OGC® DOCUMENT: 22-024R2

External identifier of this OGC® document: http://www.opengis.net/doc/PER/T18-D002

TESTBED-18: FILTERING SERVICE AND RULE SET ENGINEERING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2022-12-24
Approval Date: 2023-01-19
Publication Date: 2023-06-16
Editor: Sergio Taleisnik

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

This Testbed-18 (TB-18) Filtering Service and Rule Set Engineering Report (ER) documents best practices identified for features filtering and describes in detail how filtering can be decoupled from data services. Further, this ER describes how filtering rules can be provided to Filtering Services at runtime.

II EXECUTIVE SUMMARY

Previous OGC work addressed the challenges of increasing interoperability between aviation data services. Recently, the OGC community has developed a new family of standardized OpenAPI-based Web APIs for various geospatial resource types. These new OGC API Standards have the potential to enhance the way in which consumers can access geospatial data from various sources. OGC Testbed-16 brought together previous work on the development of OGC API Standards, the use of semantics to enrich data and SWIM data processing, and demonstrated an OpenAPI-based API implementation instance serving SWIM data. OGC Testbed-17 took lessons learned and recommendations from Testbed-16 and focused on further testing the value of standards-based APIs within the SWIM program.

OGC API-Features endpoints define their filtering capabilities. Filtering is standardized across different parts of OGC API-Features (see section Previous Work). As of December 2022, two parts were still in draft status. Advanced filtering capabilities require sophisticated server software. Not all data providers will be able to operate such a powerful service endpoint. FAA SWIM Data Services currently produce data from the National Airspace System (NAS) to consumers using various protocols and service offerings in both synchronous and asynchronous messaging formats. OGC Testbed-18 explored filtering mechanisms for feature data served by OGC API-Features instances. The experiments included filtering of native and fused SWIM data and experimented with filtering services.

The research questions for the Advanced Filtering of SWIM Feature Data Task were as follows.

- How does filtering of SWIM data served by OGC API-Features endpoints work?
- Is the metadata required by the various OGC API-Features parts sufficient to allow clients to fully understand the filtering capabilities of a service endpoint?
- OGC API — Features — Part 3: Filtering and the Common Query Language (CQL) supports queryables that are not directly represented as resource properties in the content schema of the resource. Is it possible to identify best practices for their usage?
- Clients may know the content schema of offered resources. How best to use this knowledge for advanced filtering beyond what is defined in particular in OGC API — Features — Part 3: Filtering and the Common Query Language (CQL)?

- How does a filtering service look that allows advanced filtering for rather simple OGC API-Features-based SWIM data endpoints?
- How does such a service work in situations where a data publisher has restricted filtering on certain properties (for example, because the backend datastore has not been configured to allow high-performance queries on those properties)?
- How can a client application support a customer that has knowledge of the content schema of an offered resource in the creation of filter statements? What are the key requirements for a developer GUI that allows visualization and management of these filtering tools?
- Is it possible to easily create a new filtered dataset by creating machine readable filtering rules based on the metadata required by the OGC API-Features standards? How can these rules be provided to the Filtering Service at runtime?

To answer these questions, this Testbed-18 Task was organized into the development and testing of a system of six interconnected components, as seen on Figure 1.

- **Façades for SWIM services with simple filtering mechanisms**. Retrieving aviation data from multiple SWIM services and serving these data through APIs built based on OGC API standards featuring basic filtering mechanisms. Three Façades were built.
  - The OGC API-Features Façade 1 (identified collectively as *D100*): Four APIs built to serve NOTAMs, Airport Layouts, and Airspaces
  - The OGC API-Features Façade 2 (identified collectively as *D101*): Three APIs built to serve aeronautical, flight, and weather features.
  - An extra façade, not originally included in the Task architecture, was offered in-kind by the company *Skymantics*, and was named OGC API-Features Façade 3: An API built to serve flight plans from the SFDPS (FAA) Service.
- **Components that serve aviation data with advanced filtering mechanisms**. Two filtering services were built, each one featuring an API.
  - The Filtering Service 1 (identified as *D102*): Built to serve SWIM data from D100 with advanced filtering mechanisms.
  - The Filtering Service 2 (identified as *D103*): Built to serve SWIM data from all three façades with advanced filtering mechanisms.

- **Client components to demonstrate consumption of filtered data and configuration of filtering mechanisms.** Two clients were built: One meant to serve an aviation domain expert and the other to serve a developer of aviation software applications.
  - The Business User Client (identified as *D104*): A client built to query filtering services and demonstrate the usage of advanced filtering mechanisms.
  - The Developer Client (identified as *D105*): A client built to define filter statements that can be expressed in a machine-readable way and exchanged with the filtering services.

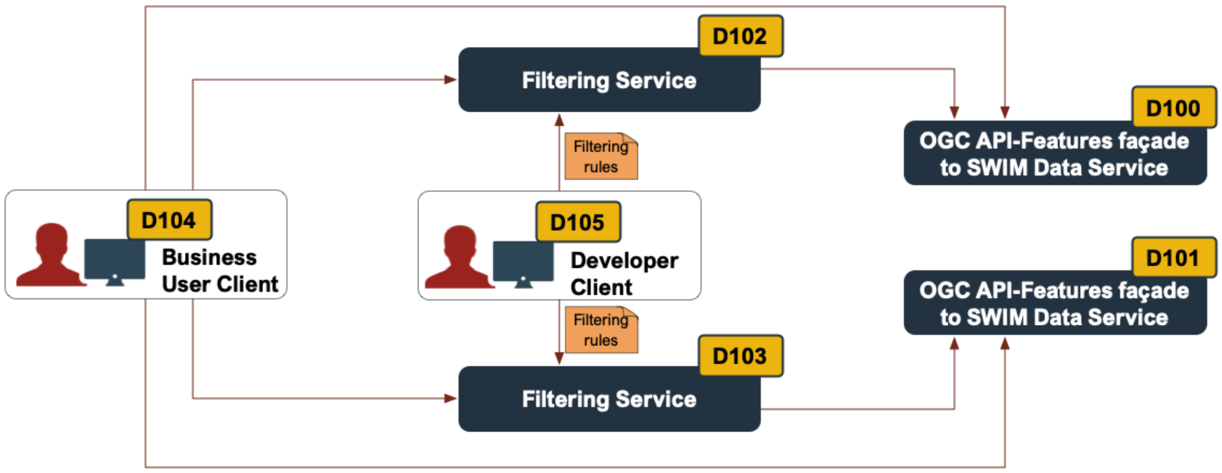


Figure 1 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task

All components were successfully developed and tested. This ER captures the operations, conformance classes, schemas, and processes meant to support an API with advanced filtering mechanisms. A comprehensive analysis of the research questions is included in a separate chapter.

III KEYWORDS

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

testbed, web service, api, standard, filter, SWIM, aviation

IV PREFACE

It is possible 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.

V SECURITY CONSIDERATIONS

No security considerations have been made for this document.

VI SUBMITTERS

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

Name	Organization	Role
Sergio Taleisnik	Skymantics, LLC	Editor
Clemens Portele	interactive instruments GmbH	Contributor
Eugene Yu	George Mason University	Contributor
Jérôme Jacovella-St-Louis	Ecere Corporation	Contributor
Patrick Dion	Ecere Corporation	Contributor
Mohammad Moallemi	Concepts Beyond LLC	Contributor

1

SCOPE

1 SCOPE

This OGC Testbed 18 Engineering Report (ER) documents best practices identified for features filtering and describes in detail how filtering can be decoupled from data services. This includes how filtering rules can be provided to Filtering Services at runtime. The ER specifies operations, schemas, and processes to support search and filtering of features. The ER also documents lessons learned and recommendations for future work.

2

NORMATIVE REFERENCES

2 NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Open API Initiative: **OpenAPI Specification 3.0.2**, 2018 https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md

van den Brink, L., Portele, C., Vretanos, P.: OGC 10-100r3, **Geography Markup Language (GML) Simple Features Profile**, 2012 http://portal.opengeospatial.org/files/?artifact_id=42729

W3C: **HTML5**, W3C Recommendation, 2019 http://www.w3.org/TR/html5/

Schema.org: http://schema.org/docs/schemas.html

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: IETF RFC 2616, *Hypertext Transfer Protocol — HTTP/1.1*. RFC Publisher (1999). https://www.rfc-editor.org/info/rfc2616.

E. Rescorla: IETF RFC 2818, *HTTP Over TLS*. RFC Publisher (2000). https://www.rfc-editor.org/info/rfc2818.

G. Klyne, C. Newman: IETF RFC 3339, *Date and Time on the Internet: Timestamps*. RFC Publisher (2002). https://www.rfc-editor.org/info/rfc3339.

M. Nottingham: IETF RFC 8288, *Web Linking*. RFC Publisher (2017). https://www.rfc-editor.org/info/rfc8288.

H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub: IETF RFC 7946, *The GeoJSON Format*. RFC Publisher (2016). https://www.rfc-editor.org/info/rfc7946.

3

TERMS, DEFINITIONS AND ABBREVIATED TERMS

3 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.

3.1. Terms and definitions

3.1.1. **Application Programming Interface (API)**

an interface that is defined in terms of a set of functions and procedures and enables a program to gain access to facilities within an application [8].

3.1.2. **Façade Service**

a component that fetches data from a specific data source and makes it available through its own interface [10]. The main reason for building this type of service is the difficulty or inability to modify the original data source with the intent of modifying:

- the underlying structure of the API; and
- the format in which the data are made available.

3.1.3. filter expression

predicate encoded for transmission between systems.

Example CQL2-Text or CQL2-JSON are examples how a predicate can be encoded as a filter expression.

[**SOURCE:** [14]]

3.1.4. parameterized stored query

a **stored query** that has one or more parameters.

Note 1 to entry: When executing a parameterized stored query, the user has to provide parameter values for each parameter of a stored query. If a parameter has a default value, a parameter can be omitted from the request to execute the stored query.

3.1.5. Predicate

set of computational operations applied to a data instance which evaluate to true or false.

Note 1 to entry: In relational algebra, this is called a selection.

[**SOURCE:** [5]]

3.1.6. property selection

operation to create a copy of a data instance, restricted to a subset of the properties of the data instance.

Note 1 to entry: In relational algebra, this is called a projection. This term was used in the OGC Web Feature Service standard, but the Features API SWG has decided not to use the term, because in the context of geographic information the term “projection” is closely associated with map projections and causes confusion if used with a different meaning.

Note 2 to entry: This topic is one of the future parts that the Features API SWG is starting to work on. See the initial proposal.

3.1.7. **query**

request for data from a dataset.

Note 1 to entry: A query will at least identify the data that the query operates on, the predicate(s) used to select the result set, the properties of the data instances that should be included in the response, the order in which the data instances should be included in the response, and the maximum number of data instances in the response.

3.1.8. **queryable**

a token that represents a property of a resource that can be used in a **filter expression**.

[**SOURCE:** [14]]

3.2. **returnable**

a token that represents a property of a resource that can be included in a representation of the resource.

Note 1 to entry: The term has been introduced in the proposal for the Schemas extension of OGC API Features, so far without a definition.

