OGC® DOCUMENT: 22-023R2

External identifier of this OGC® document: http://www.opengis.net/doc/PER/T18-D001

TESTBED-18: FEATURES FILTERING SUMMARY ENGINEERING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2022-12-24
Approval Date: 2023-01-19
Publication Date: 2023-07-14
Editor: Sergio Taleisnik

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

Open Geospatial Consortium

License Agreement

Use of this document is subject to the license agreement at https://www.ogc.org/license

Copyright notice

Copyright © 2023 Open Geospatial Consortium
To obtain additional rights of use, visit https://www.ogc.org/legal

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

I. ABSTRACT vii

II. EXECUTIVE SUMMARY vii

III. KEYWORDS x

IV. PREFACE xi

V. SECURITY CONSIDERATIONS xii

VI. SUBMITTERS xii

1. SCOPE 2

2. NORMATIVE REFERENCES 4

3. TERMS, DEFINITIONS AND ABBREVIATED TERMS 6
3.1. Terms and definitions 6
3.2. Abbreviated terms 8

4. INTRODUCTION 10
4.1. Background 10
4.2. Requirements Statement 12
4.3. Functional Overview 13

5. OGC API-FEATURES FAÇADE 1 (D100) 20
5.1. Internal Architecture 20
5.2. Differences to the component from Testbed 17 24
5.3. Challenges and Lessons Learned 26

6. OGC API-FEATURES FAÇADE 2 (D101) 29
6.1. Status Quo 29
6.2. Internal Architecture 30
6.3. Feature collections 33
6.4. Filtering Capabilities 35
6.5. Challenges and Lessons Learned 36

7. FILTERING SERVICE 1 (D102) 40
7.1. Status Quo 40
7.2. Internal Architecture 40

7.3. Challenges and Lessons Learned 47
7.4. Recommendations and Future Work 51

8. FILTERING SERVICE 2 (D103) 53
8.1. Internal Architecture 53
8.2. ldproxy 53
8.3. Filtering Capabilities 54
8.4. HTML Support 66
8.5. Challenges and Lessons Learned 68

9. BUSINESS USER CLIENT (D104) 71
9.1. Internal Architecture 71
9.2. Recommendations and Future Work 76

10. DEVELOPER CLIENT (D105) 78
10.1. Internal Architecture 78
10.2. Challenges and Lessons Learned 84
10.3. Recommendations and Future Work 85

11. TECHNOLOGY INTEGRATION EXPERIMENTS (TIES) 88
11.1. TIE Summary Table 88
11.2. TIE Functional Tests 88

ANNEX A (INFORMATIVE) REVISION HISTORY 99

BIBLIOGRAPHY 101

LIST OF TABLES

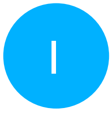
Table 1 — Server Endpoints 32
Table 2 — AIXM Feature Collections from SWIM Data Services 33
Table 3 — FIXM and Traffic Flow Feature Collections from SWIM Data Services 34
Table 4 — ITWS Weather Feature Collections from SWIM Data Services 34
Table 5 — Technology Integration Experiments Overview 88
Table 6 — TIE Overview of the Business User Client With the D103 Filtering Service 94
Table 7 — TIE Overview of the Developer Client With the D103 Filtering Service 95

LIST OF FIGURES

Figure 1 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task ix
Figure 2 — History of OGC experiments to enhance SWIM 12

Figure 3 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task 14
Figure 4 — Workflow from the perspective of a business user that needs filtered data 15
Figure 5 — First Workflow Sequence Diagram 16
Figure 6 — Workflow from the perspective of a filtering rules developer 17
Figure 7 — Second Workflow Sequence Diagram 18
Figure 8 — D100 Component Overview 21
Figure 9 — Information Flow for Data Requests 23
Figure 10 — Communicating Data Changes to ldproxy 24
Figure 11 — D100 Simple Filtering in the Web Browser 26
Figure 12 — D101 Component Overview 30
Figure 13 — API-Features Data Interaction with Backend SWIM Messaging Services 31
Figure 14 — Major Entity Relationship Diagram for Managed SWIM Features 31
Figure 15 36
Figure 16 — Single feature returned from CQL2 filter query on test collection 42
Figure 17 — Single flight plan feature cascaded from Skymantics service 43
Figure 18 — Paging through an output collection resulting from the above filter query pre-defined using OGC API - Processes - Part 3 46
Figure 19 47
Figure 20 47
Figure 21 — Expressions UML Conceptual Model, covering CQL2 capabilities 49
Figure 22 — Operators UML Conceptual Model, covering CQL2 capabilities 49
Figure 23 — Standard functions UML Conceptual Model, covering CQL2 capabilities 50
Figure 24 57
Figure 25 — Executing a Stored Query from the Web Browser 67
Figure 26 — Response from a Stored Query in the Web Browser (all Pittsburgh airport features) 68
Figure 27 — Developer Client Component Breakdown 71
Figure 28 — Business Client Query List 73
Figure 29 — Business Client Showing Airspaces 73
Figure 30 — Business Client Showing Airports 74
Figure 31 — Business Client Showing Intersections 75
Figure 32 — Business Client Showing NOTAMs 76
Figure 33 — Developer Client Component Breakdown 78
Figure 34 — Developer Client Queries Table 80
Figure 35 — Developer Client Collection Table 81
Figure 36 — Query Editing in the Developer Client 81
Figure 37 — Developer Client Query Creation 82
Figure 38 — Developer Client Collections Table 83
Figure 39 — Developer Client Displaying Queryables 83
Figure 40 — Developer Client Displaying Sortables 84
Figure 41 — Map of LAX (from interactive instruments' D100 service) as it appears on collection page 89

Figure 42 — Map of JFK airport from interactive instruments' D100 service 90
Figure 43 — Paginated filtered features returned for a query on the EWR collection 90
Figure 44 — Feature #1413 from the EWR airport collection from interactive instruments' D100 service 91
Figure 45 — Ecere's D102 Filtering Service cascading features from George Mason University's D101 service 93
Figure 46 — Ecere's D102 Filtering Service cascading and filtering features from Skymantics service 94



ABSTRACT

This OGC Testbed-18 (TB-18) Features Filtering Summary Engineering Report (ER) summarizes the implementations, findings, and recommendations that emerged from the efforts to better understand the current OGC API-Features filtering capabilities and limitations and how filtering can be decoupled from data services.

This ER describes:

- two façades built to interface SWIM services and serve aviation data through APIs (built with OGC API Standards) including basic filtering capabilities;
- the two filtering services built to consume SWIM data and serve it through OGC based APIs featuring advanced filtering mechanism;
- the client application built to interface with the filtering services; and
- the developer client built to define filter statements that can be expressed in a machine-readable way and exchanged with the filtering service.

EXECUTIVE SUMMARY

Previous OGC work has addressed the challenges of increasing interoperability between aviation data services. Recently, the OGC community has developed a new family of standardized OpenAPI-based Web API Standards for various geospatial resource types. These OGC APIs have the potential to enhance the way in which consumers can access geospatial data from various sources. Testbed-16 brought together previous work on the development of OGC API Standards, the use of semantics to enrich data, SWIM data processing, and demonstrated an OpenAPI-based API serving SWIM data. Testbed-17 used the 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), with two parts 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) that is provided to consumers using various protocols and service offerings in both synchronous and asynchronous messaging formats. Testbed-18 (TB-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 listed below.

- 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 should this knowledge be used 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 that allows advanced filtering for rather simple OGC API-Features-based SWIM data endpoints look like?
- 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 and listed below.

- **Façades for SWIM services with simple filtering mechanisms**, retrieving aviation data from multiple SWIM services, serving these data through APIs built based on OGC API standards, and 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; and
  - an extra façade, not originally included in the Task architecture, offered in-kind by the company *Skymantics*, and was named OGC API-Features Façade 3: An APIs 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; and
  - 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; and
  - 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.

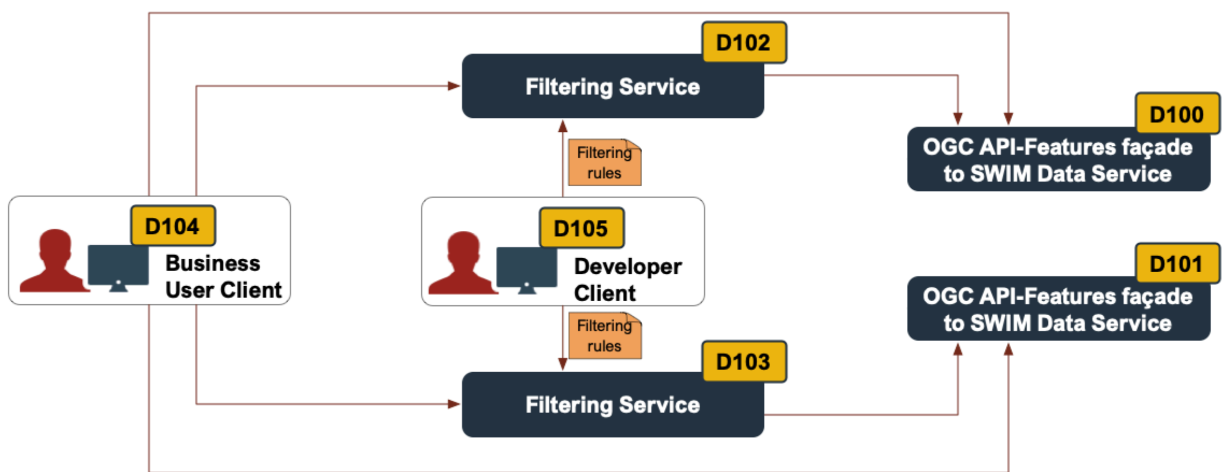
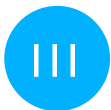


Figure 1 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task

All components were successfully developed and tested. The lessons learned throughout the Testbed are documented in this ER and help respond to the questions posed above. The following is a set of recommendations for future work.

- **Technical Design and CQL Standard**: The implementations in this Testbed provided feedback to the technical design and helped to improve the specification in Testbed-18 Filtering Service and Rule Set Engineering Report (D002) as well as in the Common Query Language (CQL2) candidate standard.
- **Filter close to the data**: In this task, the deliverables architecture called for one component performing the filtering, while another acts as an OGC API façade. The Testbed initiative highlighted that this architecture is not ideal, suggesting the exploration of filtering performed as close to the source of data as possible.

- **Explore the potential of spatial joins**: Future work could explore, in the context of queries spanning multiple collections, the capability of performing queries that not only filter and combine the results from multiple collections, but also perform join operations on them, including spatiotemporal joins.
- **First-Filtering**: Future work could explore how the filtering service might be able to perform a first-filtering pass that requires retrieving less data from the original data source.
- **Interactive Query Building Interface**: Future initiatives could explore an Interactive query builder web interface providing graphical and pre-selectable queryables, sortables, and conformances that are fetched from the SWIM Filtering service. This has the potential to speed query building and validation.
- **Smart Query Building Interface**: Another shortcoming is the lack of real-time hints on the impact of changes to the query structure and on the results of the query. Having graphical query results that are tied back to the query expression helps the user understand the impact of each queryable or conformance used in the query as well as the result of the query. The results of the query would be graphically displayed in real-time to the Developer User while the query is being built, helping the Developer User to more quickly adjust the query to achieve the desired results.
- **Data Correlation from multiple SWIM Services**: In TB-18 the queries built using CQL2 language are limited to each SWIM data service, and these queries are segregated from each other. Future work should explore filtering services supporting data fusion based on parameters in the data that are of similar type that support the same conformances.



KEYWORDS

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

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

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 upon by any implementation of the standard set forth in this document and to provide supporting documentation.

SECURITY CONSIDERATIONS

No security considerations have been made for this document.

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) summarizes the implementations, findings, and recommendations that emerged from Testbed-18 efforts regarding the current filtering capabilities and limitations with the OGC API — Features Standard and how filtering can be decoupled from data services. The ER describes two façades built to interface SWIM services and serve aviation data through APIs built using OGC API Standards.

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 [7].

3.1.2. **Façade Service**

a component that fetches data from a specific data source and makes it available through its own interface [9]. 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 either:

- the underlying structure of the API; or
- the format in which the data is made available.

3.1.3. 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, 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.

3.1.4. 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.

3.1.5. SWIM Data

any data provided through the SWIM System.

3.1.6. Web API

an API using an architectural style that is founded on the technologies of the Web [8].

Note 1 to entry: Best Practice 24: Use Web Standards as the foundation of APIs in the W3C Data on the Web Best Practices [8] provides more detail.

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

3.2. Abbreviated terms

AIXM	Aeronautical Information Exchange Model
API	Application Programming Interface
CRS	Coordinate Reference System
ER	Engineering Report
FAA	Federal Aviation Administration
FIXM	Flight Information Exchange Model
NAS	National Airspace System
NOTAM	Notice to Airmen
OGC	Open Geospatial Consortium
SCDS	SWIM Cloud Distribution Service
SFDPS	SWIM Flight Data Publication Service
SWIM	System Wide Information Management
TB	Testbed
TFMS	Traffic Flow Management System
TIE	Technology Integration Experiment
WFS	Web Feature Service
WXXM	Weather Information Exchange Model

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 for data access. 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. Since consumers can retrieve data from several SWIM services there is the need for interoperability between the services.

4.1.2. OGC API Standards

For several years, 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 continue to evolve, are approved by the OGC, and are implemented by the community, the aviation industry can subsequently experiment with 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 four parts and a fifth proposed part.

- OGC API — Features — Part 1: Core. Approved September 2019, this standard defines discovery and query operations. [11]
- 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. [12]

- 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. [13]
- 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. [14]
- 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. [5]

A Common Query Language (CQL2) is being developed together with Part 3 to standardize a language that is recommended for filter expressions. [15]

OGC API – Processes: An approved (August 2021) OGC API Standard that 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 enables 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 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 [14].

Two of TB-16 recommendations were to integrate OGC API requirement classes within SWIM Data Services and to demonstrate interoperability between diverse Aviation APIs [14]. 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 [14]. 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.

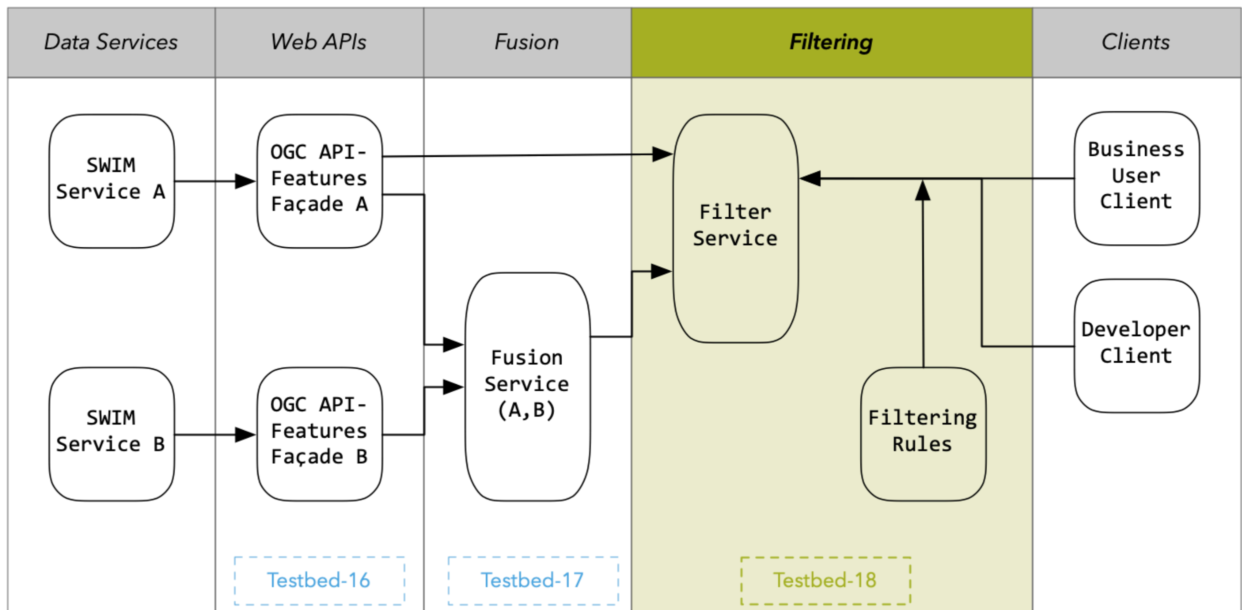


Figure 2 — History of OGC experiments to enhance SWIM

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 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 like 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 easily creating a new filtered dataset possible 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 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 can be exchanged with the filtering services.

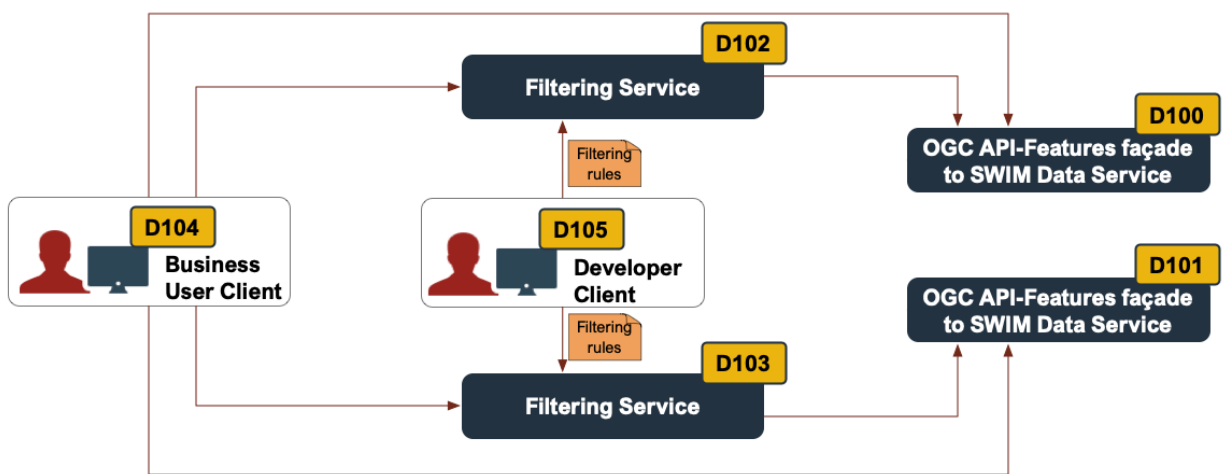


Figure 3 — Component Diagram for the Advanced Filtering of SWIM Feature Data Task

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.

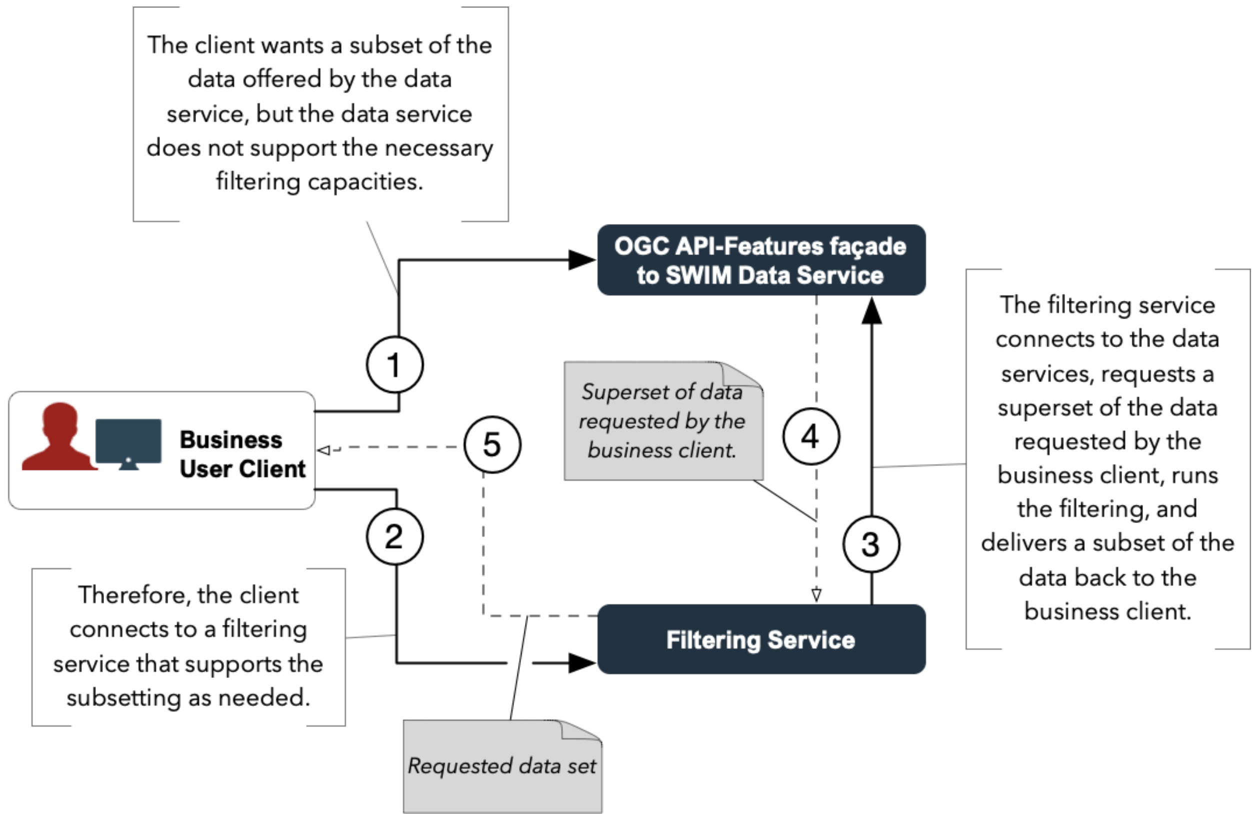


Figure 4 — Workflow from the perspective of a business user that needs filtered data

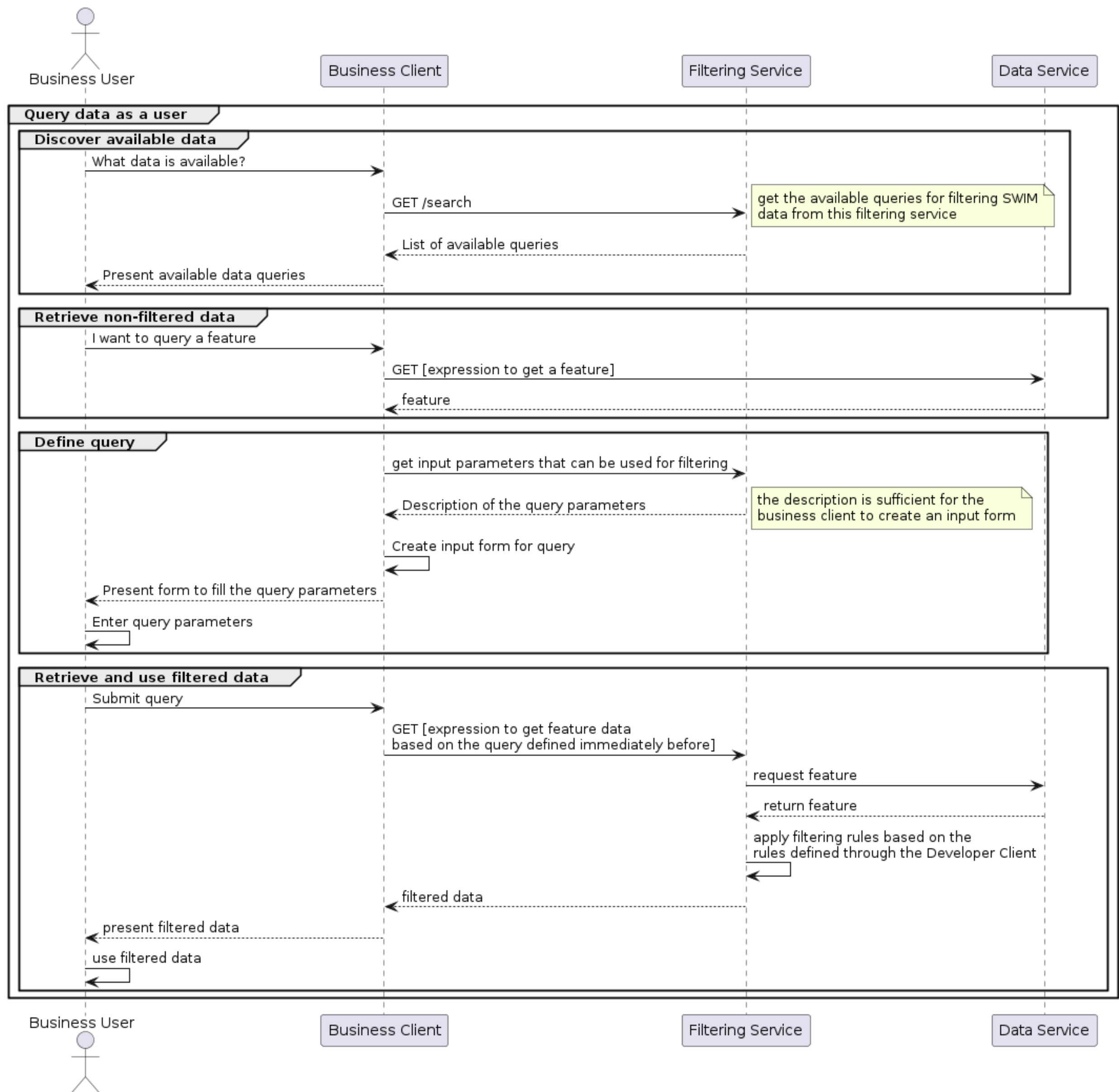


Figure 5 — First Workflow Sequence Diagram

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.

