OGC Testbed-16

Aviation Engineering Report

Publication Date: 2021-01-13

Approval Date: 2020-12-14

Submission Date: 2020-11-07

Reference number of this document: OGC 20-020

Reference URL for this document: http://www.opengis.net/doc/PER/t16-D001

Category: OGC Public Engineering Report

Editor: Sergio Taleisnik

Title: OGC Testbed-16: Aviation Engineering Report

OGC Public Engineering Report

Table of Contents

Chapter 1. Subject

This Testbed-16 Aviation Engineering Report (ER) summarizes the implementations, findings and recommendations that emerged from the efforts of further advancing interoperability and usage of Linked Data within the Federal Aviation Administration (FAA) System Wide Information Management [https://www.faa.gov/air_traffic/technology/swim/] (SWIM) context. The goal of this effort was to experiment with OpenAPI and Linked Data to explore new ways for locating and retrieving SWIM data in order to enable consumers to consume SWIM data more easily in their business applications, and enable the discovery of additional relevant information for their needs.

Specifically, this ER documents the possibility of querying and accessing data (and its metadata) using Semantic Web Technologies as well as interlinking heterogeneous semantic data sources available on the Web. Together with an analysis on the potential for using OpenAPI-based Application Programming Interface (API) definitions to simplify access to geospatial information, an exploration of solutions for data distribution that complement those currently used by SWIM is presented.

This ER describes a solution architecture consisting of a collection of interoperable components developed to demonstrate technologies that can help achieve Testbed 16 objectives. This document describes 1.) a component built to relay SWIM data through an OGC API – Features endpoint, and 2.) a Client Application built to interact with this API. This document also describes:

- A Semantic Registry built to serve SWIM service and collection metadata,
- A Triple Builder and a Triple Store designed to generate aviation linked data, and
- A Semantic Web Client built to interact with the Triple Store.

Finally, this ER captures lessons learned and recommendations for future work.

Chapter 2. Executive Summary

FAA SWIM Data Services currently produce data from the National Airspace System (NAS) to consumers using various protocols and service offerings in both synchronous and asynchronous messaging formats. These services are documented and described in the FAA's NAS Service Registry/Repository (NSRR). Consumers can access the NSRR to obtain this information for each SWIM service offering in order to develop their business applications. Often, business applications require multiple SWIM data. Therefore, efforts were made to standardize the SWIM data models for their domains (i.e. weather, aeronautical, and flight) based on XML standards such as Weather Information Exchange Model (WXXM), Flight Information Exchange Model (FIXM), and Aeronautical Information Exchange Model (AIXM).

The use of standardized data models improves the interoperability of data processing. SWIM Data Services are designed with "data-centric approaches" consisting on defining and standardizing logical data models. These in turn drive the implementation of dictionaries (what to call things), metadata (a means to discover things) and services (how to access and process things). As the implementations that result from these approaches fall short of providing sufficient semantics and context, the interoperability between them becomes limited. These circumstances can be improved by taking a semantic-enabled approach which adds a machine-encoded, semantics-based knowledge layer to NAS data and services, helping ensure that SWIM objects and entities are properly interpreted and employed, in a useful mission context [1].

Previous OGC work has addressed the challenges of increasing interoperability and semantically-enabling aviation data services. Recently, the OGC community has developed a new family of standardized OpenAPI-based Web APIs for the various geospatial resource types with the potential to enhance the way in which consumers can access geospatial data from various sources. One initiative in particular, reflected in the Testbed-14 (OGC 18-035) *Semantically Enabled Aviation Data Models Engineering Report*, provided the baseline and recommendations that shaped the requirements for the Testbed-16 Aviation Task.

The goals for the Aviation Task were classified into two main areas:

- **API Modernization**
  - Evaluating solutions for data distribution that complement those currently used by FAA SWIM
  - Advancing data integration in the aviation community by facilitating flexibility in deployment solutions
  - Exploring the potential of OpenAPI on simplifying and modernizing access to geospatial information
- **Linked Data**
  - Querying and accessing SWIM data (and associated metadata) using Semantic Web technologies
  - Interlinking heterogeneous aviation related semantic data sources available on the Web

To achieve these goals, the Aviation Task was organized into the development and testing of a system of six interconnected components, as shown in Figure 1:

- A **SWIM Data Relay API** (identified as *D100*): An OGC API - Features endpoint relaying SWIM data retrieved from SWIM Data Services.
- A **Semantic Registry** (identified as *D101*): An API serving metadata for SWIM services and metadata for the datasets these services provide.
- A **Triple Builder and a Triple Store** (identified as *D103*): A generator and provider of aviation linked data by combining aviation ontologies with data retrieved from either the SWIM Data Relay API or from other external data sources.
- A set of **Aviation Ontologies**: Built to support the generation of aviation linked data by the Triple Builder.
- A **Semantic Web Client** (identified as *D105*): A client application built to consume SWIM linked data from the Triple Store and use it in several types of user interfaces.
- A **SWIM Data Client** (identified as *D106*): A client application built to consume SWIM data from the SWIM Data Relay API and use it in a 3D map environment.

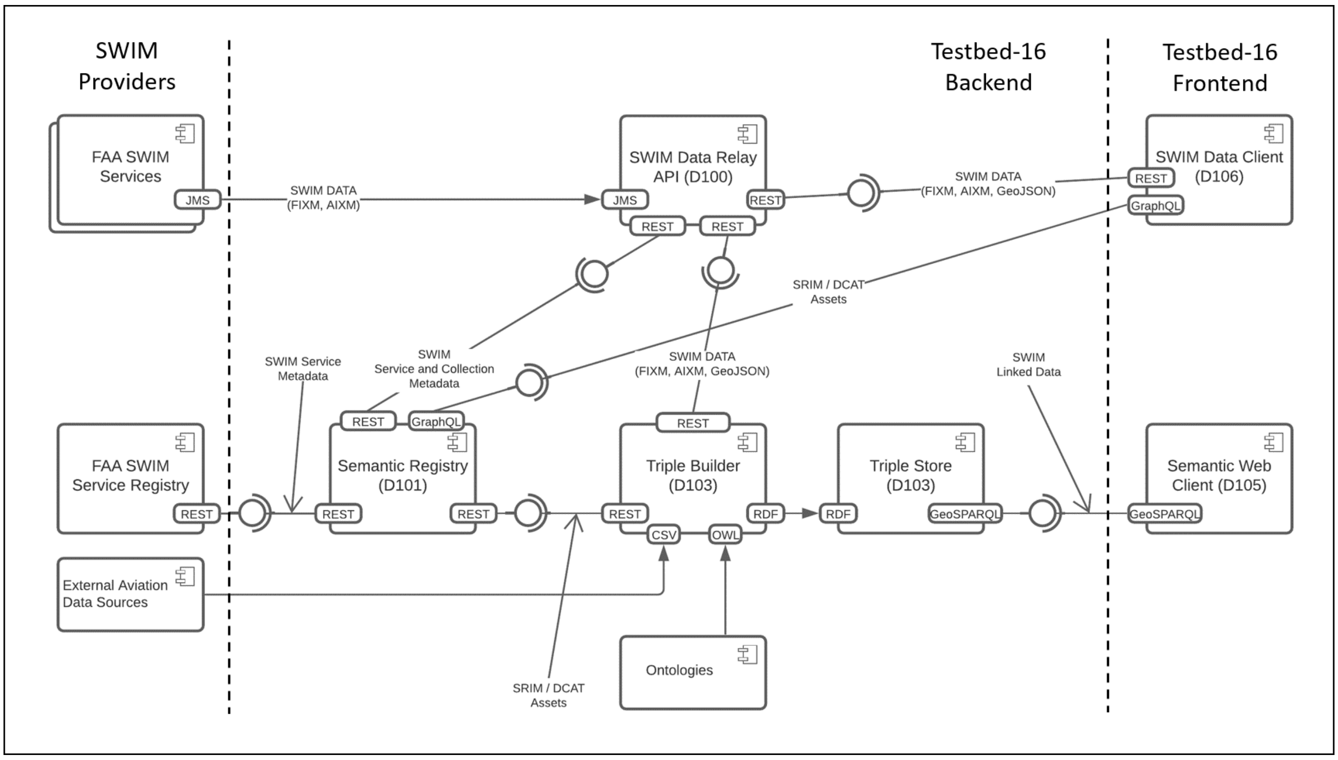


Figure 1. Testbed-16 Component Breakdown and Data Flow

- The *API Modernization* goals were explored through the work in the SWIM Data Relay API (D100) and the SWIM Data Client (D106). The development and testing of these components demonstrated the use of OGC API - Features as a proxy to fetch, serve and ultimately consume heterogeneous SWIM data.
- The *Linked Data* goals were explored through the work in the Semantic Registry (D101), the Triple Builder / Triple Store (D103), and the Semantic Web Client (D105). The development and testing of these components demonstrated the generation and consumption of aviation linked data, as well as the generation and consumption of semantically-enriched SRIM assets.