Note 2 to entry: APIs implementing OGC API Features will include all returnables in a response unless the property has no value for the instance or if the property is not included in the list of requested properties (see **property selection**).

3.3. sortable

a token that represents a property of a resource that can be used to sort a collection of resource instances.

NOTEThe term has been introduced in OGC API Records without a definition. Note that the content will be moved from OGC API Records to a new part of OGC API Features.

3.3.1. sorting

operation to order the data instances in a set based on the values of selected properties of each data instance.

Note 1 to entry: This capability is currently part of OGC API Records, but will be moved to a new part of OGC API Features.

3.3.2. Standardized API

an API that is intended to be deployed by multiple API providers with the same API definition.

Note 1 to entry: The only difference between the API definitions will be the URL(s) of the API deployment. All other aspects are identical (resources, content schemas, content constraints and business rules, content representations, parameters, etc.) so that any client that can use one deployment of the standardized API definition can also use all other deployments, too.

Note 2 to entry: If the API provides access to data, different deployments of the API will typically share different content.

3.3.3. Standards-based API

an API that conforms to one or more conformance classes specified in one or more standards.

Note 1 to entry: Since almost all APIs will conform to some standard, the term is usually used in the context of a specific standard or a specific family of standards. This ER considers Web APIs with a specific focus on the OGC API standards. Therefore, whenever the term is used in this ER, it is meant as an alias for an API that conforms to one or more conformance classes as defined in the OGC API standards.

3.3.4. stored query

a predefined **query** that is available a resource in a Web API.

Note 1 to entry: Stored queries can be used for two purposes. The first is to save users of the API the effort of creating their own queries. The second is to constrain what users may receive and how. The second purpose was the main purpose in the testbed where a developer creates stored queries for use by business users.

Note 2 to entry: The Testbed 18 requirements state that “filtering rules” must be defined “in some machine readable way.” This Engineering Report uses **stored query** in the OGC standards and **filtering rule** in the Testbed 18 requirements as synonyms. The “JSON file with filtering rules” in Figure 11 of the Testbed 18 Call for Participation is a query encoded as a JSON object that includes **filter expressions** using CQL2-Text or CQL2-JSON.

3.3.5. SWIM Data

any data provided through the SWIM System.

3.3.6. Web API

an API using an architectural style that is founded on the technologies of the Web [9].

Note 1 to entry: Best Practice 24: Use Web Standards as the foundation of APIs in the W3C Data on the Web Best Practices [9] provides more detail.

Note 2 to entry: A Web API is basically an API based on the HTTP standard(s).

3.4. Abbreviated terms

AIXM	Aeronautical Information Exchange Model
API	Application Programming Interface
CQL2	OGC Common Query Language
CRS	Coordinate Reference System
ER	Engineering Report
FAA	Federal Aviation Administration
JSON	JavaScript Object Notation
NAS	National Airspace System

NOTAM	Notice to Airmen
OGC	Open Geospatial Consortium
SFDPS	SWIM Flight Data Publication Service
SWIM	System Wide Information Management
TB	Testbed
TIE	Technology Integration Experiment
WFS	Web Feature Service

4

INTRODUCTION

4 INTRODUCTION

4.1. Background

4.1.1. SWIM

The System-Wide Information Management (SWIM) initiative supports the sharing of aeronautical, air traffic, and weather information. This is accomplished by providing communications infrastructure and architectural solutions for identifying, developing, provisioning, and operating a network of highly distributed, interoperable, and reusable services.

As part of the SWIM architecture, data providers create services for consumers to access their data. Each service is designed to be stand-alone. However, the value of data increases when combined with other data. Real-world situations are often not related to data from one but instead from several SWIM feeds. The need for consumers to retrieve data from several SWIM services creates the need of interoperability between those services.

4.1.2. OGC API Standards

For several years, the OGC members have worked on developing a family of OGC Web API standards for various geospatial resource types. These OGC API Standards are defined using OpenAPI. As the OGC API standards keep evolving, are approved by the OGC, and are implemented by the community, the aviation industry can subsequently experiment and implement them.

The following OGC API Standards and Draft Specifications were used for the development of APIs during Testbed 18.

OGC API – Features: A multi-part standard that defines the capability to create, modify, and query vector feature data on the Web and specifies requirements and recommendations for APIs to follow a standard way of accessing and sharing feature data. It currently consists of the following four parts.

- OGC API — Features — Part 1: Core. Approved September 2019, this standard defines discovery and query operations. [12]
- OGC API — Features — Part 2: Coordinate Reference Systems by Reference. This standard, approved October 2020, extends the core capabilities specified in Part 1: Core with the ability to use coordinate reference system (CRS) identifiers other than the defaults defined in the core. [13]

- Draft OGC API — Features — Part 3: Filtering. Part 3 specifies an extension to the OGC API — Features — Part 1: Core standard that defines the behavior of a server that supports enhanced filtering capabilities. [14]
- Draft OGC API — Features — Part 4: Create, Replace, Update, and Delete. Part 4 specifies an extension that defines the behavior of a server that supports operations to add, replace, modify, or delete individual resources from a collection. [15]
- Proposal OGC API — Features — Part 5: Search. The proposal is an initial draft for Query resources that support queries on multiple collections in the same request, parameterized stored queries and join queries. [6]

A Common Query Language (CQL2) is being developed together with Part 3 to standardize a language that is recommended for filter expressions. [16]

OGC API – Processes: An approved (August 2021) OGC API Standard, specifies requirements for implementing a Web API that enables the execution of computing processes and the retrieval of metadata describing their purpose and functionality. Typically, these processes combine raster, vector, coverage, and/or point cloud data with well-defined algorithms to produce new information. [1]

Draft OGC API – Tiles: This recent OGC API Standard defines how to discover which resources offered by the Web API can be retrieved as tiles, retrieve metadata about the tile set (including the supported tile matrix sets, the limits of the tiled set inside the tile matrix set), and how to request a tile. [2]

Draft OGC API – Styles: This draft OGC API specifies building blocks for implementing OGC Standards based Web APIs that enable map servers, clients, and visual style editors to manage and fetch styles. [3]

4.1.3. Exploration of OGC API Standards by SWIM

Over the years, the FAA and the OGC have jointly explored making SWIM data more easily accessible and more valuable. As part of these past efforts, Testbed-16 brought together previous work on the development of OGC APIs, the use of semantics to enrich data, and SWIM data processing. The objectives were to deliver the first demonstration of an OpenAPI-based API serving SWIM data, a component generating aviation Linked Data, and two client applications querying and displaying that data [15].

Two of TB-16 recommendations were to integrate OGC API requirement classes within SWIM Data Services and to demonstrate interoperability between diverse Aviation APIs [15]. In order to advance these recommendations, TB-17 focused on the development of eleven APIs based on OGC API Standards and the completion of Technology Integration Experiments (TIEs) between these APIs.

During TB-16, the development of the API serving aviation data resulted in numerous lessons learned and recommendations [15]. TB-16 saw the development of one aviation-related API based on an OGC API Standard (OGC API — Features). The APIs developed during TB-17 ([4])

addressed many of those lessons learned and implemented additional OGC API Standards (draft and approved) which have been maturing since. This process is reflected in Figure 2.

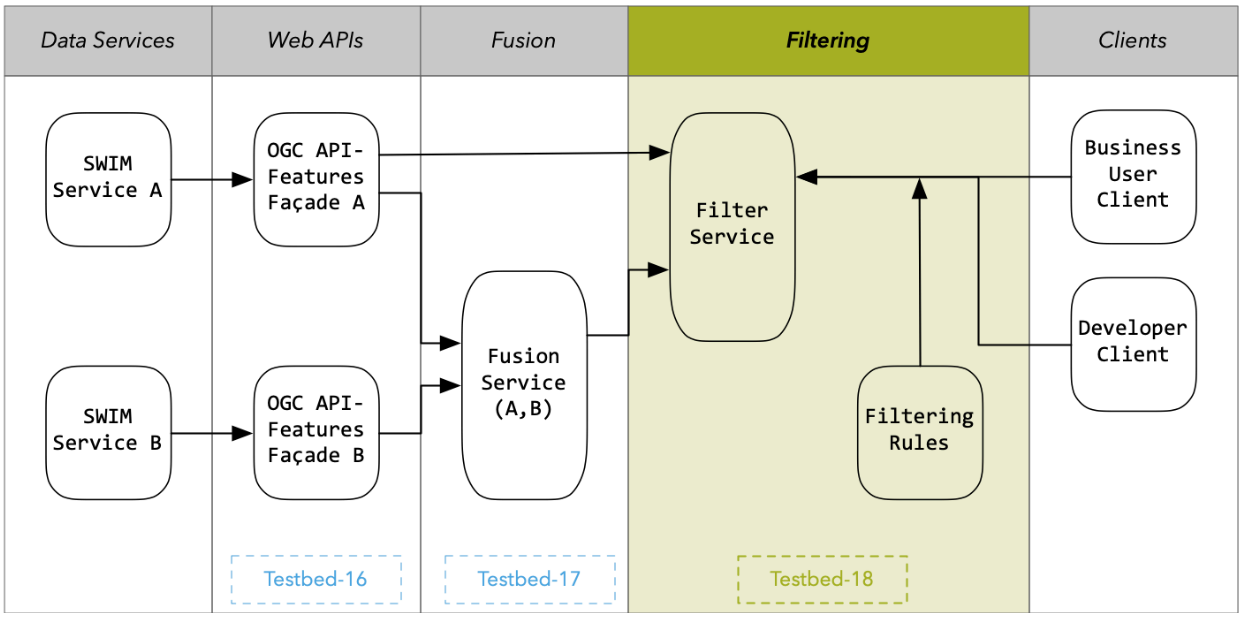


Figure 2 — History of OGC experiments to enhance SWIM

4.2. Requirements Statement

Testbed-18 required investigating the potential of filtering using OGC API Standards in the context of the SWIM Program.

The original goals of the TB-18 Advanced Filtering of SWIM Feature Data Task were as follows.

- Experiment with OGC API-Features filtering mechanisms.
- Explore if best practices for advertising filtering capabilities are required beyond what is already defined in the various OGC API-Features Parts.
- Demonstrate advanced filtering in situations where the data endpoints support only rudimentary filtering by introducing a new service type “Filtering Service.”
- Allow filtering rules for a specific data service to be provided at runtime in a machine-readable manner.

The research questions for the Advanced Filtering of SWIM Feature Data Task were as follows.

- How does filtering of SWIM data served by OGC API-Features endpoints work?
- Is the metadata required by the various OGC API-Features parts sufficient to allow clients to fully “understand” the filtering capabilities of a service endpoint?

- OGC API — Features — Part 3: Filtering and the Common Query Language (CQL) supports queryables that are not directly represented as resource properties in the content schema of the resource. Is it possible to identify best practices for their usage?
- Clients may know the content schema of offered resources. How best to use this knowledge for advanced filtering beyond what is defined in OGC API — Features — Part 3: Filtering and the Common Query Language (CQL)?
- How does a filtering service look that allows advanced filtering for rather simple OGC API-Features-based SWIM data endpoints?
- How does such a service work in situations where a data publisher has restricted filtering on certain properties (for example, because the backend datastore has not been configured to allow high-performance queries on those properties)?
- How can a client application support a customer who has knowledge of the content schema of an offered resource in the creation of filter statements? What are the key requirements for a developer GUI that supports visualization and management of these filtering tools?
- Is it possible to easily create a new filtered dataset by creating machine readable filtering rules based on the metadata required by the OGC API-Features standards? How can these rules be provided to the Filtering Service at runtime?