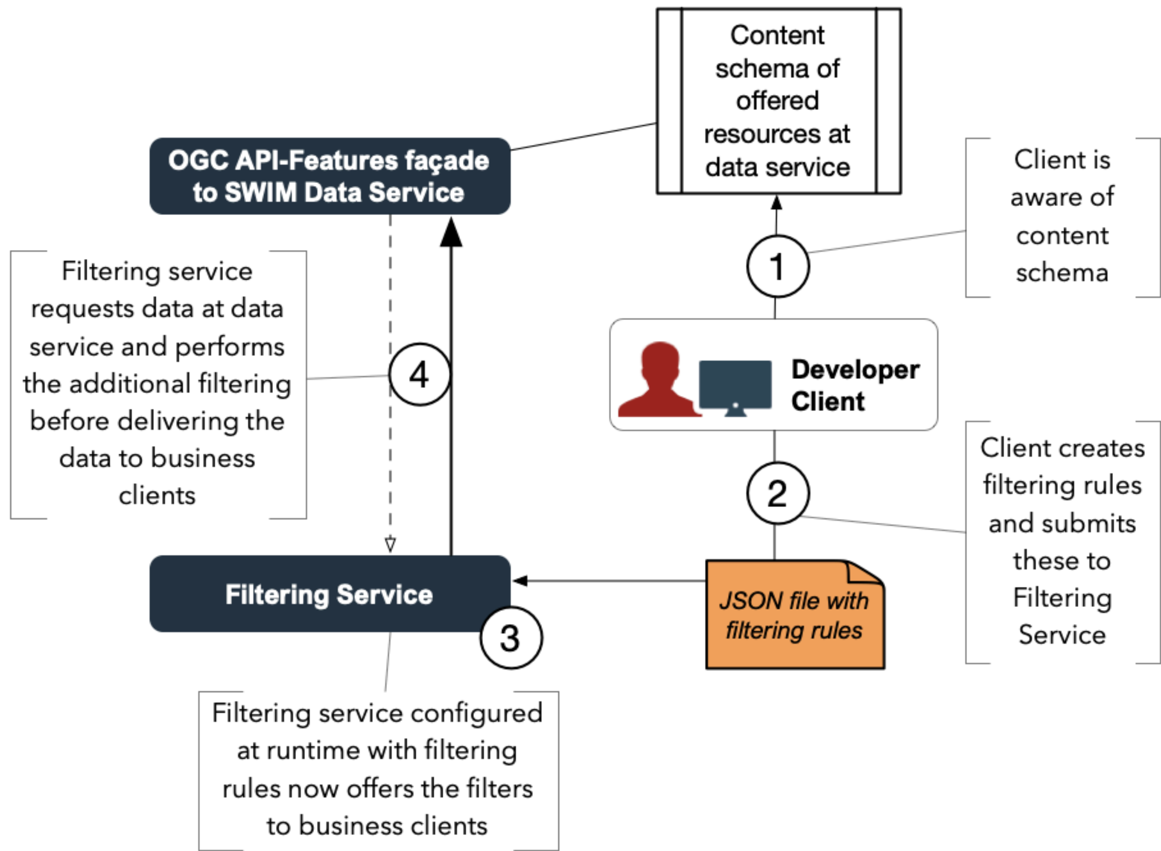


Figure 6 — Workflow from the perspective of a filtering rules developer

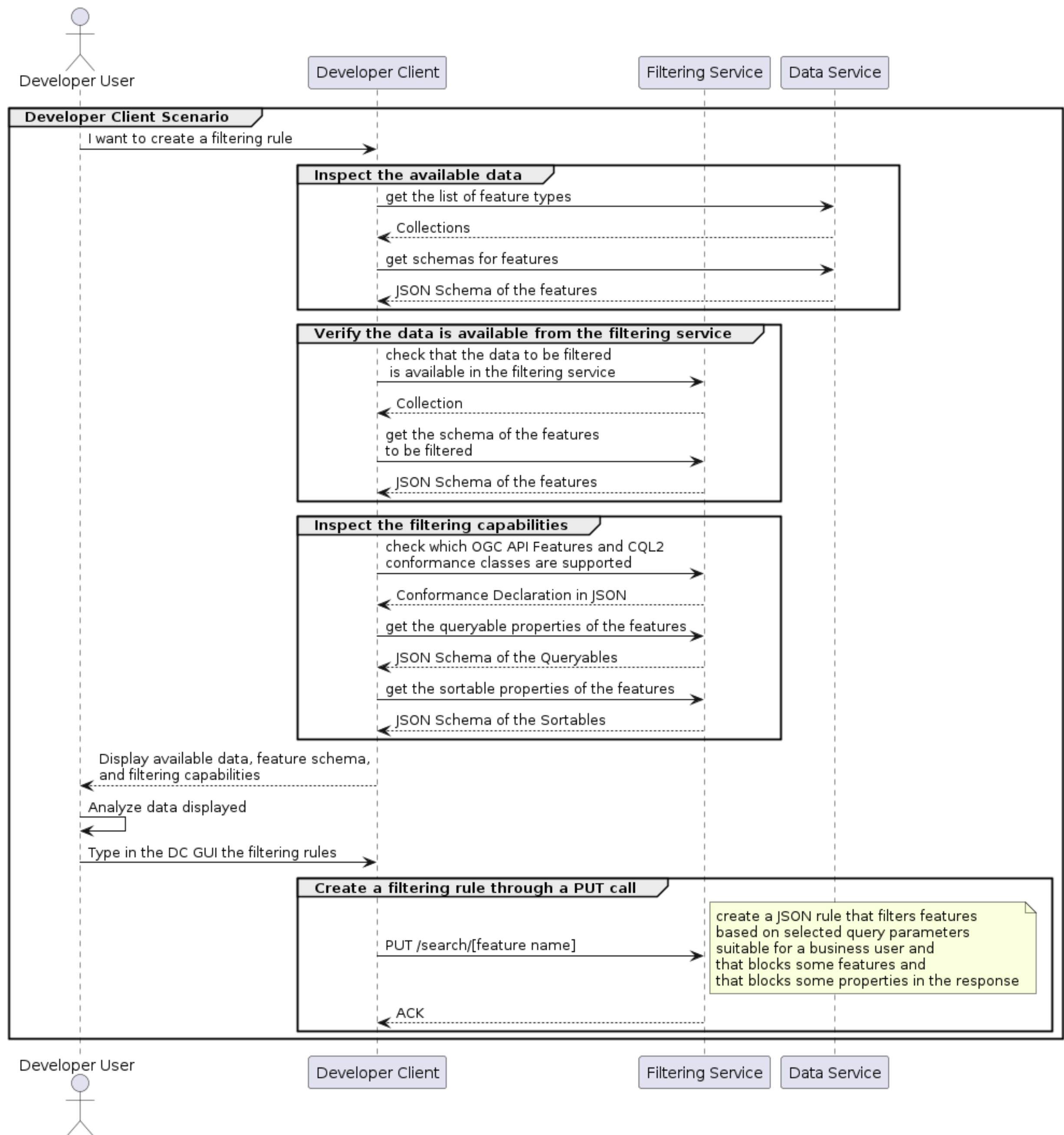


Figure 7 — Second Workflow Sequence Diagram

5

OGC API-FEATURES FAÇADE 1 (D100)

5 OGC API-FEATURES FAÇADE 1 (D100)

The OGC API-Features Façade 1, identified as deliverable 100 or D100, is a component built to demonstrate a service consuming data from a SWIM data service and serving that data through an API built based on OGC API standards offering a limited set of filtering capabilities. The component was demonstrated by interactive instruments GmbH.

The Façade Service was originally set up in Testbed 17 (see the description of D104 in chapter 4 of [4]) and updated for Testbed 18.

5.1. Internal Architecture

5.1.1. Component Overview

This Façade Service features the following.

- Two data retrieval subcomponents: one for the Federal Notice to Airmen System (AIM-FNS) and another one for airspace and airport static data sources.
- A database to store the extracted data.
- ldproxy which delivers the APIs that serve the extracted data. Figure 8 describes the Façade Service subcomponents, the data sources (note that for AIM-FNS the retriever also accesses static sources on some occasions), and the Testbed-18 components consuming from the Façade APIs.

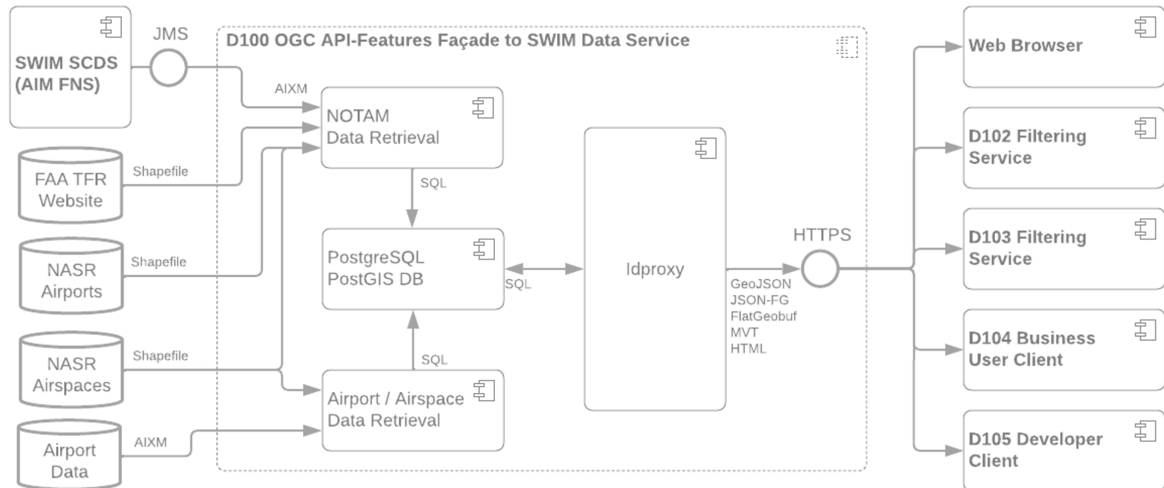


Figure 8 — D100 Component Overview

5.1.2. ldproxy (Web API)

The main component that delivers the Aeronautical Data APIs is ldproxy. ldproxy is a software product written in the Java programming language that implements the OGC API family of Standards. ldproxy enables sharing geospatial data using Web APIs based on OGC API Standards.

The component provides the following four Data APIs that have been deployed.

- Airports (access by AIXM feature type)
- Airports (access by airport)
- Airspaces
- Federal Notice to Airmen System (FNS)

Responses with feature data could be requested in the following representations/formats.

- GeoJSON (media type `application/geo+json`, query parameter `f=json`)
- JSON-FG (media type `application/vnd.ogc.fg+json`, query parameter `f=jsonfg`)
- HTML (media type `text/html`, query parameter `f=html`)
- FlatGeobuf (media type `application/flatgeobuf`, query parameter `f=fgb`)

In addition, some of the APIs also provided access to vector tiles and styles for those tiles. These can be useful for exploring the data but are not directly related to the research questions.

For more information about the APIs, see section 4.2 of the Testbed 17 Aviation API Engineering Report.

5.1.3. PostgreSQL/PostGIS (Database)

The data available for API access are stored in a PostgreSQL/PostGIS database. That is, the data are stored in tables with the geometries stored as Simple Features geometries in Well-Known-Text format.

The following are the aeronautical datasets.

- All Controlled Airspaces of the Classes B, C, D, and E from the National Airspace System Resource (NASR) Subscription
- Airport data for the major airports in the United States provided by Hexagon (AIXM 5.1)
- Notices to Airmen (NOTAMs) received from a Federal Notice to Airmen System (AIM-FNS) subscription in FAA's SWIM Cloud Distribution Service (SCDS), a cloud-based infrastructure dedicated to providing near real-time FAA SWIM data to the public via Solace JMS messaging

5.1.3.1. Communication Between ldproxy and PostgreSQL

ldproxy converts API requests to SQL queries and processes the results to convert them to API responses in the GeoJSON, Features and Geometries JSON (JSON-FG), HTML, or Mapbox Vector Tile (MVT) formats.

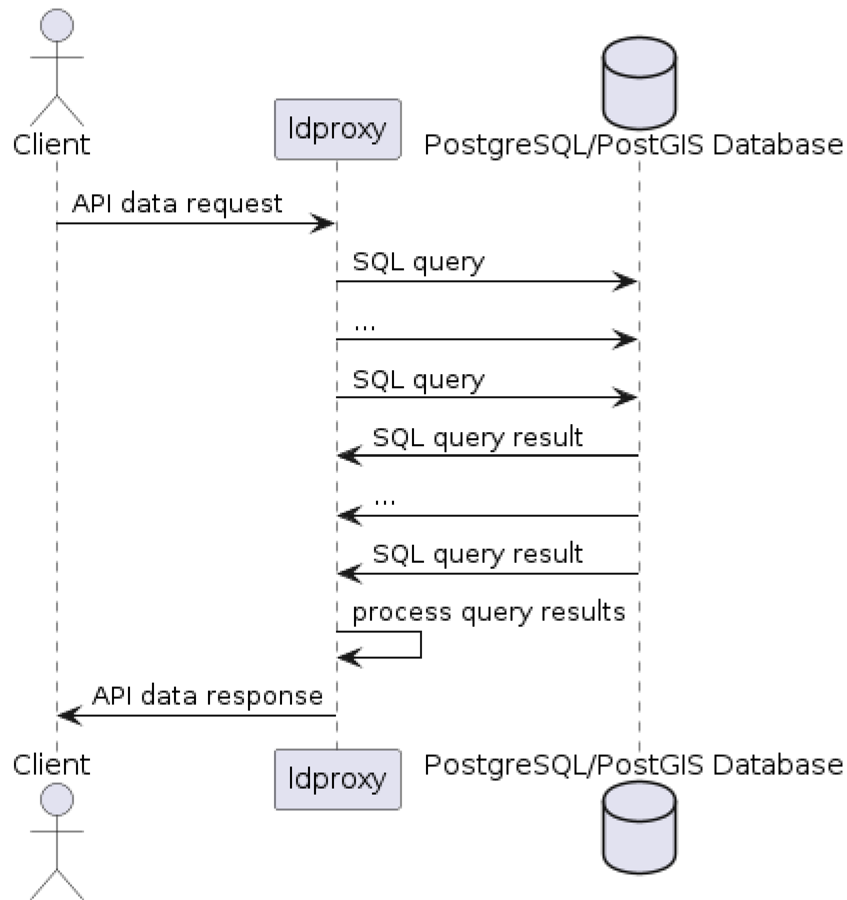


Figure 9 — Information Flow for Data Requests

While the first two datasets are static and do not change, the NOTAMs are dynamic. New NOTAMs are added to the database as they are received from the SCDS subscription. Since a new NOTAM may change information that is cached in ldproxy for performance reasons (in particular, the spatial and temporal extent of the NOTAM dataset, but also vector tiles) a database trigger is used to notify ldproxy about a new NOTAM. For each new NOTAM, the spatial and temporal extents are evaluated and, if needed, the extents of the NOTAM feature collection are updated. Additionally, vector tiles that include the spatial extent are invalidated in the tile cache.

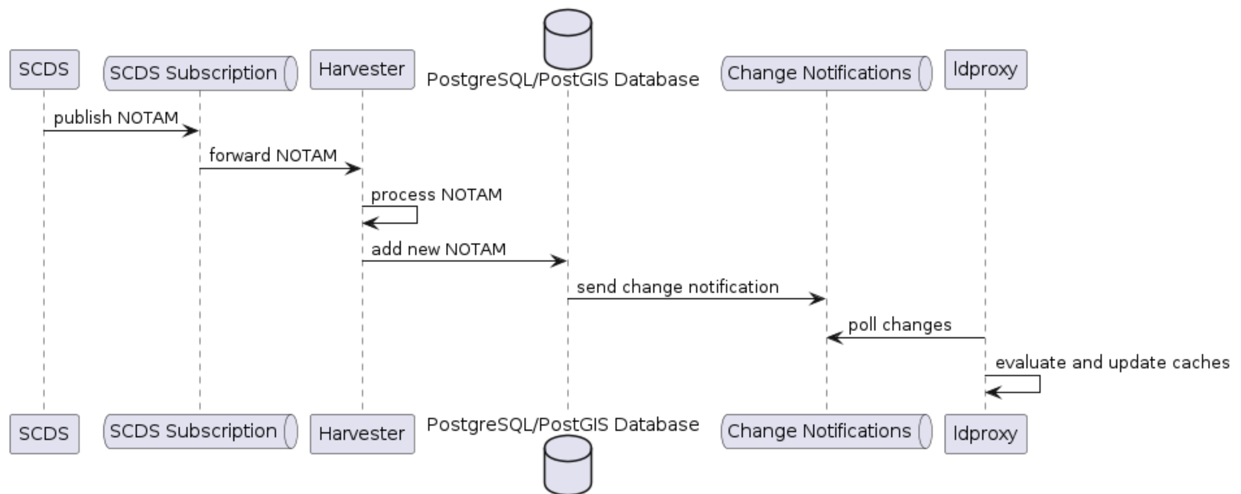


Figure 10 — Communicating Data Changes to ldproxy

The communication between ldproxy and PostgreSQL uses the standard PostgreSQL protocol, which is TCP/IP-based.

5.1.4. Data Retrieval

See sections 4.1.4 and 4.1.5 of the Tested-17 Aviation API Engineering Report on details how the SWIM datasets are loaded into the database.

5.2. Differences to the component from Testbed 17

The component was updated to the latest version of ldproxy and to the new version 1.0.1 of OGC API Features Part 1: Core.

The APIs mainly supported simple access to the data, based on OGC API Features Part 1: Core. That is, paging was supported as well as filtering using `bbox`, `datetime`, and selected feature attributes.

Some sample requests for data requests with simple filter capabilities were included in the description of the NOTAM API, by clicking on "Sample requests to filter NOTAM events" on the HTML landing page for more information. This gave clients an idea of how to construct data requests with simple but already useful filter capabilities.

Example — Example simple filtering requests in the NOTAM API:

- For spatial filtering, the "bbox" query parameter can be used. https://t18.ldproxy.net/d100_fns/collections/notam/items?bbox=-77.6,38.4,-76.2,39.6 selects events in the Washington area.
- For temporal filtering, the "datetime" query parameter can be used. https://t18.ldproxy.net/d100_fns/collections/notam/items?datetime=now selects events that are currently

valid, https://t18.ldproxy.net/d100_fns/collections/notam/items?datetime=2021-06-

27T01:00:00Z selects events that are effective at 1AM UTC on 2021-06-27, and https://t18.ldproxy.net/d100_fns/collections/notam/items?datetime=2021-07-01T00:00:00Z/2021-07-31T23:59:59Z selects NOTAMs that are effective during July 2021.

- For filtering on attributes that are queryables, a query parameter with the attribute name can be used.

  - https://t18.ldproxy.net/d100_fns/collections/notam/items?location=IAD selects events related to Dulles airport.
  - https://t18.ldproxy.net/d100_fns/collections/notam/items?icao_location=KIAD also selects events related to Dulles airport.
  - https://t18.ldproxy.net/d100_fns/collections/notam/items?notam_keyword=RWY selects events related to runways.
  - https://t18.ldproxy.net/d100_fns/collections/notam/items?affected_fir=ZDC selects events affecting the Washington Air Route Traffic Control Center (ZDC).
  - https://t18.ldproxy.net/d100_fns/collections/notam/items?scenario=82 selects the events related to scenario 82 (no documentation about the meaning of the scenarios could be identified).
  - Any of these query parameters can be combined. Events are selected that meet all predicates. https://t18.ldproxy.net/d100_fns/collections/notam/items?datetime=now¬am_keyword=RWY&affected_fir=ZDC selects all NOTAMs that are currently in effect, related to runways and affecting the Washington Air Route Traffic Control Center (ZDC).

The HTML representation of the Features resource also included a simple HTML form to submit requests with these simple filters.

