                                                   |
      |     "valid_time_begin": "2021-08-02T09:58:00Z",                    |
      |     "valid_time_end": "2021-08-05T09:58:00Z",                      |
      |     "issued": "2021-08-02T09:58:00Z"                               |
      |   },                                                               |
      |   "links": [                                                       |
      |     {                                                              |
      |       "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/
NOTAMS/collections/notams/schemas/feature?f=json",
      |       "rel": "describedby",                                        |
      |       "type": "application/schema+json"                            |
      |     },                                                             |
      |     .                                                              |
      |     .                                                              |
      |     .                                                              |
      |   ]                                                                |
      | }                                                                  |
      |<-------------------------------------------------------------------+
```


Requesting a JSON-FG response using the `Accept` header

```
https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS/collections/
notams/items/CWFID.NOTAMS.0.33?f=jsonfg
```

Requesting a JSON-FG response using the `f` query parameter

The response in this case would be identical to that shown above.

7.5.5. Encoding the spatial geometry of a feature

If the primary geometry of a feature can be represented as a valid GeoJSON geometry, `cubeserv` encodes the feature's geometry as the value of the GeoJSON `geometry` key and sets the value of JSON-FG `where` key to `null`.

If the primary geometry of a feature cannot be represented as a valid GeoJSON geometry — for example because of the geometry's type or the requested output CRS — `cubeserv` encodes the geometry as the value of the `where` key. If that geometry value can also be transformed into a valid GeoJSON geometry then `cubeserv` performs the transformation and sets the value of the GeoJSON `geometry` accordingly. Otherwise (i.e. no transformation is possible) the `geometry` key is set to `null`. The transformation to a GeoJSON geometry can be a simple CRS transformation but it can also be a simplification of the `where` value or even a dimensionally collapsed version of the `where` value as might be the case when a 3-D `where` value is projected to a 2-D footprint.

The following example shows a NOTAM feature where the primary geometry is in a CRS that cannot be represented using the GeoJSON `geometry` key. In this case the server generates the geometry as the value of the `where` key. The server then transforms the geometry to **CRS84** and uses that transformed value to populate the `geometry` key. In this way both GeoJSON clients and JSON-FG clients can consume the content.

Example — A NOTAM feature with both the `geometry` and `where` keys set.:


```
 {
```

```
   "type": "Feature",
   "featureType": "notams",
   "id": "CWFID.NOTAMS.0.33",
   "coord-ref-sys": "http://www.opengis.net/def/crs/EPSG/0/2955",
   "geometry": {
     "type": "Point",
     "coordinates": [
       -14.39366667,
       -7.969666667
     ]
   },
   "where": {
     "type": "Point",
     "coordinates": [
       13529365.67,
       -16236678.87
     ]
   },
   "when": {
     "interval": [
       "2021-08-02T05:58:00Z",
       "2021-08-05T05:58:00Z"
     ]
   },
   "properties": {
     "notam_function": "NOTAMC",
     "text": "M0080/21 NOTAMC M0079/21 A) FHAW",
     "year": "2021",
     "number": 80,
     "scenario": "401",
     "location": "FHAW",
     "icao_location": "FHAW",
     "series": "M",
     "type": "C",
     "valid_time_begin": "2021-08-02T09:58:00Z",
     "valid_time_end": "2021-08-05T09:58:00Z",
     "issued": "2021-08-02T09:58:00Z"
   },
   "links": [
     {
       "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS/
collections/notams/schemas/feature?f=json",
       "rel": "describedby",
       "type": "application/schema+json"
     },
     .
     .
     .
   ]
 }
```


7.5.6. Encoding the primary temporal extent of a feature

The primary temporal extent is encoded in a JSON-FG document using the `when` key. The following JSON fragment highlights the temporal extent shown in the example from the previous section.

Example — The temporal extent of the feature encoded using the `when` key.:


```
{
```

```
   .
   .
   .
   "when": {
     "interval": [
       "2021-08-02T05:58:00Z",
       "2021-08-05T05:58:00Z"
     ]
   },
   .
   .
   .
 }
```

An internal per-collection `cubeserv` configuration parameter is used to designate which property values of a feature should be used to generate the value for the temporal extent of a feature (i.e. `when`). If the configuration parameter is not set, then the value of the `when` key is set to `null`. In this case, the configuration parameter was set to indicate that the values of the `valid_time_begin` and `valid_time_end` properties should be used to populate the `when` key.

`cubeserv` also implements an *experimental* value of `auto` for the aforementioned internal configuration parameter. A value of `auto` causes the server to use a heuristic algorithm to try and identify properties in the feature that might be used to set the value of the `when` key. This capability was implemented for the testbed but was not tested.

7.5.7. Encoding of reference systems

If a client requests a CRS other than the GeoJSON default of CRS84, `cubeserv` uses the `coord-ref-sys` to asset the CRS being used to encode the value of the `where` key in the response document as illustrated in the following example:

```
   Client                                                              Server
     |                                                                   |
     |   GET /collection/notams/items/CWFIS.NOTAMS.0.33?crs=http://www.opengis.
net/def/crs/EPSG/0/2955    HTTP 1.1
     |   Accept: application/vnd.ogc.fg+json                             |
     +------------------------------------------------------------------>|
     |                                                                   |
     | {                                                                 |
     |   "type": "Feature",                                              |
     |   "featureType": "notams",                                        |
     |   "id": "CWFID.NOTAMS.0.33",                                      |
     |   "coord-ref-sys": "http://www.opengis.net/def/crs/EPSG/0/2955",
     |   "geometry": {                                                   |
     |     "type": "Point",                                              |
     |     "coordinates": [                                              |
     |       -14.39366667,                                               |
     |       -7.969666667                                                |
     |     ]                                                             |
     |   },                                                              |
     |   "where": {                                                      |
     |     "type": "Point",                                              |
     |     "coordinates": [                                              |
     |       13529365.67,                                                |
     |       -16236678.87                                                |
     |     ]                                                             |
     |   },                                                              |
```

```
      |    "when": {                                                    |
      |      "interval": [                                              |
      |        "2021-08-02T05:58:00Z",                                  |
      |        "2021-08-05T05:58:00Z"                                   |
      |      ]                                                          |
      |    },                                                           |
      |    "properties": {                                              |
      |      "notam_function": "NOTAMC",                                |
      |      "text": "M0080/21 NOTAMC M0079/21 A) FHAW",                |
      |      "year": "2021",                                            |
      |      "number": 80,                                              |
      |      "scenario": "401",                                         |
      |      "location": "FHAW",                                        |
      |      "icao_location": "FHAW",                                   |
      |      "series": "M",                                             |
      |      "type": "C",                                               |
      |      "valid_time_begin": "2021-08-02T09:58:00Z",                |
      |      "valid_time_end": "2021-08-05T09:58:00Z",                  |
      |      "issued": "2021-08-02T09:58:00Z"                           |
      |    },                                                           |
      |    "links": [                                                   |
      |    {                                                            |
      |      "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/
NOTAMS/collections/notams/schemas/feature?f=json",
      |      "rel": "describedby",                                      |
      |      "type": "application/schema+json"                          |
      |    },                                                           |
      |    .                                                            |
      |    .                                                            |
      |    .                                                            |
      |    ]                                                            |
      |  }                                                              |
      |<----------------------------------------------------------------+
```

Using the `crs` parameter to request a response in a different CRS.

NOTE: The initial proposal for the name of the key used to assert a CRS in a JSON-FG response document was `coord-ref-sys`. During the course of the testbed, but also fairly late in the process, the JSON-FG SWG decided to rename the key to `coordRefSys` to be consistent with the casing used for other key names. In order to not disrupt thread client development, `cubeserv` continued to use the old name, `coord-ref-sys`, but can be configured to generate `coordRefSys` instead.

7.5.8. Identifying the schema of a feature / feature collection

As illustrated by the following JSON fragment taken from the NOTAM JSON-FG example, the schema of the feature is indicated by a link (*rel=describedby*) in the links section of the feature.

Link pointing to the schema for a feature.:

```
{
  .
  .
  .
  "links": [
    {
      "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS/
collections/notams/schemas/feature?f=json",
      "rel": "describedby",
```

```
      "type": "application/schema+json"
    },
    .
    .
    .
  ]
  .
  .
  .
}
```

The `cubeserv` server also generates a link (rel=describedby) at the collection level of a response.

Link pointing to the schema for the collection.:

```
{
  "type": "FeatureCollection",
  "timeStamp": "2021-10-31T00:47:14-04:00",
  "numberMatched": 262274,
  "numberReturned": 10,
  .
  .
  .
  "links": [
    .
    .
    .
    {
      "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS/
collections/notams/schemas/collection?f=json",
      "rel": "describedby",
      "type": "application/schema+json"
    },
    .
    .
    .
  ],
  "features": [
    .
    .
    .
  ]
}
```

The feature-level schema may be used to validate the feature instance to which it is associated. The collection-level schema may be used to validation a response from the server's features endpoint (i.e. `/items`).

NOTE: At the moment, `cubeserv` will generate a schema for the GeoJSON representation of the feature or collection. The server still needs to be updated to generate a schema for the JSON-FG representation.

7.5.9. Identifying the feature type

Features are often categorized by type which typically means that all instances of the feature have at least a common subset of properties. In JSON-FG a key named `featureType` is used to denote the type of the feature as illustrated in the following JSON-FG fragment.

Example — The type of a feature denoted using the `featureType` key.:

```
{
  "type": "Feature",
  "featureType": "notams",
  "id": "CWFID.NOTAMS.0.33",
  "coord-ref-sys": "http://www.opengis.net/def/crs/EPSG/0/2955",
  .
  .
  .
  "links": [
    {
      "href": "https://test.cubewerx.com/cubewerx/cubeserv/tb17/ogcapi/NOTAMS/
collections/notams/schemas/feature?f=json",
      "rel": "describedby",
      "type": "application/schema+json"
    },
    .
    .
    .
  ]
}
```

Typically, a persistent resource exists that describes the feature type. If such a resource exists JSON-FG recommends that a link (rel=type) be added in the links section of the feature. Since no normative links to feature types accompanied that data are loaded into the CubeWerx server, no such links (rel=type) are generated in the server's responses.

7.5.10. Issues encountered

No issues were encountered implementing the extensions. Any deviations from the specification were identified by client TIEs and resolved.

7.6. D102 Features and Geometries JSON Client for Aviation (Hexagon)

7.6.1. Implementation

The Hexagon Aviation Client was developed using Hexagon's LuciadRIA API. The LuciadRIA API is an easy-to-use library that allows developers to implement web applications that run in any

modern web browser, offering many geospatial capabilities. LuciadRIA supports the visualization of geospatial data either as vector or raster data as 2D and 3D maps.

LuciadRIA supports GeoJSON data decoding. In Testbed-17 and as part of D102 Hexagon implemented a custom decoder that is capable of interpreting geospatial features encoded in JSON-FG and is able to map those features into the LuciadRIA native model to present the features on a map. The features are styled according to a predefined style that matches the feature type.

7.6.2. Client

The Hexagon aviation client supports JSON-FG elements as detailed in the TIE table below. As part of the implementation, Hexagon added support for CRS's other than CRS84. Support for 'where' and 'when' properties was also added. Support of the 'geometry' property is kept for backwards compatibility.

Hexagon did not include support for polyhedron geometries as part of Testbed-17, but it is considered for a future implementation. The geometries supported were the standard GeoJSON geometries.

7.6.3. TIEs

At the Aviation client-side the following was supported;

1. Landing Page: The Aviation Client connects only to the JSON version of the API. The data is retrieved and presented to the user as a form. The user can select from the form the information of his interest.
2. Collections: The Aviation Client connects only to the JSON version of the API. The data is retrieved and presented to the user as a form. The User can select the collection of its interest.
3. Detect JSON-FG: The Aviation Client connects only to the JSON version of the API. The data is retrieved and presented to the user as a form. When JSON-FG is detected, the user is informed and can select their preferred format from the list of available formats provided by the backend.
4. CoordRefSys: The Aviation Client connects only to the JSON version of the API. The data is retrieved and presented to the user as a form. The user can select his preferred CRS from the list of available CRS's provided by the back end.
5. Where: Implemented, it is supported with limitations (no support for the new geometry polyhedron).
6. When: All aspects are fully supported.
7. Metadata (optional): Not implemented.
8. CRS (optional): Not implemented.

7.6.4. Issues

No major issues were detected. The deliverable implemented most of the JSON-FG specifications with one main exception (polyhedron as an additional geometry).

Other issues detected were mainly on the interaction of the client with the backend services. Some of the services had a slow response, especially when large datasets were requested. This slow response is also an issue for datasets that require significant processing. This implies that the user of the client is forced to set a limit on the number of features requested. This can be achieved by specifying the maximum number of features to load or by restricting the bounding box or the time window to request. Some improvement could be done at the server-side to improve this behavior.

7.6.5. Screenshots

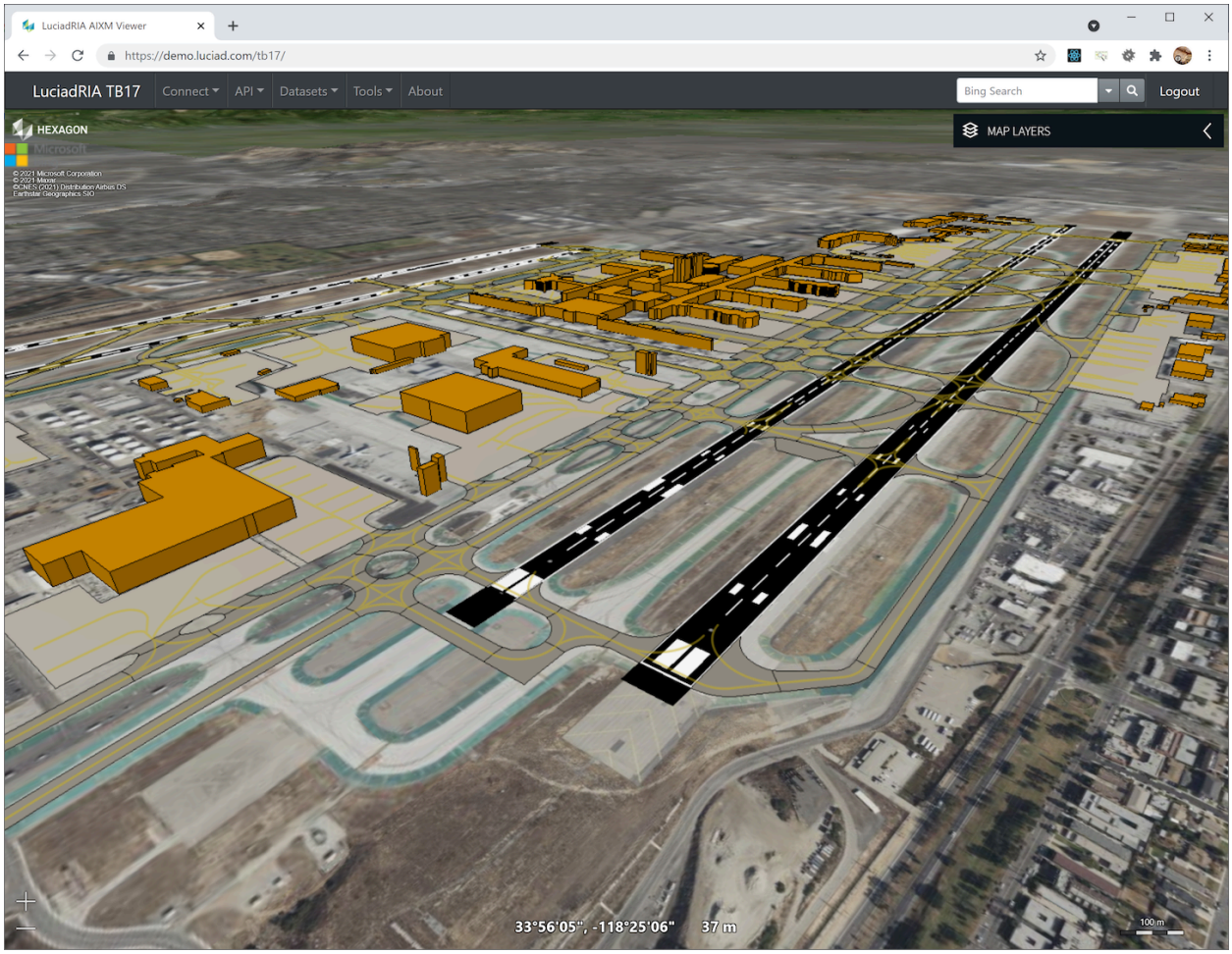

Figure 5 — Hexagon Aviation client requests feature for airport elements to Interactive Instruments API

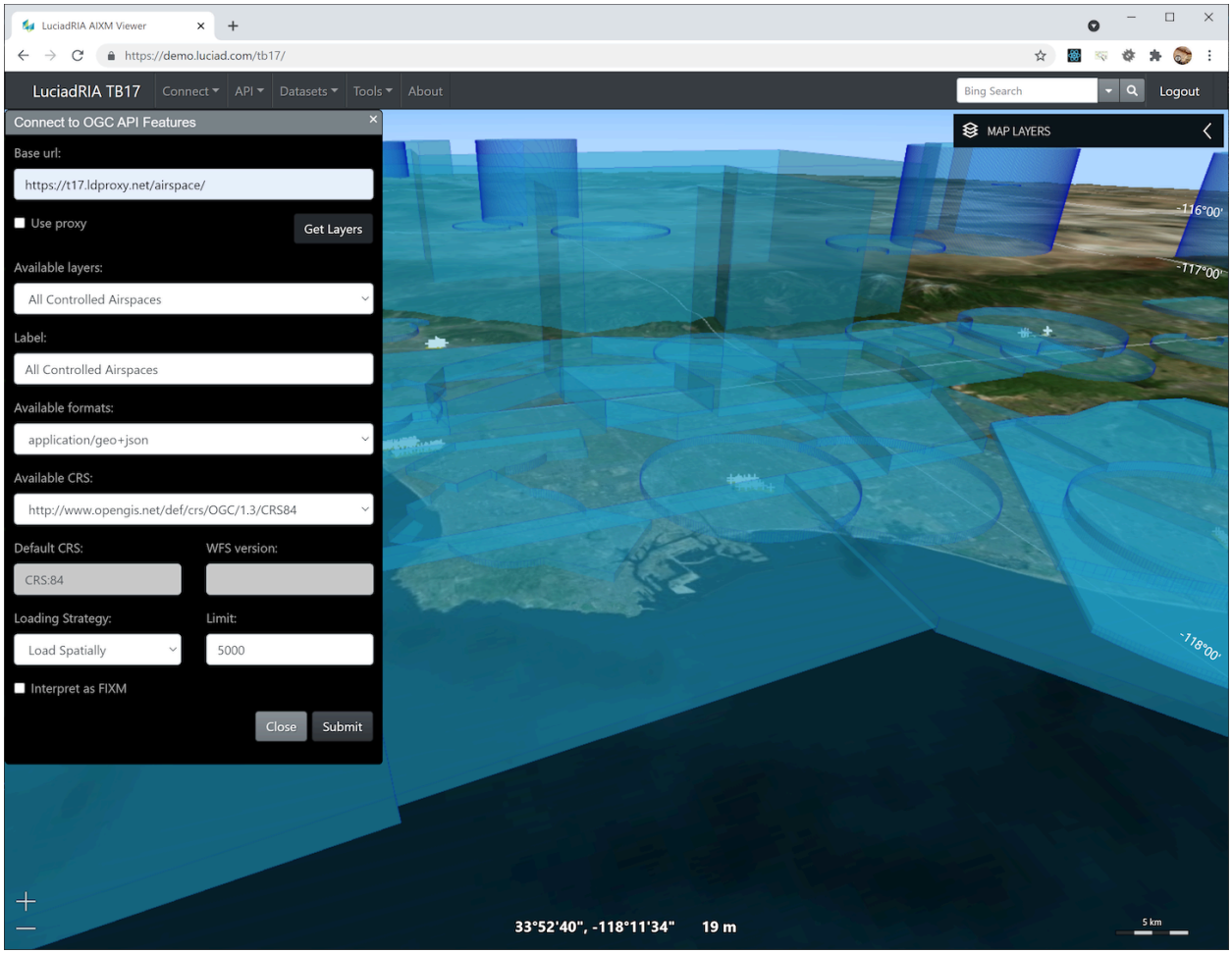

Figure 6 — Hexagon Aviation client requests feature for airport elements to Interactive Instruments API

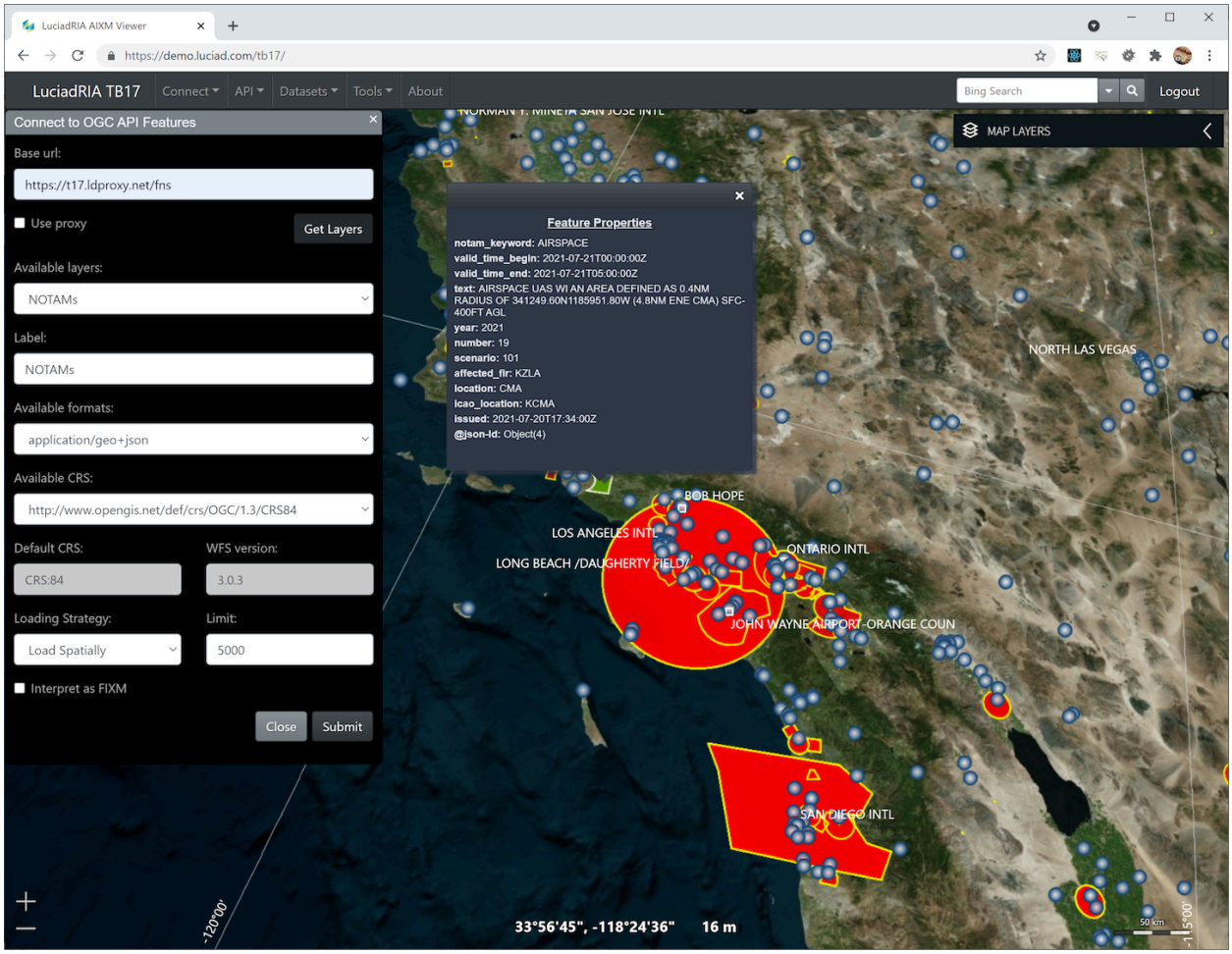

Figure 7 — Hexagon Aviation client requests feature for airport elements to Interactive Instruments API

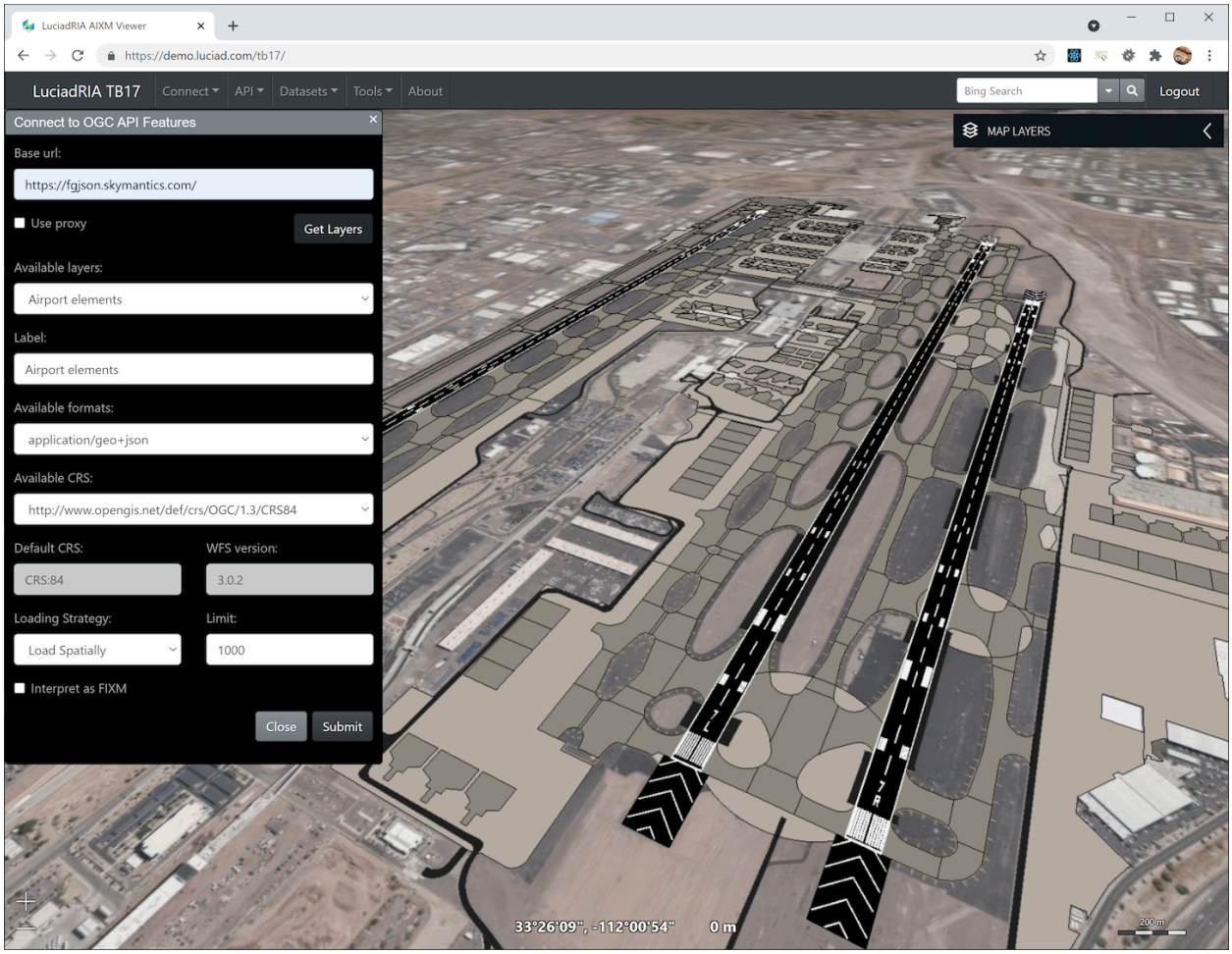

Figure 8 — Hexagon Aviation client requests feature for airport elements to Interactive Instruments API

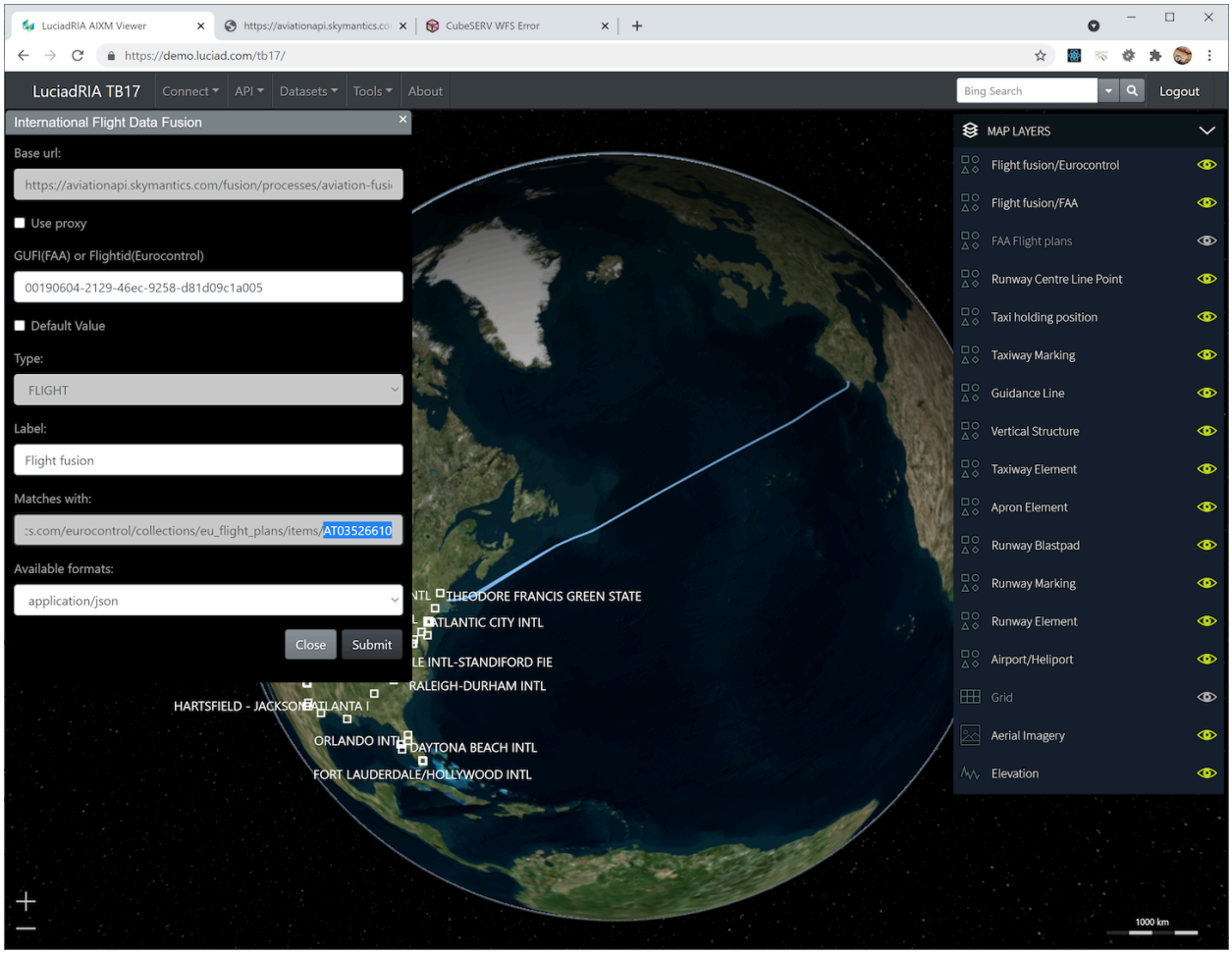

Figure 9 — Hexagon Aviation client requests the International Flight Service for an FAA flight and its matching Eurocontrol Flight

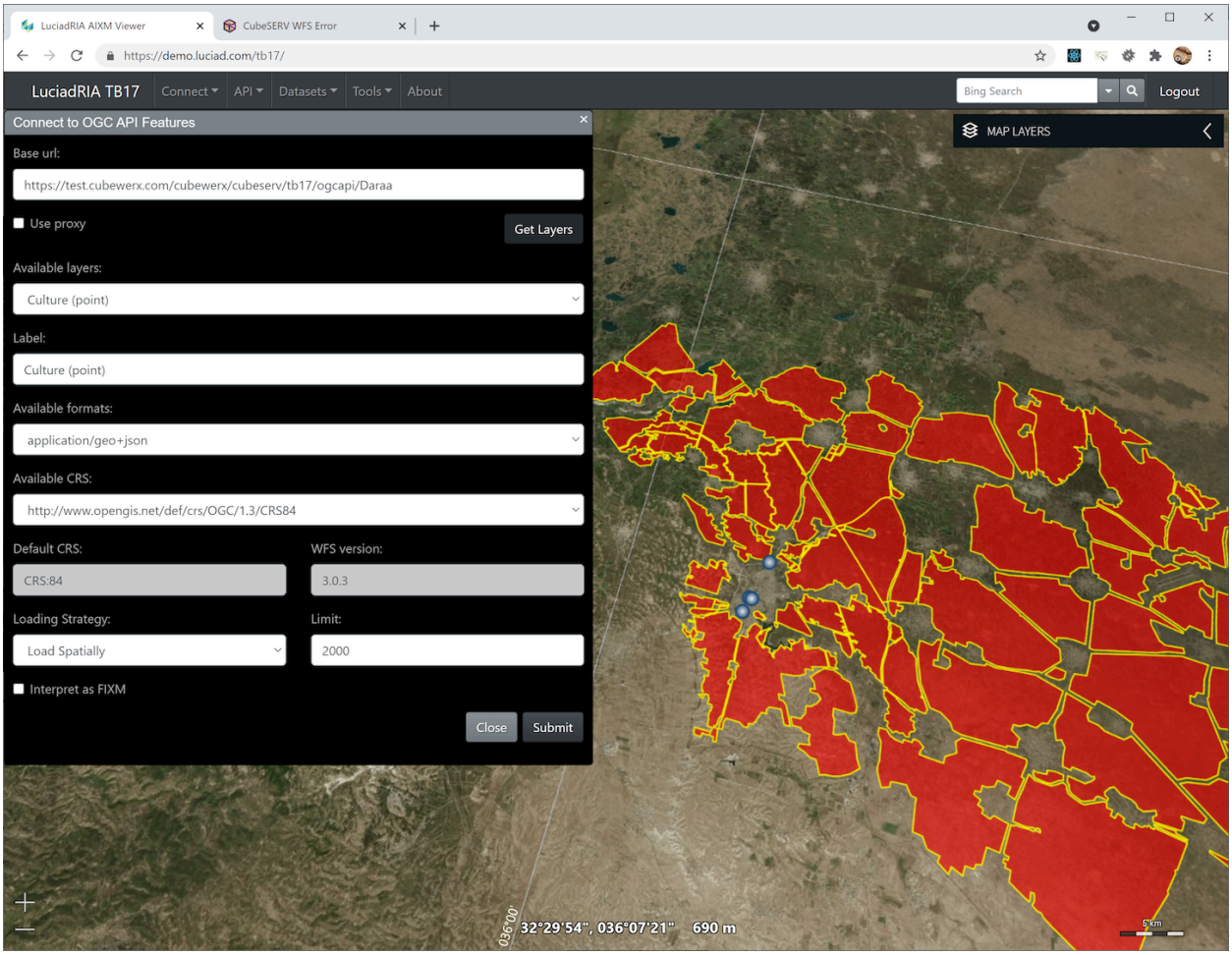

Figure 10 — Hexagon Aviation client requests feature for airport elements to Cuberwerx API

7.7. D103 Features and Geometries JSON Client for Aviation (Ecere)

7.7.1. Client implementation

Ecere's GNOSIS Cartographer is a cross-platform 3D visualization client built upon the GNOSIS SDK geospatial visualization and processing library. By extending the library's GeoJSON parsing and loading module, Ecere added support for loading JSON-FG features and feature collections, including its capabilities for specifying a temporal instant or interval for features (`when`), describing the CRS in `coord-ref-sys`, and using a CRS other than CRS84 in the `where` property for the geometry.

The OGC API client module automatically recognizes the availability of JSON-FG from links with the corresponding media type (specified as `application/vnd.ogc.fg+json`), and selects