4.3. Functional Overview

As shown in Figure 3, the Advanced Filtering of SWIM Feature Data Task architecture was organized into a system of seven interconnected components. All seven components were developed simultaneously throughout the Testbed, with permanent communication and cooperation among participant organizations.

The components can be divided into the following three groups.

- **Façades for SWIM services with simple filtering mechanisms**. Retrieve aviation data from multiple SWIM services and serve these data through APIs built based on OGC API Standards featuring basic filtering mechanisms. Three Façades were built.
  - The OGC API-Features Façade 1 (identified collectively as *D100*): Four APIs built to serve NOTAMs, Airport Layouts, and Airspaces
  - The OGC API-Features Façade 2 (identified collectively as *D101*): Three APIs built to serve aeronautical, flight, and weather features.
  - An extra façade, not originally included in the Task architecture, was offered in-kind by the company *Skymantics*, and was named OGC API-Features Façade 3: An API built to serve flight plans from the SFDPS (FAA) Service.

- **Components that serve aviation data with advanced filtering mechanisms**. Two filtering services were built, each one featuring an API.
    - The Filtering Service 1 (identified as *D102*): Built to serve SWIM data from D100 with advanced filtering mechanisms.
    - The Filtering Service 2 (identified as *D103*): Built to serve SWIM data from all three façades with advanced filtering mechanisms.
- **Client components to demonstrate consumption of filtered data and configuration of filtering mechanisms**. Two clients were built: One meant to serve an aviation domain expert and the other to serve a developer of aviation software applications.
    - The Business User Client (identified as *D104*): A client built to query filtering services and demonstrate the usage of advanced filtering mechanisms.
    - The Developer Client (identified as *D105*): A client built to define filter statements that can be expressed in a machine-readable way and exchanged with the filtering services.

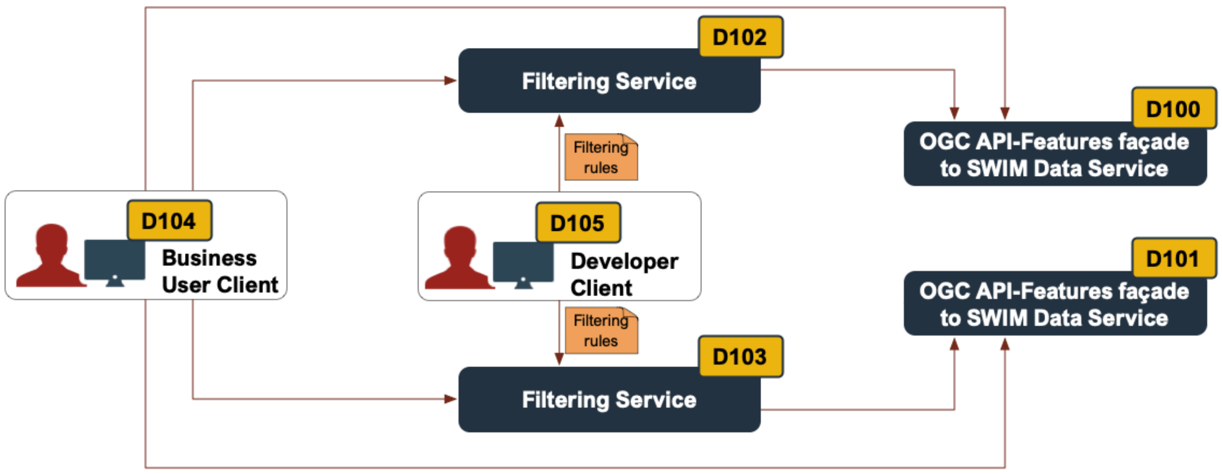


Figure 3 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task

4.3.1. Component Interactions

The following two figures illustrate the intended interactions between the components described in the Work Items & Deliverables section of this ER. The two figures illustrate the workflows for using the filtering service for data subsetting (Figure 4) from the perspective of a business client and for configuring the filtering service at runtime (Figure 6) from the perspective of the filtering rules developer.

In the first workflow, illustrated in Figure 4, an *OGC API-Features façade to SWIM Data Service* data service offers insufficient filtering capabilities to its customers. The Business User Client does not want to access large data sets and then perform filtering itself. Instead, the client wants to make use of a *Filtering Service* that can handle the filtering of the data and provide the subset of the data that the client is interested in. If the filtering service receives a data request from the client, it connects to the data service to access the necessary data, filters out everything that is not requested by the client, and eventually delivers the result to the client.

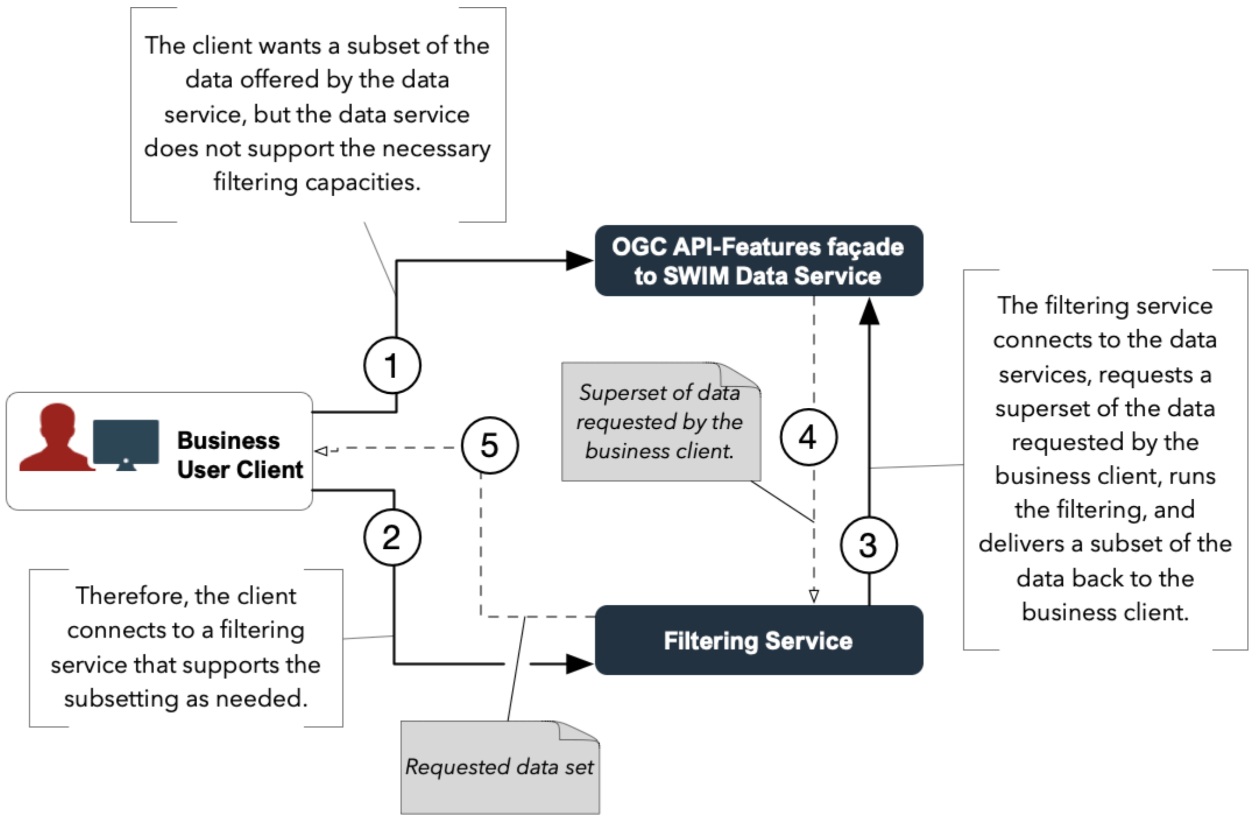

The client wants a subset of the data offered by the data service, but the data service does not support the necessary filtering capacities.
OGC API-Features façade to SWIM Data Service
1
Business User Client
5
Superset of data requested by the business client.
4
The filtering service connects to the data services, requests a superset of the data requested by the business client, runs the filtering, and delivers a subset of the data back to the business client.
2
3
Therefore, the client connects to a filtering service that supports the subsetting as needed.
Filtering Service
Requested data set

Figure 4 — Workflow from the perspective of a business user that needs filtered data