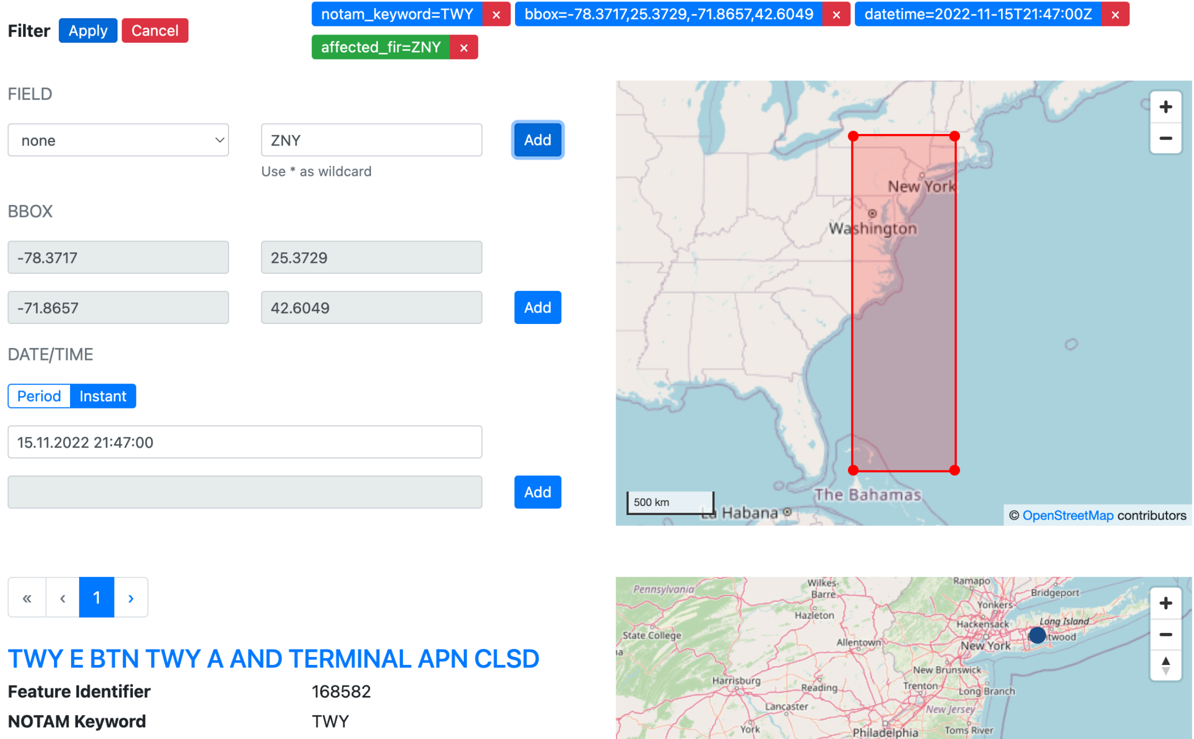


Figure 11 — D100 Simple Filtering in the Web Browser

In Testbed 17 the APIs supported more OGC API building blocks, in particular they supported filtering using OGC API — Features — Part 3: Filtering and the Text encoding of Common Query Language (CQL2). Since the Filtering Services D102 and D103 were used for Filtering in Testbed 18, these capabilities were removed from the Façade Service in the testbed. In addition, other capabilities were removed, like the ability to request the data in various coordinate reference systems.

5.3. Challenges and Lessons Learned

Since the component was in large part reused for Testbed 18, there were no new challenges regarding the pre-existing APIs. See section 4.3 of the Tested-17 Aviation API Engineering Report for several challenges and lessons learned during Testbed 17.

Furthermore, there was a goal to add additional weather datasets to the component in Testbed 18. However, getting access to weather data with a coverage of the continental United States

within the time frame of Testbed 18 was not possible. Therefore, no additional SWIM data was added during Testbed 18 to the component.

6

OGC API-FEATURES FAÇADE 2 (D101)

6 OGC API-FEATURES FAÇADE 2 (D101)

OGC API-Features Façade 2, identified as deliverable 101 or D101, is a component built to demonstrate a service consuming data from a SWIM data service and serving that data through an API built based on OGC API standards offering a limited set of filtering capabilities. George Mason University demonstrated the D101 component.

6.1. Status Quo

For Testbed 16 (OGC document 20-020) the OGC API-Feature façade for the SWIM data messaging services was initially implemented as a RESTful relay Web service using Java, Spring framework, and Java Messaging Service (JMS) clients. A Testbed 18 objective was to test integrated filtering capabilities of OGC API-Features, especially with the OGC API-Features draft Part 3 filtering capability of OGC document 19-079r1. The following summarizes the main gaps or the advances to be made to the API-Features façade Web service for the SWIM implemented in the Testbed 16.

- **Filtering capabilities**: The API-Features façade for the SWIM messaging services implemented for Testbed 16 lacks the support of advanced filtering as specified in the OGC API-Features Part 3 (OGC 19-079r1). There is a filter parameter declared in the OpenAPI specification of the OGC API — Features Standard. The OGC API — Features service will be enhanced with the support of filtering using Filtering / Common Query Language (CQL2) languages in both text and JSON encoding.

- **Additional SWIM data**: The Testbed 16 API-Feature façade service does not handle any weather data. That activity focused on relaying data and caching features from messages in Aeronautical Information Exchange Model (AIXM) and Flight Information Exchange Model (FIXM). Extensions to AIXM and FIXM were handled in Testbed 16, such as the US Federal Aviation Administration Notice to Air Missions System Event (FNSE) extension and the US National Airspace System (NAS) extension. The API-Features service in Testbed 18 will add weather data provided in the Weather Information Exchange Model (WXXM) or its local extensions.

- **Updates of OGC API-Features specification**: The OGC API-Features Standard has evolved and been revised for several years. The core Standard has a new update officially released as version 1.0.1. The API-Features service in Testbed 18 adopted the latest revision. Related tools and software development kits (SDKs) for OpenAPI have also evolved and advanced recently from community efforts. The implementation of the API-Features service in Testbed 18 leverages the advancements of these SDKs, especially the evolutions in the open source community. The interface parsing and validation will be done with the API-first approach in the Python open source community. This would be a different experience from the previous Java development.

6.2. Internal Architecture

The OGC API-Feature façade for SWIM services was re-implemented and enhanced using the OpenAPI stub generator (python-aiohttp) to support standard search and retrieval of SWIM messaging-accumulated feature data with filtering languages of Filtering / Common Query Language (CQL2) in text format or JSON format. In addition to the filtering capabilities, new data was added with new messaging-harvester for the Integrated Terminal Weather System (ITWS) of the System Wide Information Management (SWIM).

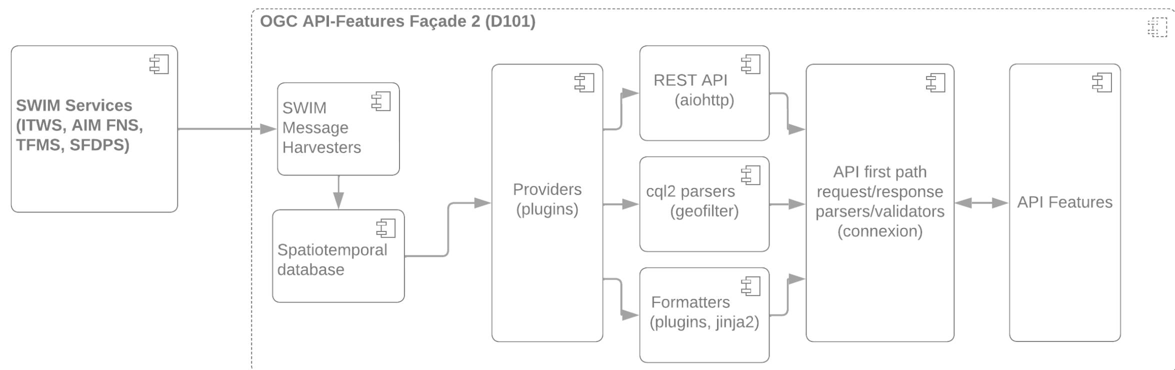


Figure 12 — D101 Component Overview

Filtering language support is primarily based on an extended version of the open source pygeofilter, a Python implementation of cql2-text and cql2-json parsers. There is an example backend translator for Django, SQLAlchemy, GeoPandas, and generic SQL. This implementation forked the open source pygeofilter to extend the SQL backend to work with the PostGIS backend spatiotemporal database.

The overall server-side API-Features library was implemented as a pluggable system to support extension of the server. Two parts are pluggable – backend data providers and frontend output formatters. The backend data providers provide customized interaction with the data source. Different types may require different providers. The current system implemented the data providers with a PostGIS spatiotemporal database. All collection metadata and features are managed with a set of relational entities (tables).

The frontend formatters produce the responses matching the request. In the Testbed 18 implementation, GeoJSON and rendered HTML are supported. The jinja2 templates are used to render HTML output. The rendered HTML page embedded a light-weight JavaScript client to support the interactive retrieval and rendering of results on maps in browsers.

The following figure shows the interaction and data flow between OGC API — Features façade services and SWIM messaging services. In the implementation, the data are first harvested, translated, and cached in a unified spatiotemporal database from the messaging stream of SWIM services. The ingestion of streaming messages is completed with two steps: a Java Messaging Service (JMS) client to receive the messages and a harvester for each data type to load message into the spatiotemporal database. The harvesters implemented in Java in

Testbed 16 were reused for Testbed 18, including harvesters for AIM FNS, TFMS, and SFDPS. The SWIMITWS, the harvesters for ingesting ITWS Weather data from SWIM ITWS messages is newly implemented in Testbed 18. The data include 28 feature collections. Each feature collection has a specific handler to complete the necessary translation, reformatting, and loading into the intermediate spatiotemporal database.

```
OGCAPIUsers-> D101 : API-Features for SWIM features

D101-> SWIMServiceConsumer: Subscription requests
SWIMServiceConsumer -> SWIM: Request for services
SWIM -> SWIMAIMFNS: Subscribe to SWIM AIM FNS service
SWIMAIMFNS -> SWIM: AIM FNS notice to airmen messaging responses
SWIM -> SWIMTFMS: Subscribe to SWIM TFMS service
SWIMTFMS -> SWIM: TFMS traffic flow messaging responses
SWIM -> SWIMSFDPS: Subscribe to SWIM SFDPS service
SWIMSFDPS -> SWIM: SFDPS flight data messaging responses
SWIM -> SWIMITWS: Subscribe to SWIM ITWS service
SWIMITWS -> SWIM: ITWS Weather data messaging responses
SWIM -> SWIMServiceConsumer: Proxy service responses
SWIMServiceConsumer-> D101 : Document-based database indexing

D101->OGCAPIUsers : Resources (GeoJSON, HTML)
```

Figure 13 — API-Features Data Interaction with Backend SWIM Messaging Services

```
' hide the spot
hide circle

' avoid problems with angled crows feet
skinparam linetype ortho

entity "Collections" as e01 {
  *e1_fid : number <<generated>>
  --
  *thegeom : geometry
  *cid : varchar
  table: varchar
  title : varchar
  description : text
  bbox: varchar
  temporal: varchar
  itemtype: varchar
  crs: varchar
  links: varchar
}

entity "Collection.Queryables" as e07 {
  *e7_id : number <<generated>>
  --
  *e2_id : number <<FK>>
  *cid : varchar <<FK>>
  type : varchar
  encoding : text
}

entity "CollectionEncoding" as e02 {
  *e2_id : number <<generated>>
  --
  *e1_id : number <<FK>>
  *type: varchar
  *encoding : text
}
```

```
entity "Document" as e03 {
  *e3_id : number <<generated>>
  --
  *file : varchar
  *status : number
}

entity "Message" as e04 {
  *e4_id : number <<generated>>
  --
  *e3_id : number <<FK>>
  xpath : varchar
  type : number
  encoding : text
}

entity "Feature" as e05 {
  *e5_id : number <<generated>>
  --
  *thegeom : geometry
  *gmlid : varchar
  *gmlidtrace : varchar
  *featuretype : varchar
  *begintime : timestamp
  *endtime : timestamp
  *isinsttime : Boolean
  elevation : double
  verticalTop : double
  elem: varchar
}

entity "FeatureEncoding" as e06 {
  *e6_id : number <<generated>>
  --
  *e4_id : number <<FK>>
  *e5_id : number <<FK>>
  xpath : varchar
  type : number
  encoding : text
}


e01 ||..|{ e02
e05 ||..o{ e06
e06 ||..o{ e04
e03 ||..o{ e04
```

Figure 14 — Major Entity Relationship Diagram for Managed SWIM Features

Table 1 — Server Endpoints

SERVER NAME	URL FOR LANDING PAGE

6.3. Feature collections

6.3.1. AIXM Features

The AIXM feature collections provide a façade the ability to access AIXM features available in SWIM services. The following table lists the collections. The direct access point for the collection can be formed by using a collection id as follows.

https://cat.csiss.gmu.edu/gmuwfs3/collections/<collection_id>

For example, the direct access to collection of Apron Element: https://cat.csiss.gmu.edu/gmuwfs3/collections/ApronElement

Table 2 — AIXM Feature Collections from SWIM Data Services

ID	TITLE	DESCRIPTION
ApronElement	Apron Element	Parts of a defined apron area. ApronElements may have functional characteristics defined in the ApronElement type. ApronElements may have jetway, fuel, towing, docking, and groundPower services.
Airspace	Airspace	A defined three dimensional region of space relevant to air traffic.
Taxiway Element	Taxiway Element	Part of a Taxiway
Runway Element	Runway Element	Runway element may consist of one or more polygons not defined as other portions of the runway class.

6.3.2. FIXM/Flight Features

The FIXM/Flight feature collections provide a façade with access to FIXM feature and traffic flow information in SWIM services. The traffic flow information is provided by the TFMData Service by the FAA's Traffic Flow Management System (TFMS). The following table lists the collections. The direct access point for the collection can be formed by using a collection id as follows.

https://cat.csiss.gmu.edu/gmuwfs3/collections/<collection_id>

For example, the direct access to collection of Track Information: https://cat.csiss.gmu.edu/gmuwfs3/collections/trackinformation

Table 3 — FIXM and Traffic Flow Feature Collections from SWIM Data Services

ID	TITLE	DESCRIPTION
trackinformation	Track information	Track information of flight.
RAPT	RAPT Timeline Data	Route Availability Planning Tool (RAPT) Timeline Data.
flightPlanInformation	Flight plan information	Flight plan information.
flightPlanAmendment Information	Flight plan amendment information	Flight plan amendment information.
NasFlightMessage	Flight Message in FIXM with US Extension	FIXM — Flight information with US Extension. Schemas are available at https://www.fixm.aero/download.pl?view=e .

6.3.3. ITWS Features

ITWS feature collections provide a façade with access to ITWS weather data in SWIM services. The following table lists the collections. The direct access point for the collection can be formed by using collection id as follows. There are currently 28 feature collections available from the SWIM ITWS service.

https://cat.csiss.gmu.edu/gmuwfs3/collections/<collection_id>

For example, the direct access to collection of Precipitation 5nm Product: https://cat.csiss.gmu.edu/gmuwfs3/collections/ITWS9849

Table 4 — ITWS Weather Feature Collections from SWIM Data Services

ID	TITLE	DESCRIPTION
ITWS9849	Precipitation 5nm Product	The 5nm precipitation product shows data from the TDWR surface level scan in National Weather Service (NWS) VIP 6-levels and areas indicated as attenuated data in a gridded format.
ITWS9845	Airport Lightning Warning	The airport lightning warning product is used to notify users that there is the potential for lightning in close proximity to predefined 'critical regions' near an airport. The source of the data is the National Lightning Detection Network (NLDN).
ITWS9911	SM SEP 5nm Product	The Storm Motion Storm Extrapolated Position (SM SEP) product shows motion vectors and contours which indicate the predicted future positions of storm cells. One set of data is generated for the current state and for 10 and 20 minutes in the future.

ID	TITLE	DESCRIPTION
ITWS9838	Tornado Detections Product	The Tornado Detections product includes the location of a tornado as determined by the NEXRAD tornado algorithm.
ITWS9903	Forecast Contour Product	The Forecast Contour product contains four predicted contour lines corresponding to the outer edges of a storm cell for what ITWS calculates to be the position of the outer boundary of the storm cell for 30 and 60 minutes in the future and for both standard and winter forecasts.

6.4. Filtering Capabilities

The filtering capability of the API-Features supports both cql2-text and cql2-json. The parsing and translating of queries into backend SQL in PostGIS/Postgresql is achieved through a forked, revised open source library, pygeofilter. Two major revisions to pygeofilter are functional mapping and field (property) mapping. The functional mapping list enables the conversion of query functions to equivalent functions in the backend PostGIS/Postgresql database. For example, temporal comparison function "BEFORE" may have to be mapped to "<" in temporal field comparison with proper conversion (i.e., converting from string expression into internal sql timestamp).

The field mapping converts the property to field or extracted field value in a backend spatiotemporal database. The spatiotemporal database in this implementation uses a generic spatiotemporal database. The data for each feature are stored as an encoded blob/text in the encoding field with a corresponding media-type. The primary encoding is GeoJSON (i.e. application/geo+json) besides the native encoding (e.g., application/aixm+xml;version=3.0.1). The field mapping uses the JSON parsing and extraction capabilities of Postgresql database. The following list shows examples of mappings for ITWS weather data.

The parsing and validation of user requests is enabled using API-first approach. The API-first approach assures the consistency of declarations in OpenAPI YAML or JSON. The implementation of OGC API — Features also uses the open-source server stub generator, OpenAPI Generator, to generate stubs for the server-side library. Models are generated using the Python-aiohttp generator specifically for Python programs. The models are linked to the interface parser, connexion. Connexion is a framework that automatically handles HTTP requests based on OpenAPI Specification. The tight links between parameters, models, and responses enable the consistency across-board.

6.4.1. Query

6.4.1.1. Query with cql2-text

The following is an example request on ITWS9849 (Precipitation 5nm Product) — Search airports.

The cql2-text is: itws_sites = 'CMH'

The URL encoded request is as follows.

https://cat.csiss.gmu.edu/gmuwfs3/collections/ITWS9849/items?limit=10&filter=itws_sites%20%3D%20%27CMH%27&filter-crs=http%3A%2F%2Fwww.opengis.net%2Fdef%2Fcrs%2FOGC%2F1.3%2FCRS84&filter-lang=cql2-text

6.4.1.2. Query with cql2-json

The cql2-json is as follows.

```
{
  "filter": {
    "op": "=",
    "args": [
      {
        "property": "itws_sites"
      },
      "CMH"
    ]
  }
}
```

Figure 15

The URL-encoded request is as follows.

https://cat.csiss.gmu.edu/gmuwfs3/collections/ITWS9849/items?limit=10&filter=%7B%22filter%22%3A%7B%22op%22%3A%22%3D%22%2C%22args%22%3A%5B%7B%22property%22%3A%22itws_sites%22%7D%2C%22CMH%22%5D%7D%7D&filter-crs=http%3A%2F%2Fwww.opengis.net%2Fdef%2Fcrs%2FOGC%2F1.3%2FCRS84&filter-lang=cql2-json

6.5. Challenges and Lessons Learned

The following summarizes challenges and lessons learned for the implementation and enablement of filtering capabilities in the API-Features services for SWIM data.

- **Filtering enablement**: The Filtering / Common Query Language (CQL2) standard is evolving. The open-source libraries for supporting the parsing and translation of CQL2

queries into common backend databases queries or query services such as SPARQL and GraphQL are not in sync with the changes to the draft standard. For example, the implementation of the OGC API — Features service in Testbed -18 using Python found pygeofilter as the library that supports both cql2-text and cql2-json.

However, the predicates for spatial and temporal conditions are still in the formats with prefixes s_ and t_ respectively. This leads to the mandatory forking and revision of pygeofilter to be usable for the latest version as defined in OGC 19-079r1 (OGC API – Features – Part 3: Filtering). The open-source library pygeofilter also has limited support on the backend. In Testbed 18, pygeofilter does not support the backend with PostGIS/postgresql. The backend translator must be extensively edited to support the evaluation of CQL2 (either cql2-text or cql2-json) into valid a SQL command for PostGIS/Postgresql.

- **Filtering capability description**: SWIM messaging services provide different features with different information models. These schemas are not easily regenerated automatically from (for example) database schemas. The extraction and encoding of queryables were done manually with extensive study of the original data models, including AIXM, FIXM, WXXM, and specialized FAA extensions. The description of queryables is flexible, but this flexibility may lead the response to be too flexible to be used properly by clients.

  The SWIM dataset has pre-defined codes for locations, facilities, and/or status. Some may not be fully expressed with "enum" attributes. For example, the International Air Transport Association's (IATA) Location Identifier is another standard that is widely used in SWIM messages as attributes. Verifying if the code is valid unless the client checks with a registry service of IATA is not possible. The specification may specify how to describe the queryable in such a case and give examples for describing the queryable.

- **Filtering performance**: Performance is an implementation-specific experience based on balancing the general applicability and the specialization of information models in the backend spatiotemporal database. This Testbed's implementation considers managing different features generated from SWIM messages. The harvester would create aa feature collection under the same type if the service has not recorded the collection. In order to achieve this flexibility, the implementation populates partially common attributes (e.g., spatial extent, temporal extent) in defined entities and encodes all into a blob field with corresponding media type (e.g., application/geo+json or native media type – application/aixm+xml;version=3.0.1).

  Most queryables are only accessible through an encoded blob. In defining the field-mapping, we used JSON query against the blob. These properties/attributes for queries are directly managed by PostGIS/Postgresql. Therefore, the performance improvement with indexing these properties is not supported with this implementation. This setup used in Testbed 18 may work fine for SWIM data services if only a limited number of features are searchable within their valid time range.

- **API-first approach for implementing services**: The Testbed 18 implementation of OGC API — Features uses the API-first approach. In other words, the specification defined in YAML is used at the interface, intermediate internal information models, and response generation. This approach needs a well-defined OpenAPI specification in either JSON or YAML. The implementation starts to form a complete OpenAPI document using the examples in API-Features Core Standard GitHub and adds the filter parameters from the examples in swaggerhub. The API-Features (OGC 17-069r5 and OGC 19-079r) schema

does not include paths although these examples include paths. The paths should be an integrated part of the specification.

Having the specification schema repository including more concrete OpenAPI specification examples with fully defined paths at different levels of conformances in YAML or JSON would be helpful. Schemas can be used directly with little configuration changes with the API-first implementation approach. The internal models generated from the YAML specification or JSON have problems in dealing with "oneOf" for parameterization and Geometrycollection GeoJSON object. The generated models need to be revised before the stubs work. This may be an OpenAPI open-source tool issue. The open-source OpenAPI generator, specifically the Python-aiohttp generator, needs to be revised to support the proper generation of models.