it by default when configured to do so. Because the latest version of the client always renders features in 3D, in practice the client must de-project features coordinates to convert them to CRS84(h) anyways, so there is little value in this case to request the features in an alternate CRS if they are also readily available in CRS84 from the service. However, for the purpose of testing the new capability to load JSON-FG specified in alternate CRS, support for *OGC API — Features — Part 2: CRS by reference* was implemented in the client, and the client was configured to request features in an alternate supported CRS such as Web Mercator (EPSG:3857) or World Mercator (EPSG:3395), even though this requires extra work from the client's perspective. Because the client makes heavy use of tiling, it normally prefers selecting *OGC API — Tiles* as an access mechanism. This preference had to be disabled for services providing feature collections as both vector tiles and features so that it uses *OGC API — Features* for the tests.

The client supports loading data sources by either pointing directly to individual OGC API collections of geospatial data, or to an OGC API landing page providing multiple collections. When a CRS other than CRS84(h) is requested, the coordinates stored in the `where` property are used as geometry instead of the usual GeoJSON `geometry` property. Two-dimensional line and polygon features can be visualized in a 3D view by draping them unto the 3D globe or terrain. Polygon features can also be rendered as 3D polyhedrons by extruding them based on their properties, as demonstrated with the air spaces specifying a minimum and maximum altitude. Points features are visualized by placing a marker at the location identified by the point coordinates. The feature schemas are used by the client to determine the list of available properties as well as their type and make these selectable from within the client's style visual style editor.

Ecere also implemented the capability to filter features based on temporal instant or interval which JSON-FG allows to specify for individual features.

7.7.2. Server implementation

As an in-kind contribution, Ecere also implemented support for JSON-FG in its GNOSIS Map Server (demonstration server available), as well as support for *OGC API — Features — Part 2: CRS by reference*, successfully passing the OGC Team Engine executable test suite conformance tests. GeoSolutions were able to perform successful Technology Integration Experiments in their client using the JSON-FG output from this implementation.

7.7.3. Technology Integration Experiments

Ecere successfully performed Technology Integration Experiments with the three server components implementing support for JSON-FG for all identified capabilities:

1. accessing the OGC API landing page;
2. listing available collections at `/collections` and parsing their content such as their spatio-temporal extent which can be previewed in the client's viewport;
3. detecting and negotiating support for features (`/collections/{collectionId}/items`) available as JSON-FG by following the `items` link relation type and recognizing and accepting the `application/vnd.ogc.fg+json` media type;

4. understanding the `coord-ref-sys` property of a JSON-FG response, validating it against the requested CRS, and taking it into consideration for interpreting the geometry coordinates;

5. using the geometry specified in the JSON-FG `where` property when applicable, or `geometry` for GeoJSON geometry types in CRS84, de-projecting the coordinates if necessary, and rendering the geometry in a 3D view;

6. using the instant or interval specified in JSON-FG `when` property for purposes of filtering based on a selected temporal extent;

7. retrieving and parsing the feature schemas to list and make them available for purposes of styling, such as for use in 3D extrusion based on a minimum and maximum altitudes (e.g. air spaces);

8. list the CRS available for each collection to validate against the CRS to be requested.

7.7.4. Issues encountered and lessons learned

7.7.4.1. Features with geometries of mixed dimensionality

The GNOSIS library does not yet natively support collections of features of mixed geometry dimensions, as the compact data structures to store collections of features are optimized for geometries of a specific number of dimensions. As some collections deployed on server components contained mixed geometry dimensions, this complicated some of the tests by requiring temporary changes to force loading of features of a particular number of dimensions.

Two options for solving this challenge were identified:

- the library could fall back to a less optimal feature collection storage when mixed geometry dimensions are detected, or
- the library could automatically split collections containing mixed geometry dimensions into separate sub-collections containing geometries of the same number of dimensions.

The latter option could either be done by additional processing on the client side, or the client could separately request features of a specific number of dimensions, if the server provides this capability as a *queryable* for use with *OGC API — Features — Part 3: Filtering*.

7.7.4.2. Mixed feature types

Similarly, the library does not yet fully support collections of features of mixed feature types, i.e. where different properties are available for different features, as the attributes caching and

access mechanisms expect a pre-initialized schema where each available property is defined and typed.

The new JSON-FG capability of identifying the `featureType` either at the feature collection level or at the individual feature level is interesting as it could provide a mechanism by which to identify the feature type each feature belongs to. A recommendation that a `featureType` queryable should be supported when a collection of features contains more than a single feature type would also facilitate filtering on the server-side.

Since a particular feature type often also consists of features with geometry of a specific dimensionality, this would also help with the first challenge of geometry dimensionality.

7.7.4.3. Schemas

During the initiative, existing proposals for describing the schemas of features underwent changes, and some new types of schemas were introduced. This caused some difficulties for Ecere's client implementation which relied on those schemas for accessing the attributes. Because the client does not use the schemas for strict JSON validation, but rather for awareness of the attributes available and their types, Ecere suggested that a separate simple schema conformance class only requires the presence of a *feature* level schema, from which a *feature collection* schema could easily be inferred. It would also be useful to standardize how schemas are defined in cases where a feature collection offers features of different feature types, e.g. with a top-level `oneOf` whose elements are the schemas for the individual feature types.