All components were successfully developed and tested. The lessons learned throughout the Testbed are captured in this ER. The following is a set of recommendations for future work:

- **Fostering the use of OGC API and Linked Data in aviation**: Work in Testbed 16 demonstrated the benefits of using OGC APIs but also documented several downsides that may require adapting the perspective on how the aviation industry pretends to leverage their power of interoperability. The Testbed also shed light on the complexities of generating and consuming aviation linked data while at the same time laying the foundation for future activities that could facilitate the adoption of linked data by the aviation community. Recommendations include:
    1. **Implementing OGC API - Features within SWIM Data Services** would eliminate the two main issues demonstrated in this Testbed of an OpenAPI-based SWIM API: the need for massive additional storage and a complex transformation logic.
    2. **Demonstrating Interoperability Between Diverse APIs** would speed up the development of OpenAPI-based SWIM APIs by troubleshooting the design and communication of both servers and clients.
    3. **Demonstrating the Value of Linked Data in Aviation** would help the aviation industry leverage the return of investment of transitioning to using linked data. The components built in this Testbed could be used as a starting point for such a demonstration.
    4. **Exploring Alternatives for a Seamless Transition to Linked Data** would make it easier for servers and clients to gradually transition to using linked data.
    5. **Exploring Linked Data Support Alternatives for OGC APIs** would help foster the adoption of linked data within OGC APIs.
- **Ontology Development**: Work on this Testbed demonstrated the critical role of ontologies to support information integration and reasoning by means of search and discovery of assets and support of aviation linked data. Recommendations include:
    1. **Expanding the Scope of Aviation Ontologies** by first building foundational cross-domain ontologies followed by aviation-specific ontologies based on the aforementioned.
    2. **Standardizing the SRIM Model** to facilitate the adoption of semantic registries.
    3. **Improving the GeoSPARQL Standard** to enhance support for querying aviation linked data

2.1. What does this ER mean for the Geosemantics Working Group and the OGC in general

The OGC Working Group for review of this ER is the Geosemantics Domain Working Group (DWG). This Testbed work may also be applicable to the Aviation DWG which is co-chaired by the FAA and EUROCONTROL.

The scope of the Geosemantics DWG is any aspect of semantic conceptual modeling and formal representation of data models that advances the geospatial interoperability mission of the OGC. The mission of the Geosemantics DWG is to establish an interoperable and actionable semantic framework for representing the geospatial knowledge domains of information communities as well as mediating between them.

The Geosemantics DWG has reviewed several ERs that served as baseline for the Testbed-16 Aviation Task, most notably the *Semantically Enabled Aviation Data Models ER* of Testbed-14 (OGC 18-035), making this DWG the ideal reviewer of this Testbed-16 Aviation ER. This Task took a step further from the conceptual elements explored in previous work and generated a working example of a modern and semantically-enabled system of interconnected aviation data sources. The lessons learnt from the development of this system will help advance the objectives of this DWG and the OGC.

2.2. Document contributor contact points

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

Contacts

Name	Organization	Role
Sergio Taleisnik	Skymantics	Editor
Charles Chen	Skymantics	Contributor
Eugene Yu	George Mason University	Contributor
Felipe Carrillo Romero	Hexagon	Contributor
Stephane Fellah	Image Matters	Contributor
Wenwen Li	Arizona State University	Contributor
Yuanyuan Tian	Arizona State University	Contributor
Scott Serich	OGC	Contributor

2.3. Foreword

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.

Chapter 3. References

The following normative documents are referenced in this document.

- OGC: OGC 17-069, OGC API - Features - Part 1: Core (2019) [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html]
- OGC: OGC 11-052r4, OGC GeoSPARQL - A Geographic Query Language for RDF Data (2012) [http://www.opengis.net/doc/IS/geosparql/1.0]
- FAA, SESAR: Service Description Conceptual Model (SDCM) 2.0 [https://www.faa.gov/air_traffic/technology/swim/governance/service_semantics/media/SDCM_v2.0/SDCM_v2.0.html]
- EUROCONTROL, FAA, NGA: AIXM 5.1 Specification (2010) [http://aixm.aero/page/aixm-51-specification]
- EUROCONTROL, FAA: FIXM Core v4.2.0 [https://www.fixm.aero/release.pl?rel=FIXM-4.2.0]
- IETF: RFC-7946 The GeoJSON Format (2016) [https://tools.ietf.org/html/rfc7946]
- Facebook: GraphQL Specification (2018) [http://spec.graphql.org/June2018/]
- W3C: RDF Schema 1.1 (2014) [http://www.w3.org/TR/rdf-schema/]
- W3C: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (2012) [https://www.w3.org/TR/owl-syntax/]

Chapter 4. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

- **Linked Data**

  a set of best practices for publishing and connecting structured data on the Web.

- **Semantics**

  a conceptualization of the implied meaning of information that requires words and/or symbols within a usage context. [1]

- **DCAT**

  Data Catalog Vocabulary — an RDF vocabulary designed to facilitate interoperability between data catalogs published on the Web.

- **SRIM**

  Semantic Registry Information Model — a superset of the DCAT ontology that defines the set of classes and properties commonly used to represent any item in a register. [2]

- **SWIM**

  System Wide Information Management — program overseen by the Federal Aviation Administration (FAA) designed to facilitate greater sharing of Air Traffic Management (ATM) system information

- **Taxonomy**

  a system or controlled list of values by which to categorize or classify objects. [1]

- **Triple**

  the most high-level abstraction in the semantic web. It describes a statement using a triple of "Subject - Predicate - Object". URIs are used to identify the subject of the statement. The object of the statement can be another URI or a literal like a string or number. [3]

4.1. Abbreviated terms

- AIXM Aeronautical Information Exchange Model
- API Application Programming Interface
- DCAT Data Catalog Vocabulary
- FIXM Flight Information Exchange Model
- OWL Web Ontology Language
- RDF Resources Description Framework
- SKOS Simple Knowledge Organization System
- SRIM Semantic Registry Information Model

- SWIM System Wide Information Management
- WSDOM Web Service Description Ontological Model
- WXXM Weather Information Exchange Model

Chapter 5. Overview

Section 6 describes the background work that preceded and laid the foundation for the work performed in the Aviation Task.

Section 7 outlines the status quo, presents the Testbed 16 Aviation Task goals and summarizes the work performed on this Testbed activity.

Section 8 describes the SWIM Data Relay API developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 9 describes the Semantic Registry developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 10 describes the Aviation Ontologies developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 11 describes the Triple Builder and Triple Store developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 12 describes the Semantic Web Client developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 13 describes the SWIM Data Client developed to meet the goals of this Testbed activity as well as documenting lessons learned and challenges.

Section 14 provides recommendations for future work.

Chapter 6. Background

This Testbed-16 task worked on better understanding the value of **semantic-enablement** and of **modern Web APIs** in the context of SWIM service integration. The work carried out in this Testbed was based on a knowledge base composed of a plethora of OGC initiatives, summarized in Figure 2 and further described in this chapter.