The API-first library, connexion, needs to be revised to support proper validation. Alternatively, the specification may avoid this complexity by leaving out the validation of certain property constraints. For example, the BBOX parameter may just be defined as a simple array with the choice of 2-D or 3-D constraints which the connexion not properly validated against the specification. The Geometrycollection may be fully defined in the schema section to enable the generator tool to create inheritable objects in the generated models.

- **Advanced filtering**: SWIM messaging services are streaming data with validated time periods while they are ingested into separate feature collections. For example, the wind information is delivered through three feature collections: itws_9840_start for wind start time, itws_9840_expiration for wind alter information expiration time, and itws_9840_speed for wind speed update. A filter based on wind event would need to be formed as an integrated query against all three feature collections to get the valid wind event information.

  Another case is the part-of relationship between SWIM feature collections which also needs an advanced filtering query to get valid information. For example, a Runway feature may consist of several RunwayElements, a Taxiway feature of multiple TaxiwayElements, and an Apron feature of multiple ApronElements. The classes at the highest level (e.g., Runway, Taxiway, Apron) are in different feature collections than their component classes (e.g., RunwayElement, TaxiwayElement, ApronElement). The CQL2 Standard is not designed to work against multiple feature collections even though they may be served in one OGC API — Features service.

- **Light-weight client for rendering response in browsers**: The implementation of OGC API — Features in Testbed 18 uses an embedded leaflet-based, light-weight JavaScript client to render the HTML response in browsers. This light-weight client helps in guiding users to input proper parameters. The jinja2 templates are used to create the templates for rendering results from different feature collections.

7

FILTERING SERVICE 1 (D102)

7 FILTERING SERVICE 1 (D102)

Ecere Corporation provided the D102 Filtering Service as a Web API instance implementing *OGC API — Features* supporting advanced filtering which can be applied to cascaded services that do not offer these capabilities themselves, such as D100 and D101. Filters can also be pre-defined using extended process execution requests. The service cascades the D100 and D101 service from interactive instruments, Skymantics and George Mason University *OGC API — Features* endpoints, with proper paging support whether filtering is used or not.

The endpoint is available at: https://maps.gnosis.earth/ogcapi/collections/swim .

7.1. Status Quo

Prior to the testbed, Ecere's GNOSIS Map Server did not support filtering using the *Common Query Language (CQL2)* or cascading other map services.

7.2. Internal Architecture

This service is a deployment of the GNOSIS Map Server implementing OGC API Standards and draft specifications. The relevant collections from the deployed endpoint used for this testbed task are found at: (https://maps.gnosis.earth/ogcapi/collections/swim.

The GNOSIS Map Server product is a certified implementation of *OGC API — Features — Part 1: Core* Standard. The server product also supports extensions including *Part 2: Coordinate Reference Systems by reference*, as well as the draft *Part 3: Filtering* specification. The implementation also supports returning filtered results as tiled vector feature data ("vector tiles") through the *OGC API — Tiles — Part 1: Core* Standard, including support for the *Mapbox Vector Tiles* encoding. The D102 service also acts as an *OGC API — Features* client for cascading to the D100, D101, and the Skymantics services.

Experiments were performed using *GeoJSON* and *Features & Geometry JSON* to encode vector features.

7.2.1. Filtering Capabilities

Filtering queries can be specified at request time using *Features — Part 3: Filtering*, using either the CMSS expression language or the OGC Common Query Language (CQL2) (new capability developed for this initiative). Filters can also be pre-defined using the *OGC API — Processes — Part 3: Workflows & Chaining* draft specification, supporting the creation of virtual collections.

Support for the *OGC API — Features* “Search” extension, discussed in more detail in the Filtering Service and Rule Set Engineering Report (ER) (OGC 22-024), is planned for future work. Ecere will be actively following the development of that extension, considering use cases relating to having common basic analytics and search capabilities across different OGC API Standards, such as planned for *OGC API — Processes — Part 3: Workflows & Chaining Input* and *Output modifiers* requirements classes, as well as OGC API — Coverages, OGC API — DGGS. Some of these aspects were previously discussed in Testbed 17 — GeoDataCube API and in the May 2022 *Space Partition* Code Sprint.

7.2.1.1. CQL2

The Ecere implementation supports specifying filters using CQL2, including support for the *Basic*, *Property-Property*, and *Arithmetic Expressions* conformance classes, and the *cql2-text* encoding. Support for additional conformance classes, as well as the `cql2-json` encoding, is planned for future work.

Test collections, using Natural Earth data, for some of the CQL2 Abstract Tests are deployed at the following location.

https://maps.gnosis.earth/ogcapi/collections/cql2-test

The following are a couple of example requests using these test collections.

https://maps.gnosis.earth/ogcapi/collections/cql2-test:ne_110m_admin_0_countries/items?filter=NAME=’Luxembourg’

https://maps.gnosis.earth/ogcapi/collections/cql2-test:ne_110m_populated_places_simple/items?filter=NOT (start is NULL) AND (pop_other<1038288 OR name<’København’) AND NOT (“date” IS NOT NULL or NOT start is NULL) AND (“date” IS NOT NULL)&f=json

GNOSIS Map Server

https://maps.gnosis.earth/ogcapi/collections/cql2-test:ne_110m_populated_places_simple/items?filter=pop_other=1038280+8

GNOSIS Map Server ecere.ca

København (feature 168)

from ne_110m_populated_places_simple

Back to data collection

*(Download **GeoJSON** representation; view on **geojson.io**)*

Geospatial extent: { { lat: 55.68051, lon: 12.5615399 } - { lat: 55.68051, lon: 12.5615399 } }

Property	Value
featurecla	Admin-0 capital
name	København
namepar	Copenhagen
nameascii	Kobenhavn
sov0name	Denmark
sov_a3	DNK
adm0name	Denmark
adm0_a3	DNK
adm1name	Hovedstaden
pop_max	1085000
pop_min	1085000
pop_other	1038288
meganame	K
ls_name	Copenhagen
date	2021-04-16
start	2021/04/16 10:15:59
end	2022/04/16 10:16:06
boolean	1

Figure 16 — Single feature returned from CQL2 filter query on test collection

7.2.2. Cascading

Ecere implemented the ability to provide access to cascaded services, while also supporting filtering features returned by those cascaded services, which may not themselves provide filtering support. Pagination challenges specific to this capability were addressed to reflect the fact that not all results returned by the cascaded service will be part of the filtering service responses.

The following link is an example items request to features originating from the cascaded service D100 service provided by interactive instruments.

http://maps.gnosis.earth/ogcapi/collections/swim:d100_airspace:class_c/items

GNOSIS Map Server

https://maps.gnosis.earth/ogcapi/collections/swim:faa:faa_flight_plans/items/86982710619143863

Feature 86982710619143863

from faa_flight_plans

Back to data collection

(Download GeoJSON representation; view on geojson.io)

Property	Value
aircraftdescription	▶ (object)
arrival	▶ (object)
centre	▼ (object) { "id" : "ZBW", "type" : "nasr:ARTCC", "url" : "https://aviationapi.skymantics.com/faa/collections/artccs/items/ZBW" }
departure	▼ (object) { "departurePoint" : { "id" : "KJFK", "type" : "schema:Airport", "url" : "https://aviationapi.skymantics.com/faa/collections/airports/items/KJFK" }, "runwayPositionAndTime" : { "runwayTime" : { "actual" : { "time" : "2023-01-08T18:41:00Z" } } }, "xsi:type" : "ns5:NasDepartureType" }
flightidentification	▼ (object) { "aircraftIdentification" : "THY4", "computerId" : "836", "siteSpecificPlanId" : "205", "xsi:type" : "ns5:NasFlightIdentificationType" }
flightplan	KN51966100
flightstatus	ACTIVE
flighttype	SCHEDULED
gufi	000dfeb7-4035-4e0c-8fa5-dd45dae372bd
lastupdated	2023-01-08T19:06:40.545000
operator	▼ (object) { "operatingOrganization" : { "organization" : { "name" : "THY" } } }
source	AH
system	SLC
airspeed	
altitude	▶ (object)
coordination	▶ (object)
enroute	▶ (object)

Figure 17 — Single flight plan feature cascaded from Skymantics service

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

The *OGC API — Features* Search extension (described in detail in OGC 22-024 ER) has much 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 follows.

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 parameter 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` as follows.

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 retrieval of 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 18 — Paging through an output collection resulting from the above filter query pre-defined *using OGC API - Processes - Part 3*

7.2.4. 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 19

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 20

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.

7.3. Challenges and Lessons Learned

Building *next* and *prev* links that reflect both cascaded and filtered features, while respecting the limits of the original queries, in order to page features properly proved to be an important challenge. Issues with some of the cascaded implementations themselves also made the task more difficult, including the following issues that were filed upstream with *pygeoapi*:

- support for describing object feature properties (#1090);
- use of invalid JSON Schema types in queryables resource (#1091); and

- 500 server error returned when using `application/geo+json` as the media type for `Accept:` header (#1108).

The use of complex objects and arrays in feature properties, as well as the occurrence of null geometry and of mixed geometry types by the cascaded services, were other major difficulties that required significant changes and improvements to Ecere's GNOSIS Map Server implementation.

While implementing support for CQL2, Ecere reviewed the draft CQL2 specification and detected several important issues and suggested additional improvements such as simplifying the grammar to facilitate parser implementations. Several of the issues were resolved during the testbed, including the following.

- CQL2 division real or integer? data type dependent? (#711)
- `/functions` — out of scope for CQL2? (in Part 3: Filtering) (#715)
- `POLYGON\(`… in CQL2 examples (#716)
- CQL2: Permission 3 (binary comparison operators for time instants) targeting clients? (#719)
- CQL2: Clarify that DATE and TIMESTAMP literals are in basic, but not INTERVAL (#720)
- CQL2: Requirement 2C is a requirement on the client; 2B "literal value" (#721)
- CQL2: Basic Spatial Operators issues (#733)
- CQL2: Spatial Operators Issues (#734)
- CQL2 (AT): A.12 CQL2 Text A.13 CQL2 JSON (#750)
- CQL2: Remove — from list of valid identifier characters? (#766)

It was agreed that the following issues are to be addressed in a future revision of CQL2.

- Simplify the cql2-text grammar (future version improvements?) (#705)
- CQL2 as an expression language (not only boolean predicates) (#723)

Other issues were still pending completion at the time of writing this report.

- CQL2 Escaping (#717)
- CQL2: Thoughts on arrays and IN (#718)
- CQL2/AT1.6: {p2} and {p3} contribute nothing to result (#768)

Ecere also worked on a UML conceptual model for expressions, operators, and standardized functions as part of the *Styles & Symbology SWG* activities with the intent for it to be compatible

with the CQL2 conformance classes and support a similar modularity, and potentially provide the foundations of an OGC conceptual basis for expressions and filtering.

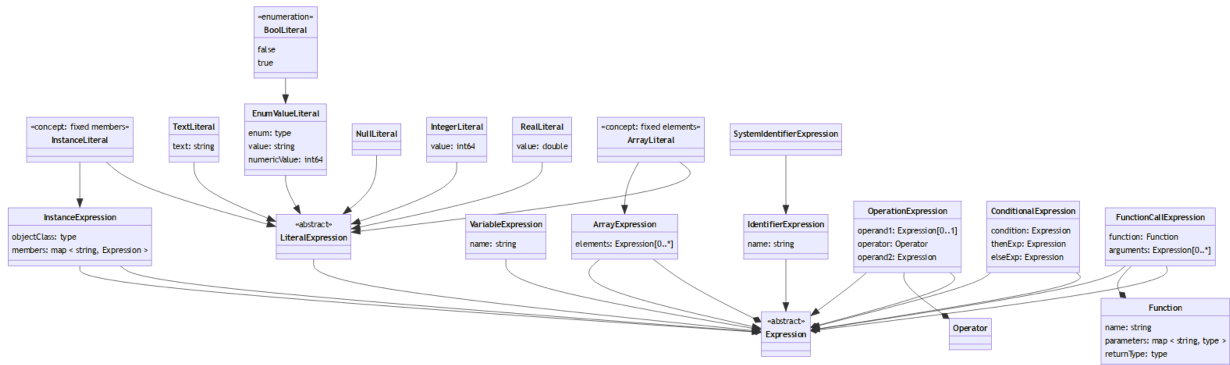


Figure 21 — Expressions UML Conceptual Model, covering CQL2 capabilities

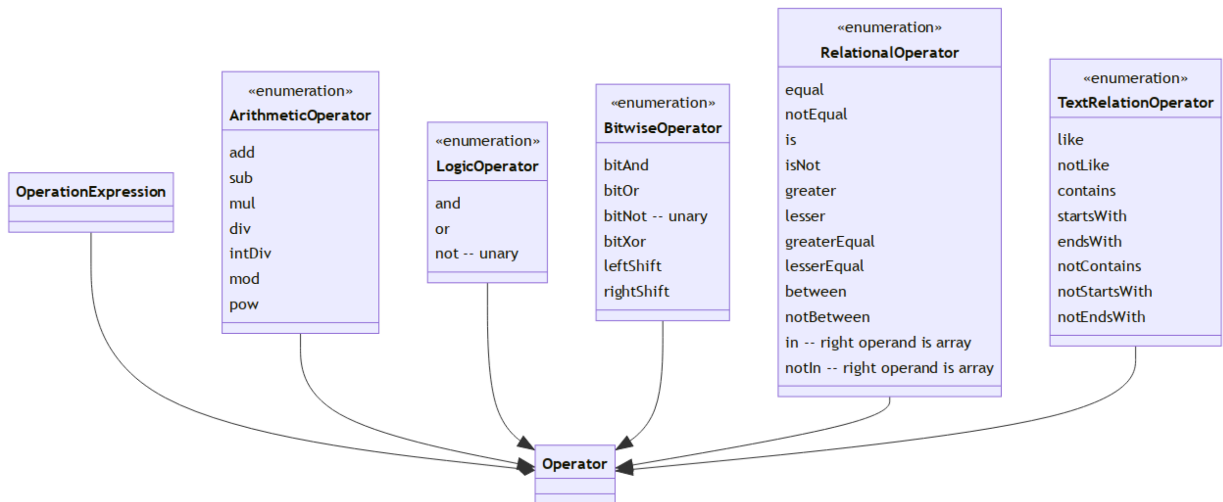


Figure 22 — Operators UML Conceptual Model, covering CQL2 capabilities

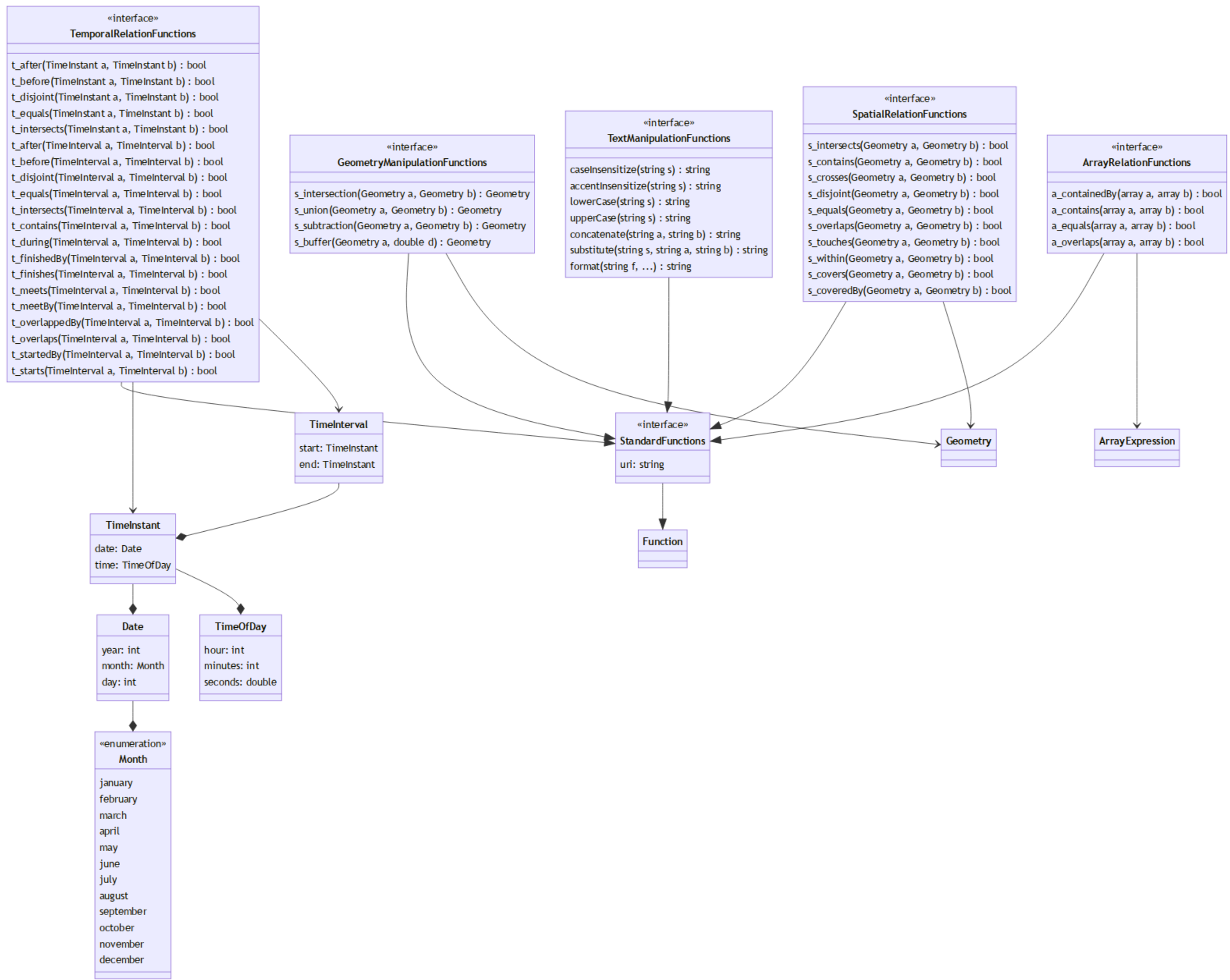


Figure 23 — Standard functions UML Conceptual Model, covering CQL2 capabilities

An important lesson to take away from the Testbed-18 initiative is that while a filtering service can be implemented separately from a distinct *OGC API — Features* façade service, in many ways this architecture really defeats the purpose of the filtering capabilities. This is a result of a request returning the full unfiltered set of features must still be made to the cascaded service. A more efficient architecture would be required to perform the filtering as close to the data source as possible.

7.4. Recommendations and Future Work

- **Filter close to the data**: In this task, the deliverables architecture called for one component to perform the filtering while another acts as an OGC API façade. The initiative highlighted that this architecture is not ideal, since the filtering service would still need to request the entire dataset prior to performing the filtering. Instead, for a practical application, the filtering should be performed as close to the source of data as possible, within the same component.

- **Explore the potential of spatial joins**: In the context of queries spanning multiple collections, participants briefly discussed the capability to perform queries that not only filter and combine the results from multiple collections, but also perform join operations, including spatiotemporal joins. A typical scenario that is relevant to aviation use cases is to query flight paths which would encounter bad weather. This could be achieved by correlating the geometry of the flight paths with the location of severe weather patterns.

  The dataset used for such correlation would not necessarily need to be on the same server. Instead, the client could possibly direct a filtering service (e.g., one with flight paths) to retrieve data from a particular OGC API instance (e.g., one with weather data). It may in fact may faster for the two servers to talk to each other directly, for example if they are both hosted on the same cloud infrastructure, as opposed to the client first retrieving the data, then submitting it as part of its filtering query.

- **First-Filtering**: In addition, the filtering service may be able to perform a first filtering pass that requires retrieving less data from the other service. This capability could be explored in greater depth in a future activity.

8 FILTERING SERVICE 2 (D103)

8 FILTERING SERVICE 2 (D103)

The Filtering Service 2, identified as deliverable 103 or D103, is an OGC Web API instance that supports advanced filtering for services that do not offer these capabilities themselves, such as D100 and D101. D103 was demonstrated by interactive instruments.

8.1. Internal Architecture

The component consists of the following three filter APIs that were deployed.

- Airports
- Airspaces
- Federal Notice to Airmen System (FNS)

These filter APIs provide access to the same feature data as the corresponding façade, but in addition also provide support for Part 2: Coordinate Reference Systems by Reference, Part 3: Filtering and Common Query Language (CQL2).

Responses with feature data could be requested in the following representations/formats:

- GeoJSON (media type `application/geo+json`, query parameter `f=json`)
- JSON-FG (media type `application/vnd.ogc.fg+json`, query parameter `f=jsonfg`)
- HTML (media type `text/html`, query parameter `f=html`)
- FlatGeobuf (media type `application/flatgeobuf`, query parameter `f=fgb`)

8.2. ldproxy

Like Clause 5, the APIs are provided by ldproxy. To support the requirements for advanced filtering of SWIM data, the following enhancements were been implemented in ldproxy during the testbed:

- updated to the latest draft version of OGC API — Features — Part 3: Filtering and Common Query Language (CQL2);
- support for the CQL2-JSON encoding of the Common Query Language (CQL2);

- support for all API building blocks specified in Clauses 4 and 5 of the Testbed-18 Filtering Service and Rule Set Engineering Report (D002); and
- support for JSON-FG was updated to the latest draft (version 0.1).

At the time of writing this ER, the updated code is in the branch "multi-queries" of ldproxy and the master branches of XtraPlatform and XtraPlatform Spatial.

8.3. Filtering Capabilities

8.3.1. Support for CQL2 and Part 3 (Filtering) of OGC API Features

All conformance classes of the Common Query Language (CQL2) were supported, except "Accent-insensitive Comparisons," "Functions," and "Arithmetic Expressions." Both the CQL2-Text and CQL2-JSON encodings were supported.

The conformance classes that are supported could be seen at the relative path `conformance` from the API landing page, for example at https://t18.ldproxy.net/d103_fns/conformance. The CQL2 conformance classes all started with the base URI `http://www.opengis.net/spec/cql2/0.0/conf`.