7.7.4.4. Heavy geometry and scale of interest

In the case of feature collections where individual vector features both span a large geospatial area and are defined at a high resolution (i.e. they are made up of a large number of coordinates), filtering by bounding box and features paging as defined by *OGC API — Features — Part 1: Core* is not sufficient to achieve efficient retrieval and visualization. A classic example is a high-resolution coastline of a country or continent. The scenario of visualizing such features at a large scale denominator would greatly benefit from the capability to request the features with a generalized (simplified) version of the geometry. When visualizing these features at a small scale denominator, it is also desirable to be able to clip these. Both of these capabilities (simplification and clipping) are integrated within the concept of vector tiles, but can also be provided as a simple extensions to OGC API — Features. Experiments and refinement to a proposal for such extensions (scale-denominator / `zoom-level` and `clip-box` query parameters) were conducted in collaboration with interactive instruments and CubeWerx during the joint ISO / OGC code sprint focused on *OGC API — Features* / ISO 19168 demonstrating the significant benefits of these extensions in terms of improved performance and reduced use of bandwidth.

The performance advantage is particularly noticeable in the Ecere client since it relies heavily on an internal tiling organization which currently results in making separate features requests for each tile, many of which may end-up returning duplicate information. An alternative would be to request all features only once and then proceed to tile them in the client, but it is very difficult for the client to judge which approach will be more efficient for different scenarios in terms of how many features will be filtered by bounding box intersection at different scales.

7.7.5. Screenshots

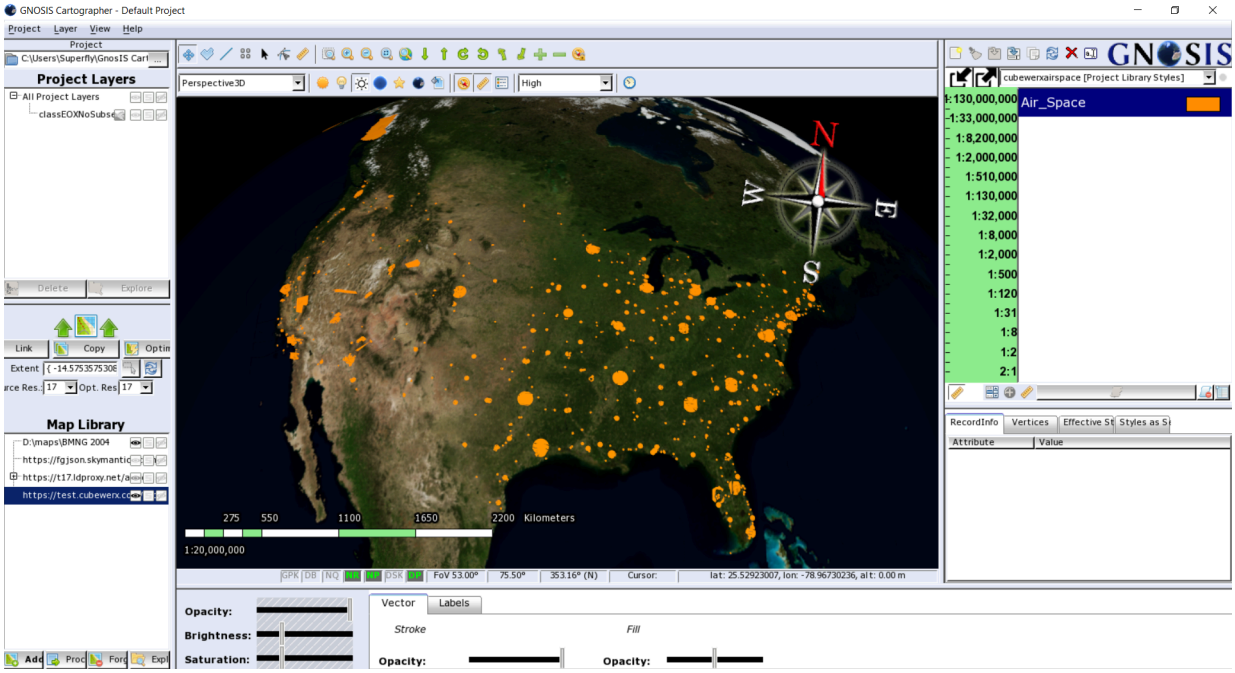

Figure 11 — Cartographer client requests simplified features from CubeWerx endpoint, loads and renders with extrusion using attributes

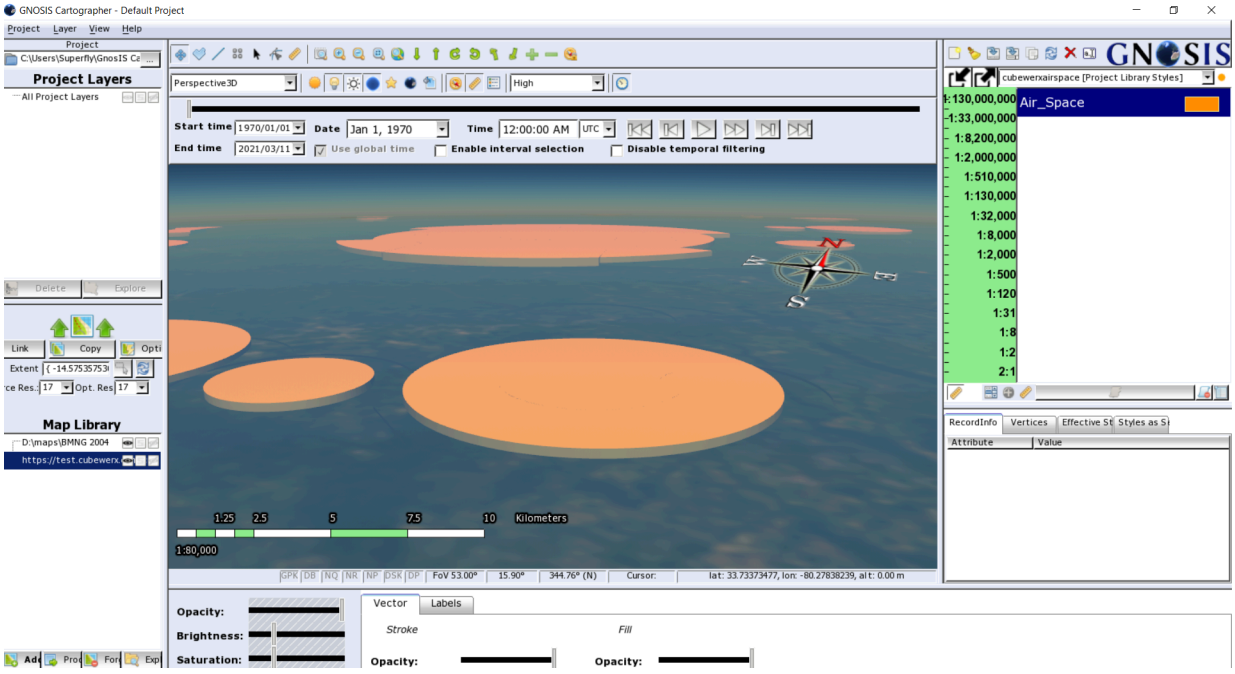

Figure 12 — Cartographer client requests non-simplified features from CubeWerx endpoint, loads and renders with extrusion using attributes

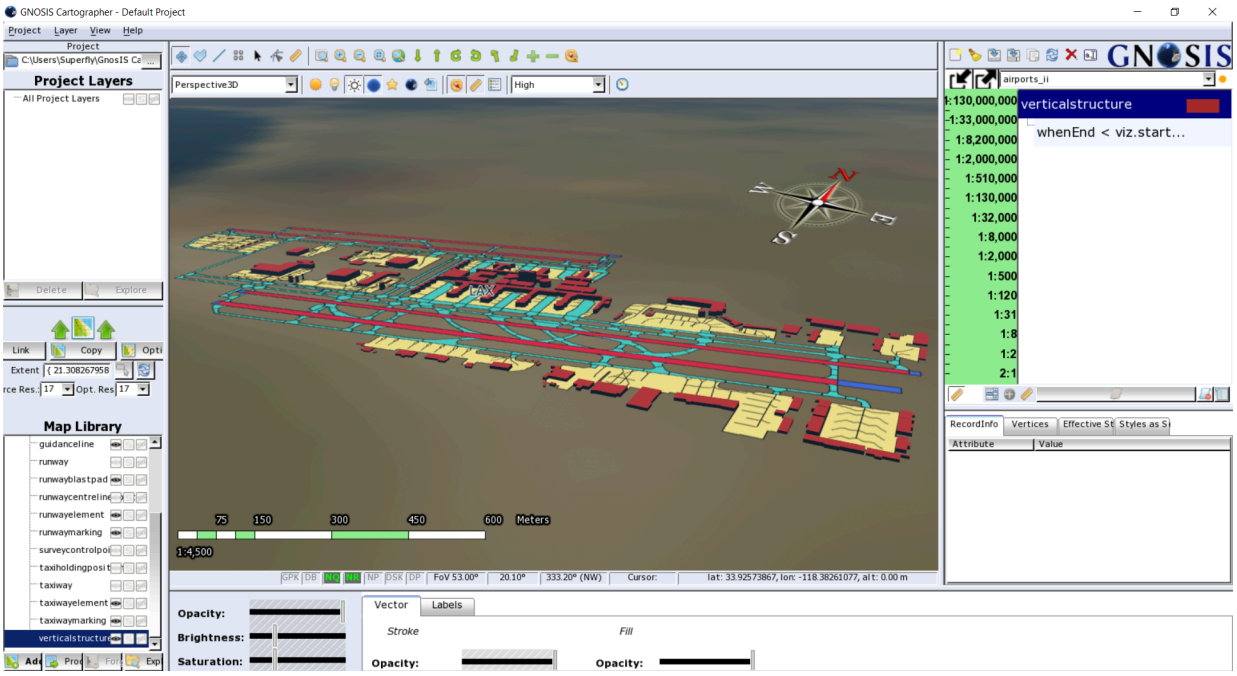

Figure 13 — Cartographer client requests features from interactive instruments endpoint zoomed at airport location, loads and renders with styling

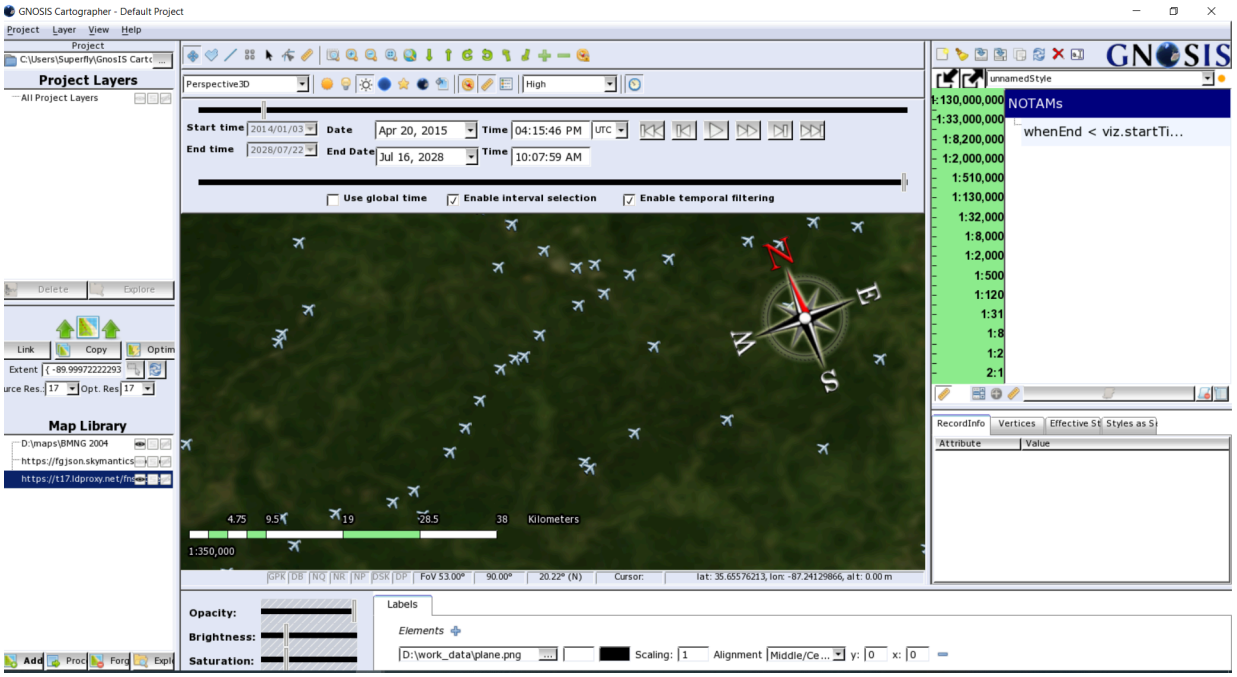

Figure 14 — Cartographer client renders features from interactive instruments NOTAMS collection filtered by time at large interval extent

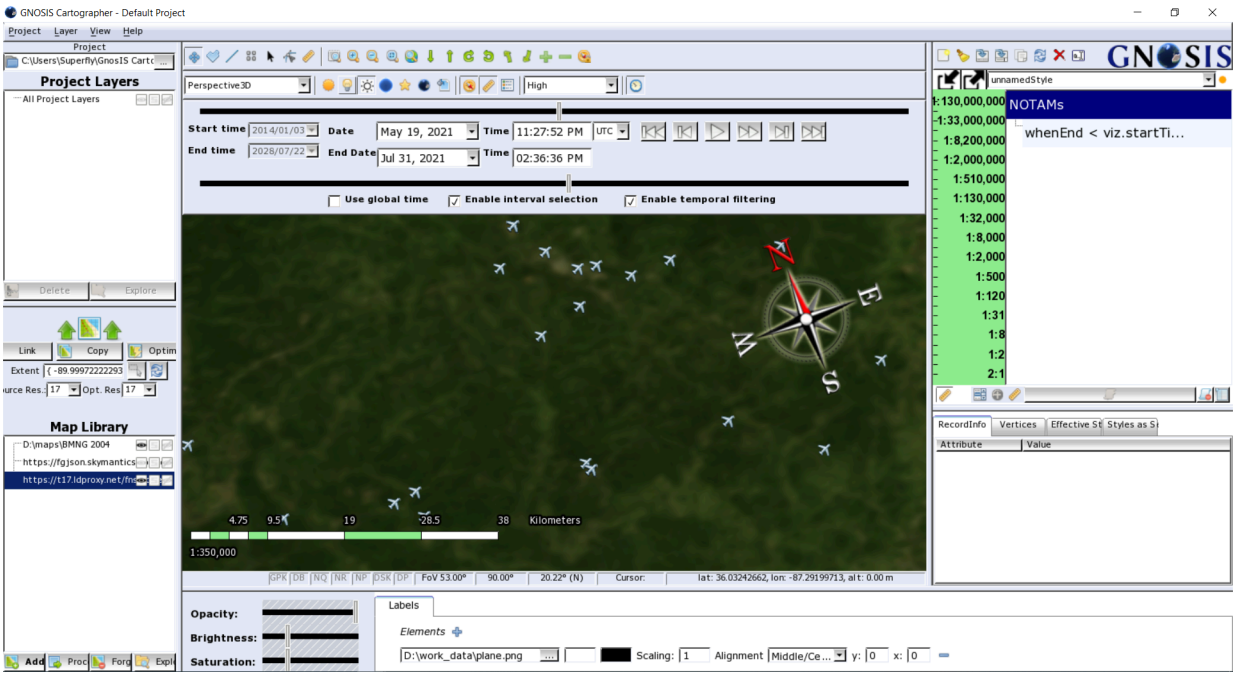

Figure 15 — Cartographer client renders features from interactive instruments NOTAMS collection filtered by time at narrow interval extent

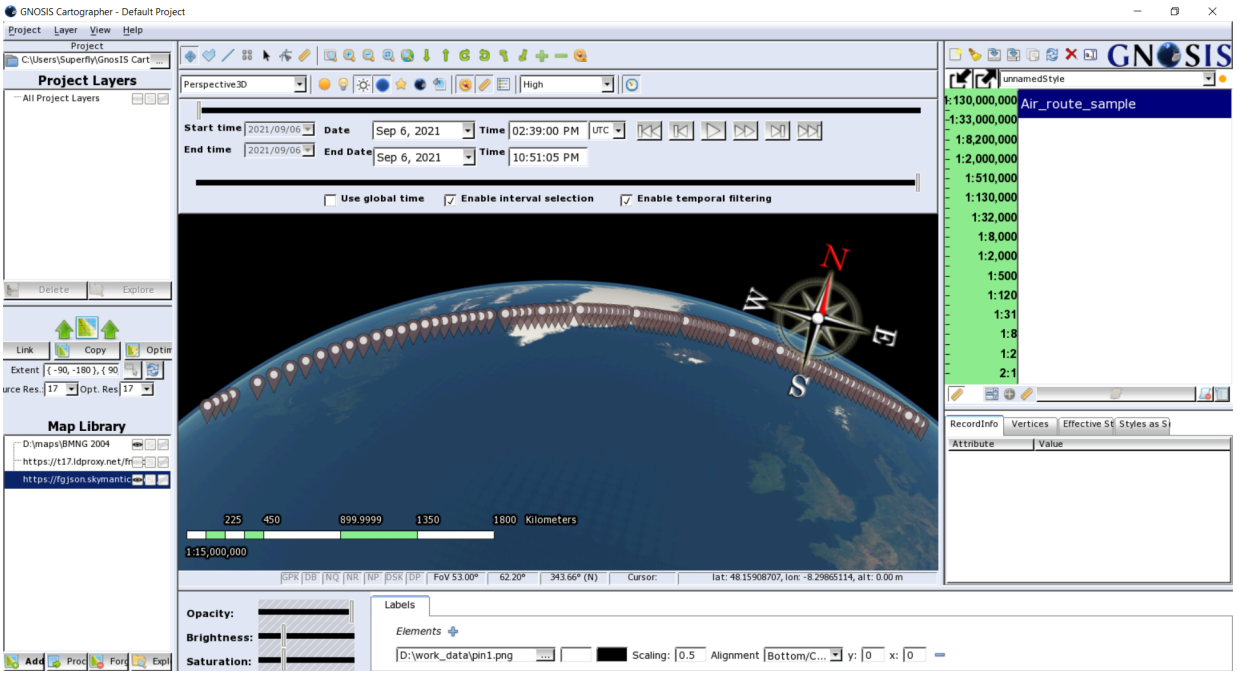

Figure 16 — Cartographer client renders features from Skymantics air route sample filtered by time at full interval extent

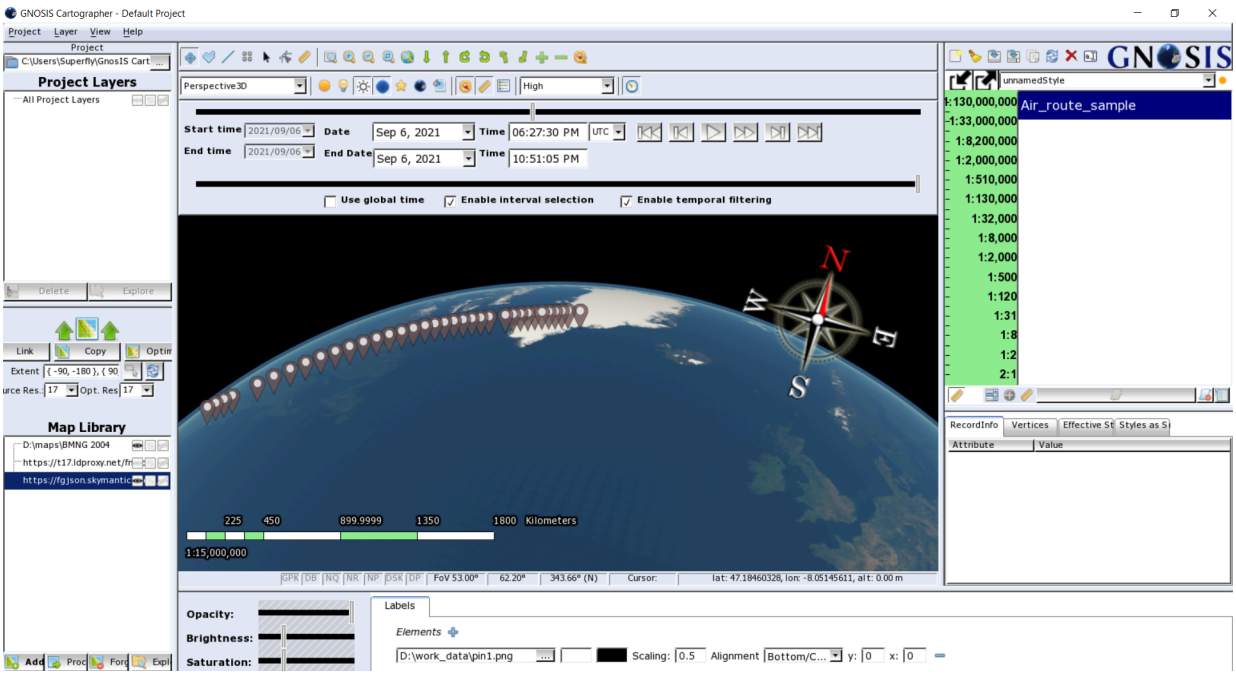

Figure 17 — Cartographer client renders features from Skymantics air route sample filtered by time at half interval extent

7.8. D116 Features and Geometries JSON Client (GeoSolutions)

The D116 components from GeoSolutions implement a web client application that interacts with the OGC API features endpoints to retrieve and render the JSON-FG data format. The web application is based on MapStore, a modular open source WebGIS framework written in JavaScript and ReactJS that allows a developer to build an interactive application around maps and spatial data.

- repository with source code
- live demo

7.8.1. Structure of the web client application

The implemented application has been built from a MapStore custom project and provides the following plugins:

1. A catalog panel to connect to an OGC API endpoint.
2. A layer tree to select and edit imported layers.

3. A layer setting panel to edit params:
    - Elevation property: If specified uses this property as elevation value.
    - Upper volume elevation property: If specified, uses this property as upper value.
    - Available CRS: Select a different CRS to apply to the features collection.
    - Max features count: Maximum number of features to request.
    - Style: A dedicated panel to edit the features style (only polygon supported).
    - JSON preview: A preview in JSON of the selected collection,
4. Map viewer: Render the imported layers and it's possible to change between 2D (OpenLayers) and 3D (Cesium) view.
5. Projection selector: Available only for the 2D view allows to change the rendered projection.
6. Time range selector: Available only for GeoJSON with the experimental property `when` supported to visualize the time range of the imported collection and filter the features base on their own interval.

Figure 18 — Structure of the web application

7.8.2. TIEs

The Technology Integration Experiments (TIEs) related to this ER tested the following workflow:

- Request JSON-FG collection from OGC API features endpoints;
- Display information of JSON-FG collection;
- Render the features on a 2D and 3D map;
- Use the new `where` property instead of geometry to render and visualize feature;
- Use the new `when` property to filter features based on a time range client side.

7.8.2.1. Issue encountered and lessons learnt

The use of the new JSON-FG properties such as `where`, `when`, and `featureType` extending GeoJSON does not create breaking changes in the application. The rendering workflow of the features could take advantage of existing utilities and approaches for using the GeoJSON format.

The main issues encountered are:

- Some geometry types introduced in the `where` property are not supported by the client so they are automatically skipped with a fallback on the default `geometry` property (e.g. Polyhedron type).