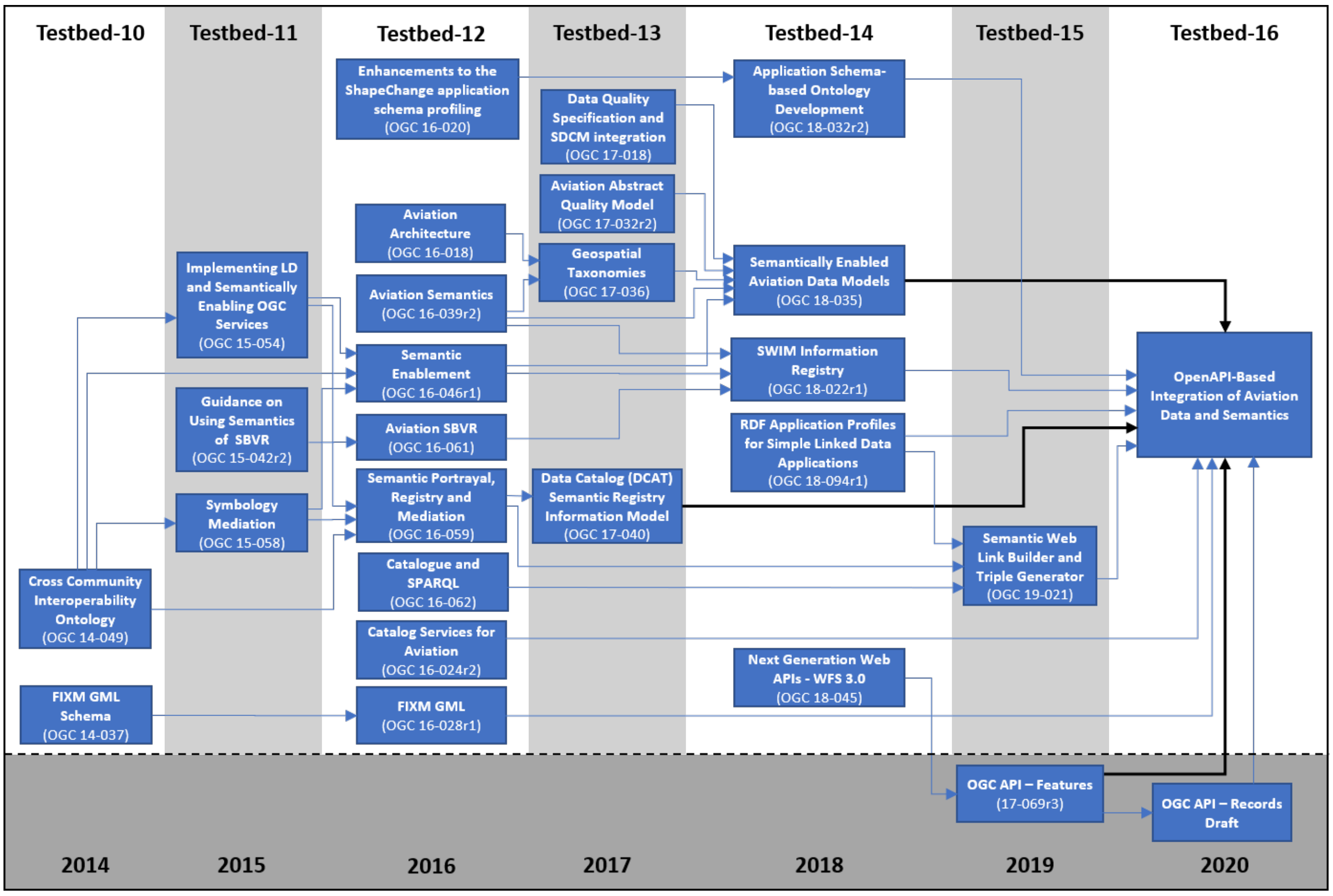

Testbed-10
Testbed-11
Testbed-12
Testbed-13
Testbed-14
Testbed-15
Testbed-16
Enhancements to the ShapeChange application schema profiling (OGC 16-020)
Data Quality Specification and SDCM integration (OGC 17-018)
Application Schema-based Ontology Development (OGC 18-032r2)
Aviation Abstract Quality Model (OGC 17-032r2)
Aviation Architecture (OGC 16-018)
Geospatial Taxonomies (OGC 17-036)
Semantically Enabled Aviation Data Models (OGC 18-035)
Implementing LD and Semantically Enabling OGC Services (OGC 15-054)
Aviation Semantics (OGC 16-039r2)
Semantic Enablement (OGC 16-046r1)
SWIM Information Registry (OGC 18-022r1)
OpenAPI-Based Integration of Aviation Data and Semantics
Guidance on Using Semantics of SBVR (OGC 15-042r2)
Aviation SBVR (OGC 16-061)
RDF Application Profiles for Simple Linked Data Applications (OGC 18-094r1)
Semantic Portrayal, Registry and Mediation (OGC 16-059)
Data Catalog (DCAT) Semantic Registry Information Model (OGC 17-040)
Symbology Mediation (OGC 15-058)
Semantic Web Link Builder and Triple Generator (OGC 19-021)
Cross Community Interoperability Ontology (OGC 14-049)
Catalogue and SPARQL (OGC 16-062)
Catalog Services for Aviation (OGC 16-024r2)
Next Generation Web APIs - WFS 3.0 (OGC 18-045)
FIXM GML Schema (OGC 14-037)
FIXM GML (OGC 16-028r1)
OGC API – Features (17-069r3)
OGC API – Records Draft
2014
2015
2016
2017
2018
2019
2020

Figure 2. OGC Work Timeline Preceding Testbed-16

6.1. Semantic-Enablement

The Semantic Web is a Web of Data — of dates and titles and any other data one might conceive of. The collection of Semantic Web technologies (RDF, OWL, SKOS, SPARQL, etc.) provides an environment where applications can query that data and draw inferences using vocabularies, among others [4].

To enable wider adoption of the Semantic Web, the term of Linked Data was introduced in 2008. Linked data provides a simplified view of the Semantic Web as a web of linkage between data nodes. The idea of linked data is similar to the web of hypertext. However the semantic web is not merely about publishing data on the web but more about making links in such a way that a person or a machine can explore the data [3]. The linked data leads to other related data. The semantic web is also constructed in such a way that it can be parsed and reasoned about. The web of hypertext is constructed with links anchored in HTML documents. At the same time the semantic web is constructed in such a way that arbitrary links between entities are described by Triples in RDF. URIs are used to identify any kind of concept [5].

OGC has investigated the use of Linked Data and Semantic Web Technologies in numerous past Testbeds:

1. OGC 18-094r1, OGC Testbed-14: Characterization of RDF Application Profiles for Simple Linked Data Application and Complex Analytic Applications Engineering Report
2. OGC 18-032r2, OGC Testbed-14: Application Schema-based Ontology Development Engineering Report
3. OGC 17-040, OGC Testbed-13: DCAT/SRIM Engineering Report
4. OGC 17-032r2, OGC Testbed-13: Aviation Abstract Quality Model Engineering Report
5. OGC 17-018, OGC Testbed-13: Data Quality Specification Engineering Report

The topic of semantic-enablement of models used in the domain of aviation has been explored previously in OGC Testbeds.

1. OGC 19-021, OGC Testbed-15: Semantic Web Link Builder and Triple Generator
2. OGC 18-035, OGC Testbed 14: Semantically Enabled Aviation Data Models Engineering Report
3. OGC 17-036, OGC Testbed-13: Geospatial Taxonomies Engineering Report
4. OGC 16-039r2, OGC Testbed-12: Aviation Semantics Engineering Report
5. OGC 16-046r1, OGC Testbed-12: Semantic Enablement Engineering Report

Testbed-12 participants focused on semantic-enablement by means of parallel initiatives that worked on topics of semantic portrayal, semantic search and semantic mediation (OGC 16-046r1). An OGC ER (OGC 16-039r2) examined the role of geospatial semantic technology within aviation by proposing extensions to WSDOM that enabled an efficient discovery of OGC-compatible web services (OWS) and helped to create a service knowledge base. A Semantic Registry Information Model (SRIM) was defined as a superset of the W3C Data Catalog (DCAT) ontology that defined the set of classes and properties commonly used to represent any item in a register (OGC 16-059).

Other initiatives within Testbed-12 reviewed the topic of cyber security within the aviation domain in conjunction with OGC Web Services (OWS) (OGC 16-040r1), provided recommendations for the implementation of GML elements into FIXM (OGC 16-028r1), and advanced previous work in the area of business rules for AIXM 5 based on SBVR (OGC 16-061).

Testbed-13 saw the implementation of a RESTful Semantic Registry that supported the Semantic Registry Information Model (SRIM) (OGC 17-040). This work served as a foundation for the Semantic Registry built in Testbed-16.

Further work in Testbed-13 focused on formulating geospatial taxonomies to support the classification of aviation services based on their geospatial characteristics and integrating them with SDCM in order to enable the discovery of those services (OGC 17-036).

Finally, Testbed-13 also focused on aviation data quality. The Aviation Abstract Quality Model ER (OGC 17-032r2) describes a taxonomy and a model for the fundamental concepts related to data quality. The OGC Data Quality Specification ER (OGC 17-018) provided methods to quantify the quality concepts described in the AQM, and extended the SDCM to be able to support this quality information for each service described in the registry.

The **Testbed-14** Semantically Enabled Aviation Data Models ER (OGC 18-035) describes work on semantically-enabling existing data and metadata models used in the aviation industry. Before Testbed-14, these models were already using Linked Data standards to represent information. However, these models were failing to provide enough semantics to facilitate the integration of information and services, improve search and discovery in the current registry, and increase the level of automation in systems. OGC 18-035 laid out a series of recommendations for future work, many of which were addressed in Testbed-16: Managing controlled vocabularies, developing a Semantic Registry, investigating the modularization and encoding of aviation ontologies, as well as demonstrating the use of semantic enablement of aviation data.

Finally, the **Testbed-15** Semantic Web Link Builder and Triple Generator (OGC 19-021) described a generalized approach towards performing data fusion from multiple heterogeneous geospatial linked data sources. Despite the use case in this Testbed was not aviation focused, the concepts developed were deemed reusable within any other domain and were therefore utilized for the Testbed 16 Aviation task.

6.2. OpenAPI-based Web APIs

There are many documented benefits for using Web APIs in the context of data retrieval and processing like SWIM. Benefits include faster time to market for products, more flexibility in deployment models, and straight forward upgrade paths as standards evolve.

The goal of the OpenAPI Specification is to define a standard, language-agnostic interface description for HTTP APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, OpenAPI removes the guesswork in calling the service [6].

OGC members have been investigating the use of OpenAPI since 2016. This investigation was in part due to recognizing that OGC's existing web service standards (aka W*S or OWS) were in effect web APIs. Modernizing the OGC model for getting content on the web required a fairly fundamental change in underlying design [7].

Work in Testbed-14 initiated OGC developments in OpenAPI. A major revision of the OGC Web Feature Service (WFS) was revised by proposing a set of innovations, including the support of the OpenAPI specification (OGC 18-021, OGC 18-045). OGC built upon that experience and one year later published OGC API - Features (OGC 17-069r3), the first OGC standard defined and documented using OpenAPI. Concurrently, Testbed-15 proposed an OpenAPI-based API specification for maps and tiles (OGC 19-069), as well as styles (OGC 19-010r2).

At the time of publication of this ER, the OGC members are actively working on nine API standards, each one focused on serving a specific set of elements:

- OGC API - Features [https://ogcapi.ogc.org/features/]
- OGC API - Common [https://ogcapi.ogc.org/common/]
- OGC API - Maps [https://ogcapi.ogc.org/maps/]
- OGC API - Tiles [https://ogcapi.ogc.org/tiles/]

- OGC API - Styles [https://ogcapi.ogc.org/styles/]
- OGC API - Environmental Data Retrieval (EDR) [https://ogcapi.ogc.org/edr/]
- OGC API - Records [https://ogcapi.ogc.org/records/]
- OGC API - Processes [https://ogcapi.ogc.org/processes/]
- OGC API - Coverages [https://ogcapi.ogc.org/coverages/]

This section next describes two of the OGC APIs that are most relevant to this Testbed-16 Aviation task.

OGC API - Features

The OGC API – Features Standard is 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 that want to follow a standard way of accessing and sharing feature data [8].

This OGC API was considered as the foundation for the SWIM Data Relay API and the SWIM Data Client developed in this Testbed, as the data retrieved from SWIM services would ultimately be displayed as geospatial elements on maps.

OGC API - Records

The OGC API – Records document is a multi-part draft specification that defines the capability to create, modify, and query metadata on the Web. The draft specification enables the discovery of geospatial resources by standardizing the way collections of descriptive information about the resources (metadata) are exposed [9].

This draft OGC API was initially considered as the foundation for the Semantic Registry developed in this Testbed, as metadata querying was a fundamental part of these components, but was eventually replaced by a standard REST API.

Chapter 7. Technical Architecture

This chapter outlines the current status of semantic-enablement and interoperability of SWIM Services, presents the goals defined for this Testbed based on the status quo, and finally describes the components and their interactions that were built to tackle those goals.