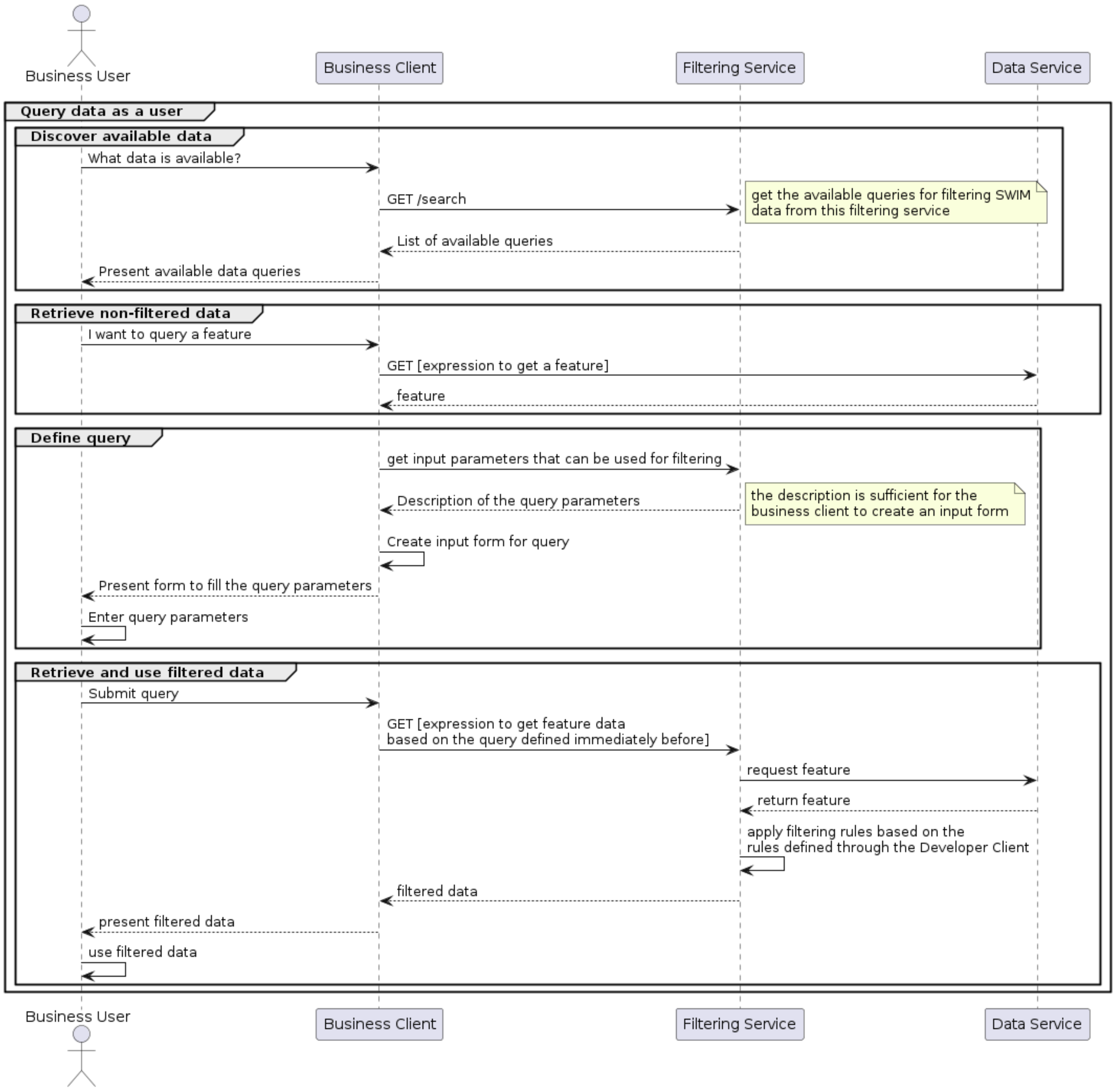

Business User
Business Client
Filtering Service
Data Service
Query data as a user
Discover available data
What data is available?
GET /search
get the available queries for filtering SWIM data from this filtering service
List of available queries
Present available data queries
Retrieve non-filtered data
I want to query a feature
GET [expression to get a feature]
feature
Define query
get input parameters that can be used for filtering
Description of the query parameters
the description is sufficient for the business client to create an input form
Create input form for query
Present form to fill the query parameters
Enter query parameters
Retrieve and use filtered data
Submit query
GET [expression to get feature data based on the query defined immediately before]
request feature
return feature
apply filtering rules based on the rules defined through the Developer Client
filtered data
present filtered data
use filtered data

Figure 5 — First Workflow Sequence Diagram

The second workflow, illustrated in Figure 6, demonstrates how a filtering service can be configured at run time. The assumption is that the Developer Client is aware of the API characteristics of the data service as well as the content schema of the data served by the data server. Based on both, the client supports the user with a GUI in the definition of the filtering rules. The user can then register these rules with the filtering service, which is now configured to run the data service specific filtering.

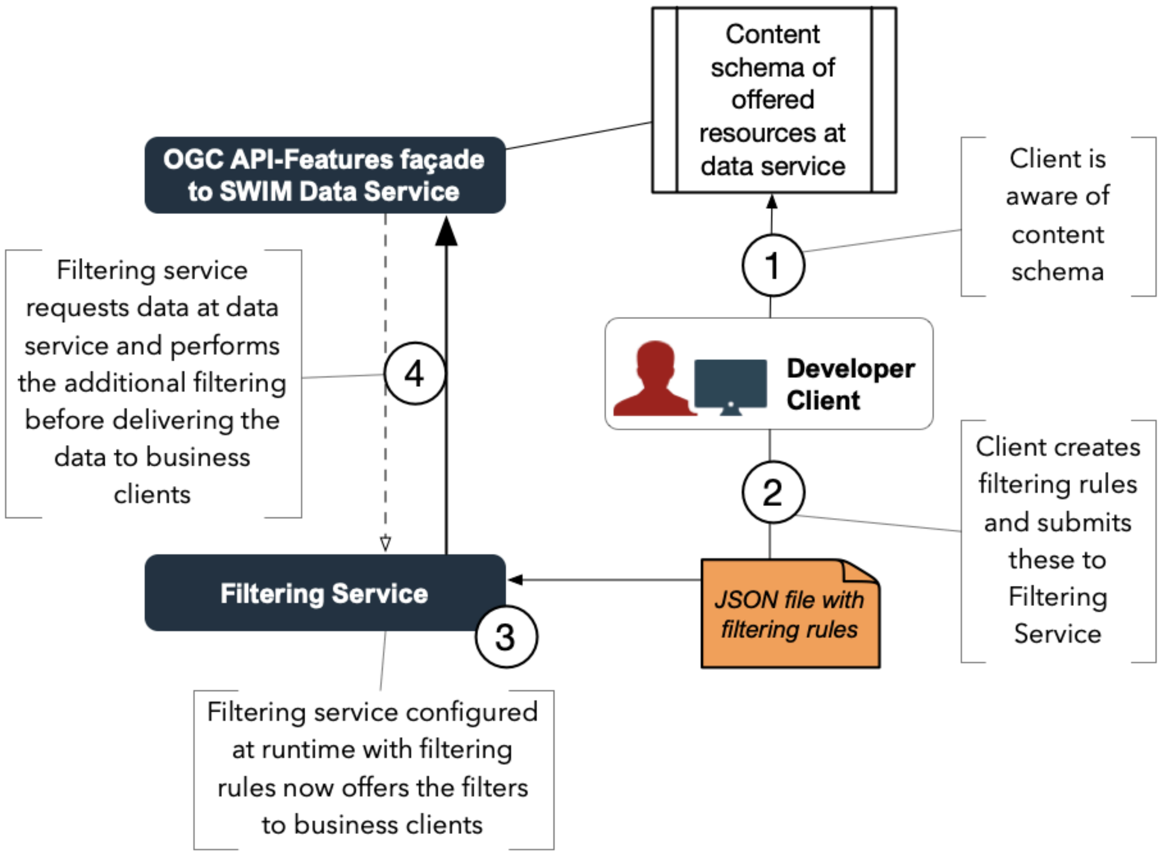

Content schema of offered resources at data service
OGC API-Features façade to SWIM Data Service
Client is aware of content schema
1
Filtering service requests data at data service and performs the additional filtering before delivering the data to business clients
4
Developer Client
2
Client creates filtering rules and submits these to Filtering Service
Filtering Service
JSON file with filtering rules
3
Filtering service configured at runtime with filtering rules now offers the filters to business clients

Figure 6 — Workflow from the perspective of a filtering rules developer

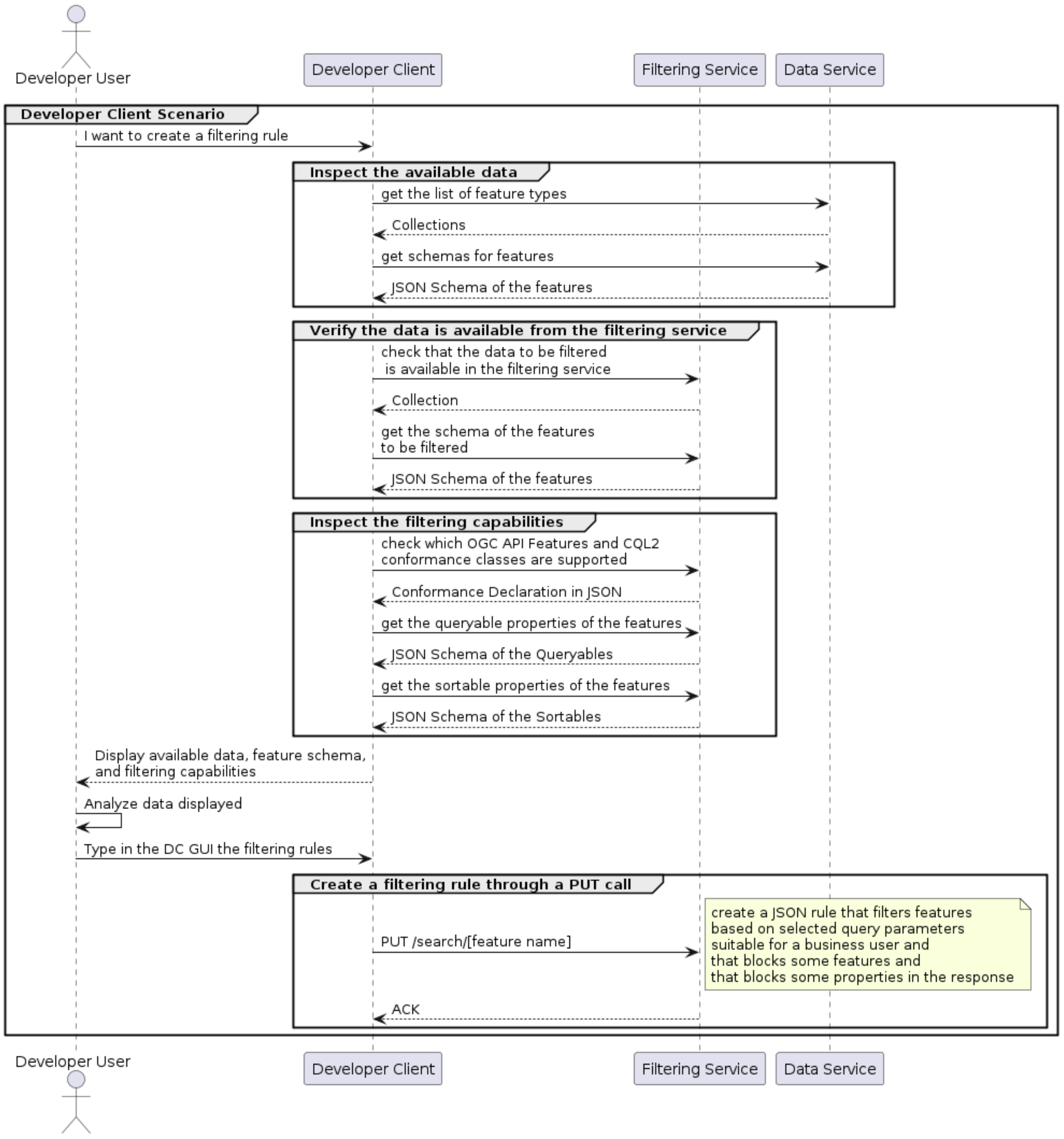

Developer User
Developer Client
Filtering Service
Data Service
Developer Client Scenario
I want to create a filtering rule
Inspect the available data
get the list of feature types
Collections
get schemas for features
JSON Schema of the features
Verify the data is available from the filtering service
check that the data to be filtered is available in the filtering service
Collection
get the schema of the features to be filtered
JSON Schema of the features
Inspect the filtering capabilities
check which OGC API Features and CQL2 conformance classes are supported
Conformance Declaration in JSON
get the queryable properties of the features
JSON Schema of the Queryables
get the sortable properties of the features
JSON Schema of the Sortables
Display available data, feature schema, and filtering capabilities
Analyze data displayed
Type in the DC GUI the filtering rules
Create a filtering rule through a PUT call
PUT /search/[feature name]
create a JSON rule that filters features based on selected query parameters suitable for a business user and that blocks some features and that blocks some properties in the response
ACK

Figure 7 — Second Workflow Sequence Diagram

5

OPERATIONS

5 OPERATIONS

Table 1 — Operations

ENDPOINT	METHOD	REQUEST	RESPONSE	DESCRIPTION
`/search`	GET	n/a	List of stored queries	Fetch the stored queries on the server
`/search`	POST	A query expression	A feature collection	Execute an ad-hoc query
`/search/{queryId}`	GET	n/a	A feature collection	Execute the stored query; parameters are submitted as query parameters
`/search/{queryId}`	POST	URL-encoded form with query parameters	A feature collection	Execute this stored query
`/search/{queryId}`	PUT	A query expression	n/a	Create or update a stored query
`/search/{queryId}`	DELETE	n/a	n/a	Delete this stored query
`/search/{queryId}/definition`	GET	n/a	A query expression	Get the definition of the stored query
`/search/{queryId}/parameters`	GET	n/a	JSON Schema of an object where each parameter is a property	Get the definition of the parameters
`/search/{queryID}/parameters/{parameterID}`	GET	n/a	JSON Schema of the parameter	Get the details of a query parameter

5.1. Conformance Classes

- Core: Support executing stored queries.
- Parameterized Stored Queries: Support executing parameterized stored queries.
- Manage Stored Queries: Support reading, creating, replacing, and deleting stored queries.
- Ad-hoc Queries: Support executing ad-hoc queries.