- The use of duplicate geometry coordinates (`where` and `geometry` properties) could increase the size of the JSON-FG file. From the web perspective this size increase could affect the performance. The proposed solution to provide two different sets of coordinates could mitigate this issue (e.g. `geometry` with a bounding box and `where` with complete coordinates).

In conclusion, the ability to fallback to GeoJSON provided by the JSON-FG format supports good flexibility for the client implementations.

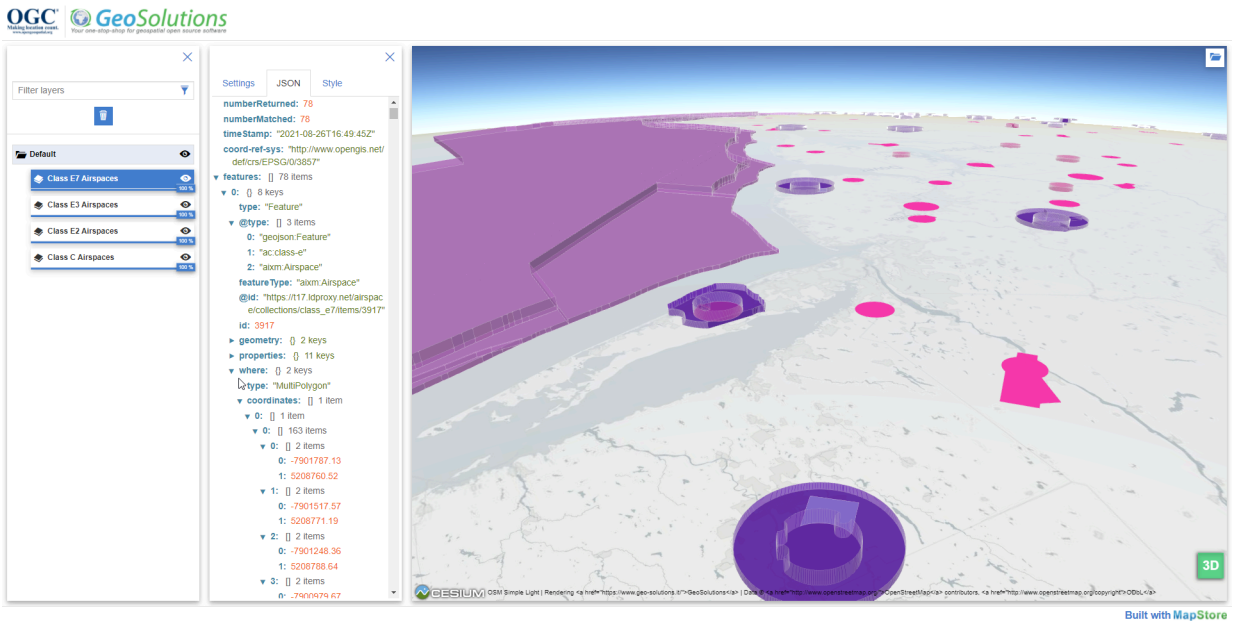

Figure 19 — MapStore client interacts with the interactive instruments service and it renders in 3D different Airspace classes and one of them is using the where property with the coord-ref-sys to EPSG:3857

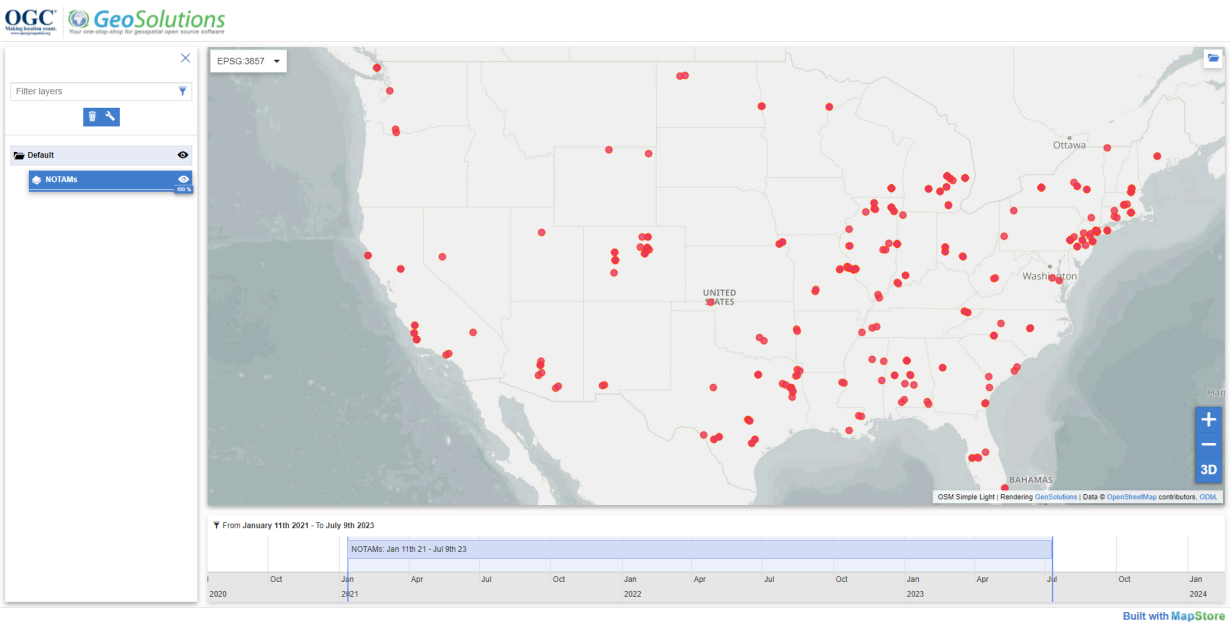

Figure 20 — MapStore client interacts with the interactive instruments service and it shows the timeline for the collection NOTAMs

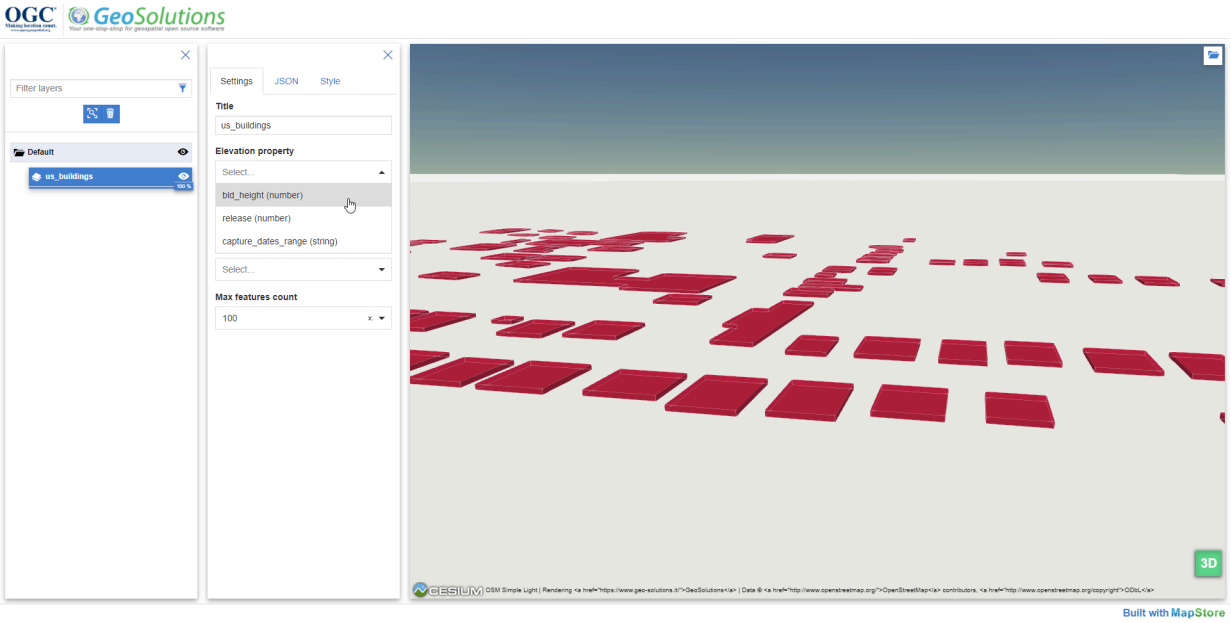

Figure 21 — MapStore client interacts with the Cubewerx service and it shows buildings in 3D

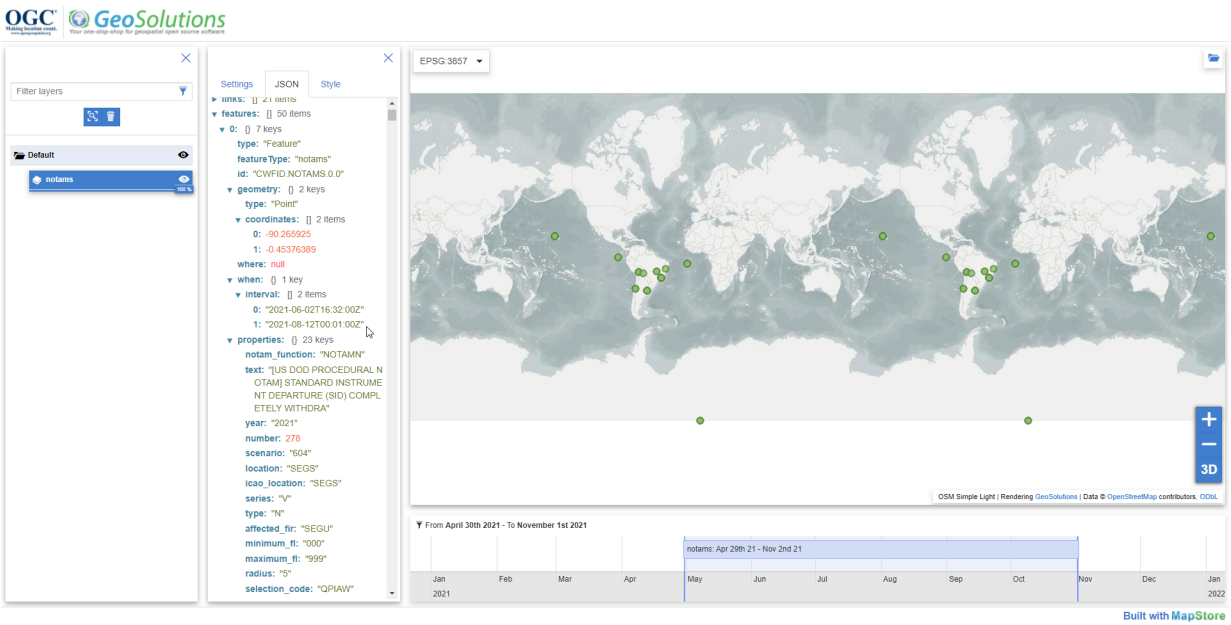

Figure 22 — MapStore client interacts with the Cubewerx service and it shows the timeline to filter the NOTAMs collection features

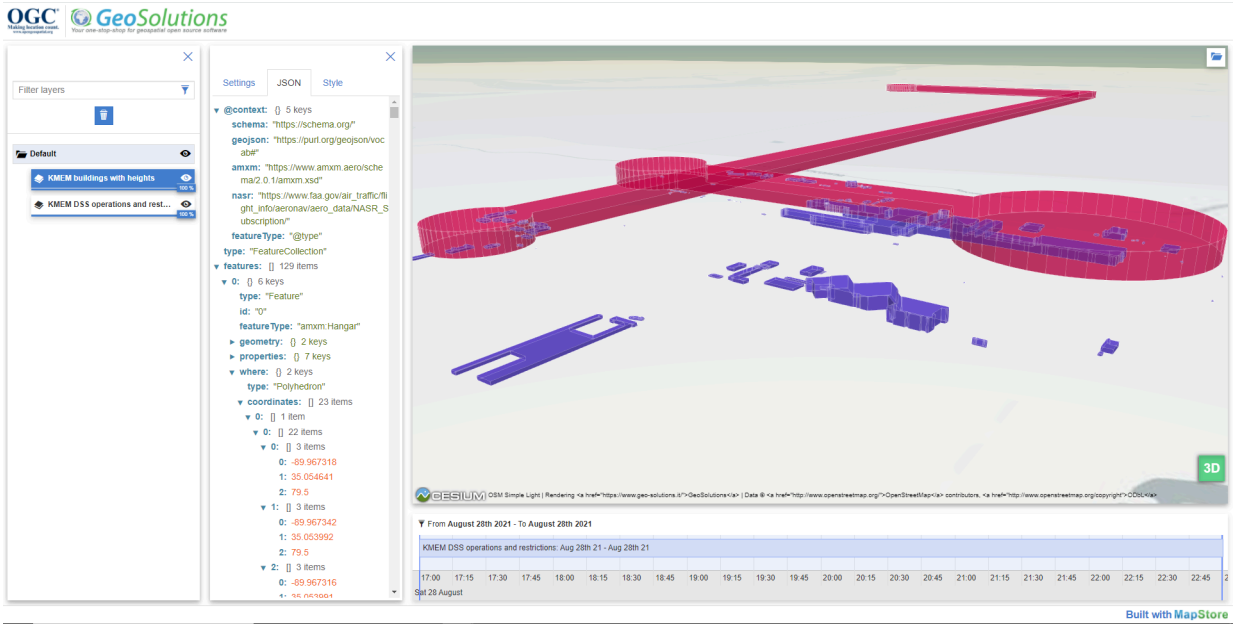

Figure 23 — MapStore client interacts with the Skymantics service and it renders in 3D the buildings and DSS from KMEM dataset and use the CRS84 geometry instead the Polyhedron one

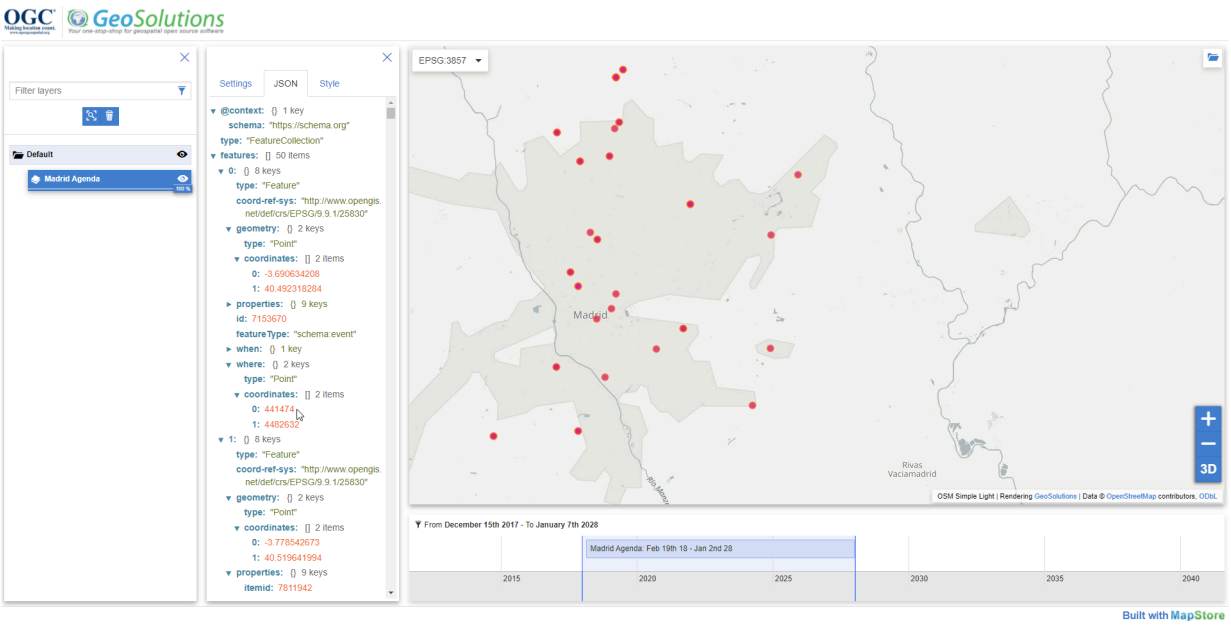

Figure 24 — MapStore client interacts with the Skymantics service and it shows the timeline to filter the Madrid Agenda collection

8

RESULTS AND RECOMMENDATIONS

8 RESULTS AND RECOMMENDATIONS

8.1. Results

The results described in the previous chapter can be summarized as follows:

- The TIEs were successful. Problems identified during initial TIEs were usually a result of simple bugs or temporary unavailability of server deployments.
- Extending existing GeoJSON writers and readers with the additional JSON members was in general straightforward as demonstrated in the six software components.
- These experiences support the current JSON-FG approach:
    - Extend GeoJSON (every JSON-FG document is a valid GeoJSON document);
    - Focus on minimal extensions to GeoJSON that are useful in many contexts relevant to OGC members and avoid edge-cases;
    - Specify the extensions as additional top-level JSON members (do not add constraints on “properties” or any other GeoJSON member); and
    - Specify the extensions in a modular way, so that implementations can pick and choose the capabilities that they need.
- The proposed member “featureType” supports a capability that was only mentioned implicitly in the charter of the Features and Geometries JSON SWG, but has been fundamental for successful TIEs.
- The potential duplication of the spatial geometry (in the “where” and “geometry” members) increases the size of the JSON-FG file and, as a consequence, affects the performance in web applications. It should be possible for clients that know that they do not need the fallback “geometry” member to suppress its content when the features are requested via an API. Another option is to return a small, simplified geometry in the fallback “geometry” member.
- The “when” member is limited in its representation of a single instant or interval. This is a conscious restriction, because this already supports many use cases and directly maps to UI elements like a time slider. More complex cases like repeating instants or interval, etc. could be added in the future, if there is enough demand. The design of the “when” member supports such extensions.
- No client implemented support for polyhedron geometry during the testbed. This may be a sign that supporting polyhedrons is more complex than the other

extensions or that it is less useful. There are other geometry representations of 3D geometries that are frequently used and that might also be considered. Examples are a planar base surface as an additional property describing the extrusion extent (beside a polygon, circles are also frequently used in some domains like aviation) or an OBJ-like geometry representation as in CityJSON.

- One client had issues with JSON-FG and GeoJSON responses from servers with mixed geometry types. This is not a JSON-FG issue per se, since JSON-FG is no different from GeoJSON in this regard, but a conformance class could be considered where all features in a JSON-FG feature collection are homogeneous with respect to the geometry type. The draft JSON-FG specification has also been extended to support that a member “geometryDimension” is added to a JSON-FG feature collection, if all features have the same geometry dimension (see section Clause 6.6.2.2).
- Parsing JSON schemas for features to retrieve information of the features is still too complex due to inconsistent levels of depth of information (for instance, inclusion or omission of geometry metadata) and different ways to construct the JSON schemas. These topics should be discussed in the SWG and tested in implementations.
- No normative statements were identified for representing properties that are relationships with other features or resources like codelists. However, three general patterns for representing such relationships have been identified as options. Depending on the data and how the data is expected to be used, the preferences of data publishers for one or the other pattern will vary.
- The JSON-FG extensions may also be useful for representing routes in the Route Exchange Model, as shown by the representation of flight plans using JSON-FG.
- The planned geometry simplification extension of OGC API Features can help to improve performance at low zoom levels. The SWG is encouraged to make progress with the extension.
- A proposal for JSON-FG requirements/conformance classes was developed as input for the SWG.

The results are consistent with the experiences of the Aviation task in Testbed-17 to utilize the JSON-FG encoding [4].

8.2. Recommendations

The following are the key recommendations based on the work and lessons learned in the Testbed 17 JSON-FG API thread.

1. The experience in the testbed was that the current draft Clause 6 provides useful capabilities, but at the same time is still simple to implement in software. However, more testing and developer feedback is needed.