7.1. Status Quo

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

As part of the SWIM architecture, data providers create services for consumers to access their data. Each service is designed to be stand-alone. However, the value of data increases when combined with other data. Real-world situations are often not related to data from not just one but from several SWIM Data Services. Having consumers retrieving data from several SWIM services raises the need for interoperability between those services. In recent years OGC members have worked on the development of a standardized family of Web APIs defined using OpenAPI for the various geospatial resource types utilized by the geospatial community. Currently, there are no SWIM services implemented using the new generation of OGC APIs.

The diversity of actors providing SWIM Services requires measures ensuring the data being transmitted can be successfully consumed. Traditionally, SWIM Data Services are designed with "data-centric approaches" consisting of defining and standardizing logical data models. These models in turn drive the implementation of dictionaries (what to call things), metadata (a means to discover things) and services (how to access and process things). As the implementations that result from these approaches fall short of providing sufficient semantics and context, the interoperability between them becomes limited [1].

The FAA has engaged in initiatives to facilitate the integration of heterogeneous data sources, including:

- **Service Description Conceptual Model (SDCM)**: A joint FAA-SESAR work, the SDCM is a graphical and lexical representation of the properties, structure, and interrelationships of all service metadata elements, collectively known as a Service Description.
- **SWIM Controlled Vocabulary (SWIM CV)**: A single source providing terms and definitions commonly employed in SWIM.
- **Web Service Description Ontological Model (WSDOM)**: An ontology model intended to be a basis for model-driven implementation of artifacts related to Service Oriented Architectures (SOA)
- **Semantic Web for Air Transportation (SWAT) Special Interest Group**: A taskforce dedicated to sharing experiences and approaches in applying Semantic Web technologies
- **semantics.aero**: An open repository for use by the international aviation community to publish artifacts developed using Semantic Web technologies

7.2. Problem Statement

Testbed-16 required investigating data integration options based on semantic web technologies and analyzing the current status of achievable interoperability.

The goals of the Testbed-16 Aviation Task were classified into two areas:

- **API Modernization**
  - Evaluating solutions for data distribution that complement those currently used by FAA SWIM.
  - Advancing data integration in the aviation community by facilitating flexibility in deployment solutions.
  - Exploring the potential of OpenAPI for simplifying and modernizing access to geospatial information.
- **Linked Data**
  - Querying and accessing SWIM data (and associated metadata) using Semantic Web technologies.
  - Interlinking heterogeneous aviation related semantic data sources available on the Web.

The following questions were proposed in the Testbed 16 Call for Participation (CFP) as guidance for the Task efforts:

1. Should the existing SWIM architecture be modernized with resource-oriented Web APIs?
2. What role can OGC Web APIs play for a modernized SWIM services?
3. How can OGC APIs be used to address the heterogeneous semantic SWIM landscape?
4. What impact do Linked Data principles and requirements have on OGC Web APIs?
5. How to deal with the various ontologies and taxonomies used in SWIM?
6. How to best enhance the various ontologies and how to build a scalable geospatial definition server?
7. How to best combine data from various SWIM data feeds to make it available for multi-source and linked data based analytics?

The long-term goal for the FAA is, as seen on Figure 3, to facilitate access to SWIM services and registries from not only the FAA but also other aviation partners as well such as EUROCONTROL or the Korea Aviation Corporation (KAC), Republic of Korea (ROK). For the purposes of this Testbed, only FAA SWIM services were considered as part of the scope.

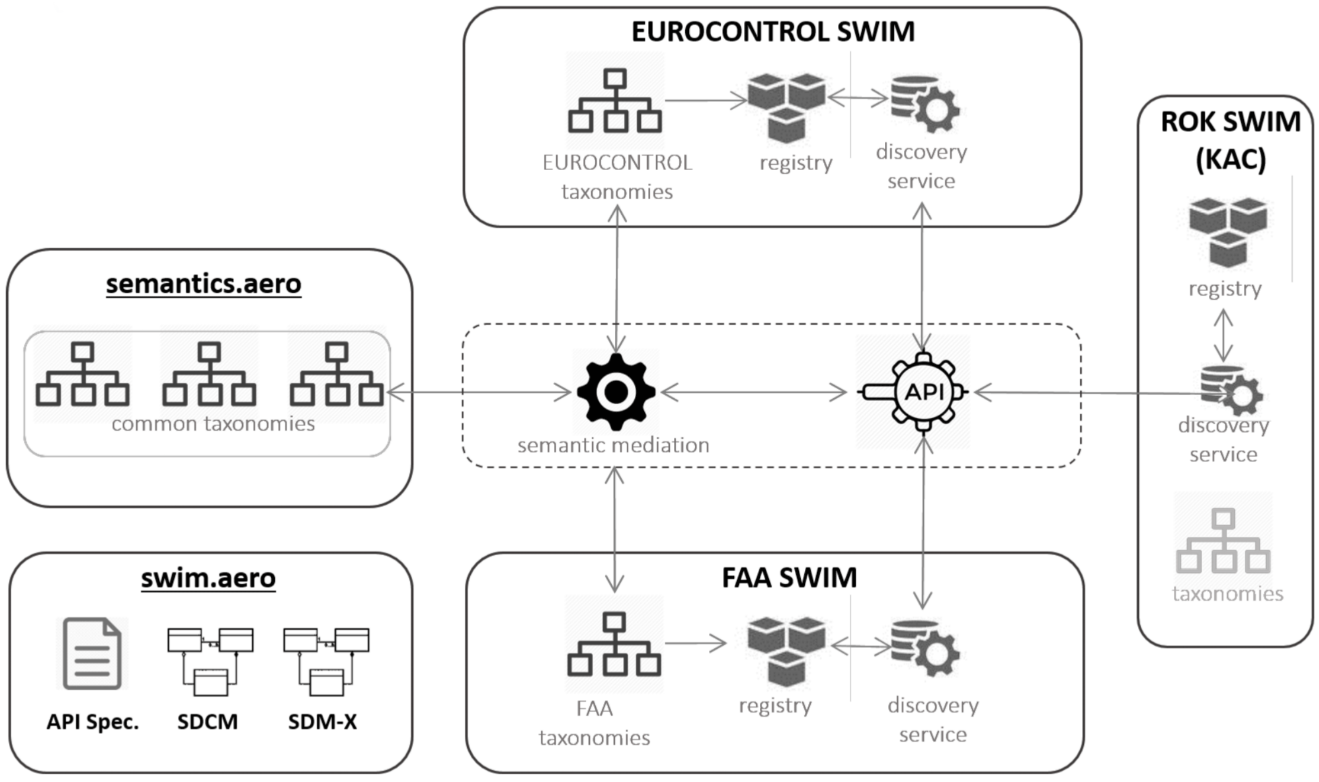


Figure 3. Deployment of API and Semantic Technology in Today's Global SWIM

7.3. Functional Overview

As shown in Figure 4, the Aviation Task architecture was organized into a system of six interconnected components. All six components were developed simultaneously throughout the Testbed, with permanent communication and cooperation among participant organizations. The following is an overview of each component and their interactions between them. A more thorough description of their internal architecture and lessons learned can be found in the following chapters in this ER.

- A SWIM Data Relay API (identified as *D100*): An OGC API - Features endpoint serving SWIM data.
- A Semantic Registry (identified as *D101*): An API serving metadata of SWIM services together with metadata of the datasets they provide.

NOTE	The Testbed CFP included two instances of the SWIM Data Relay API and did not include the Semantic Registry. The need for a Semantic Registry, and the replacement of the second SWIM Data Relay API instance with the Semantic Registry, was proposed and accepted early in the Testbed.

- A Triple Builder and Triple Store (identified as *D103*): A generator and provider of aviation linked data by combining aviation ontologies with data retrieved from either the SWIM Data Relay API or from other external data sources.
- A set of Aviation Ontologies: Built to support the generation of aviation linked data by the Triple Builder.

NOTE The development of ontologies was not part of the requirements specified in the Testbed CFP. The need for ontologies was identified in the early stages of the Testbed and acquired enough significance to be considered a separate component.

- A Semantic Web Client (identified as *D105*): A client application built to consume SWIM linked data from the Triple Store and use it in several types of user interfaces.
- A SWIM Data Client (identified as *D106*): A client application built to consume SWIM data from the SWIM Data Relay API and use it in a 3D map environment.

Each component, as well as their interactions, was designed to help answer the questions laid out by the main Testbed goals:

- The *Linked Data* goals were explored through the work in the Semantic Registry (D101), the Triple Builder / Triple Store (D103), and the Semantic Web Client (D105). The development and testing of these components demonstrated the generation and consumption of aviation linked data, as well as the generation and consumption of semantically-enriched SRIM assets.
- The *API Modernization* goals were explored through the work in the SWIM Data Relay API (D100) and the SWIM Data Client (D106). The development and testing of these components demonstrated the use of OGC API - Features as a proxy to fetch and serve heterogeneous SWIM data.