Table 2 — Conformance Classes

ENDPOINT	METHOD	CONFORMANCE CLASS
`/search`	GET	Core
`/search`	POST	Ad-hoc Queries
`/search/{queryId}`	GET	Core
`/search/{queryId}`	POST	Core
`/search/{queryId}`	PUT	Manage Stored Queries
`/search/{queryId}`	DELETE	Manage Stored Queries
`/search/{queryId}/definition`	GET	Manage Stored Queries
`/search/{queryId}/parameters`	GET	Parameterized Stored Queries
`/search/{queryID}/parameters/{parameterID}`	GET	Parameterized Stored Queries

Note that `/search/{queryId}/definition` is part of the 'Manage Stored Queries' Conformance Class. This is because in the current design `GET` on `/search` returns only the main query metadata and, if applicable, the parameter descriptions. This supports use cases where a query provider wants to keep the query expressions hidden from regular users. That is, the query expression of a stored query should only be visible to those managing the query. Consequently, `GET` on `/search/{queryId}/definition` is part of the Manage Stored Queries conformance class.

6 SCHEMAS

6 SCHEMAS

6.1. Query Expression

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "QueryExpression.json",
  "oneOf": [
    {
      "allOf": [
        {"$ref": "#/$defs/query"},
        {
          "type": "object",
          "properties": {
            "title": {"type": "string"},
            "description": {"type": "string"},
            "limit": {"$ref": "#/$defs/limit"},
            "parameters": {"$ref": "Parameters.json"}
          }
        }
      ]
    },
    {
      "type": "object",
      "required": ["queries"],
      "properties": {
        "title": {"type": "string"},
        "description": {"type": "string"},
        "queries": { "type": "array", "minItems": 1, "items": {"$ref": "#/
$defs/query"} },
        "filter": {"$ref": "#/$defs/filter"},
        "filterOperator": { "type": "string", "enum": ["and", "or"], "default":
 "and" },
        "properties": {"$ref": "#/$defs/properties"},
        "limit": {"$ref": "#/$defs/limit"},
        "parameters": {"$ref": "Parameters.json"}
      }
    }
  ],
  "$defs": {
    "query": {
      "type": "object",
      "required": ["collections"],
      "properties": {
        "collections": { "type": "array", "minItems": 1, "items": {"type":
"string"} },
        "filter": {"$ref": "#/$defs/filter"},
        "properties": {"$ref": "#/$defs/properties"},
        "sortby": {"$ref": "#/$defs/sortby"}
      }
    },
    "filter": {"type": "object"},
    "properties": { "type": "array", "minItems": 1, "items": {"type": "string",
 "minLength": 1} },
    "sortby": {
      "type": "array",
      "minItems": 1,
      "items": {"type": "string", "pattern": "[+|-]?.+"}
    },
    "limit": {"type": "integer", "minimum": 1, "default": 1000, "maximum":
10000}
  }
}
```

A query expression can contain a single query (the properties "collections," "filter," "properties," and "sortby" are members of the query expression object) or multiple queries (a "queries" member with an array of query objects is present) in a single request.

For each query

- The value of "filter" is a CQL2 JSON filter expression.
- The value of "properties" is an array with the names of properties to include in the response.
- The value of "sortby" is used to sort the features in the response.
- Multiple entries in the "collections" member represent a join between the specified collections. Just like in SQL, the properties of each collection participating in the join are combined and presented in the features in the result set. Property names in the request and in the response have to be prefixed with the collection name plus a period ("."), e.g., "collection.property."
  - Support for joins was a stretch goal in Testbed 18 and not discussed or tested in detail.
  - There are several open questions related to joins that should be considered by the Features API SWG.
    - How should the id of the joined feature be assigned? A combination of all features in the tuple?
    - How should the primary geometry be assigned (in the GeoJSON representation) if there are multiple geometry properties?
    - Should it also be allowed to "nest" joined properties, so instead of `{ …, "col.prop1": 1, "col.prop2": "a", … }` encode it as `{ …, "col": { "prop1": 1, "prop2": "a" }, … }`?

For multiple queries

- If multiple queries are specified, the results are concatenated. The response is a single feature collection.
  - The feature ids in the response to a multi-collection query must be unique. Since the `featureId` of a feature only has to be unique per collection, they need to be combined with the `collectionId`. The server could determine how the id values are constructed or require a specific approach. The latter has the advantage that the source feature could be identified (if, e.g., a concatenation of the `collectionId` and `featureId` such

    as “apronelement.123456” is used). However, if this is a requirement, then maybe a “self” link in each feature would be cleaner? In addition, if an API already uses feature identifiers that are unique across all collections (e.g., a UUID), then a mandatory collection prefix would be unnecessary.

    - Another aspect is cases where a feature is included in the result set of multiple queries. The feature should be included only once in the result. However, if the ids are constructed in a way that there is no conflict, if the same feature is included multiple times (e.g., just an auto-incrementing index value as the id), maybe it could also be tolerated if the same feature is included more than once?

    - Another approach to both issues could be to return an array of feature collections in the response, one for each query. Similar to what was done in WFS 2.0. This seems cleaner and clearer as it avoids the “hacks” and issues discussed in the previous bullet items. Since `/search` is a different resource than `/items` this would be conceptually clean. This leaves the question of usability: a single feature collection is easier if the result is just fed into some GeoJSON tooling. However, in a JavaScript based client it seems there is no overhead to process such a response. It could also be beneficial for use with JSON-FG where each feature collection would more likely be homogeneous and could include metadata that simplify parsing the response.

- The direct members “filter” and “properties” represent “global” constraints that must be combined with the corresponding member in each query. The global and local property selection list are concatenated and then the global and local filters are combined using the logical operator specified by the “filterOperator” member.

    - The global member “filter” should only reference queryables that are common to all collections being queried. If a queryable is specified that is not defined or does not exist for a particular collection then the value of the property is `null`.

    - The global member “properties” should only reference presentables that are common to all collections being queried. If a presentable is specified that is not defined or does not exist for a particular collection then the property is omitted from the response.

General rules

- A “title” and “description” for the query expression can be added. Providing both is strongly recommended to explain the query to users.

- The “limit” member applies to the entire result set.

- Note that “sortby” will only apply per query. A global “sortby” would require that the results of all queries are compiled first and then the combined result set is sorted. This would not support “streaming” the response.

- In case of a parameterized stored query, the query expression may contain JSON objects with a member “$parameter.” The value of “$parameter” is an object with a member where the key is the parameter name and the value is a JSON schema describing the parameter. When executing the stored query, all objects with a “$parameter” member are replaced with the value of the parameter for this query execution. Comma-separated parameter values are converted to an array if the parameter is of type “array”.

- Parameters may also be provided in a top-level member “parameters” and referenced using “$ref”.

6.2. Stored Queries

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "StoredQueries.json"
  "type": "object",
  "properties": {
    "queries"    : { "type": "array" , "items": {"$ref": "#/$defs/
StoredQueryDescription"} },
    "links"      : { "type": "array" , "items": {"$ref": "#/$defs/Link"}
            },
    "title"      : { "type": "string"
            },
    "description": { "type": "string"
            }
  },
  "$defs": {
    "StoredQueryDescription": {
      "required": ["id"],
      "type": "object",
      "properties": {
        "id": {"type": "string"},
        "title": {"type": "string"},
        "description": {"type": "string"},
        "parameters": { "$ref": "Parameters.json" },
        "links": { "type": "array", "items": {"$ref": "#/$defs/Link"} }
      }
    },
    "Link": {
      "type": "object",
      "required": ["href", "rel"],
      "properties": {
        "href": {"type": "string"},
        "rel": {"type": "string"},
        "title": {"type": "string"},
        "type": {"type": "string"},
        "hreflang": {"type": "string"},
        "length": {"type": "integer"},
        "templated": {"type": "boolean", "default": false}
      }
    }
  }
}
```

6.3. Parameters

The parameters are described as a JSON object where each parameter is a property with its JSON Schema as its value.

Providing sufficient information that allows clients to generate meaningful queries is essential. Recommendation 1 in OGC API — Features — Part 3 has recommendations for schemas that are straightforward to parse by clients but that are expressive enough to allow clients to generate forms to provide parameter values.

In addition, if a parameter declares a default value, the API will use that default value if no value is provided in the request to execute a query.

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "Parameters.json"
  "type": "object",
  "description": "Each parameter is described by a property where each value
is a JSON Schema object.",
  "additionalProperties": {
    "$ref": "https://json-schema.org/draft/2019-09/schema"
  }
}
```

7

PROCESSES — PART 3 APPROACH

7 PROCESSES — PART 3 APPROACH

7.1. Pre-defining queries based on *Processes — Part 3* extension

The *OGC API — Features* Search extension (described in detail in the other section of this document) shares a lot in common with the idea of allowing `filter`, `properties` (for selection and derived fields/properties), and `sortBy` to qualify inputs in *OGC API — Processes — Part 3: Workflows and Chaining* (*input* and *output modifiers* requirements classes).

A well-known pass-through process (with support for collection input including filtering) could support an execution request with a syntax equivalent to the *Search* extension endpoint, similar to how the *OGC API — Routes* `/routes` endpoint shares a `POST` payload syntax with an eventual definition of a well-known routing process. Pre-defined queries could also be parameterized by deploying them as processes, as suggested in the *Deployable workflows* requirements class of *Processes — Part 3: Workflows and Chaining*.

The modifiers introduced include the same `filter`, `properties`, and `sortBy` parameters to qualify inputs originating from a data collection or process, whether they are local or remote, as well as outputs resulting from a process. In addition to the ability to select specific fields/properties, the `properties` parameter can also be used to derive new fields, for example using CQL2 arithmetic expressions.

Processes — Part 3 also defines a *Collection output* requirements class where the output of the workflow execution is either a dataset landing page (which can contain multiple collections), or a single collection.

For example, the following parameterized query, addressing similar use case as the one described further in section *8.3.5 — Support for the Search resources*'s *Example 2*, taking two parameters, a string enumeration named *composition* and a string array named *airports*, could be expressed in a Part 3-extended *OGC API — Processes* execution request (using *Collection input* and *Output modifiers*) as shown below.

Example 1 — Example parameterizable query as a Part 3-extended execution request