Example 1 — Conformance Declaration of the NOTAM Filter API


```
{
  "links": [
    {
      "rel": "self",
      "type": "application/json",
      "title": "Dieses Dokument",
      "href": "https://t18.ldproxy.net/d103_fns/conformance?f=json"
    },
    {
      "rel": "alternate",
      "type": "text/html",
      "title": "Dieses Dokument als HTML",
      "href": "https://t18.ldproxy.net/d103_fns/conformance?f=html"
    }
  ],
  "conformsTo": [
    "http://www.opengis.net/spec/cql2/0.0/conf/advanced-comparison-operators",
    "http://www.opengis.net/spec/cql2/0.0/conf/array-operators",
    "http://www.opengis.net/spec/cql2/0.0/conf/basic-cql2",
    "http://www.opengis.net/spec/cql2/0.0/conf/basic-spatial-operators",
    "http://www.opengis.net/spec/cql2/0.0/conf/case-insensitive-comparison",
    "http://www.opengis.net/spec/cql2/0.0/conf/cql2-json",
    "http://www.opengis.net/spec/cql2/0.0/conf/cql2-text",
    "http://www.opengis.net/spec/cql2/0.0/conf/property-property",
    "http://www.opengis.net/spec/cql2/0.0/conf/spatial-operators",
    "http://www.opengis.net/spec/cql2/0.0/conf/temporal-operators",
    "http://www.opengis.net/spec/ogcapi-common-1/1.0/conf/core",
    "http://www.opengis.net/spec/ogcapi-common-1/1.0/conf/html",
    "http://www.opengis.net/spec/ogcapi-common-1/1.0/conf/json",
```

```
    "http://www.opengis.net/spec/ogcapi-common-1/1.0/conf/oas30",
    "http://www.opengis.net/spec/ogcapi-common-2/0.0/conf/collections",
    "http://www.opengis.net/spec/ogcapi-common-2/0.0/conf/html",
    "http://www.opengis.net/spec/ogcapi-common-2/0.0/conf/json",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/core",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/geojson",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/html",
    "http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/oas30",
    "http://www.opengis.net/spec/ogcapi-features-2/1.0/conf/crs",
    "http://www.opengis.net/spec/ogcapi-features-3/0.0/conf/features-filter",
    "http://www.opengis.net/spec/ogcapi-features-3/0.0/conf/filter",
    "http://www.opengis.net/spec/ogcapi-features-n/0.0/conf/ad-hoc-queries",
    "http://www.opengis.net/spec/ogcapi-features-n/0.0/conf/core",
    "http://www.opengis.net/spec/ogcapi-features-n/0.0/conf/manage-stored-
queries"
  ]
}
```

To construct a query with a richer filter than was supported by the D100 Data APIs, clients must inspect the queryable properties of the feature collection ("queryables"). The response is a JSON Schema and every property can be used in a filter expression. Each queryable is described with a title, a description, and a type. Where known, constraints on the range of values are also included. This should support generating meaningful queries as well as clients that dynamically generate a UI for the data. For example, a property with an `enum` constraint could be represented using a drop-down list.

Example 2 — Queryables of the NOTAM features

```
{
  "title" : "NOTAMs",
  "description" : "Notice to Airmen",
  "properties" : {
    "notam_keyword" : {
      "title" : "NOTAM Keyword",
      "description" : "Keyword associated with the NOTAM",
      "type" : "string",
      "enum" : [ "AD", "APRON", "AIRSPACE", "CHART", "COM", "IAP", "NAV",
"OBST", "ODP", "ROUTE", "RWY", "SECURITY", "SID", "SPECIAL", "STAR", "SVC",
"TWY", "VFP", "CONSTRUCTION", "LTA" ]
    },
    "notam_function" : {
      "title" : "NOTAM Function",
      "description" : "Function of the NOTAM (New, Replacement, Cancelled)",
      "type" : "string",
      "enum" : [ "NOTAMN", "NOTAMR", "NOTAMC" ]
    },
    "valid_time_begin" : {
      "title" : "Valid time (begin)",
      "format" : "date-time",
      "type" : "string"
    },
    "valid_time_end" : {
      "title" : "Valid time (end)",
      "format" : "date-time",
      "type" : "string"
    },
    "text" : {
      "title" : "Text",
      "description" : "NOTAM condition text.",
      "type" : "string"
    },
```

```
    "selection_code" : {
      "title" : "Q Code",
      "description" : "Q Code value for the NOTAM.",
      "type" : "string"
    },
    "year" : {
      "title" : "Year",
      "description" : "NOTAM year values per ICAO Annex-15.",
      "type" : "string"
    },
    "number" : {
      "title" : "Number",
      "description" : "NOTAM number value per ICAO Annex-15.",
      "type" : "integer"
    },
    "scenario" : {
      "title" : "Scenario",
      "description" : "Identifier of the event scenario used for digital
encoding. The mapping can be found in the Event Scenario documents.",
      "type" : "string"
    },
    "affected_fir" : {
      "title" : "Flight Information Region",
      "description" : "Flight Information Region (FIR) that is impacted by the
NOTAM.",
      "type" : "string"
    },
    "location" : {
      "title" : "Location designator",
      "description" : "NOTAM location designator of the affected airport/
heliport or facility.",
      "type" : "string"
    },
    "icao_location" : {
      "title" : "ICAO location designator",
      "description" : "ICAO location designator, if published.",
      "type" : "string"
    },
    "series" : {
      "title" : "Series",
      "description" : "NOTAM series value per International Civil Aviation
Organization (ICAO) Annex-15.",
      "type" : "string"
    },
    "type" : {
      "title" : "Type",
      "description" : "NOTAM type value per ICAO Annex-15. Accepted values are:
 New (N), Replace (R), Cancel (C).",
      "type" : "string",
      "enum" : [ "N", "R", "C" ]
    },
    "issued" : {
      "title" : "Issued",
      "description" : "Issue date/time of the NOTAM.",
      "format" : "date-time",
      "type" : "string"
    },
    "traffic" : {
      "title" : "Traffic",
      "description" : "NOTAM traffic value per ICAO Annex-15.",
      "type" : "string"
    },
    "purpose" : {
```

```
        "title" : "Purpose",
        "description" : "NOTAM purpose value per ICAO Annex-15.",
        "type" : "string"
      },
      "scope" : {
        "title" : "Scope",
        "description" : "NOTAM scope value per ICAO Annex-15.",
        "type" : "string"
      },
      "minimum_fl" : {
        "title" : "Minimum flight level",
        "description" : "NOTAM minimum flight level value per ICAO Annex-15.",
        "type" : "string"
      },
      "maximum_fl" : {
        "title" : "Maximum flight level",
        "description" : "NOTAM maximum flight level value per ICAO Annex-15.",
        "type" : "string"
      },
      "coordinates" : {
        "title" : "Coordinates",
        "type" : "string"
      },
      "radius" : {
        "title" : "Radius",
        "type" : "string"
      },
      "schedule" : {
        "title" : "Schedule",
        "description" : "Contains a schedule of activity/outage if the hours of
effect are less than 24 hours a day.",
        "type" : "string"
      },
      "lower_limit" : {
        "title" : "Lower limit",
        "description" : "Specifies the lower height restriction of the NOTAM.",
        "type" : "string"
      },
      "upper_limit" : {
        "title" : "Upper limit",
        "description" : "Specifies the upper height restriction of the NOTAM.",
        "type" : "string"
      },
      "geometry" : {
        "$ref" : "https://geojson.org/schema/Geometry.json"
      }
    },
    "additionalProperties" : false,
    "type" : "object",
    "$schema" : "https://json-schema.org/draft/2019-09/schema",
    "$id" : "https://t18.ldproxy.net/d103_fns/collections/notam/queryables"
}
```

The code below shows an example of a query with a CQL2-Text filter expression. This request selects events related to runways/taxiways/airspaces along a trajectory in Florida with a buffer that was effective between 6PM and 10PM UTC on July 12th, 2021.

Without the percent encoding:

```
https://t18.ldproxy.net/d103_fns/collections/notam/items?filter=
S_INTERSECTS(geometry,POLYGON((-79.53576165455738 25.18569533817705, -
79.50845450086926 25.091558846018916, -79.50003716505086 24.993903798755063,
```

```
-79.51083312059717 24.896483025086265, -79.54042748499097 24.803040350710415,
 -79.58768296340534 24.71716672520052, -79.65078355430599 24.64216222382644,
 -79.7273043373718 24.580909227512972, -79.81430466182294 24.53576165455737,
 -79.90844115398109 24.508454500869263, -80.00609620124493 24.50003716505086,
 -80.10351697491373 24.510833120597162, -80.19695964928958 24.54042748499097,
 -80.28283327479947 24.58768296340534, -80.35783777617355 24.650783554305985,
 -80.41909077248702 24.72730433737181, -80.46423834544262 24.81430466182295,
-82.46423834544262 29.81430466182295, -82.49154549913074 29.908441153981084,
-82.49996283494914 30.006096201244937, -82.48916687940283 30.103516974913735,
 -82.45957251500903 30.196959649289585, -82.41231703659466 30.28283327479948,
 -82.34921644569401 30.35783777617356, -82.2726956626282 30.419090772487028,
-82.18569533817706 30.46423834544263, -82.09155884601891 30.491545499130737,
-81.99390379875507 30.49996283494914, -81.89648302508627 30.489166879402838,
-81.80304035071042 30.45957251500903, -81.71716672520051 30.41231703659466, -
81.64216222382645 30.349216445694015, -81.58090922751298 30.27269566262819, -
81.53576165455738 30.18569533817705, -79.53576165455738 25.18569533817705)))
AND
notam_keyword IN ('RWY', 'TWY', 'AIRSPACE') AND
T_INTERSECTS(INTERVAL(valid_time_begin,valid_time_end), INTERVAL('2021-07-
12T18:00:00Z','2021-07-12T22:00:00Z'))
```

Figure 24

8.3.2. Support for property selection

The Filter APIs not only supported filtering, but also supported restricting the feature properties that are returned by implementing the "Property Selection" proposal for OGC API — Features.

To reduce the properties in the response to a subset of all feature properties, these had to be listed in a query parameter `properties`, e.g., `properties=text,notam_keyword,issued`.

8.3.3. Support for sorting

In addition to selecting the properties to be included in the response, sorting the features in the response was supported by implementing the "Sorting" requirements class of OGC API Records.

To sort the features in the response, the feature properties that should be used to determine the sort order had to be listed in a query parameter `sortby`, e.g., `sortby=notam_keyword,text`.

Similar to the queryables, clients could identify the sortable properties of the feature collection ("sortables"). Again, the response was a JSON Schema, and every property could be used in the "sortby" query parameter.

Example — Sortables of the NOTAM features: The APIs were configured so that the same properties could be used for sorting and in filter expressions.


```
{
  "title" : "NOTAMs",
  "description" : "Notice to Airmen",
  "properties" : {
    "notam_keyword" : {
      "title" : "NOTAM Keyword",
      "description" : "Keyword associated with the NOTAM",
      "type" : "string",
```

```
      "enum" : [ "AD", "APRON", "AIRSPACE", "CHART", "COM", "IAP", "NAV",
"OBST", "ODP", "ROUTE", "RWY", "SECURITY", "SID", "SPECIAL", "STAR", "SVC",
"TWY", "VFP", "CONSTRUCTION", "LTA" ]
    },
    "notam_function" : {
      "title" : "NOTAM Function",
      "description" : "Function of the NOTAM (New, Replacement, Cancelled)",
      "type" : "string",
      "enum" : [ "NOTAMN", "NOTAMR", "NOTAMC" ]
    },
    "valid_time_begin" : {
      "title" : "Valid time (begin)",
      "format" : "date-time",
      "type" : "string"
    },
    "valid_time_end" : {
      "title" : "Valid time (end)",
      "format" : "date-time",
      "type" : "string"
    },
    "text" : {
      "title" : "Text",
      "description" : "NOTAM condition text.",
      "type" : "string"
    },
    "selection_code" : {
      "title" : "Q Code",
      "description" : "Q Code value for the NOTAM.",
      "type" : "string"
    },
    "year" : {
      "title" : "Year",
      "description" : "NOTAM year values per ICAO Annex-15.",
      "type" : "string"
    },
    "number" : {
      "title" : "Number",
      "description" : "NOTAM number value per ICAO Annex-15.",
      "type" : "integer"
    },
    "scenario" : {
      "title" : "Scenario",
      "description" : "Identifier of the event scenario used for digital
encoding. The mapping can be found in the Event Scenario documents.",
      "type" : "string"
    },
    "affected_fir" : {
      "title" : "Flight Information Region",
      "description" : "Flight Information Region (FIR) that is impacted by the
NOTAM.",
      "type" : "string"
    },
    "location" : {
      "title" : "Location designator",
      "description" : "NOTAM location designator of the affected airport/
heliport or facility.",
      "type" : "string"
    },
    "icao_location" : {
      "title" : "ICAO location designator",
      "description" : "ICAO location designator, if published.",
      "type" : "string"
    },
```

```
    "series" : {
      "title" : "Series",
      "description" : "NOTAM series value per International Civil Aviation
Organization (ICAO) Annex-15.",
      "type" : "string"
    },
    "type" : {
      "title" : "Type",
      "description" : "NOTAM type value per ICAO Annex-15. Accepted values are:
 New (N), Replace (R), Cancel (C).",
      "type" : "string",
      "enum" : [ "N", "R", "C" ]
    },
    "issued" : {
      "title" : "Issued",
      "description" : "Issue date/time of the NOTAM.",
      "format" : "date-time",
      "type" : "string"
    },
    "traffic" : {
      "title" : "Traffic",
      "description" : "NOTAM traffic value per ICAO Annex-15.",
      "type" : "string"
    },
    "purpose" : {
      "title" : "Purpose",
      "description" : "NOTAM purpose value per ICAO Annex-15.",
      "type" : "string"
    },
    "scope" : {
      "title" : "Scope",
      "description" : "NOTAM scope value per ICAO Annex-15.",
      "type" : "string"
    },
    "minimum_fl" : {
      "title" : "Minimum flight level",
      "description" : "NOTAM minimum flight level value per ICAO Annex-15.",
      "type" : "string"
    },
    "maximum_fl" : {
      "title" : "Maximum flight level",
      "description" : "NOTAM maximum flight level value per ICAO Annex-15.",
      "type" : "string"
    },
    "coordinates" : {
      "title" : "Coordinates",
      "type" : "string"
    },
    "radius" : {
      "title" : "Radius",
      "type" : "string"
    },
    "schedule" : {
      "title" : "Schedule",
      "description" : "Contains a schedule of activity/outage if the hours of
effect are less than 24 hours a day.",
      "type" : "string"
    },
    "lower_limit" : {
      "title" : "Lower limit",
      "description" : "Specifies the lower height restriction of the NOTAM.",
      "type" : "string"
    },
```

```
    "upper_limit" : {
      "title" : "Upper limit",
      "description" : "Specifies the upper height restriction of the NOTAM.",
      "type" : "string"
    }
  },
  "additionalProperties" : false,
  "type" : "object",
  "$schema" : "https://json-schema.org/draft/2019-09/schema",
  "$id" : "https://t18.ldproxy.net/d103_fns/collections/notam/sortables"
}
```

8.3.4. Support for basic filtering according to OGC API Features Core

The Filter APIs also supported the simple filtering mechanisms supported by the corresponding Data API in D100. See here for details and sample requests.

8.3.5. Support for the Search resources

The requirements for advanced filtering of SWIM data extended beyond these existing draft specifications, in particular:

- to retrieve features from multiple collections with a single request;
- to restrict the query capabilities of a business user; and
- to simplify the execution of queries by business users.

The additional API building blocks needed to support these requirements are specified in the Testbed-18 Filtering Service and Rule Set Engineering Report (D002) and have been implemented in the Filter Service D103.

The filtering rules / stored queries used in the scenario were generated by the Developer Client (D105), but as examples the following stored queries were published in the Filter APIs.

Default values were provided for all parameters so that all queries could be executed without providing a parameter.

Example 1 — Stored Query: Concrete apron, runway, and taxiway elements at NYC area airports: A simple query without parameters in the Airports API. It selects all apron, runway, and taxiway elements with concrete at the major NYC area airports.
The response was restricted to the properties “geometry,” “airport,” and “type”.

```
{
  "id": "nyc-concrete-surface-areas",
  "title": "Concrete apron, runway and taxiway elements at NYC area airports",
  "queries": [
    {"collections":["apronelement"]},
    {"collections":["runwayelement"]},
    {"collections":["taxiwayelement"]}
  ],
  "filter": {
```

```
    "op": "and",
    "args": [
      { "op": "=", "args": [{ "property": "composition" }, "CONC"] },
      { "op": "in", "args": [{ "property": "airport" }, ["JFK", "EWR", "LGA"]]
}
    ]
  },
  "properties": ["geometry", "airport", "type"],
  "limit": 1000
}
```

Example 2 — Stored Query: Apron, runway, and taxiway elements of a specific type at one or more airports: A similar query in the Airports API, but with two parameters, the type (normal use, parking, shoulder, intersection) and the airports (one or more from a list of airports in the eastern US).
The response was restricted to the properties "geometry," "airport," "type," and "composition".

```
{
  "id": "surface-areas",
  "title": "Apron, runway and taxiway elements of a specific type at one or
more airports.",
  "queries": [
    { "collections": ["apronelement"]   },
    { "collections": ["runwayelement"]  },
    { "collections": ["taxiwayelement"] }
  ],
  "filter": {
    "op": "and",
    "args": [
      {
        "op": "=",
        "args": [
          {"property": "type"},
          {
            "$parameter": {
              "type": {
                "title": "Type of the apron, runway or taxiway element",
                "description": "The following types are distinguished: normal
use, parking, shoulder, intersection.",
                "type": "string",
                "enum": ["NORMAL", "PARKING", "SHOULD", "INTERS"],
                "default": "NORMAL"
              }
            }
          }
        ]
      },
      {
        "op": "in",
        "args": [
          {"property": "airport"},
          {
            "$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"],
  "limit": 1000
}
```

Example 3 — Stored Query: NOTAMS along the route IAD to JFK: This query in the NOTAM API filtered NOTAMs by location, by keyword, and by time interval. The default values for the parameters were: the route from IAD to JFK with a 50km buffer (location), airspace-related NOTAMs (keyword), and between 15:05 and 16:30 UTC on August 18th, 2022 (time interval).

```
{
  "id": "generic-notam-query",
  "title": "filter NOTAMS",
  "description": "NOTAMs filtered by location, by keyword and by time interval.
",
  "collections": ["notam"],
  "filter": {
    "op": "and",
    "args": [
      {
        "op": "s_intersects",
        "args": [
          {"property": "geometry"},
          {
            "$parameter": {
              "geometry": {
                "title": "NOTAM geometry",
                "description": "NOTAMs with a location that intersects the
geometry are selected.",
                "type": "object",
                "required": ["type", "coordinates"],
                "properties": { "type": {"type": "string"}, "coordinates":
{"type": "array"} },
                "default": {
                  "type": "Polygon",
                  "coordinates": [
                    [
                      [ -74.07517, 41.03269 ],
                      [ -73.96881, 41.06885 ],
                      [ -73.85509, 41.08851 ],
                      [ -73.73847, 41.09087 ],
                      [ -73.62351, 41.07587 ],
                      [ -73.51473, 41.04408 ],
                      [ -73.41637, 40.99675 ],
                      [ -73.33224, 40.93573 ],
                      [ -73.26558, 40.86340 ],
                      [ -73.21892, 40.78257 ],
                      [ -73.19399, 40.69637 ],
                      [ -73.19167, 40.60812 ],
                      [ -73.21195, 40.52123 ],
                      [ -73.25399, 40.43900 ],
                      [ -73.31608, 40.36459 ],
```

```
                    [ -73.39583, 40.30082 ],
                    [ -73.49016, 40.25010 ],
                    [ -77.14409, 38.57116 ],
                    [ -77.24494, 38.53223 ],
                    [ -77.35344, 38.50940 ],
                    [ -77.46549, 38.50353 ],
                    [ -77.57685, 38.51484 ],
                    [ -77.68334, 38.54291 ],
                    [ -77.78091, 38.58666 ],
                    [ -77.86586, 38.64445 ],
                    [ -77.93495, 38.71408 ],
                    [ -77.98551, 38.79290 ],
                    [ -78.01555, 38.87792 ],
                    [ -78.02384, 38.96588 ],
                    [ -78.01000, 39.05341 ],
                    [ -77.97446, 39.13715 ],
                    [ -77.91854, 39.21387 ],
                    [ -77.84433, 39.28058 ],
                    [ -77.75467, 39.33469 ],
                    [ -74.07517, 41.03269 ]
                  ]
                ]
              }
            }
          }
        }
      ]
    },
    {
      "op": "in",
      "args": [
        {"property": "notam_keyword"},
        {
          "$parameter": {
            "notamKeywords": {
              "title": "NOTAM keywords",
              "description": "A list of NOTAM keywords to filter only NOTAMs
with the selected keywords. For more information about the NOTAM keywords see
https://www.faa.gov/air_traffic/publications/atpubs/notam_html/chap1_section_2.
html.",
              "type": "array",
              "items": {
                "type": "string",
                "enum": [
                  "AD", "APRON", "AIRSPACE", "CHART", "COM", "IAP", "NAV",
"OBST", "ODP",
                  "ROUTE", "RWY", "SECURITY", "SID", "SPECIAL", "STAR",
"SVC", "TWY", "VFP",
                  "CONSTRUCTION", "LTA"
                ]
              },
              "default": ["AIRSPACE"]
            }
          }
        }
      ]
    },
    {
      "op": "t_intersects",
      "args": [
        {
          "interval": [ {"property": "valid_time_begin"}, {"property":
"valid_time_end"} ]
```

```
          },
          {
            "interval": [
              {
                "$parameter": {
                  "begin": {
                    "title": "Begin of time interval",
                    "description": "UTC timestamp for the begin of time
interval written according to RFC 3339. Example: '2022-08-18T15:05:00Z'. Only
NOTAMs with temporal validity in the time interval are selected.",
                    "type": "string",
                    "format": "date-time",
                    "default": "2022-08-18T15:05:00Z"
                  }
                }
              },
              {
                "$parameter": {
                  "end": {
                    "title": "End of time interval",
                    "description": "UTC timestamp for the end of time interval
written according to RFC 3339. Example: '2022-08-18T16:30:00Z'. Only NOTAMs
with temporal validity in the time interval are selected.",
                    "type": "string",
                    "format": "date-time",
                    "default": "2022-08-18T16:30:00Z"
                  }
                }
              }
            ]
          }
        ]
      }
    ]
  },
  "limit": 10
}
```

Example 4 — Stored Query: All features of an airport: This query selects all features of an airport.

```
{
    "id": "features-yb-airport",
    "title": "All features of an airport",
    "queries": [
        { "collections": ["airportheliport"      ] },
        { "collections": ["apron"                ] },
        { "collections": ["apronelement"         ] },
        { "collections": ["guidanceline"         ] },
        { "collections": ["runway"               ] },
        { "collections": ["runwayblastpad"       ] },
        { "collections": ["runwaycentrelinepoint"] },
        { "collections": ["runwayelement"        ] },
        { "collections": ["runwaymarking"        ] },
        { "collections": ["surveycontrolpoint"   ] },
        { "collections": ["taxiholdingposition"  ] },
        { "collections": ["taxiway"              ] },
        { "collections": ["taxiwayelement"       ] },
        { "collections": ["taxiwaymarking"       ] },
        { "collections": ["verticalstructure"    ] }
    ],
    "filter": {
        "op": "=",
```

```
        "args": [
            {"property": "airport"},
            {
                "$parameter": {
                    "airport": {
                        "title": "Airport",
                        "description": "The 3-letter IATA airport codes of
available airports.",
                        "type": "string",
                        "enum": ["JFK", "EWR", "LGA", "BOS", "PIT", "PHL",
"DCA", "BWI", "IAD"],
                        "default": "JFK"
                    }
                }
            }
        ]
    },
    "limit": 10000
}
```

8.4. HTML Support

Support for HTML in the OGC API – Features Standard is recommended. A server that supports HTML will support browsing the data with a web browser and will enable search engines to crawl and index the dataset.