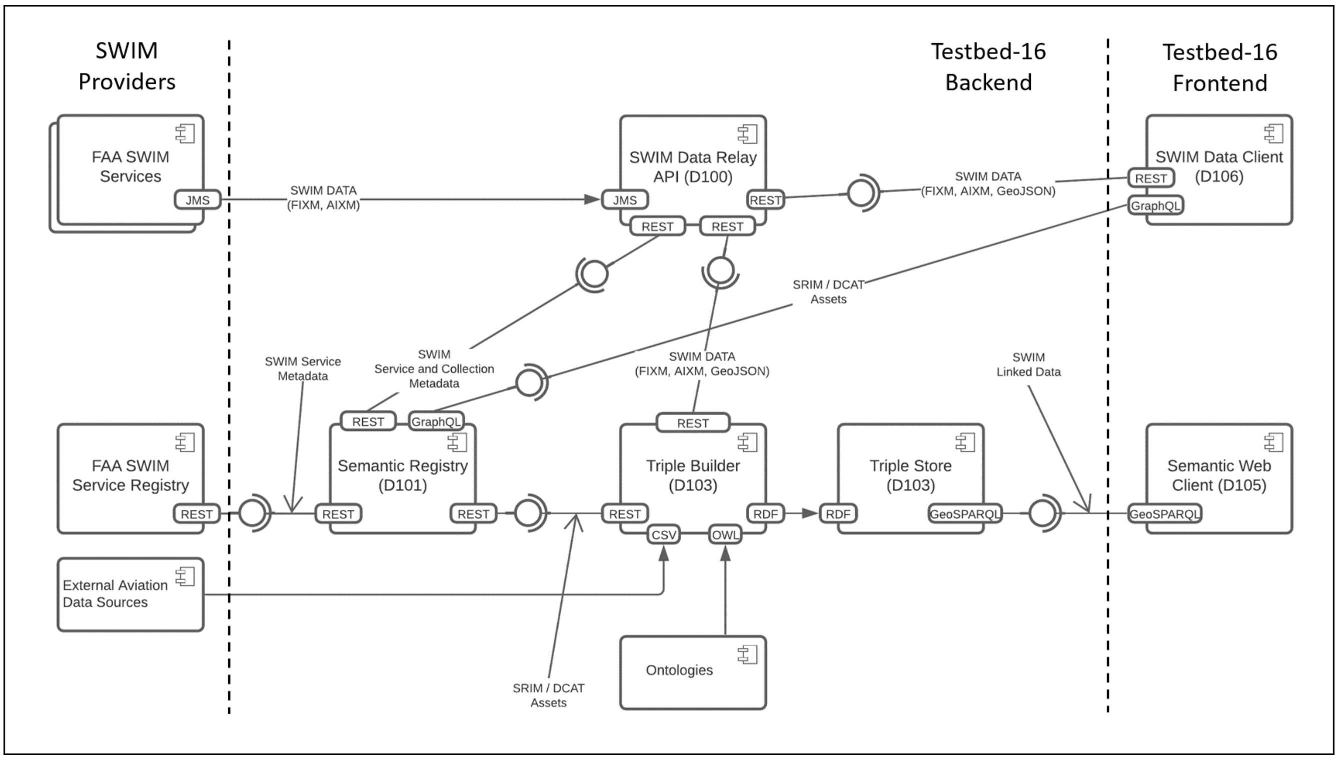


Figure 4. Testbed-16 Aviation Task Components

7.3.1. Frontend Component Interactions

Two scenarios involving end users were identified - one for each of the two clients developed in this Testbed: The **SWIM Data Client** (D106) and the **Semantic Web Client** (D105). Both scenarios consist of the end user interfacing a client application to visualize aviation data. The major differences between the scenarios are:

- Clients have different visual interfaces to display aviation data. The SWIM Data Client is a 3D map environment displaying aeronautical infrastructure, airspaces, and routes. The Semantic Web Client displays linked data about airports and flights on 2D maps, tables and graphs.
- The SWIM Data Client retrieves SWIM data via the SWIM Data Relay API, while the Semantic Web Client retrieves SWIM linked data from the Triple Store.
- The SWIM Data Client discovers data source endpoints through the Semantic Registry, while the Semantic Web Client has the endpoints hardcoded.

The frontend scenarios are described in Figure 5. Detailed descriptions and lessons learned from each component are presented in the following chapters in this ER.

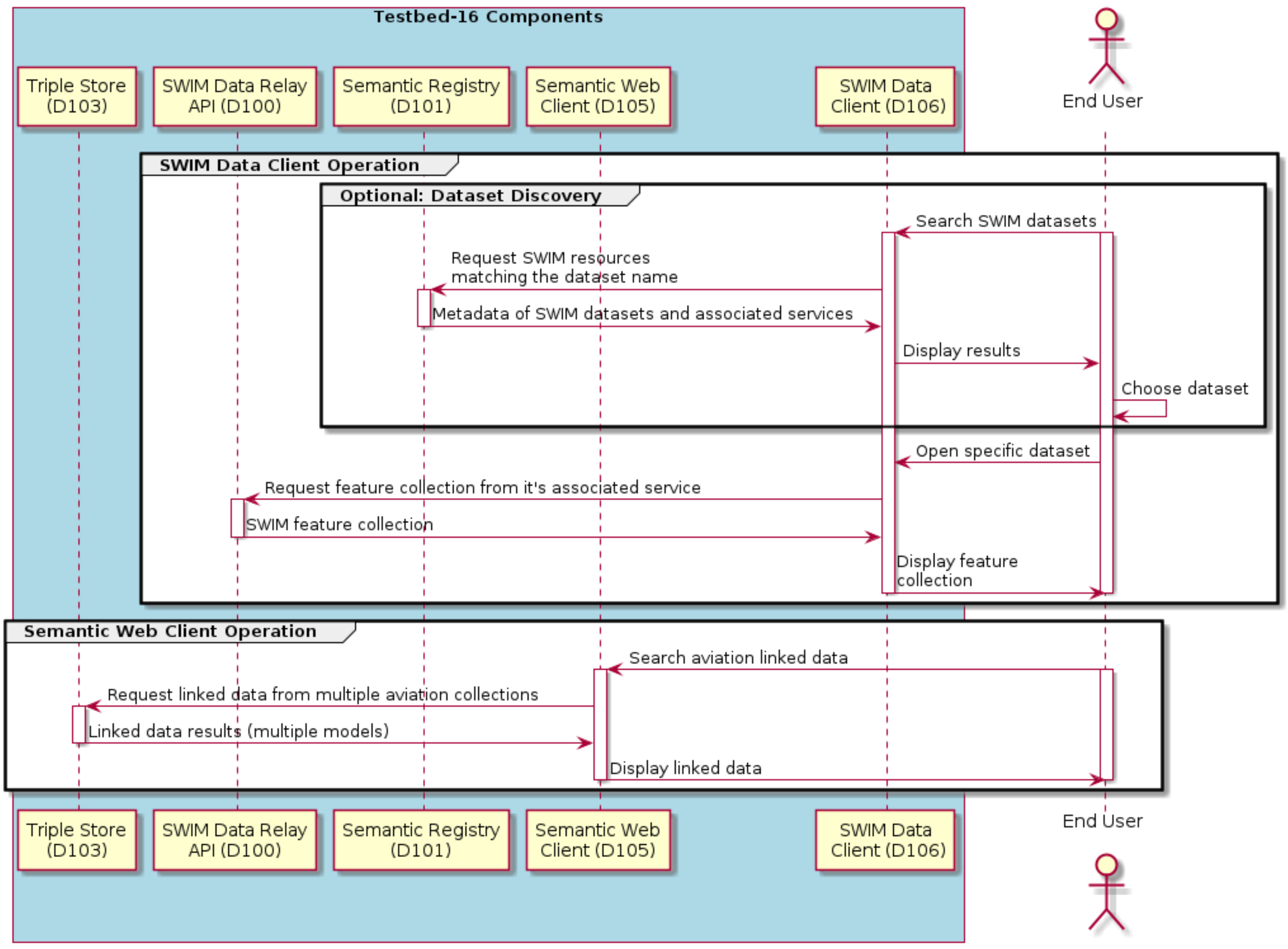

Testbed-16 Components
Triple Store (D103)
SWIM Data Relay API (D100)
Semantic Registry (D101)
Semantic Web Client (D105)
SWIM Data Client (D106)
End User
SWIM Data Client Operation
Optional: Dataset Discovery
Search SWIM datasets
Request SWIM resources matching the dataset name
Metadata of SWIM datasets and associated services
Display results
Choose dataset
Open specific dataset
Request feature collection from it's associated service
SWIM feature collection
Display feature collection
Semantic Web Client Operation
Search aviation linked data
Request linked data from multiple aviation collections
Linked data results (multiple models)
Display linked data