```
{
 "process" : "PassThrough",
 "inputs" : {
  "data" : [ {
    "collection" : "apronelement",
    "filter": {
      "op": "and",
      "args": [
        {
          "op": "=",
          "args": [
            {"property": "composition"},
            {
              "$input": {
                "composition": { "type": "string", "enum": ["CONC", "..."] }
              }
            }
          ]
        },
        {
          "op": "in",
          "args": [
            {"property": "airport"},
            {
              "$input": {
                "airports": {
                  "type": "array",
                  "items": { "type": "string", "enum": ["JFK", "EWR", "LGA", ".
..."] }
                }
              }
            }
          ]
        }
      ]
    },
    "properties": ["geometry", "airport", "type"],
    "sortBy": ["airport"]
  } ]
 }
}
```

In this example, `data` is an input defined in the *PassThrough* well-known process with multiplicity *1..** which is returned as the process output.

The `collection` property is defined in the *Collection input* requirements class of *OGC API — Processes — Part 3*, whereas the `filter`, `properties`, and `sortby` elements specifying a *cql2-json* filtering expression, selected properties and an ascending sort order by airport, are defined in Part 3's *Input modifiers* requirements class.

These field modifiers can also be used in the context of the *Output modifiers* requirements class together with the *Collection output* requirement class or with regular process execution outputs. In the context of a feature collection output, these query parameters building blocks also correspond to the functionality provided by *OGC API — Features — Part 3: Filtering*, as well as a planned extensions for *Coverages* and *Discrete Global Grid Systems*.

This execution request could be deployed as a new process, e.g., *ApronFiltering*, using the *Processes — Part 2: Deploy, Replace, Undeploy* extension's `POST` operation to `/processes` together with the *Deployable workflows* requirements class of Part 3. The resulting process would get listed at `/processes` with a process description including the input parameters (`composition` and `airports`) and could itself be executed by POSTing to `/processes/ApronFiltering/execution`.

Example 2 — Example execution request of parameterized query deployed as a process

```
{
  "inputs" : {
    "composition" : "CONC",
    "airports": [ "JFK", "LGA" ]
  }
}
```

Posting this execution request to the execution endpoint without a `Prefer:` header would result in a synchronous execution that returns the features. With support for the *Collection output* requirements class, specifying as parameter `response=collection` would instead return a collection description, as for a `GET` request to `/collections/{collectionId}` in *Common — Part 2* and *Features — Part 1*. With `response=landingPage`, a landing page would be returned for the filtered dataset, allowing to retrieve multiple collections.

Such virtual collections could also be published to a dataset API as a persistent collection, e.g., as `/collections/concApronsJFKLGA`, using a non-parameterizable execution request as the payload of a `POST` operation to `/collections` to create the dynamic collection, with a *Processes execution request* content media type to be registered, e.g., `application/ogcexecreq+json` as suggested in *Part 3 — Section 14. Media Types*.

During this Testbed-18 advanced filtering task Ecere successfully demonstrated the use of Part 3 extensions to pre-define filtering queries to the cascading service, including the deployment of a sample *PassThrough* process with support for the `filter` and `properties` modifiers, as well as for the *Collection Input* and *Collection output* requirements class, with the following limitations:

- the filters were expressed using the *cql2-text* encoding rather than *cql2-json*;
- the input and output were limited to a single collection;
- the `sortBy` modifier (and sorted feature collections in general) remained to be implemented; and
- support for parameterized queries and deployable workflows was not yet implemented.

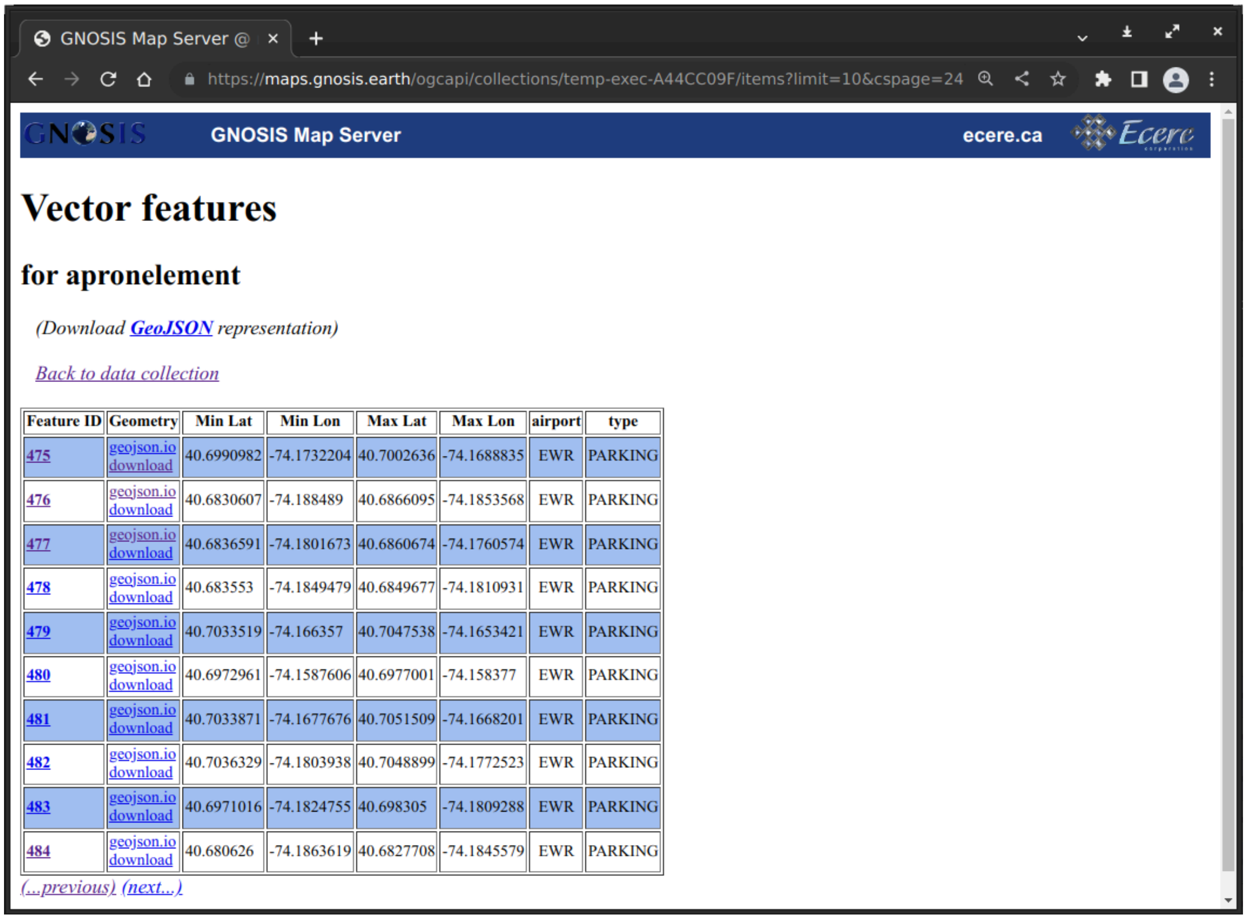
A sample pre-defined query is available from the endpoint:

https://maps.gnosis.earth/ogcapi/processes/PassThrough/execution?response=collection

including the following default pre-defined filter execution request:

Example 3 — Working execution request of filtering query from D100 cascaded collection

```
{
   "process" : "https://maps.gnosis.earth/ogcapi/processes/PassThrough",
   "inputs" : {
      "data" : [
         {
            "collection" : "https://maps.gnosis.earth/ogcapi/collections/swim:
d100_airports:apronelement",
            "filter" : "composition = 'CONC' and airport in ('JFK', 'EWR',
'LGA')",
            "properties" : [ "geometry", "airport", "type" ]
         }
      ]
   }
}
```


GNOSIS Map Server @
https://maps.gnosis.earth/ogcapi/collections/temp-exec-A44CC09F/items?limit=10&cspage=24
GNOSIS Map Server
ecere.ca

Vector features

for apronelement

(Download ***GeoJSON*** *representation)*

Back to data collection

Feature ID	Geometry	Min Lat	Min Lon	Max Lat	Max Lon	airport	type
475	geojson.io download	40.6990982	-74.1732204	40.7002636	-74.1688835	EWR	PARKING
476	geojson.io download	40.6830607	-74.188489	40.6866095	-74.1853568	EWR	PARKING
477	geojson.io download	40.6836591	-74.1801673	40.6860674	-74.1760574	EWR	PARKING
478	geojson.io download	40.683553	-74.1849479	40.6849677	-74.1810931	EWR	PARKING
479	geojson.io download	40.7033519	-74.166357	40.7047538	-74.1653421	EWR	PARKING
480	geojson.io download	40.6972961	-74.1587606	40.6977001	-74.158377	EWR	PARKING
481	geojson.io download	40.7033871	-74.1677676	40.7051509	-74.1668201	EWR	PARKING
482	geojson.io download	40.7036329	-74.1803938	40.7048899	-74.1772523	EWR	PARKING
483	geojson.io download	40.6971016	-74.1824755	40.698305	-74.1809288	EWR	PARKING
484	geojson.io download	40.680626	-74.1863619	40.6827708	-74.1845579	EWR	PARKING

(...previous) (next...)

Figure 8 — Paging through an output collection resulting from the above filter query pre-defined *using OGC API - Processes - Part 3*

7.2. Cross-collections queries

Among advanced filtering capabilities are cross-collections queries, whereby the server is instructed to perform a join between data sources (which could potentially be hosted in two different servers) as a multi-collection query. Although there was no time to implement this capability during the Testbed, some thinking and discussion focused on researching that capability. Whereas the search extension defines a new endpoint at `/search`, Ecere suggested supporting cross-collection queries for the usual `/items` endpoint. An example `GET` request would appear as follows:

```
GET /collections/apronelement/items?
   collections=apron&
   properties=*,apron.otherProperty&
   filter=associatedApron=apron.id and airport in (JFK, EWR, LGA)&
   sortby=airport&
   limit=1000
```

Figure 9

If there are multiple airport aprons matching the same apronelement, this would likely return more entries than available in the *apronelement* collection. In this case, the items IDs would need to be disambiguated and would not correspond to the typical `/collections/apronelement/items/{itemId}`.

The following request illustrates a weather use case, where wind speed information could be available either as *OGC API — Features* or as *OGC API — Coverages* (in the case of a coverage, `winds.geometry` would refer to the geometry of each cell):