2. The SWG should take the current draft extensions, convert them into a candidate standard, tag it as version “1.0.0-draft.1” and look for additional feedback from implementations.

3. OGC Innovation Program initiatives such as sprints, pilots and testbeds continue to be a good environment for collaborative development and testing. During Testbed-17 this included the OGC Code Sprint in November 2021, which was focused on OGC API Features and included multiple discussions and implementations related to JSON-FG. It is recommended to look for opportunities to mature the JSON-FG draft through additional OGC Innovation Program initiatives.

4. Testing JSON-FG with initial implementations in commonly used software including GDAL, QGIS or web mapping libraries like Leaflet, OpenLayers or MapLibre would be good.

5. Before moving forward with the candidate standard to OAB / Public review, the open questions in the chapter Clause 6 should be resolved and all aspects of JSON-FG should have been tested in more detail.

6. The feedback from the TIEs noted above should be discussed in the Features and Geometries JSON SWG and, if possible, future Innovation Program initiatives:

    - Recommendations when to include the fallback GeoJSON “geometry” member or not and OGC API building blocks to control the behavior.
    - Support for 3D geometries through polyhedron geometry objects or other encodings (base surface plus height, support for circles, more compact coordinate encodings).
    - How to simplify the parsing of the JSON schemas describing the feature schemas?
    - Continue to investigate the options for representing relationships with or links to other resources.
    - Potential support a “geometryDimension” member and a potential conformance class for homogeneous feature collections.

7. Working Groups in OGC should be encouraged to investigate whether the JSON-FG extensions can be useful for their encodings (for example, JSON-FG might be relevant for future versions of the Route Exchange Model).

8. The Features API SWG is encouraged to make progress with the planned geometry simplification extension of OGC API Features and potential support for versioned feature data.

A

ANNEX A (INFORMATIVE) JSON SCHEMA DOCUMENTS

A

ANNEX A (INFORMATIVE) JSON SCHEMA DOCUMENTS

A.1. JSON Schema of the JSON-FG feature extensions

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/Feature.json",
  "title": "JSON-FG Feature",
  "type": "object",
  "required": [
    "when",
    "where"
  ],
  "properties": {
    "featureType": {
      "oneOf": [
        {
          "type": "string"
        },
        {
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      ]
    },
    "links": {
      "type": "array",
      "items": {
        "$ref": "#/$defs/Link"
      }
    },
    "when": {
      "oneOf": [
        {
          "type": "null"
        },
        {
          "type": "object",
          "required": [
            "instant"
          ],
          "properties": {
            "instant": {
              "oneOf": [
```

```
              {
                "type": "string",
                "format": "date"
              },
              {
                "type": "string",
                "format": "date-time"
              }
            ]
          }
        }
      },
      {
        "type": "object",
        "required": [
          "interval"
        ],
        "properties": {
          "interval": {
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
            "items": {
              "oneOf": [
                {
                  "type": "string",
                  "format": "date"
                },
                {
                  "type": "string",
                  "format": "date-time"
                },
                {
                  "type": "null"
                }
              ]
            }
          }
        }
      }
    ]
  },
  "coordRefSys": {
    "$ref": "#/$defs/refsys"
  },
  "where": {
    "oneOf": [
      {
        "type": "null"
      },
      {
        "title": "GeoJSON Point",
        "type": "object",
        "required": [
          "type",
          "coordinates"
        ],
        "properties": {
          "type": {
            "type": "string",
            "enum": [
              "Point"
            ]
```

```
      },
      "coordinates": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "number"
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON LineString",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "LineString"
        ]
      },
      "coordinates": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "array",
          "minItems": 2,
          "items": {
            "type": "number"
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON Polygon",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "Polygon"
```

```
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "minItems": 4,
          "items": {
            "type": "array",
            "minItems": 2,
            "items": {
              "type": "number"
            }
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON MultiPoint",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiPoint"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "minItems": 2,
          "items": {
            "type": "number"
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON MultiLineString",
    "type": "object",
    "required": [
      "type",
```

```
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiLineString"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "minItems": 2,
          "items": {
            "type": "array",
            "minItems": 2,
            "items": {
              "type": "number"
            }
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON MultiPolygon",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiPolygon"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "items": {
            "type": "array",
            "minItems": 4,
            "items": {
              "type": "array",
              "minItems": 2,
              "items": {
                "type": "number"
              }
            }
          }
        }
      },
```

```
    "bbox": {
      "type": "array",
      "minItems": 4,
      "items": {
        "type": "number"
      }
    }
  }
},
{
  "title": "JSON-FG Polyhedron",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Polyhedron"