Figure 5. Testbed-16 Aviation Task Frontend Components Interactions

7.3.2. Backend Component Interactions

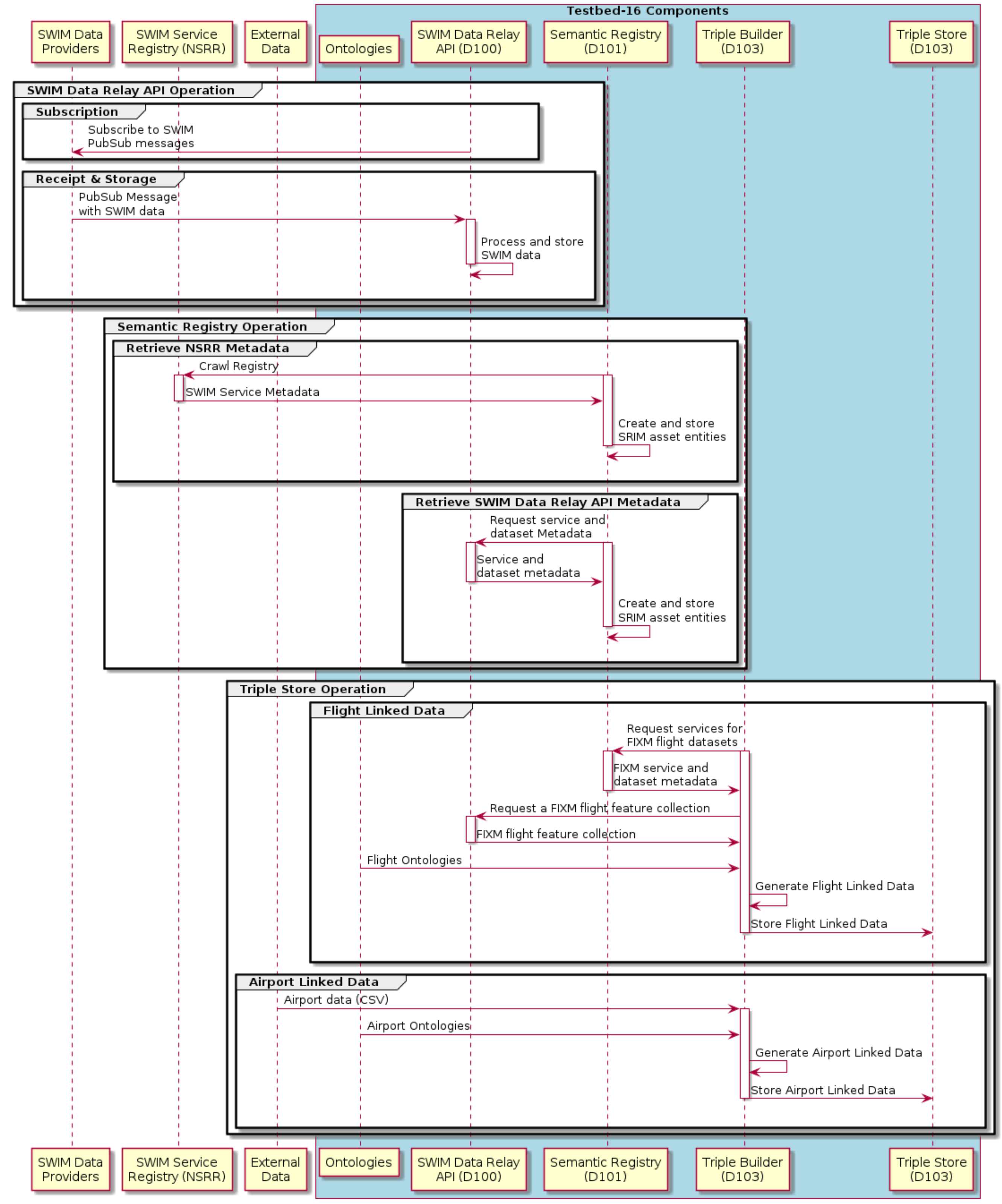
The backend processes support the operations of the frontend. There are three main backend scenarios, one for each of the backend components. These have in common the consumption, processing and storage of data related to aviation.

- The operation of the **SWIM Data Relay API** (D100) consists of extracting features from SWIM Publish/Subscribe (Pub/Sub) messages and making them available for other components to consume through an OGC API - Features endpoint.
- The operation of the **Semantic Registry** (D101) consists of extracting service and collection metadata from providers of SWIM data and making them available for other components to

consume as SRIM/DCAT assets through a REST or a GraphQL endpoint.

- The operation of the **Triple Builder and Triple Store** (D103) consists of extracting aviation data, transforming it into linked data, and making it available for other components to consume through a GeoSPARQL endpoint.

The backend scenarios are described in Figure 6. Detailed descriptions and lessons learned from each component are presented in the following chapters in this ER.



Testbed-16 Components
SWIM Data Providers
SWIM Service Registry (NSRR)
External Data
Ontologies
SWIM Data Relay API (D100)
Semantic Registry (D101)
Triple Builder (D103)
Triple Store (D103)
SWIM Data Relay API Operation
Subscription
Subscribe to SWIM PubSub messages
Receipt & Storage
PubSub Message with SWIM data
Process and store SWIM data
Semantic Registry Operation
Retrieve NSRR Metadata
Crawl Registry
SWIM Service Metadata
Create and store SRIM asset entities
Retrieve SWIM Data Relay API Metadata
Request service and dataset Metadata
Service and dataset metadata
Create and store SRIM asset entities
Triple Store Operation
Flight Linked Data
Request services for FIXM flight datasets
FIXM service and dataset metadata
Request a FIXM flight feature collection
FIXM flight feature collection
Flight Ontologies
Generate Flight Linked Data
Store Flight Linked Data
Airport Linked Data
Airport data (CSV)
Airport Ontologies
Generate Airport Linked Data
Store Airport Linked Data

Figure 6. Testbed-16 Aviation Task Backend Components Interactions

Chapter 8. SWIM Data Relay API

The SWIM Data Relay API was the component designed to fetch information from SWIM data sources, receive requests from the SWIM Data Client or the Triple Builder, and relay that information through an OGC API - Features service. This Component provides unified discovery operations that allow users to retrieve metadata (capabilities and information about the distribution of data) and data (encoded in standard simple feature formats, e.g. GeoJSON).

George Mason University participants did the development of this component. The main design objective of this component was to demonstrate the distribution of SWIM data through an OpenAPI-based API.

8.1. Status Quo

There are different services to enable the discovery and access of data within a System Wide Information Management (SWIM) system for aeronautic information. For the Federal Aviation Administration (FAA) SWIM, the NAS (National Aviation Services) Service Registry and Repository (NSRR) provides a centralized registry to hold detailed information about all existing and planned SWIM-enabled services. This includes information on where services and documents can be registered and searched. In addition, the FAA System Wide Information Management (SWIM) Cloud Distribution Service (SCDS) provides near real-time FAA SWIM data to the public via Solace Java Message Service (JMS) messaging.

Among the data services in SCDS, there are:

- SWIM Terminal Data Distribution System (STDDS) for surface movement data (Airport Surface Detection System — Model X (ASDE-X), Airport Surface Surveillance Capability (ASSC)), approach surveillance radar data from STARS systems, Runway Visual Range (RVR), and a variety of departure event data; Integrated Terminal Weather System (ITWS) for a variety of weather information in graphic and textual forms from different sites;
- Traffic Flow Management(TFMS) for both flight data (Flight Plans, Departure/Arrival Notifications, Position Reports) and flow information (Traffic Management Initiative (TMI), Ground Stop (GS), Ground Delay Program (GDP));
- Time Based Flow Management (TBFM) metering information data for knowledge of when metering is in effect at an Air Route Traffic Control Center (ARTCC) when Adjacent Center Metering (ACM) is occurring and at which sites, flight Scheduled Time of Arrivals (STAs) to the runway threshold, meter fix and all arcs, and flight Estimate Time of Arrivals (ETAs) to the runway threshold;
- SWIM Flight Data Publication Service (SFDPS) for a variety of En Route flight data (such as flight plans, beacon codes, and handoff status) and airspaces (such as sector configuration data, route status, Special Activity Airspace (SAA) status, and altimeter settings, and general data including En Route Automation Modernization (ERAM) status information);
- The Aeronautical Information Management (AIM) Federal Notices to Airmen (NOTAM) System (AIM FNS) for information on temporary changes to components of, or hazards in the National Airspace System (NAS).

There are different schemas for discovering and accessing data with the FAA SWIM. For example, the AIM FNS delivers messages that contain encoding in the Aeronautical Information Exchange Model (AIXM) 5.1 format with different extensions (e.g. FAA extension, Event extension). The SFDPS delivers en-route flight information in Flight Information Exchange Model (FIXM) NAS 3.0. The ITWS in production delivers weather information in legacy ITWS data models.

Standard exchange information models have been developed in the aviation domain. AIXM is designed to exchange information on aerodrome/heliport including movement areas, services, facilities, airspace structures, organizations and units (including services), points and NAVAIDs, procedures, routes, and flying restrictions. The latest version of AIXM is 5.1.1. FIXM is designed to exchange information on flight and flow Information. Version 4.2.0 is the latest version of FIXM. The Weather Information Exchange Model (WXXM) is designed to deliver weather data. The latest version of WXXM is version 2.0. The implementation and support of these standard exchange information models differs among different services and organizations.

SWIM data are time sensitive. Most SWIM data are delivered through message services in near real time or real time. Data volume is high volume and velocity. Over 1.6 trillion messages per day and 4.3 terabytes of data per day are passing through the SWIM. There is no direct support of spatiotemporal searching.

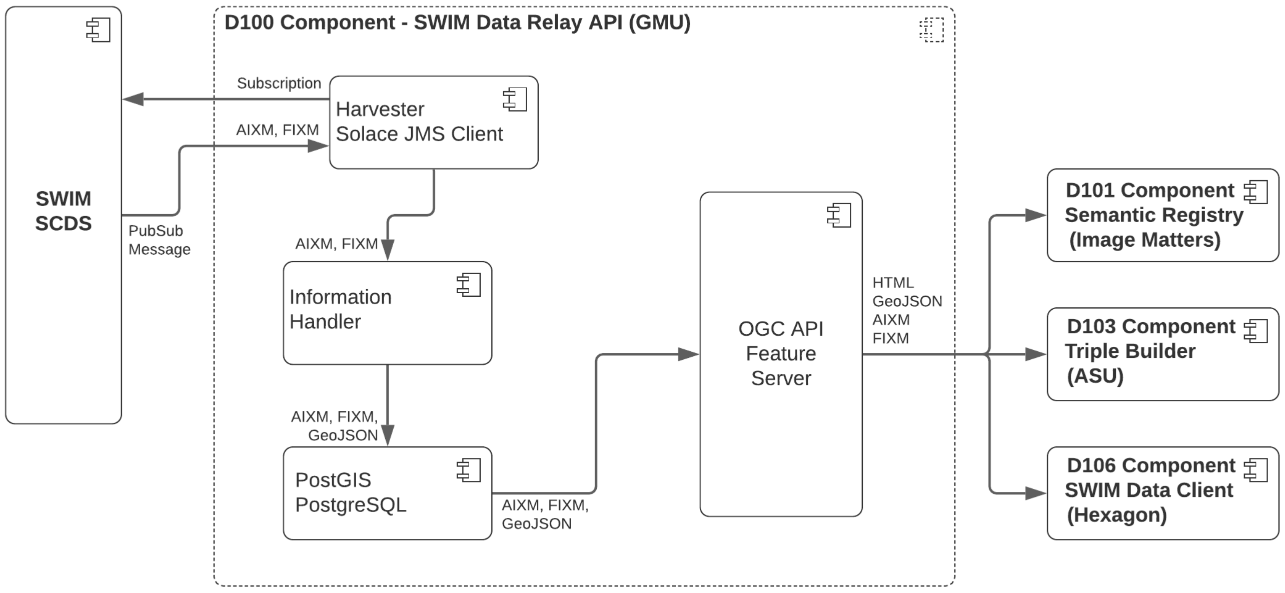
The emerging suite of OGC APIs is a new generation of OGC standard interfaces to enable the interoperation of geospatial resource discovery and access. The currently approved OGC API – Features is defined and documented using the OpenAPI 3.0 specification. OpenAPI supports the automation of coding for servers and clients in unified RESTful services, i.e. those implementing Representational State Transfer (REST). To the knowledge of this Testbed’s participants, there is currently no SWIM data service implemented as a service described using OpenAPI.