For these reasons, support for both the HTML and the JSON resources were implemented during Testbed 18. As a result, the site itself already provides a Business Client.

The HTML representation of the Stored Queries resource included an HTML form for each stored query. The form includes:

- the descriptive elements (title and descriptions);
- a drop-down list for the desired response format for the feature collection; and
- For each parameter:
  - the descriptive elements (title, description and data type); and
  - a field, drop-down list or multi-selection list depending on the schema of the parameter.

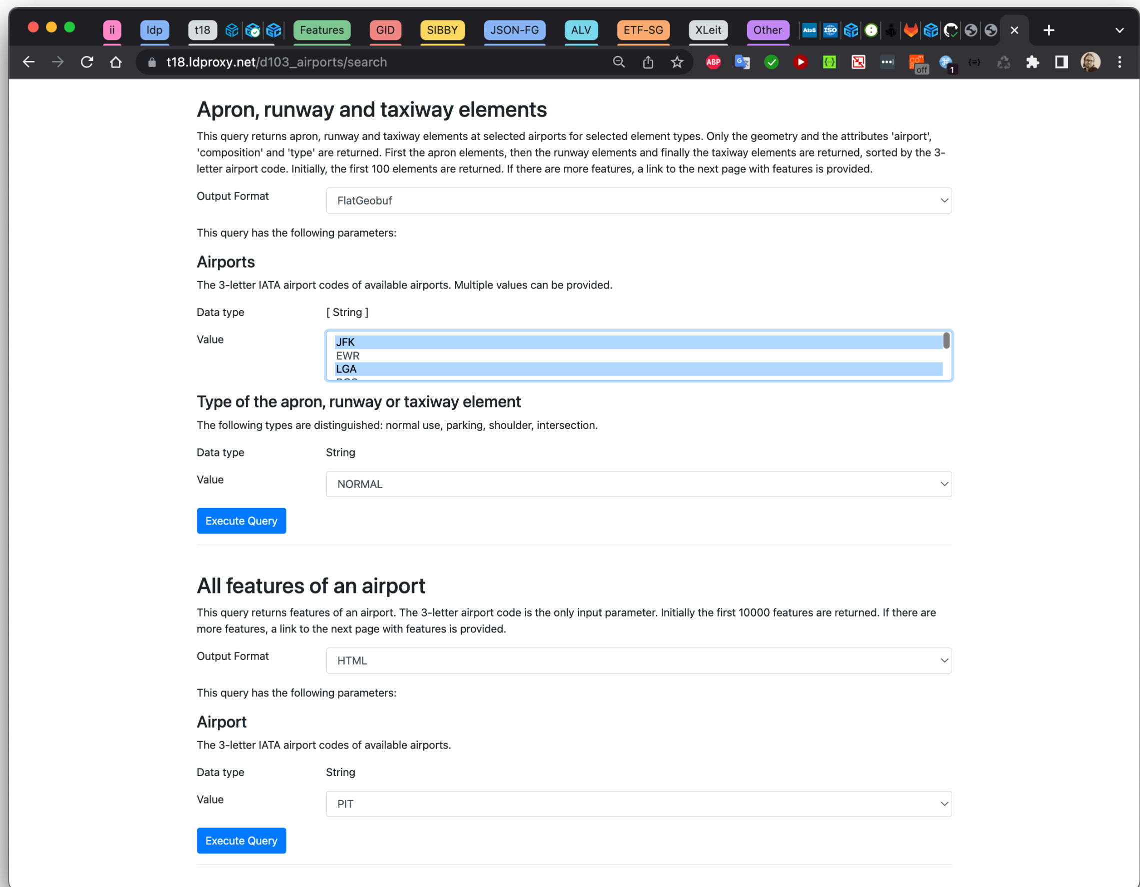


Figure 25 — Executing a Stored Query from the Web Browser

If "HTML" is selected as the response format, the features in the response are presented in the browser. This is the response of the second stored query in the screenshot above, which selects all Pittsburgh airport features.

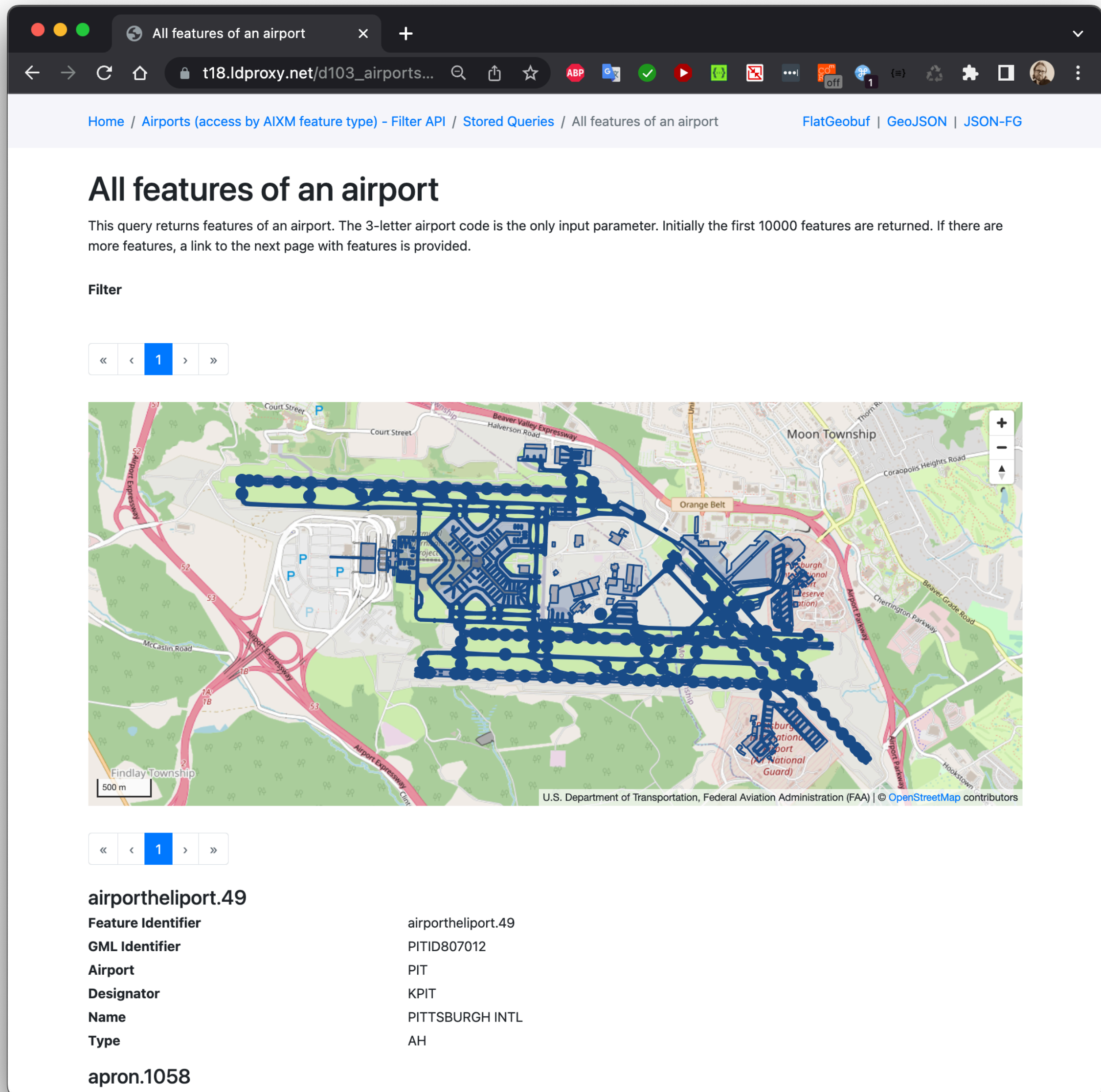


Figure 26 — Response from a Stored Query in the Web Browser (all Pittsburgh airport features)

8.5. Challenges and Lessons Learned

The implementation helped to improve the specification in Testbed-18 Filtering Service and Rule Set Engineering Report (D002) as well as in the Common Query Language (CQL2) candidate standard.

No additional issues were identified in the TIEs.

See also sections 6.2 and 6.3 of the Testbed-18 Filtering Service and Rule Set Engineering Report (D002).

9

BUSINESS USER CLIENT (D104)

9 BUSINESS USER CLIENT (D104)

The Business User Client, identified as deliverable 104 or D104, is a component built to demonstrate a client that executes Technical Interoperability Experiments (TIEs) with the filter services D102/D103 and the data services D100/101. The component was demonstrated by Concepts Beyond LLC.

9.1. Internal Architecture

The Developer Client and the Business Client were built as part of a single web application that offers specific functionalities for each one of the demonstrated Clients. The web application was built using ASP.NET web development technology with the jQuery front-end component written in JavaScript programming language, and Model-View-Controller (MVC) back-end approach written in C#. The final application will be mounted on an Amazon Web Service (AWS) cloud server. The application is composed of the following three components that are also illustrated in Figure 27.

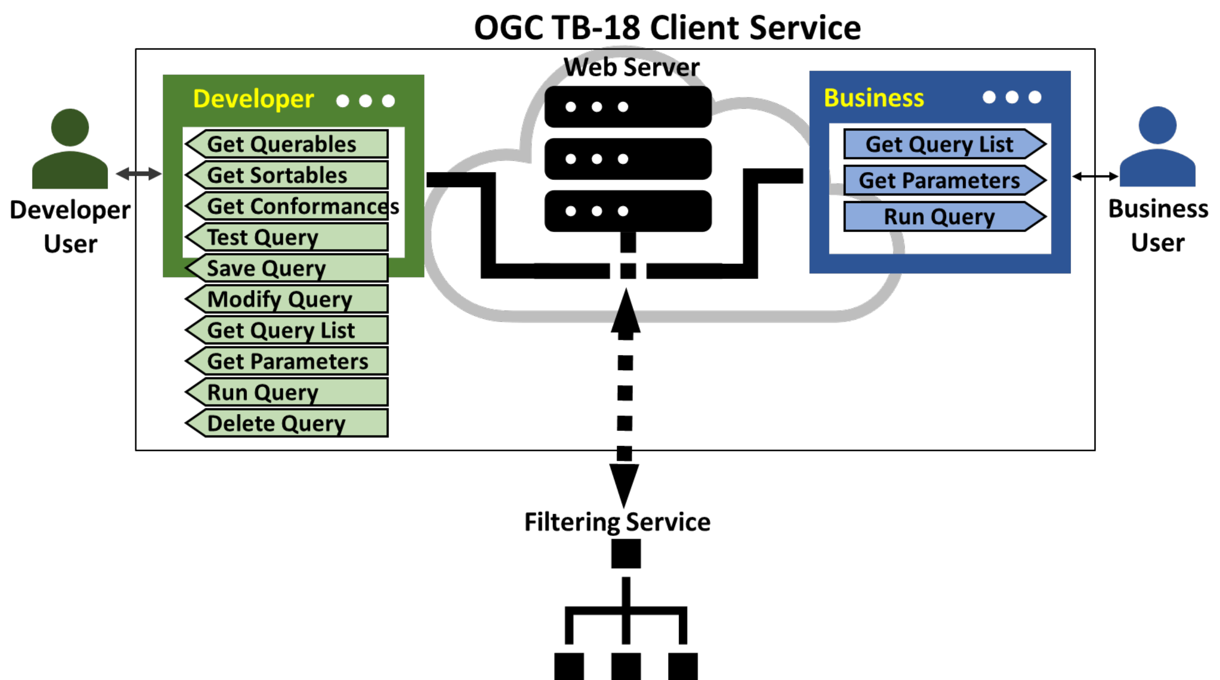


Figure 27 — Developer Client Component Breakdown

The **Web Server** is the module that acts as the back end for all user requests and is a communication hub between the clients and the SWIM Filtering Service (D102 and D103). This web server was built to interface the filtering services and fetch and relay data to and from the web application whenever a functionality of either or both clients require it. The following are

the functionalities that the web server provides to the business and developer clients (Refer to TIE Summary Table for details of the API call for each function).

- Get Queryables of a Collection
- Get Sortables of a Collection
- Get Conformances of the Collection of Collections
- Test Query
- Save Query
- Modify Query
- Get List of Stored Queries
- Get Parameters of a Query
- Run Query
- Delete Query

The **Developer Client** enables more advanced users to build complex queries using CQL2 constructs with access to the available sortables, queryables, and conformances for each data collection. As can be seen on the left side of the figure, the Developer client has access to all of the services provided by the Web Server that are being exchanged with the Filtering Service. Each service is achieved via a REST Web API call to the SWIM Filtering service in the associated format described in D102 and D103.

The **Business Client** enables end users of the SWIM data to input the parameters of each query (previously saved by the Developer user), run the query, and observe the results. The business client will be able to see the geographic data on a map, as described in D104.

The Business Client is designed to run predefined queries available by Filtering Services (D102 and D103). Currently the Business Client is using D103 Airport and NOTAM API services. These services provide a tabular list of all available queries to the user. Currently the query parameters are not shown and are to be added in the future.

Figure 28 shows a snapshot of the Business Client with NOTAM queries.

OGCTestbed18 Developer Client Business Client

NOTAMs Query List

ID	Title	Description	Parameters	Actions
test	NOTAMS along the route IAD to JFK	NOTAMs filtered by keyword affecting the route from IAD to JFK with a 50km buffer in a time interval.		Run
iad-jfk	NOTAMS along the route IAD to JFK	NOTAMs filtered by keyword affecting the route from IAD to JFK with a 50km buffer in a time interval.		Run
test7	DL 5517, 2022-08-18: NOTAMS along the planned flight path	NOTAMs related to airspaces along the planned flight path of DL 5517 from IAD to JFK with a 50km buffer on 2022-08-18.		Run
iad-jfk-dl5517-220818	DL 5517, 2022-08-18: NOTAMS along the planned flight path	NOTAMs related to airspaces along the planned flight path of DL 5517 from IAD to JFK with a 50km buffer on 2022-08-18.		Run
generic-notam-query	filter NOTAMS	NOTAMs filtered by keyword, by keyword and by time interval.		Run

Figure 28 — Business Client Query List

After running each query, the result is presented to the user both textually and graphically. Figure 29 shows an example of the result for NOTAMs related to airspaces along the planned flight path of DL 5517 from IAD to JFK with a 50km buffer on 2022-08-18.

Run

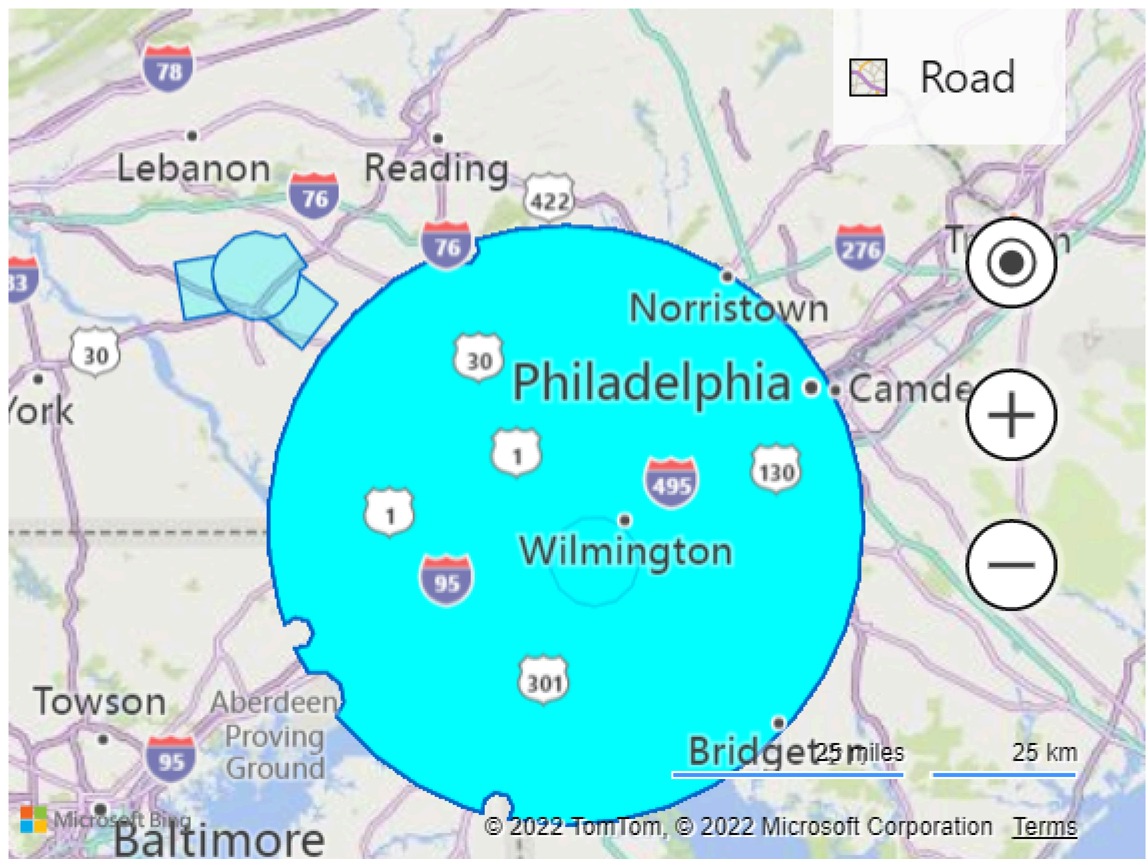


Figure 29 — Business Client Showing Airspaces

Figure 30 shows an example of running an airport collection query returning apron, runway, and taxiway elements of a specific type at one or more airports. In this query the airport parameter was set to JFK.

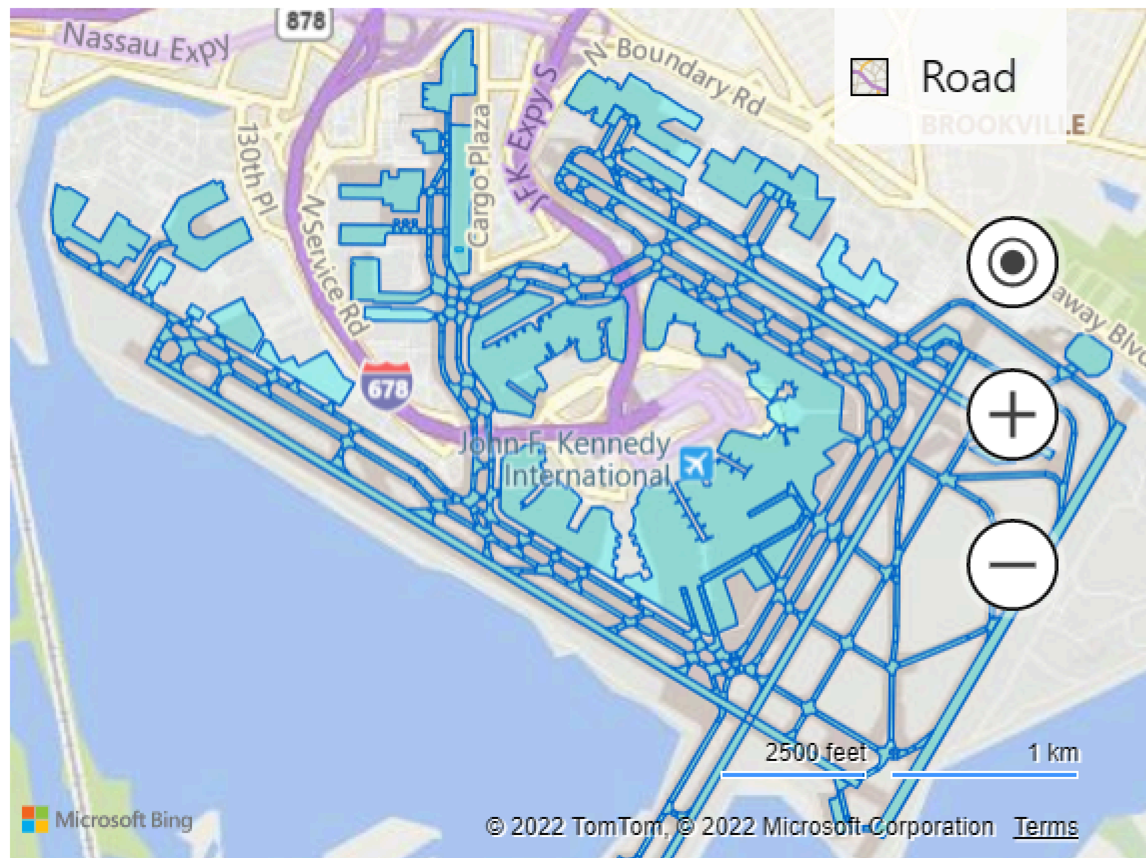


Figure 30 — Business Client Showing Airports

In Figure 31, the same query was run, with the "type" parameter set to intersections. This query returns intersection geometries (blue diamonds in the picture) at airports, in this case DCA airport.