```
GET /collections/flightRoutes/items?
   collections=https://weather.com/ogcapi/collections/winds&
   filter=winds.speed > 100 and s_intersects(geometry, winds.geometry) and
departingAirport in (JFK, EWR, LGA)&
   sortby=-winds.speed,departingAirport&
   limit=1000
```

Figure 10

If the winds collection supports *OGC API — Environmental Data Retrieval (EDR)*, the flight routes service could use a trajectory request including the flight route geometry to the weather data API, as one potential way to make this efficient. This would avoid the client fetching the full set of weather data, then transmitting it all again to the flight route service, exchanging the full data collection twice, when in fact all that may have been needed is for the first service to send the smaller flight routes geometry to the second service in a trajectory request. In addition, the two services may often be hosted nearby (e.g., both being hosted on Amazon Web Services), while the client is located elsewhere.

8

CONCLUSIONS

8 CONCLUSIONS

8.1. Research questions

The CFP included several research questions that drive the architecture used in Testbed 18 Filter activities. This section is intended to explain how this architecture addressed the research questions.

How does filtering of SWIM data served by OGC API-Features endpoints work?

See the sequence diagram. Part 1 of OGC API Features provides only simple query capabilities (bounding box, time interval or instant, discrete attribute values). Part 3 and CQL2 extend this with support for more advanced filter expressions. The design described in this Engineering Report extends this capability with additional capabilities necessary for this task, including the following.

- A capability to not only filter the data, but also return only a subset of the properties.
- A capability for an authorized developer to restrict the query capabilities of the filtering services at runtime.

Is the metadata required by the various OGC API-Features parts sufficient to allow clients to fully understand the filtering capabilities of a service endpoint?

In general, the following information should be sufficient for many use cases.

- The APIs document the necessary metadata to query the data (spatial and temporal extent, supported coordinate reference systems, etc.).
- The APIs document the schema of the response for each data collection.
- The APIs document the properties and their characteristics that can be queried for each data collection (Queryables).
- The APIs document the properties and their characteristics that can be used to sort the features of each data collection (Sortables).
- The Filter Services document the filtering rules / queries that can be executed including the parameters for each rule / query.

OGC API — Features — Part 3: Filtering and the Common Query Language (CQL) supports queryables that are not directly represented as resource properties in the content schema of the resource. Is it possible to identify best practices for their usage?

This aspect was not addressed in Testbed 18 as it was not necessary to implement the scenario and use case.

Clients may know the content schema of offered resources. How best to use this knowledge for advanced filtering beyond what is defined in OGC API — Features — Part 3: Filtering and the Common Query Language (CQL)?

Additional capabilities are specified in the sections Clause 5 and Clause 6.

How does a filtering service look that supports advanced filtering for rather simple OGC API-Features-based SWIM data endpoints?

If the Data Services are restricted to OGC API Features Part 1 with Schema support, a Filtering Service that would implement all capabilities specified in this architecture would support the following advanced Feature-based query capabilities that go beyond the Data Services.

- Rich filter expressions that include spatial and temporal characteristics are supported through OGC API Features Part 3 as well as CQL2.
- Query multiple data collections with a single request.
- Filtering rules / queries are managed and stored in the service.
- These stored queries can be parameterized. The parameters are specified when the query is executed.
- Not only filtering is supported. Changing the presentation of the matched features through selection of properties to include in the response or by sorting according to an order specified in the filtering rule or a parameter is also supported.

How does such a service work in situations where a data publisher has restricted filtering on certain properties (for example, because the backend datastore has not been configured to allow high-performance queries on those properties)?

In those cases, the data must be streamed to the filtering service and filtering has to occur in the filtering service (if the service supports such a capability). There is no difference in the API, the filtering rule / query / filter expressions are independent of the implementation at the backend of the Filtering Service.

How can a client application support a customer that has knowledge of the content schema of an offered resource in the creation of filter statements? What are the key requirements for a developer GUI that allows visualization and management of these filtering tools?

To query a data collection, the Developer Client at least needs to understand the JSON Schema that describes the Queryables (provided by the Filtering Service) and the supported conformance classes describing the provided CQL2 and API capabilities (Conformance Declaration of the Filtering Service).

To restrict the properties that can be returned for a data collection, the Developer Client needs to be able to understand the JSON Schema that describes the content of the data (provided by the Data Service or/and the Filtering Service).

To manage the filtering rules in a Filtering Service, the Developer Client needs to support the Manage Stored Queries conformance class specified Clause 5.

Is it possible to easily create a new filtered dataset by creating machine readable filtering rules based on the metadata required by the OGC API-Features standards?

Yes. Each filtering rule specifies a filtered and potentially restricted view on a dataset. In that sense, this is a derived dataset.

How can these rules be provided to the Filtering Service at runtime?

Through the operations specified in the Manage Stored Queries conformance class.

8.2. Lessons learned

The architecture documented in this Engineering Report leveraged existing candidate specifications and proposals for querying data from the Features API Standards Working Group (SWG). These were in particular:

- the Common Query Language (CQL2) ([16]) as the language for filter expressions; and
- the proposed Search ([6]) extension to retrieve features from multiple collections in a single query and to execute parameterized stored queries.

Pre-existing resources turned out to be a good match for addressing the requirements stated in the Testbed 18 Call-for-Proposals. Therefore, these pre-existing specifications were used as a starting point and feedback was provided to the SWG.

- Several issues were opened by participants based on their implementation experience. These issues focused on improving the clarity of the CQL2 documentation and the language design.
- The Search proposal is an initial draft that has not yet been discussed in detail. The proposal was used as a starting point and an updated design was developed, discussed, implemented, and tested in Testbed 18. While some of the updates were driven by the Testbed 18 requirements, the design specified in this ER intentionally is applicable in general, not just in the context of the scenario used in Testbed 18. These results will be input to the work on the Search extension by the Features API SWG (expected for 2023). The results were initially presented to the Features API SWG in the October 2022 Member Meeting.
- One aspect that was out-of-scope for Testbed 18 was security/access control. In almost all cases, the operations in the Manage Stored Queries conformance class will require authentication and access control. Depending on the context, an API provider may also want to control access to specific stored queries or the feature collections.

8.3. Future work

- One research question was not addressed in Testbed 18 and could be subject to research in future initiatives: queryables that are not directly represented as resource properties in the content schema of the resource are explicitly supported by OGC API Features. Is it possible to identify best practices for their usage?
- Support for queries that include joins between features of different collections was discussed but considered out-of-scope of this Testbed 18.
- As mentioned above, authentication and access control were out-of-scope for the testbed. In general, it would be helpful to also test the specification with APIs that use the commonly used OAuth2 / OpenID Connect security scheme.

A

ANNEX A (INFORMATIVE) SAMPLE EXPRESSIONS AND REQUESTS

A ANNEX A (INFORMATIVE) SAMPLE EXPRESSIONS AND REQUESTS

A.1. Example query expressions

A.1.1. A simple, single query

```
{
  "title": "Fetch all apron areas",
  "collections": ["apronelement"]
}
```

A.1.2. A simple query with a filter, property selection and sorting of the response document

```
{
  "title": "Fetch all concrete apron areas of the main New York area airports
(JFK, EWR and LGA)",
  "description": "Returned are the geometry, the airport and the apron type,
sorted by airport.",
  "collections": ["apronelement"],
  "filter": {
    "op": "and",
    "args": [
      { "op": "=", "args": [{ "property": "composition" }, "CONC"] },
      { "op": "in", "args": [{ "property": "airport" }, ["JFK", "EWR", "LGA"]]
}
    ]
  },
  "properties": ["geometry", "airport", "type"],
  "sortby": ["airport"],
  "limit": 1000
}
```

A.1.3. The same query with two parameters (one string and one string array)

```
{
  "title": "Fetch the apron areas of selected airports",
  "description": "Returned are the geometry, the airport, the apron type and
the material, sorted by airport. The query accepts two parameters: the airport
and the type of the apron area.",
  "collections": ["apronelement"],
  "filter": {
    "op": "and",
    "args": [
      {
        "op": "=",
        "args": [
          {"property": "type"},
          {
            "$parameter": {
              "type": {
                "title": "Type of the apron, runway or taxiway element",
                "description": "The following types are distinguished: normal
use, parking, shoulder, intersection.",
                "type": "string",
                "enum": ["NORMAL", "PARKING", "INTERS", "SHOULD"],
                "default": "NORMAL"
              }
            }
          }
        ]
      },
      {
        "op": "in",
        "args": [
          {"property": "airport"},
          "$parameter": {
            "airports": {
              "title": "Airports",
              "description": "The 3-letter IATA airport codes or the airports
to filter. Specify multiple values as a comma-separated list.",
              "type": "array",
              "items": {
                "type": "string",
                "enum": ["JFK", "EWR", "LGA", "BOS", "PIT", "PHL", "DCA",
"BWI", "IAD"]
              },
              "default": ["JFK", "EWR", "LGA"]
            }
          }
        ]
      }
    ]
  },
  "properties": ["geometry", "airport", "type", "composition"],
  "sortby": ["airport"],
  "limit": 1000
}
```

A.1.4. The same query with the two parameters declared at the top level

```
{
  "title": "Fetch the apron areas of selected airports",
  "description": "Returned are the geometry, the airport, the apron type and
the material, sorted by airport. The query accepts two parameters: the airport
and the type of the apron area.",
  "collections": ["apronelement"],
  "filter": {
    "op": "and",
    "args": [
      {
        "op": "=",
        "args": [
          {"property": "type"},
          {"$parameter": {"$ref": "#/parameters/type"}}
        ]
      },
      {
        "op": "in",
        "args": [
          {"property": "airport"},
          {"$parameter": {"$ref": "#/parameters/airport"}}
        ]
      }
    ]
  },
  "properties": ["geometry", "airport", "type", "composition"],
  "sortby": ["airport"],
  "limit": 1000,
  "parameters": {
    "airports": {
      "title": "Airports",
      "description": "The 3-letter IATA airport codes or the airports to
filter. Specify multiple values as a comma-separated list.",
      "type": "array",
      "items": {
        "type": "string",
        "enum": ["JFK", "EWR", "LGA", "BOS", "PIT", "PHL", "DCA", "BWI", "IAD"]
      },
      "default": ["JFK", "EWR", "LGA"]
    },
    "type": {
      "title": "Type of the apron, runway or taxiway element",
      "description": "The following types are distinguished: normal use,
parking, shoulder, intersection.",
      "type": "string",
      "enum": ["NORMAL", "PARKING", "INTERS", "SHOULD"],
      "default": "NORMAL"
    }
  }
}
```

A.1.5. A query on multiple collections

```
{
  "queries": [
    {"collections":["apronelement"]},
    {"collections":["runwayelement"]},
    {"collections":["taxiwayelement"]}
  ]
}
```

A.1.6. A query on mulitple collections with sorting as well as global filtering and property selection

```
{
  "queries": [
    {"collections":["apronelement"], "sortby": ["airport"]},
    {"collections":["runwayelement"], "sortby": ["airport"]},
    {"collections":["taxiwayelement"], "sortby": ["airport"]}
  ],
  "filter": {
    "op": "and",
    "args": [
      { "op": "=", "args": [{ "property": "composition" }, "CONC"] },
      { "op": "in", "args": [{ "property": "airport" }, ["JFK", "EWR", "LGA"]]
}
    ]
  },
  "properties": ["geometry", "airport", "type"],
  "limit": 1000
}
```

A.1.7. A query on multiple collections with a different filter expressions for each collection

```
{
  "queries": [
    {
      "collections":["apronelement"],
      "filter": { ... }
    },
    {
      "collections":["runwayelement"],
      "filter": { ... }
    },
    {
      "collections":["taxiwayelement"],
      "filter": { ... }
    }
  ],
  "limit": 1000
}
```

A.1.8. A query on multiple collections with sorting and a different filter for each collection and a global filter plus property selection