8.2. Functional Overview

This component acts as a proxy service to relay information from SWIM data services. SWIM data is normally delivered by SWIM Data Providers through publish/subscribe services. Since APIs defined using the OpenAPI specification deliver data through on-demand RESTful calls, the SWIM Data Relay API Component had to restructure the retrieved SWIM data before making it available through its REST endpoint. To achieve this, the component consists of four interconnected subcomponents:

1. A **Harvester** periodically captures SWIM messages.
2. A **Feature Handler** maps the incoming messages into a common feature model that supports both spatial and temporal searches.
3. A geospatially-aware **database** manages and stores the harvested data.
4. An **OGC API - Feature Service** publishes and relays SWIM data stored in the database.

Figure 7 breaks down the SWIM Data Relay API, and outlines how data flows through its subcomponents. The workflow sequence of its elements, as well as their connections to external components, is outlined on Figure 8.



SWIM SCDS
D100 Component - SWIM Data Relay API (GMU)
Subscription
AIXM, FIXM
Harvester Solace JMS Client
PubSub Message
AIXM, FIXM
Information Handler
AIXM, FIXM, GeoJSON
PostGIS PostgreSQL
AIXM, FIXM, GeoJSON
OGC API Feature Server
HTML GeoJSON AIXM FIXM
D101 Component Semantic Registry (Image Matters)
D103 Component Triple Builder (ASU)
D106 Component SWIM Data Client (Hexagon)

Figure 7. SWIM Data Relay API Component Breakdown and Data Flow

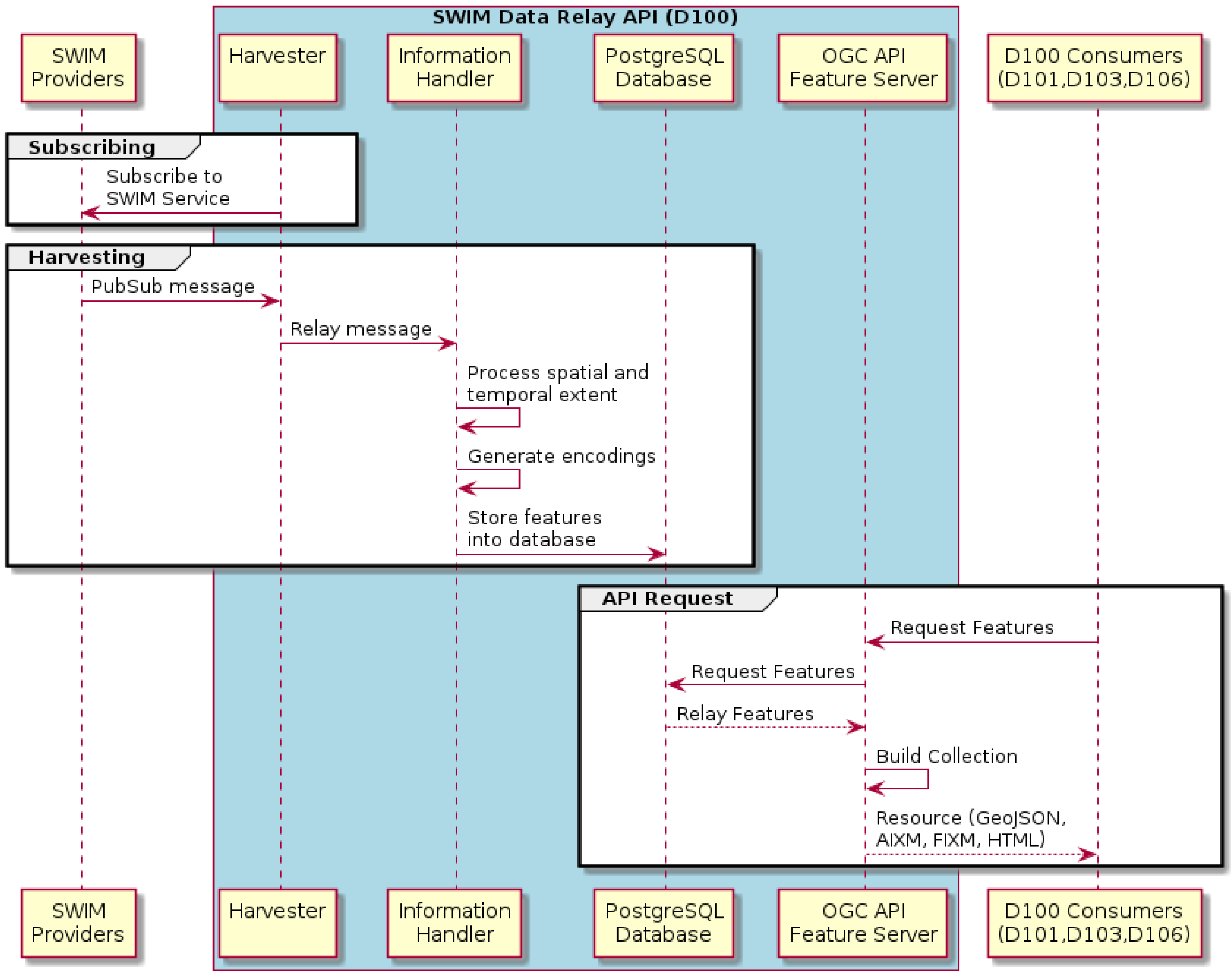


Figure 8. SWIM Data Relay API Subcomponent Interaction

8.2.1. Harvester

SWIM data services are primarily publish/subscribe services for messages that flow from providers to users. Publish/subscribe messaging, or pub/sub messaging, is a form of asynchronous service-to-service communication. In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic. Regular SWIM data consumers first need to subscribe to approved services they may be interested in, then start a client service to receive messages delivered in near real time from those SWIM data services they subscribed to.

The harvester consists of a Solace JMS (Java Message Service) **Message Client**. The JMS message client is responsible for listening to subscribed services in SWIM and relay the incoming messages to the feature handlers that process the messages.

8.2.2. Feature Handler

For each service, there is an information handler application built to analyze and process each service feed. Message handlers are implemented using Java. All these handlers run as services to continuously monitor and analyze incoming messages.

Processing spatial and temporal extent:

In each SWIM message, features have location, coverage, or geometric dimensions, which are used to define the spatial extent of each feature. Each message also includes temporal validation, distribution timestamp, or creation timestamp, which are used to define the temporal extent of each feature described. The handlers extract geospatial information and temporal extent from each message and store spatiotemporal information into the geo-database for indexing and searching.

Specific attributes are also extracted from each feed as required. Selection of attributes depend on the services and the data types to be handled. As an example, flight data may have speed and altitude, while airspace data may have elevation and upper limit of vertical structures. During the harvest stage, some of these native attributes are extracted and elevated as properties in GeoJSON encoding. For example, speed and altitude of a flight from FIXM-NAS flight data are exposed as properties.

Feature description, abstract, and topics are extracted from their schema description, or relevant online document. For example, AIXM features have an online UML model document [http://www.aixm.aero/sites/aixm.aero/files/imce/AIXM51HTML/index.html]. The online document was used to extract relevant information about each feature, such as AIXM data representing airspace [http://aixm.aero/sites/aixm.aero/files/imce/AIXM51HTML/AIXM/Class_Airspace.html]. Title and description are directly extracted from the heading and the table element, respectively. Schema information is extracted from the section for the table of “List of attributes”. These are used in describing each airspace collection.

Generating encodings:

To support different media-types in the SWIM Data Relay API Component, the harvesting stage includes a preliminary encoding process. Each message is converted and encoded in a supported format with native or default projections. The projection may be changed by reprojection in the server of the Feature Server. Additional attributes may be added to combine each message into feature collections. For this Testbed, the component was designed with three types of encodings supported:

1. First, there is a native encoding which depends on the message service that provided the data.

For example, the message may be kept in AIXM, FIXM-NAS, or TFM Data Service schema.

2. Second, a conversion to GeoJSON is performed based on a common model. Spatial and temporal extents are extracted from the original message. Geocoding may be applied to get spatial extent in actual longitude and latitude coordinates. Selected properties may be elevated and extracted as properties from its original message. For example, altitude and speed are extracted for flight and elevation and top limit for airspace. The original message is also converted into JSON and kept completely in a property “json”.
3. Third, the conversion to Text/HTML is done on the fly by using a writer under media negotiation support through JAX-RS service implementation. The conversion is primarily from GeoJSON but adding some necessary JavaScript functions to support map viewer and active links.