      ]
    },
    "coordinates": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "array",
        "minItems": 1,
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
            "minItems": 4,
            "items": {
              "type": "array",
              "minItems": 3,
              "maxItems": 3,
              "items": {
                "type": "number"
              }
            }
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 6,
      "maxItems": 6,
      "items": {
        "type": "number"
      }
    }
  }
},
{
  "title": "JSON-FG MultiPolyhedron",
  "type": "object",
  "required": [
    "type",
    "coordinates"
```

```
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiPolyhedron"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "array",
            "minItems": 1,
            "items": {
              "type": "array",
              "minItems": 1,
              "items": {
                "type": "array",
                "minItems": 4,
                "items": {
                  "type": "array",
                  "minItems": 3,
                  "maxItems": 3,
                  "items": {
                    "type": "number"
                  }
                }
              }
            }
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 6,
        "maxItems": 6,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON GeometryCollection",
    "type": "object",
    "required": [
      "type",
      "geometries"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "GeometryCollection"
        ]
      },
      "geometries": {
        "type": "array",
        "items": {
          "oneOf": [
```

```
{
  "title": "GeoJSON Point",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Point"
      ]
    },
    "coordinates": {
      "type": "array",
      "minItems": 2,
      "items": {
        "type": "number"
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 4,
      "items": {
        "type": "number"
      }
    }
  }
},
{
  "title": "GeoJSON LineString",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "LineString"
      ]
    },
    "coordinates": {
      "type": "array",
      "minItems": 2,
      "items": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "number"
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 4,
      "items": {
        "type": "number"
      }
    }
  }
```

```
},
{
  "title": "GeoJSON Polygon",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "Polygon"
      ]
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "array",
          "minItems": 2,
          "items": {
            "type": "number"
          }
        }
      }
    },
    "bbox": {
      "type": "array",
      "minItems": 4,
      "items": {
        "type": "number"
      }
    }
  }
},
{
  "title": "GeoJSON MultiPoint",
  "type": "object",
  "required": [
    "type",
    "coordinates"
  ],
  "properties": {
    "type": {
      "type": "string",
      "enum": [
        "MultiPoint"
      ]
    },
    "coordinates": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 2,
        "items": {
          "type": "number"
        }
      }
    },
    "bbox": {
```

```
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON MultiLineString",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiLineString"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
          "minItems": 2,
          "items": {
            "type": "array",
            "minItems": 2,
            "items": {
              "type": "number"
            }
          }
        }
      },
      "bbox": {
        "type": "array",
        "minItems": 4,
        "items": {
          "type": "number"
        }
      }
    }
  },
  {
    "title": "GeoJSON MultiPolygon",
    "type": "object",
    "required": [
      "type",
      "coordinates"
    ],
    "properties": {
      "type": {
        "type": "string",
        "enum": [
          "MultiPolygon"
        ]
      },
      "coordinates": {
        "type": "array",
        "items": {
          "type": "array",
```

```
                        "items": {
                          "type": "array",
                          "minItems": 4,
                          "items": {
                            "type": "array",
                            "minItems": 2,
                            "items": {
                              "type": "number"
                            }
                          }
                        }
                      },
                      "bbox": {
                        "type": "array",
                        "minItems": 4,
                        "items": {
                          "type": "number"
                        }
                      }
                    }
                  }
                ]
              }
            },
            "bbox": {
              "type": "array",
              "minItems": 4,
              "items": {
                "type": "number"
              }
            }
          }
        }
      ]
    }
  },
  "$defs": {
    "Link": {
      "type": "object",
      "required": [
        "href",
        "rel"
      ],
      "properties": {
        "href": {
          "type": "string",
          "format": "uri-reference"
        },
        "rel": {
          "type": "string"
        },
        "anchor": {
          "type": "string"
        },
        "type": {
          "type": "string"
        },
        "hreflang": {
          "type": "string"
        },
        "title": {
          "type": "string"
```

```
        },
        "length": {
          "type": "string"
        }
      }
    },
    "refsysSimpleref": {
      "type": "string",
      "format": "uri"
    },
    "refsysByref": {
      "type": "object",
      "required": [ "href" ],
      "properties": {
        "href": {
          "type": "string",
          "format": "uri"
        },
        "epoch": {
          "type": "string"
        }
      }
    },
    "refsys": {
      "oneOf": [
        { "$ref": "#/$defs/refsysSimpleref" },
        { "$ref": "#/$defs/refsysByref" },
        {
          "type": "array",
          "items": {
            "oneOf": [
              { "$ref": "#/$defs/refsysSimpleref" },
              { "$ref": "#/$defs/refsysByref" }
            ]
          }
        }
      ]
    }
  }
}
```