```
{
  "queries": [
    {
      "collections":["apronelement"],
      "filter": { ... },
      "sortby": ["airport"]
    },
    {
      "collections":["runwayelement"],
      "filter": { ... },
      "sortby": ["airport"]
    },
    {
      "collections":["taxiwayelement"],
      "filter": { ... },
      "sortby": ["airport"]
    }
  ],
  "filter": {
    "op": "and",
    "args": [
      { "op": "=", "args": [{ "property": "composition" }, "CONC"] },
      { "op": "in", "args": [{ "property": "airport" }, ["JFK", "EWR", "LGA"]]
}
    ]
  },
  "properties": ["geometry", "airport", "type"],
  "limit": 1000
}
```

A.1.9. A single join query

```
{
  "collections": ["apronelement", "apron"],
  "filter": {
    "op": "and",
    "args": [
      { "op": "=", "args": [{ "property": "apronelement.associatedApron" }, {
"property": "apron.id" }] },
      { "op": "in", "args": [{ "property": "apronelement.airport" }, ["JFK",
"EWR", "LGA"]] }
    ]
  },
  "sortby": ["apronelement.airport"],
  "limit": 1000
}
```

A.2. Example requests

This section illustrates example requests and responses in the scenario.

A.2.1. Developer Client analyzes the available data

A.2.2. Developer Client creates a stored query (“filtering rule”)

With the following request, the Developer Client creates a parameterized stored query on a dataset with AIXM-based airport data on the Filtering Service.

The query has three parameters.

- “collection”: The feature type that is queried. Restricted to one of “apronelement,” “runwayelement,” or “taxiwayelement.” The default is “runwayelement” if no parameter value is specified in the query.

- “type”: The type of the apron/runway/taxiway element. One of “NORMAL,” “PARKING,” “INTERS,” or “SHOULD”. If no parameter value is specified in the query the default is “NORMAL.”
- “airports”: The airports for which the query should return data which is restricted to selected airports. The default is “IAD” if no parameter value is specified in the query.

The query:

- returns only four feature properties (“geometry,” “airport,” “type,” “composition”), all additional properties in the data are not returned;
- sorts the response by airport; and,
- limits the response to 10000 features: if more are matched, the response will include a “next” link to the next page.

```
PUT /d103_airports/search/elements-by-type-and-airport HTTP/1.1
Host: t18.ldproxy.net
Content-Type: application/json

{
  "title": "Fetch apron, taxiway or runway elements based on their type and
airport",
  "description": "This query fetches apron, taxiway or runway elements based
on their type and airport. The response uses paging, if more than 10000
features match the query. The result is sorted by airport.",
  "query": {
    "collections": [
      {
        "$parameter": {
          "collection": { "type": "string", "enum": ["apronelement",
"runwayelement", "taxiwayelement"], "default": "apronelement" }
        }
      }
    ],
    "filter": {
      "op": "and",
      "args": [
        {
          "op": "=",
          "args": [
            {"property": "type"},
            {
              "$parameter": {
                "type": { "type": "string", "enum": ["NORMAL", "PARKING",
"INTERS", "SHOULD"], "default": "NORMAL" }
              }
            }
          ]
        },
        {
          "op": "in",
          "args": [
            {"property": "airport"},
            {
              "$parameter": {
                "airports": {
                  "type": "array",
                  "items": { "type": "string", "enum": ["JFK", "EWR", "LGA",
"BOS", "PIT", "DCA", "IAD", "BWI", "PHL"] },
                  "default": ["IAD", "DCA", "BWI"]
                }
              }
            }
          ]
        }
      ]
    },
    "properties": ["geometry", "airport", "type", "composition"],
    "sortby": ["airport"]
  },
  "limit": 10000
}
```

The server creates the stored query. If the stored query already existed, the query would be updated with the new definition.

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

A.2.3. Business Client fetches the available queries

The Business Client wants to filter data and asks the Filtering Service for the available queries.

NOTEIn addition to the queries stored on the server, the Filtering Service can also allow the execution of ad-hoc queries specified by a client. This scenario assumes that the Business Client does not have the capability to construct query expressions, only the Developer Client has that capability.

```
GET /d103_airports/search HTTP/1.1
Host: t18.ldproxy.net
Accept: application/json
```

The Filtering Service responds with the list of stored queries available on the server:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "queries": [
    {
      "id": "elements-by-type-and-airport",
      "title": "Fetch apron, taxiway or runway elements based on their type
and airport",
      "description": "This query fetches apron, taxiway or runway elements
based on their type and airport. The response uses paging, if more than 10000
features match the query. The result is sorted by airport.",
      "links": [
        {
          "rel": "self",
          "title": "Query 'Fetch apron, taxiway or runway elements based on
their type and airport'",
          "href": "https://t18.ldproxy.net/d103_airports/search/elements-by-
type-and-airport"
        },
        {
          "rel": "describedby",
          "title": "Definition of query 'Fetch apron, taxiway or runway
elements based on their type and airport'",
          "href": "https://t18.ldproxy.net/d103_airports/search/elements-by-
type-and-airport/definition"
        }
      ],
      "parameters": {
        "type": {
          "title": "Type of the apron, runway or taxiway element",
          "description": "The following types are distinguished: normal use,
parking, shoulder, intersection.",
          "type": "string",
          "enum": ["NORMAL", "PARKING", "SHOULD", "INTERS"],
          "default": "NORMAL"
        },
        "airports": {
          "title": "Airports",
          "description": "The 3-letter IATA airport codes or the airports to
filter. Specify multiple values as a comma-separated list.",
          "type": "array",
          "items": {
            "type": "string",
            "enum": ["JFK", "EWR", "LGA", "BOS", "PIT", "PHL", "DCA", "BWI",
"IAD"]
          },
          "default": ["IAD", "DCA", "BWI"]
        }
      }
    },
    ...
  ],
  "links": [
    {
      "rel": "self",
      "type": "application/json",
      "title": "This document",
      "href": "https://t18.ldproxy.net/d103_airports/search?f=json"
    }
  ]
}
```

The Business Client is interested in the first query and retrieves information about the parameters of the query with the following request:

```
GET /d103_airports/search/elements-by-type-and-airport/parameters HTTP/1.1
Host: t18.ldproxy.net
Accept: application/json
```

The Filtering Service responds with the list of parameters specified for the query:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "parameters": {
    "collection": {
      "type": "string",
      "enum": ["apronelement", "runwayelement", "taxiwayelement"],
      "default": "runwayelement"
    },
    "type": {
      "type": "string",
      "enum": ["NORMAL", "PARKING", "INTERS", "SHOULD"],
      "default": "NORMAL"
    },
    "airports": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": ["JFK", "EWR", "LGA", "BOS", "PIT", "DCA", "IAD", "BWI", "PHL"]
      },
      "default": [["IAD", "DCA", "BWI"]
    }
  }
}
```

A.2.4. Business Client executes the selected stored query and retrieves data

With this information, the business client can execute the stored query.

Since the query has default values for all parameters, the simplest query is one without parameter values.

```
GET /d103_airports/search/elements-by-type-and-airport HTTP/1.1
Host: t18.ldproxy.net
Accept: application/geo+json
```

The Filtering Service will respond with the normal runway elements of Dulles International Airport:

```
HTTP/1.1 200 OK
Content-Type: application/geo+json

{
  "type": "FeatureCollection",
  "features": [
    ...
  ]
}
```

A query with other parameter values can be executed as a GET or a POST request. The apron elements for parking from the three New York area airports will be requested.

First the GET variant:

```
GET /d103_airports/search/elements-by-type-and-airport?collection=
apronelement&type=PARKING&airports=EWR,JFK,LGA HTTP/1.1
Host: t18.ldproxy.net
Accept: application/geo+json
```

And the POST variant, which will typically be necessary for larger parameter values, e.g. a polygon geometry:

```
POST /d103_airports/search/elements-by-type-and-airport HTTP/1.1
Host: t18.ldproxy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 57
Accept: application/geo+json

collection=apronelement&type=PARKING&airports=EWR,JFK,LGA
```

In both cases the Filtering Service will respond with the selected features as a GeoJSON feature collection:

```
HTTP/1.1 200 OK
Content-Type: application/geo+json

{
  "type": "FeatureCollection",
  "features": [
    ...
  ]
}
```

B

ANNEX B (INFORMATIVE) REVISION HISTORY

B

ANNEX B (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	AUTHOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2022-09-30	0.5	S. Taleisnik	all	Draft Engineering Report (DER)
2022-11-29	0.9	S. Taleisnik	all	Version Posted to Pending
2022-12-19	1.0	S. Taleisnik	all	Final Edits

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Pross, B., Vretanos, P.A.,: OGC API — Processes- Part 1: Core. Open Geospatial Consortium, https://docs.ogc.org/is/18-062r2/18-062r2.html.

[2] Masó, J., Jacovella-St-Louis, J.: OGC API — Tiles — Part 1: Core. Open Geospatial Consortium, https://docs.ogc.org/is/20-057/20-057.html (2022).

[3] Portele, C.: OGC API — Styles. Open Geospatial Consortium, http://docs.opengeospatial.org/DRAFTS/20-009.html .

[4] Taleisnik, S.: OGC Testbed-17: Aviation API ER. Open Geospatial Consortium, https://docs.ogc.org/per/21-039r1.html, (2022).

[5] Vretanos, P.: OGC Filter Encoding 2.0 Encoding Standard. Open Geospatial Consortium, http://docs.ogc.org/is/09-026r2/09-026r2.html, (2014).

[6] Vretanos, P.: OGC API — Features — Part 5: Search (PROPOSAL). Open Geospatial Consortium, https://github.com/opengeospatial/ogcapi-features/tree/master/proposals/search, (2022)

[7] Jacovella-St-Louis, J., Vretanos, P.A.: OGC API — Processes — Part 3: Workflows and Chaining (draft). Open Geospatial Consortium, https://opengeospatial.github.io/ogcna-auto-review/21-009.html (2023).

[8] Dictionary of Computer Science — Oxford Quick Reference, (2016).

[9] Lóscio, B.F, Calegari, N., Burle, C.: Data on the Web Best Practices. W3C, https://www.w3.org/TR/dwbp/ (2017).

[10] Service Facade Pattern, https://www.ibm.com/docs/pt-br/integration-bus/9.0.0?topic=SSMKHH_9.0.0/com.ibm.etools.mft.pattern.sen.doc/sen/sf/overview.htm.

[11] SWIM Questions & Answers, https://www.faa.gov/air_traffic/technology/swim/questions_answers/, (2021).

[12] Portele, C., Vretanos, P.A., Heazel, C.: OGC API — Features — Part 1: Core. Open Geospatial Consortium, https://docs.opengeospatial.org/is/17-069r4/17-069r4.html (2022).

[13] Portele, C., Vretanos, P.A.: OGC API — Features — Part 2: Coordinate Reference Systems by Reference. Open Geospatial Consortium, https://docs.ogc.org/is/18-058r1/18-058r1.html (2022).

[14] Vretanos, P.A., Portele, C.: OGC API — Features — Part 3: Filtering. Open Geospatial Consortium, https://docs.ogc.org/DRAFTS/19-079.html .

[15] Taleisnik, S.: OGC Testbed-16: Aviation Engineering Report. Open Geospatial Consortium, https://docs.ogc.org/per/20-020.html (2021).

[16] Vretanos, P.A., Portele, C.: Common Query Language (CQL2). Open Geospatial Consortium, https://docs.ogc.org/DRAFTS/21-065.html.