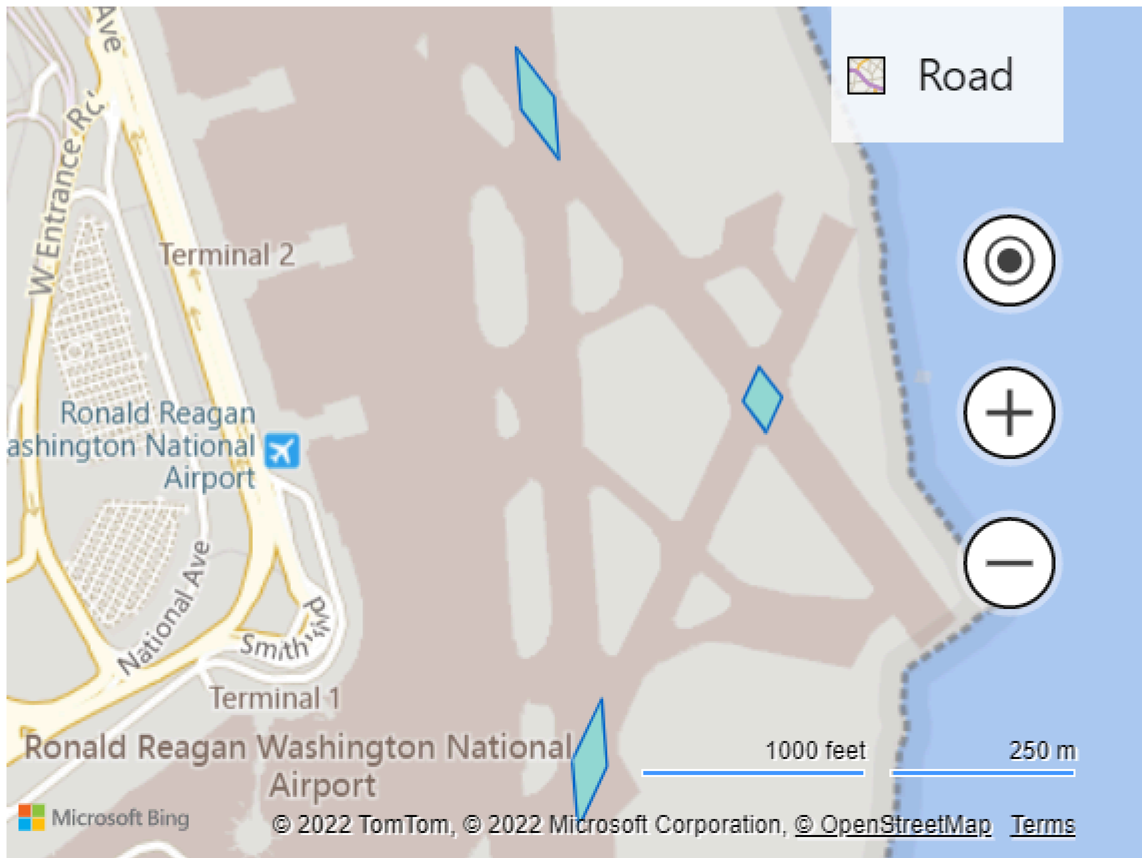


Figure 31 — Business Client Showing Intersections

Figure 32 shows a NOTAM query result that is identifying NOTAMs published for APRONs at multiple airports in eastern US. The NOTAM Id is being displayed on the map.

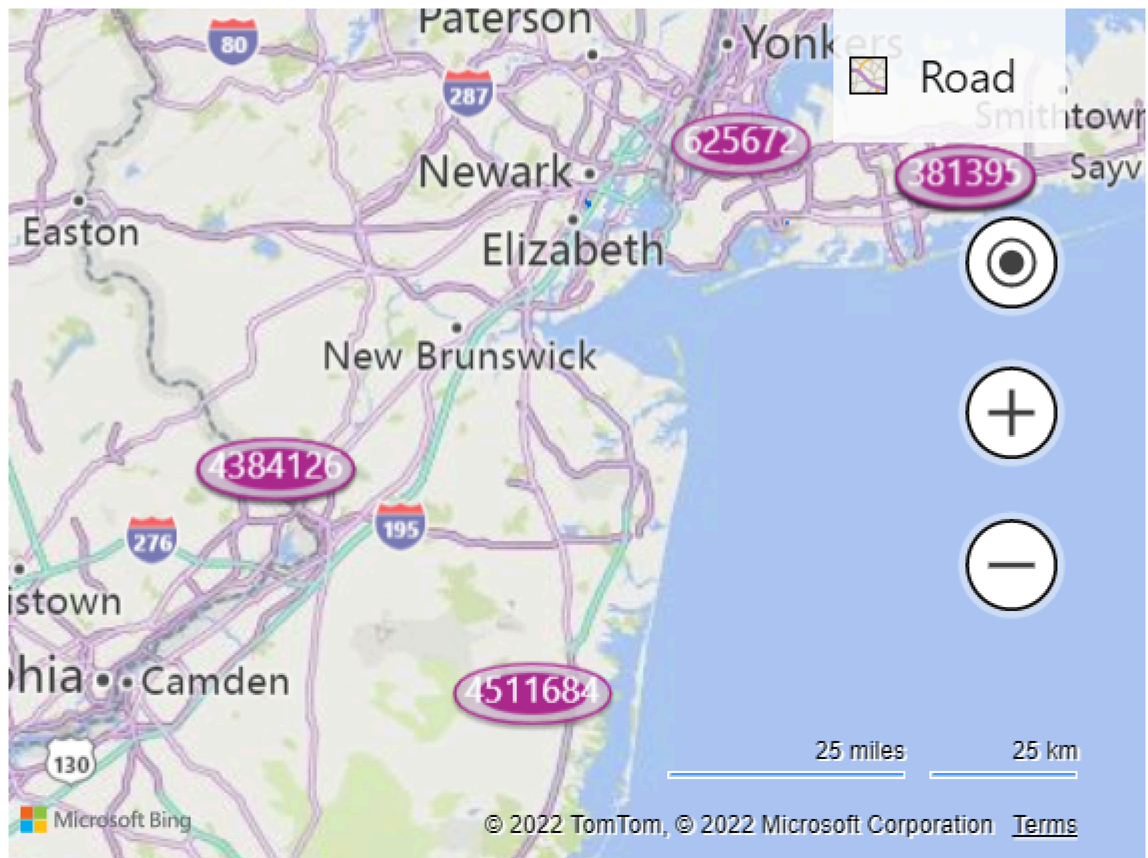


Figure 32 — Business Client Showing NOTAMs

9.2. Recommendations and Future Work

- **Explore Graphical Interfaces**: Creating a graphical interface for the user to be able to understand the result of the stored queries was a significant step that would require more investigation on how to represent data properties to the user that help the user in conceiving the impact of parameter selection on the result of the query.
- **Security**: Providing access and authorization to the data sets, properties, and even parameters of the queries will allow more control and security measures over the sensitive data.

10

DEVELOPER CLIENT (D105)

10 DEVELOPER CLIENT (D105)

The Developer Client, identified as deliverable 105 or D105, was a component built to demonstrate a client application that supports the customer in defining filter statements that can be expressed in a machine-readable way and exchanged with the filtering service. The component was demonstrated by Concepts Beyond LLC.

10.1. Internal Architecture

The Developer Client and the Business Client were built as part of a single web application that offers specific functions for each one of the demonstrated clients. The web application was built using ASP.NET web development technology with the jQuery front-end component written in JavaScript programming language and Model-View-Controller (MVC) back-end approach written in C#. The final application will be deployed on an Amazon Web Service (AWS) cloud server. The application is composed of the following three components that are also illustrated in Figure 33.

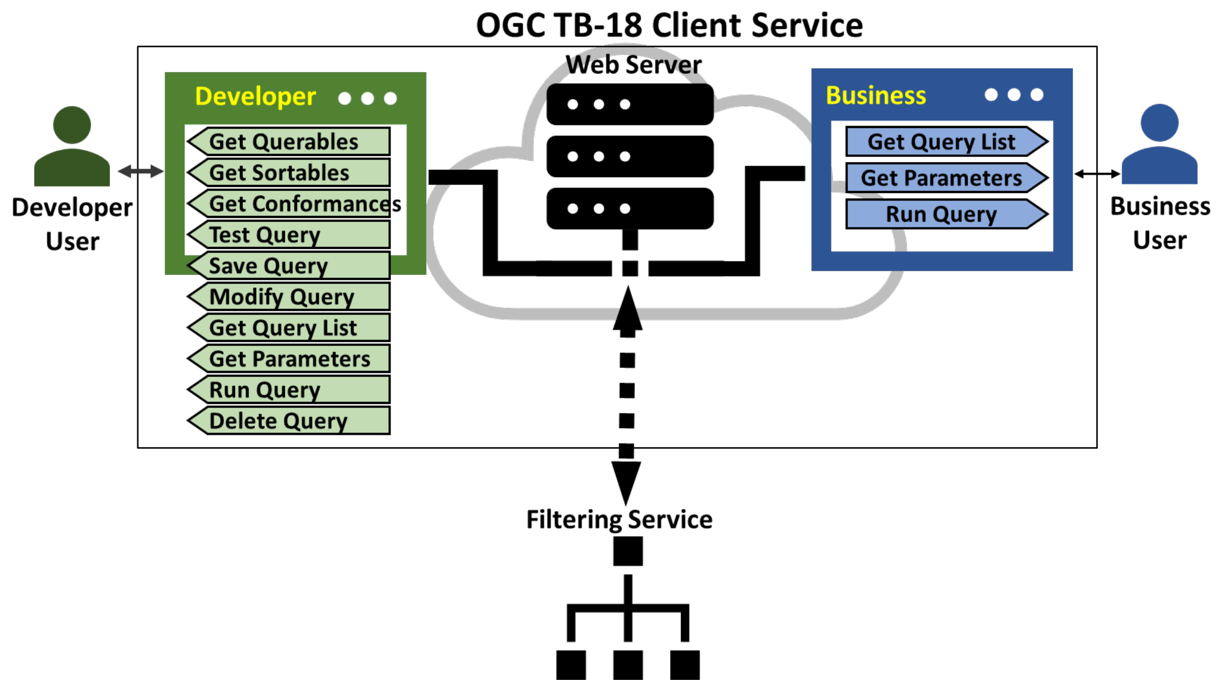


Figure 33 — Developer Client Component Breakdown

The **Web Server** is the module that acts as the backend for all user requests and is a communication hub between the clients and the SWIM Filtering Service (D102 and D103). This web server is built to interface the filtering services and fetch and relay data to and from the web application whenever a functionality of any of both clients require it. The following are

the functions that the web server provides to the business and developer clients (Refer to TIE Summary Table for details of the API call for each function).

- Get Queryables of a Collection
- Get Sortables of a Collection
- Get Conformances of the Collection of Collections
- Test Query
- Save Query
- Modify Query
- Get List of Stored Queries
- Get Parameters of a Query
- Run Query
- Delete Query

The **Business Client** enables end users of SWIM data to input the parameters for each query (previously saved by the Developer user), run the query, and observe the results. The business client will be able to see the geographic data on a map, as described in D104.

The **Developer Client** enables more advanced users to build complex queries using CQL2 constructs with access to the available sortables, queryables, and conformances for each data collection. As can be seen on the left side of the figure, the Developer client has access to all the services provided by the Web Server that are being exchanged with the Filtering Service. Each service is achieved via a REST Web API call to the SWIM Filtering service in the associated format described in D102 and D103.

The Developer Client allows the user to create, test, and save queries that can then be used by the business user to retrieve operational data. The Client also allows users to edit, delete, and run previously saved queries. To this end, the Developer Client is provided with the queryables of each data collection, determining what properties of the SWIM data must be queries or filtered. Using the conformance list, the developer user can determine what logical, mathematical, or complex operation can be carried out on the queryables of that data collection. Using sortables, the client user can sequence the data returned for multiple queryables and present them is an order, useful to the Business user.

10.1.1. List of Stored Queries

The developer client retrieves queries stored on each Filtering Service and displays them on a table on the top of the developer client page. This table lists all the available/stored queries with their Id, title, description, list of parameters, and actions that can be taken on each query. The parameters column lists the parameters needed to run the query. Figure 34 shows a snapshot

of the Developer Client displaying the query table from the NOTAMs provided by the D103 Filtering Service:

NOTAM Collections

ID	Title	Description	Parameters	Action
test	NOTAMS along the route IAD to JFK	NOTAMs filtered by keyword affecting the route from IAD to JFK with a 50km buffer in a time interval.	ID: Title end: End of time interval notamKeywords: NOTAM keywords begin: Begin of time interval	Run Edit Delete
generic-notam-query2	filter NOTAMS	NOTAMs filtered by location, by keyword and by time interval.	ID: Title notamKeywords: NOTAM keywords geometry: NOTAM geometry end: End of time interval begin: Begin of time interval	Run Edit Delete
iad-jfk	NOTAMS along the route IAD to JFK	NOTAMs filtered by keyword affecting the route from IAD to JFK with a 50km buffer in a time interval.	ID: Title keywords: Keywords end: End of time interval begin: Begin of time interval	Run Edit Delete
iad-jfk-dl5517-220818	DL 5517, 2022-08-18: NOTAMS along the planned flight path	NOTAMs related to airspaces along the planned flight path of DL 5517 from IAD to JFK with a 50km buffer on 2022-08-18.	ID: Title	Run Edit Delete

Figure 34 — Developer Client Queries Table

10.1.2. Running and Editing a Stored Query

When the “Run” button is pressed, a popup window opens and provides input entries for the parameters of the query. If the parameters have default values set by the creator of the query, the default values will automatically be populated in a drop-down selection box. These selection boxes appear for parameters with finite sets of values. Figure 35 shows a query run popup windows with the parameters to be input.

OGCTestbed18 Developer Client Business Client

ID	Title	Description	Parameters	Action
test	NOTAMS along the route IAD to J			Run Edit Delete
iad-jfk	NOTAMS along the route IAD to J			Run Edit Delete
test7	DL 5517, 2022-08-18: NOTAMS along the flight path			Run Edit Delete
iad-jfk-dl5517-220818	DL 5517, 2022-08-18: NOTAMS along the flight path	flight path of DL 5517 from IAD to JFK with a 50km buffer on 2022-08-18.		Run Edit Delete

- OGCTestbed18 - Google Chrome

localhost:5129/DeveloperClient/ParameterChoice

OGCTestbed18 Developer Client Business Client

NOTAM geometry

End of time interval 8/18/2022 4:30:00 PM

NOTAM keywords AD

Begin of time interval 8/18/2022 3:05:00 PM

Run

Your Chosen Parameters are :

© 2022 - OGCTestbed18 - Privacy

Figure 35 — Developer Client Collection Table

When the user presses the “Edit” Button for any query, the associated query id and query JSON text are populated in the textbox below the table, as shown in Figure 36.

			Parameters ID	Parameters Title	
generic-notam-query	Filter NOTAMs	NOTAMs filtered by keyword, location and time interval.	begin	Begin of time interval	Run Edit Delete
			keywords	Keywords	
			end	End of time interval	
			location	Location	

QueryName

generic-notam-query

JsonQuery

```
{
  "id": "generic-notam-query",
  "title": "Filter NOTAMs",
  "description": "NOTAMs filtered by keyword, location and time interval.",
  "collections": [
    "notam"
  ],
  "filter": {
    "op": "and",
    "args": [
      {
        "op": "s_intersects",
```

Figure 36 — Query Editing in the Developer Client

10.1.3. Creating A Query

The same textbox used for editing queries can also be used to create and test a new query. In order to first build and test the query, the user must write the JSON query text inside the

“JsonQuery” textbox. The “TestQuery” button allows the user to run the query before saving it. The “SaveQuery” button sends the query to the Filtering Service in order to store it there.

OGCTestbed18 Developer Client Business Client

AirportCollections

ID	Title	Description	Parameters	Action
nyc-concrete-surface-areas	Concrete apron, runway and taxiway elements at NYC area airports		ID / Title	Run Edit Delete
surface-areas	Apron, runway and taxiway elements of a specific type at one or more airports.		ID / Title type / Type of the apron, runway or taxiway element airports / Airports	Run Edit Delete

QueryName

JsonQuery

SaveQuery TestQuery

Figure 37 — Developer Client Query Creation

To assist the user in the query creation process, the developer client displays on a table the collections found on the filtering services it is connected to. Users can access the sortables and queryables for each collection by clicking on buttons located next to each collection description. Figure 38 shows this list from D103 SWIM Filtering Service for Airports.

Title	Description	Queryables	Sortable
Airport/Heliport	A defined area on land or water (including any buildings, installations and equipment) intended to be used either wholly or in part for the arrival, departure and surface movement of aircraft/helicopters.	Get Queryables	Get Sortable
Apron	A defined area, on a land aerodrome/heliport, intended to accommodate aircraft/helicopters for purposes of loading and unloading passengers, mail or cargo, and for fuelling, parking or maintenance.	Get Queryables	Get Sortable
Apron Element	Parts of a defined apron area. Apron Elements may have functional characteristics defined in the ApronElement type. Apron Elements may have jetway, fuel, towing, docking and groundPower services.	Get Queryables	Get Sortable
Guidance Line	A line used to guide aircraft on and between airport movement areas.	Get Queryables	Get Sortable
Runway	A defined rectangular area on a land aerodrome/heliport prepared for the landing and take-off of aircraft. Note: this includes the concept of Final Approach and Take-Off Area (FATO) for helicopters.	Get Queryables	Get Sortable
Runway Blast Pad	Specially prepared surface placed adjacent to the end of a runway to eliminate the erosive affect of the high wind forces produced by airplanes at the beginning of their takeoff rolls.	Get Queryables	Get Sortable
Runway Centreline Point	An operationally significant position on the centre line of a runway direction. A typical example is the runway threshold.	Get Queryables	Get Sortable
Runway Element	Runway element may consist of one ore more polygons not defined as other portions of the runway class.	Get Queryables	Get Sortable

Figure 38 — Developer Client Collections Table

Clicking on each "Queryables" or "Sortables" button for each collection will display a table of the queryables or sortables with names and types identified. A button at the end of the collections table opens a view of the conformance declaration for each collection. (see Figure 39 for queryables and Figure 40 for sortables).

OGCTestbed18 Developer Client Business Client

Queryables

Title: Airport/Heliport

Title	Type
GML Identifier	string
Airport	string
Designator	string
Name	string
Type	string
$ref	https://geojson.org/schema/Point.json

Back to Home

Figure 39 — Developer Client Displaying Queryables

Sortable

Title: Airport/Heliport

Title	Type
GML Identifier	string
Designator	string
Name	string
Type	string

Figure 40 — Developer Client Displaying Sortables

10.2. Challenges and Lessons Learned

- **Metadata Usage**: The metadata provided by the Filtering Services seemed mature and flexible enough to be expanded for each collection. The queryables allowed for flexible and expandable definitions of metadata for each data collection
- **Queryables not Represented as Properties**: Making sure that the name and type of the queryables defined for a data collection exactly represent the associated property of the data is key to supporting queryables that are not directly represented as resource properties in the content schema of the resource.
- **Benefits of a Filtering Service**: Knowing ahead the content schema enables a developer client to write complex queries relating various data types and collections together, providing needed operational data to the business client.
- **Best Practices for Filtering Services**: Based on the experience gained in Testbed 18, the filtering service should provide testing and trial of the queries, storing queries, and definition of the parameters of the query.

  In the future, the filtering service endpoint could potentially provide authorization to each API and access restrictions to certain types of data or queryables.
- **Best Practices for Developer Clients**:
  - Having an interactive and graphical query building web client is key to helping the developer client to create complex queries rapidly. This includes providing a tree-based structure to the query in the textbox, with queryables, sortables, and conformance selection dropdown boxes to help build valid and complex queries. Developer clients should provide interactive access to the queryables, sortables, and conformances when writing a query to allow the user to rapidly create advanced queries.
  - Graphically presenting the results of a query could help the user quickly validate and accelerate the query creation process. The client interface could include graphical display of query results. As an example, geographic data could be displayed on a map.

- Developer clients could also provide smart data property aggregation between two or more data collections where specific data properties can be matched (e.g., location of weather with the coordinates of the route of flight).

10.3. Recommendations and Future Work

- **Interactive Query Building Interface**: One of the shortcomings of current TB-18 SWIM data filtering clients is the lack of an interactive ability, helping the developer to build complex and effective queries using CQL2 language. A web based Interactive query builder providing graphical and pre-selectable queryables, sortables, and conformances that are fetched from the SWIM Filtering service is needed.

  Such an interface will help the Developer client use means such as dropdown boxes that are prepopulated with the specific queryables for each data collection that is selected to be queried, as well as offering prepopulated available conformances that can be applied to each expression, complemented with selection of the available sortable parameters for the data collection. This interface must convert the graphical query structure built by the developer user to the JSON query structure that can be saved and executed on the SWIM filtering service.

  This client could speed query building and validation by saving the user the time and hassle of looking into queryables, sortables, and conformance lists to understand which one must be used at which statement. This interface must also validate and provide hints on potential syntax, and logical errors during query building.

- **Smart Query Building Interface**: Another shortcoming is the lack of real-time hints on the impact of changes to the query structure or on the results of the query. Having graphical query results that are tied back to the query expression helps the user understand the impact of each queryable or conformance used in the query on the result of the query. The results of the query will be graphically displayed in real-time to the Developer User while they are building the query helping them adjust the query to achieve the desired results much faster.

- **Data Correlation from multiple SWIM Services**: A major milestone that is needed for SWIM is data fusion using multiple services. In TB-18 the queries built using CQL2 language are limited to each SWIM data service, and these queries are segregated from each other. For example, if a user wants to know the weather impacting their route of flight, they need to access at least two separate SWIM data services, 1- The Weather Data service, 2- the Flight Data service. However, writing such a query was not possible in TB-18.

  The filtering service must be able to support this data fusion, based on parameters in the data that are of similar type that support the same conformances. This is essential in enabling SWIM users, such as airlines and Air Navigation Service Providers (ANSPs) to fuse various SWIM data and use them effectively. This feature will improve the SWIM data usability and attract more consumers to this technology. Concepts such as Flight and Flow Information for a Collaborative Environment (FF-ICE) require such SWIM fusion services

as enablers of the technology, helping airlines and ANSPs interact with various types of live data published on a SWIM network.

11

TECHNOLOGY INTEGRATION EXPERIMENTS (TIES)

11 TECHNOLOGY INTEGRATION EXPERIMENTS (TIES)

The TB-18 Technology Integration Experiments (TIEs) focused on the usage of filters within the exchange of aviation data through APIs built using OGC API standards. This chapter summarizes all TIEs performed during this testbed task.

11.1. TIE Summary Table

The following table summarizes the TIEs performed during Testbed-18

Table 5 — Technology Integration Experiments Overview

SERVER\CLIENT	D102 FILTERING SERVICE 1	D103 FILTERING SERVICE 2	D104 BUSINESS USER CLIENT	D105 DEVELOPER CLIENT
D100 OGC API-Features Façade 1	Clause 11.2.1	Clause 11.2.2	N/A	N/A
D101 OGC API-Features Façade 2	Clause 11.2.3	N/A	N/A	N/A
OGC API-Features Façade 3 (Skymantics)	Clause 11.2.4	N/A	N/A	N/A
D102 Filtering Service 1	N/A	N/A	Clause 11.2.5	Clause 11.2.6
D103 Filtering Service 2	N/A	N/A	Clause 11.2.7	Clause 11.2.8

11.2. TIE Functional Tests

11.2.1. D102 Filtering Service 1 cascading D100 OGC API — Features Façade 1

The D102 filtering service was demonstrated cascading successfully to the interactive instruments (D100) *OGC API — Features* façade.