Figure A.1

A.2. JSON Schema of the JSON-FG feature collection extensions

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://www.opengis.net/tbd/FeatureCollection.json",
  "title": "JSON-FG Feature Collection",
  "type": "object",
  "required": [
  ],
  "properties": {
    "featureType": {
      "oneOf": [
        {
```

```
            "type": "string"
          },
          {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        ]
      },
      "geometryDimension": {
        "type": "integer"
      },
      "coordRefSys": {
        "$ref": "#/$defs/refsys"
      },
      "links": {
        "type": "array",
        "items": {
          "$ref": "#/$defs/Link"
        }
      }
    },
    "$defs": {
      "Link": {
        "type": "object",
        "required": [
          "href",
          "rel"
        ],
        "properties": {
          "href": {
            "type": "string",
            "format": "uri-reference"
          },
          "rel": {
            "type": "string"
          },
          "anchor": {
            "type": "string"
          },
          "type": {
            "type": "string"
          },
          "hreflang": {
            "type": "string"
          },
          "title": {
            "type": "string"
          },
          "length": {
            "type": "string"
          }
        }
      },
      "refsysSimpleref": {
        "type": "string",
        "format": "uri"
      },
      "refsysByref": {
        "type": "object",
        "required": [ "href" ],
        "properties": {
```

```
        "href": {
          "type": "string",
          "format": "uri"
        },
        "epoch": {
          "type": "string"
        }
      }
    },
    "refsys": {
      "oneOf": [
        { "$ref": "#/$defs/refsysSimpleref" },
        { "$ref": "#/$defs/refsysByref" },
        {
          "type": "array",
          "items": {
            "oneOf": [
              { "$ref": "#/$defs/refsysSimpleref" },
              { "$ref": "#/$defs/refsysByref" }
            ]
          }
        }
      ]
    }
  }
}
```

Figure A.2

B

ANNEX B (INFORMATIVE) REVISION HISTORY

B

ANNEX B (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	AUTHOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
May 24, 2021	n/a	C. Portele	all	initial version
November 11, 2021	n/a	C. Portele, I. Correas, S. Bovio, P. Dion, F. Carrillo Romero, P. Vretanos	all	complete draft
November 19, 2021	21-017	C. Portele	all	update based on review comments

BIBLIOGRAPHY

BIBLIOGRAPHY

1. H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub: RFC 7946, *The GeoJSON Format*. Internet Engineering Task Force, Fremont, CA (2016). https://raw.githubusercontent.com/relaton/relaton-data-ietf/master/data/reference.RFC.7946.xml

2. G. Klyne, C. Newman: RFC 3339, *Date and Time on the Internet: Timestamps*. Internet Engineering Task Force, Fremont, CA (2002). https://raw.githubusercontent.com/relaton/relaton-data-ietf/master/data/reference.RFC.3339.xml

3. Vretanos, P.A.: OGC Testbed-17: Features and Geometries JSON CRS Analysis of Alternatives Engineering Report. OGC OGC 21-018, Open Geospatial Consortium (2021)

4. Taleisnik, S.: OGC Testbed-17: Aviation API Engineering Report. OGC 21-039r1, Open Geospatial Consortium (2021)

5. Geographic information — Features and geometry — Part 2: Metrics (2019)

6. Geographic information — Simple feature access — Part 1: Common architecture (2011)

7. JSON Schema: A Media Type for Describing JSON Documents, https://json-schema.org/draft/2020-12/json-schema-core.html