The corresponding collections for the filtering cascading service, supporting CQL2 expressions as value to the `filter=` parameter for the `/items` resources, are available from the following endpoints:

https://maps.gnosis.earth/ogcapi/collections/swim:d100_airports (by AIXM)

https://maps.gnosis.earth/ogcapi/collections/swim:d100_airports2 (by airport)

https://maps.gnosis.earth/ogcapi/collections/swim:d100_notam

https://maps.gnosis.earth/ogcapi/collections/swim:d100_airspace

Figure 41 — Map of LAX (from interactive instruments' D100 service) as it appears on collection page

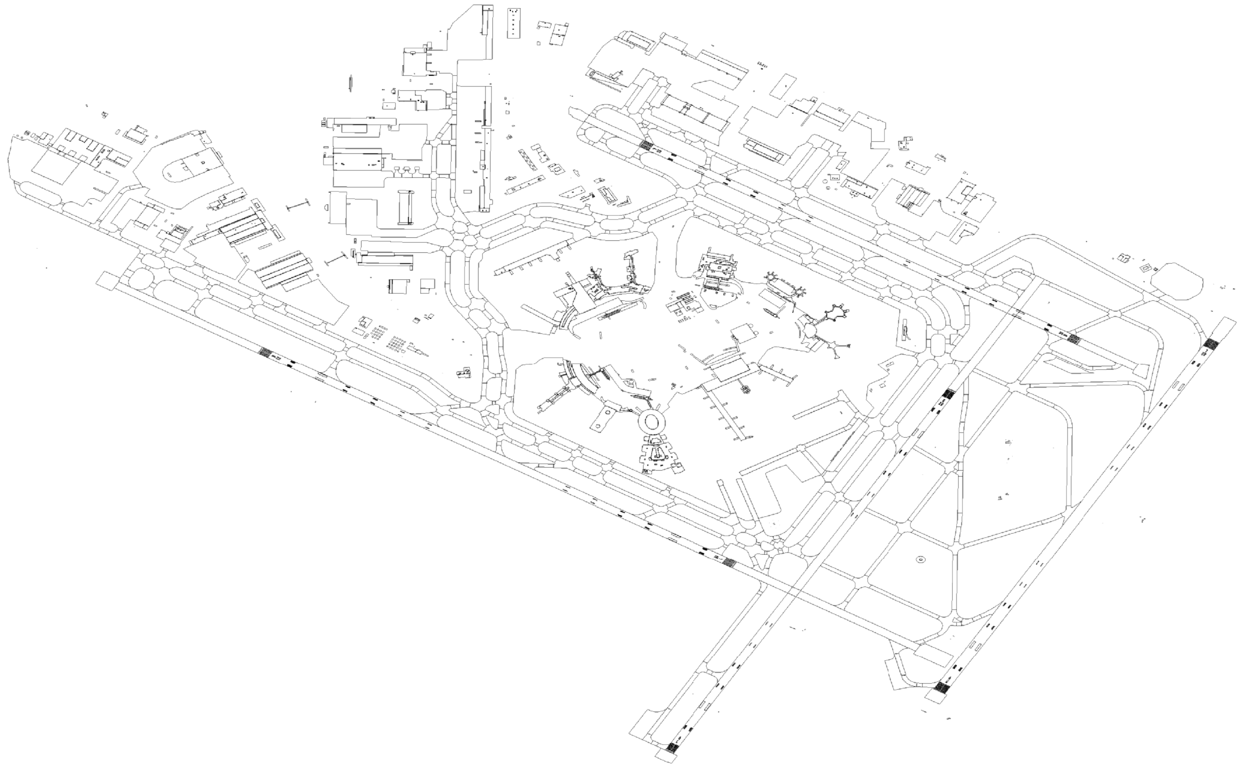


Figure 42 — Map of JFK airport from interactive instruments' D100 service

GNOSIS Map Server @ ×

https://maps.gnosis.earth/ogcapi/collections/swim:d100_airports2:EWR/items?filter=featureType=%27ApronElement%27%20and%20type=%27PARKING%27&properties=associatedApron,composition,featureType,gml_id,type

GNOSIS GNOSIS Map Server

Vector features

for EWR

*(Download **GeoJSON** representation)*

Back to data collection

Feature ID	Geometry	Min Lat	Min Lon	Max Lat	Max Lon	associatedApron	composition	featureType	gml_id	type
1470	geojson.io download	40.680626	-74.1863619	40.6827708	-74.1845579	▼ (object) { "href" : "https://t18.ldproxy.net/d100_airports2/collections/EWR/items/765", "title" : "autogeneratedID332556" }	CONC	ApronElement	EWRID332551	PARKING
1477	geojson.io download	40.6801342	-74.1883162	40.6814882	-74.1856544	▼ (object) { "href" : "https://t18.ldproxy.net/d100_airports2/collections/EWR/items/769", "title" : "autogeneratedID332563" }	CONC	ApronElement	EWRID332558	PARKING
1479	geojson.io download	40.6819289	-74.1874709	40.6833213	-74.1860049	▶ (object)	CONC	ApronElement	EWRID332565	PARKING
1526	geojson.io download	40.6802996	-74.184512	40.6805907	-74.1839846	▶ (object)	CONC	ApronElement	EWRID332572	PARKING
1973	geojson.io download	40.7036329	-74.1803938	40.7048899	-74.1772523	▶ (object)	CONC	ApronElement	EWRID332537	PARKING
1978	geojson.io download	40.6972961	-74.1587606	40.6977001	-74.158377	▶ (object)	CONC	ApronElement	EWRID332523	PARKING
1985	geojson.io download	40.6836591	-74.1801673	40.6860674	-74.1760574	▶ (object)	CONC	ApronElement	EWRID332502	PARKING
2000	geojson.io download	40.683553	-74.1849479	40.6849677	-74.1810931	▶ (object)	CONC	ApronElement	EWRID332509	PARKING
2002	geojson.io download	40.7033519	-74.166357	40.7047538	-74.1653421	▶ (object)	CONC	ApronElement	EWRID332516	PARKING
2003	geojson.io download	40.7033871	-74.1677676	40.7051509	-74.1668201	▶ (object)	CONC	ApronElement	EWRID332530	PARKING

(next...)

Figure 43 — Paginated filtered features returned for a query on the EWR collection

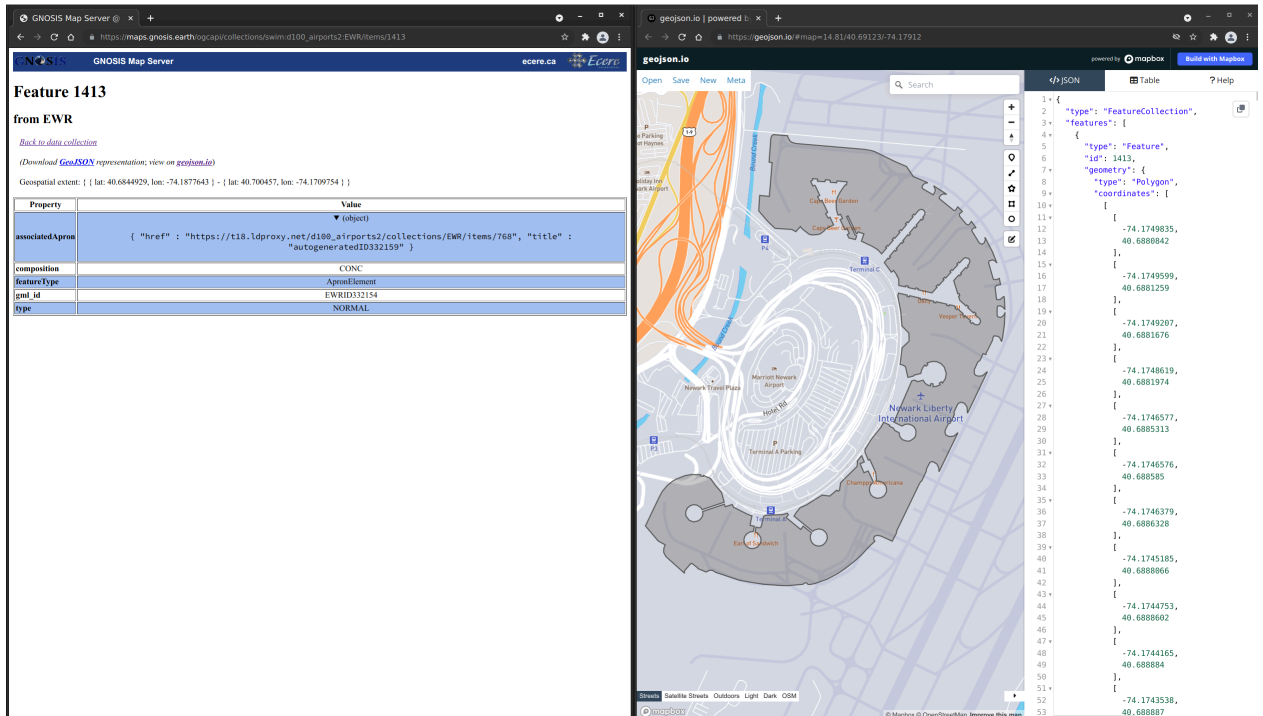


Figure 44 — Feature #1413 from the EWR airport collection from interactive instruments' D100 service

11.2.2. D103 Filtering Service 2 for D100 OGC API — Features Façade 1

For the TIE, the Filter APIs for the SWIM data façades for the Airports, Airspaces, and NOTAMs datasets were set up.

Like the corresponding D100 Data APIs, the D103 Filter APIs implement the OGC API building blocks for features, but there are the following differences.

- Scope of the D100 Data APIs:
  - the D100 Data APIs intentionally did not support rich filtering capabilities. They were limited to the building blocks from OGC API Features Core, which support limited filtering capabilities; however,
  - as Data APIs, they also provide the data as tiled vector data (aka vector tiles) together with associated styles to support presentation of the data in maps.
- Scope of the D103 Filter APIs:
  - the D103 Filter APIs are focused on filtering and, therefore, do not support access to vector tiles or styles; and
  - for feature data, they not only support OGC API Features Core, but also CRS support, filtering with CQL2-Text and CQL2-JSON, the new advanced filtering capabilities

specified in the Testbed-18 Filtering Service and Rule Set Engineering Report, and several proposed extensions for sorting and property selection.

As such, the D103 Filter APIs provide advanced filtering capabilities for the feature data published using the D100 Data APIs.

Several sample filtering rules (with and without parameters, for a single feature collection or multiple feature collections, with and without property selection and sorting, with and without paging of the response, etc.) were created. A representative set of filtering rules are documented in the description of component D103.

The filtering rules were executed in the Filter APIs, using both from the command line (cURL) and in the web browser using the HTML forms. For parameterized rules, several combinations of values were tested. All responses were checked to contain the correct results.

During the testing a few bugs were identified in the implementation of D103, which were corrected. Eventually all TIEs were executed successfully.

NOTEIn practice, supporting filtering capabilities already in the Data API may often be required. Otherwise, it may be necessary to transfer large amounts of data from the Data API to the Filter API, where a significant amount of that data does not match the filtering rule and will not be delivered to the client that submitted the filtering request. For large datasets (e.g., the D100 NOTAM API provides access to more than 5 million NOTAMs), this will not be practical for many filters. The D103 Filter APIs for the corresponding D100 Data APIs were, therefore, optimized for the specific datasets. The component did not support filtering on other OGC Web APIs implementing OGC API Features, for example, the D101 Data APIs.

11.2.3. D102 Filtering Service 1 cascading D101 OGC API — Features Façade 2

The D102 filtering service was demonstrated cascading successfully to the GMU (D101) *OGC API — Features* façade.

The corresponding collections for the filtering cascading service, supporting CQL2 expressions as value to the `filter=` parameter for the `/items` resources, are available from the following endpoint.

https://maps.gnosis.earth/ogcapi/collections/swim:gmu

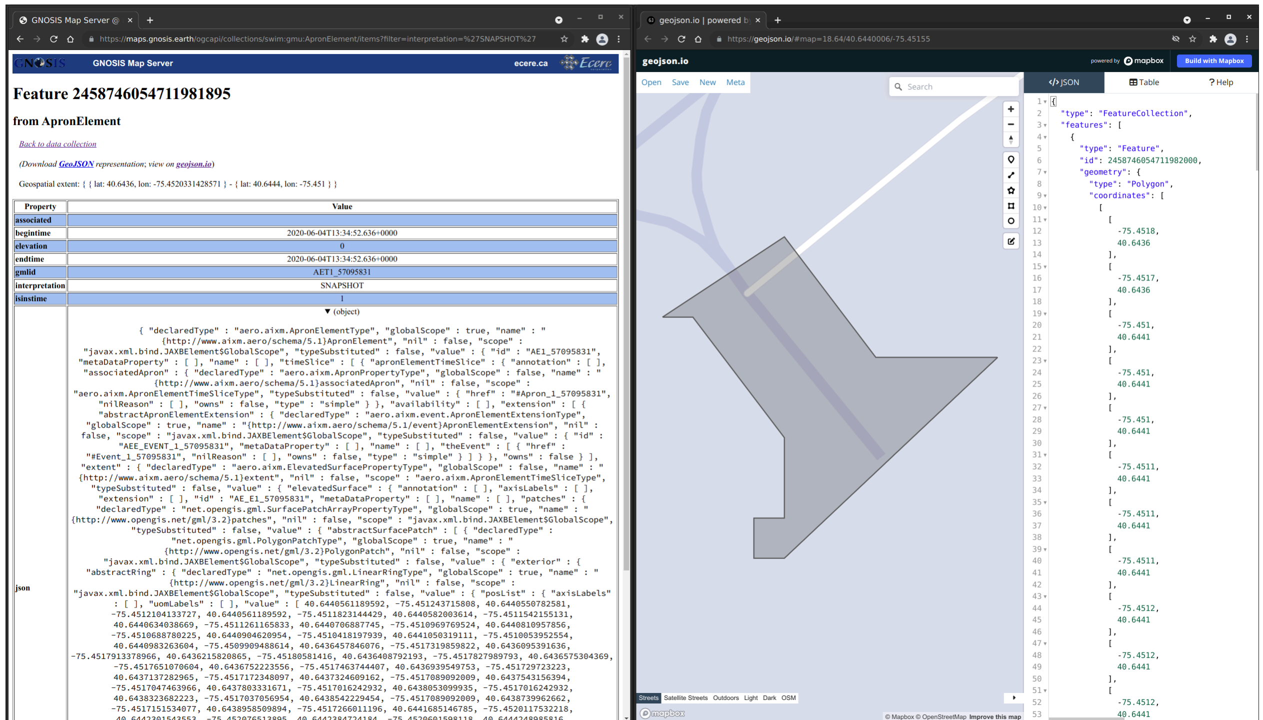


Figure 45 — Ecere's D102 Filtering Service cascading features from George Mason University's D101 service

11.2.4. D102 Filtering Service 1 cascading OGC API — Features Façade 3 (Skymantics)

The D102 filtering service was demonstrated cascading successfully to the Skymantics *OGC API — Features* endpoint.

The corresponding collections for the filtering cascading service, supporting CQL2 expressions as value to the `filter=` parameter for the `/items` resources, are available from the following endpoint.

https://maps.gnosis.earth/ogcapi/collections/swim:faa

GNOSIS Map Server

Vector features

for faa_flight_plans

(Download GeoJSON representation)

Back to data collection

Feature ID	Geometry	Min Lat	Min Lon	Max Lat	Max Lon	aircraftdescription	arrival	arrivaltime	centre	departure	departuretime	feattype	flightidentification
78705142540873208	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
594520870401030297	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
825978663549457847	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
1239845461674562965	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
2185898401869682379	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
2467752013237059537	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
2495525167867485406	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)
3174224304328308594	geojson.io download					▶ (object)	▶ (object)		▶ (object)	▶ (object)			▶ (object)

Feature ID	flightplan	flightstatus	flighttype	gufi	lastupdated	operator	source	system	airspeed	altitude	coordination	enroute	originator
78705142540873208	KN81334200	PROPOSED	NON_SCHEDULED	3f252df8-fe0f-450a-954f-fa30ea9d175a	2023-01-16T22:35:29.395000	▶ (object)	FH	SLC		▶ (object)	▶ (object)		▶ (object
594520870401030297	KN03791300	PROPOSED	NON_SCHEDULED	2bcf3899-c3a3-4077-9bb1-bc69e344fc50	2023-01-17T01:03:06.951000	▶ (object)	FH	SLC		▶ (object)	▶ (object)	▶ (object)	▶ (object
825978663549457847	KR42362600	PROPOSED	NON_SCHEDULED	a0ead5b7-6f4b-448f-bdb9-6aa1dc047f03	2023-01-13T20:03:15.058000	▶ (object)	AH	SLC		▶ (object)	▶ (object)	▶ (object)	
1239845461674562965	KN81419200	PROPOSED	NON_SCHEDULED	1f7e1595-753c-45d7-b93b-3bc7f49365dd	2023-01-16T22:36:54.219000	▶ (object)	FH	SLC		▶ (object)	▶ (object)		▶ (object
2185898401869682379	KN82854100	PROPOSED	NON_SCHEDULED	1e8956cb-9d2e-46e8-ac3a-4bdadfcc52cf	2023-01-15T23:00:49.686000		FH	SLC		▶ (object)	▶ (object)		▶ (object
2467752013237059537	KN06222300	PROPOSED	NON_SCHEDULED	5ebfffd1-7775-4897-bf8f-016fd4bdcdee	2023-01-17T01:49:00.429000	▶ (object)	AH	SLC		▶ (object)	▶ (object)	▶ (object)	▶ (object
2495525167867485406	KN83361200	PROPOSED	NON_SCHEDULED	56c9dcde-9a0b-4a4f-abcb-740a65d3ad75	2023-01-16T23:44:00.438000	▶ (object)	AH	SLC		▶ (object)	▶ (object)	▶ (object)	▶ (object
3174224304328308594	KW57472100	PROPOSED	NON_SCHEDULED	e0e67772-6fcd-4938-81af-d58703bcf915	2023-01-15T23:00:28.708000		AH	SLC		▶ (object)	▶ (object)		▶ (object

Figure 46 — Ecere's D102 Filtering Service cascading and filtering features from Skymantics service

11.2.5. D104 Business User Client accessing D102 Filtering Service 1

This TIE was still in development at the time of writing this report.

11.2.6. D105 Developer Client acessing D102 Filtering Service 1

This TIE was still in development at the time of writing this report.

11.2.7. D104 Business User Client accessing D103 Filtering Service 2

Table 6 — TIE Overview of the Business User Client With the D103 Filtering Service

#	FUNCTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
1	Get Query List	Once the Developer client page of a SWIM data is loading, the Developer Client front-end page makes a request to the Web Server. The Webserver makes an HTTP Get call at `{FilteringServiceURL}/search` on the SWIM Filtering Service	Receive query list in (`f=json`) format	Developer Client displays the list of stored queries in a table

#	FUNCTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
2	Get Parameters of a Query	Once the Developer client page of a SWIM data is loading, the Developer Client front-end page makes a request to the Web Server. The Webserver makes an HTTP Get call at `{FilteringServiceURL}/search/{queryId}/parameters` on the SWIM Filtering Service for each stored query	Receive parameter list in (`f=json`) format	Developer Client displays the list of parameters for each stored query in a table
3	Run Query	Once the request is received from Developer Client front-end page including the query Id, the Webserver makes an HTTP Post call at `{FilteringServiceURL}/search/{queryId}` on the SWIM Filtering Service	Receive query result in (`f=json, geojson`) format	Developer Client displays the results in simple text, or if the result is in Geo JSON format, on a map

11.2.8. D105 Developer Client accessing D103 Filtering Service 2

Table 7 — TIE Overview of the Developer Client With the D103 Filtering Service

#	FUNCTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
1	Get Queryables of a Collection	Once the request is received from Developer Client front-end page, the Webserver makes an HTTP Get call at `{FilteringServiceURL}/{CollectionId}/queryables` on the SWIM Filtering Service	Receive list of queryables in (`f=json`) format	Developer Client displays the list of queryables in a table
2	Get Sortables of a Collection	Once the request is received from Developer Client front-end page, the Webserver makes an HTTP Get call at `{FilteringServiceURL}/{CollectionId}/sortables` on the SWIM Filtering Service	Receive list of sortables in (`f=json`) format	Developer Client displays the list of sortables in a table
3	Get Conformances of the Collection of Collections	Once the request is received from Developer Client front-end page, the Webserver makes an HTTP Get call at `{FilteringServiceURL}/conformance` on the SWIM Filtering Service	Receive list of conformances in (`f=json`) format	Developer Client displays the list of conformances in a table
4	Test Query	Once the request is received from Developer Client front-end page including the query expression, the	Receive query result in (`f=json,`	Developer Client displays the results in simple text, or if the

#	FUNCTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
		Webserver makes an HTTP Post call at `{FilteringServiceURL}/search` on the SWIM Filtering Service, with the JSON query string included in the body of the request	`geojson`) format	result is in GeoJSON format, on a map
5	Save Query	Once the request is received from Developer Client front-end page including the query Id and expression, the Webserver makes an HTTP Put call at `{FilteringServiceURL}/search/{queryId}` on the SWIM Filtering Service, with the JSON query string included in the body of the request	HTML OK code when successful	Developer Client displays a successful or error notification
6	Modify Query	Once the “Edit” button is pressed on the Developer Client, the client makes an HTTP Get call at `{FilteringServiceURL}/search/{queryId}/definition` on the SWIM Filtering Service to receive the full query definition.	HTML OK code when successful	Developer Client then populates two text boxes on the client page with the query Id and definition, allowing the user to modify the query expression. Then the Save Query function can be used to save the query on the Filtering Service
7	Get Query List	Once the Developer client page of a SWIM data is loading, the Developer Client front-end page makes a request to the Web Server. The Webserver makes an HTTP Get call at `{FilteringServiceURL}/search` on the SWIM Filtering Service	Receive query list in (`f=json`) format	Developer Client displays the list of stored queries in a table
8	Get Parameters of a Query	Once the Developer client page of a SWIM data is loading, the Developer Client front-end page makes a request to the Web Server. The Webserver makes an HTTP Get call at `{FilteringServiceURL}/search/{queryId}/parameters` on the SWIM Filtering Service for each stored query	Receive parameter list in (`f=json`) format	Developer Client displays the list of parameters for each stored query in a table
9	Run Query	Once the request is received from Developer Client front-end page including the query Id, the Webserver makes an HTTP Post call at `{FilteringServiceURL}/search/`	Receive query result in (`f=json`, `geojson`) format	Developer Client displays the results in simple text, or if the result is in GeoJSON format, on a map

#	FUNCTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
		`{queryId}` on the SWIM Filtering Service		
10	Delete Query	Once the “Edit” button is pressed on the Developer Client, the client makes an HTTP Delete call at `{FilteringServiceURL}/search/{queryId}` on the SWIM Filtering Service to delete the query	HTML OK code when successful	Developer Client then reloads the page to update the table of query, removing the deleted query

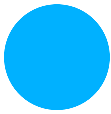
A

ANNEX A (INFORMATIVE) REVISION HISTORY

ANNEX A (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 API — Features — Part 5: Search (PROPOSAL). Open Geospatial Consortium, https://github.com/opengeospatial/ogcapi-features/tree/master/proposals/search, (2022)

[6] 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).

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

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

[9] 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.

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

[11] 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).

[12] 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).

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

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

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