8.2.3. Database

The database manages the harvested information as documents. Attributes commonly found in AIXM, FIXM or GeoJSON objects do not have a corresponding table or field in the database. The data schema is treated within each document (feature) encoding. Attributes commonly indexed and frequently searched could be eventually elevated as fields in the database for performance and convenience. PostGIS was used to maintain and index spatial properties. PostgreSQL was used to manage and archive indexes.

To store the same SWIM data in different encodings, the database has an encoding table, where two attributes *"type"* and *"encoding"* pair the data to its corresponding different encodings. The field *"type"* uses the actual media type supported by the service. The media types are currently (or proposed to be) registered with the Internet Assigned Numbers Authority (IANA).

The metadata is stored in a collection table (`Collections`) and a linked encoding table (`CollectionEncoding`). For example, title and description for a collection are stored in corresponding fields in the collection table. The "list of attributes" are extracted, re-formatted, encoded, and stored in the encoding table by linking to the collection identifier. The pair of `type` and `encoding` has the similar relationship as that used in feature encoding table, i.e. type is media type and encoding store the stubs of collection class in related encoding type.

Reusable and common attributes for a collection are directly stored in the `Collection` table while details and customized attributes are encoded and stored in the `CollectionEncoding` table. With the current implementation, customized attributes are only available for AIXM feature because that is the only one that has corresponding pages to scrape information from. Only text/html encoding is stored for now. Because of this limitation, the details in the encoding table are not necessarily available/used during the serving stage for the current implementation.

Figure 9 shows the Entity-Relationship Diagram of the SWIM Data Relay API Component.

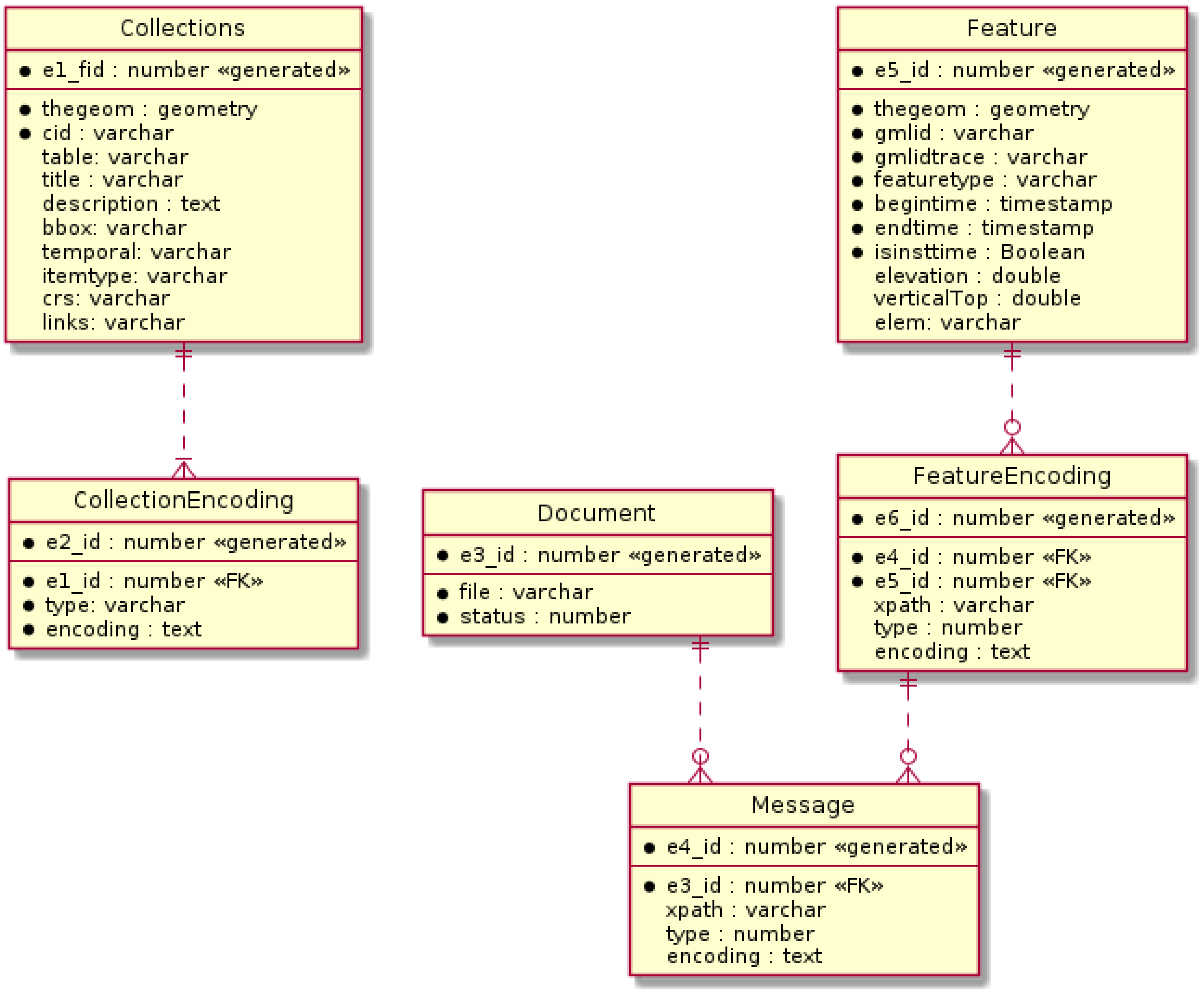


Figure 9. Entity-Relationship Diagram of the SWIM Data Relay API Component

8.2.4. OGC API - Features Implementation

An OGC API Feature Service for SWIM was deployed providing RESTful endpoints for consumers to access SWIM data. The OGC 17-069r3 (OGC API – Features – Part 1: Core) Standard was used in this Testbed. Two types of resources are managed by the OGC API - Features Service:

- **Metadata**, primarily used to describe each feature collection in the OGC API Feature Service.
- **Data**, which can either be retrieved by collection or individually.

Available media types include HTML, AIXM, FIXM, and GeoJSON. Geometries are represented in the WGS 84 Coordinate Reference System (CRS) with axis order longitude/latitude.

Media type negotiation was implemented following the IETF RFC 2616 HTTP specification, specifically Section 14.1 [https://tools.ietf.org/html/rfc2616#section-14.1]. If a RESTful client is used, the client can specify alternative media types supported by the service and retrieve the data as requested. Client programs should negotiate with the proper media type by adding a proper `Accept` header in its request. This mediation negotiation rule is applicable to any of the negotiable paths in the OGC API Feature service.

Each endpoint provides a dedicated path for each supported media type. In general, the pattern takes the extension of contracted sub-media type registered in IANA media type registry [http://www.iana.org/assignments/media-types/media-types.xhtml]. For example, the media type "application/json" will have an extension of “.json”. For collections, the type would have the path of "/collections.json". When subtype is combination of two types, the “+” will be replaced with “.”. For example, media type "application/geo+json" would have an extension of “.geo.json”.

Whenever a collection of features is requested from the OGC API - Features service, a unique identifier is generated on-the-fly by combining the service UUID and a sequence number for each record/feature/message. This process is performed during the “combining” stage when a feature collection is created from pieces of features matching the search criteria. The combining process is the process of adding each feature to a feature collection. Each feature is attributed with the generated unique identifier. A detailed description of this challenge is presented in the Challenges Section.

Appendix A describes an example of a FIXM message converted into a GeoJSON feature and its encoding of `application/fixm+nas+xml;version=3.0` in OGC API - Features output.

The Feature Server was designed with seven endpoints:

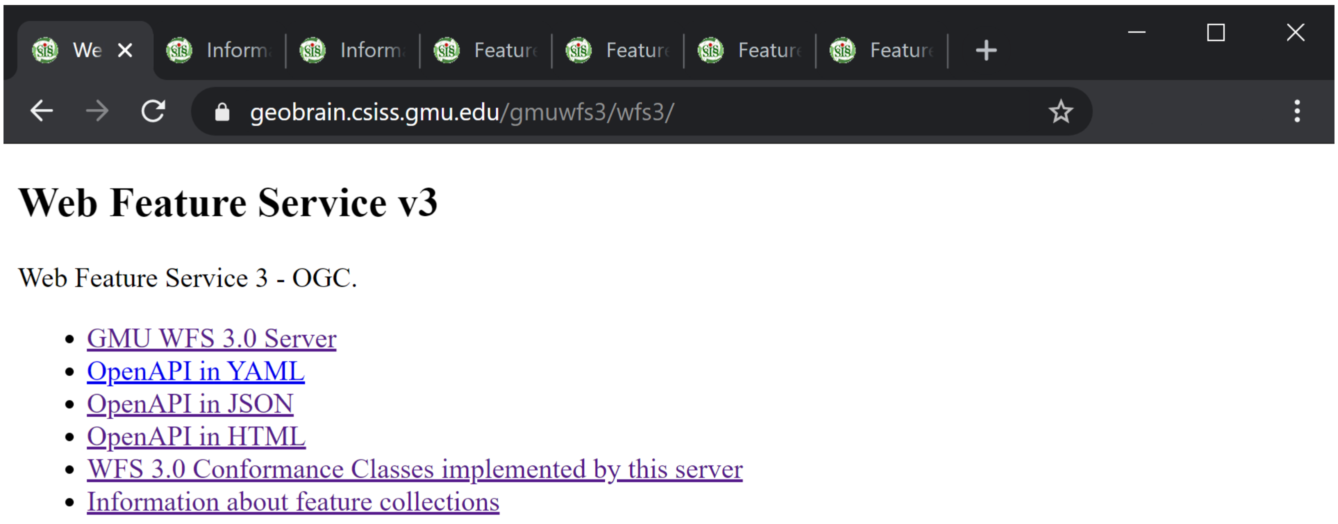
- UC1 - Get the landing page
- UC2 - Get descriptions of collections
- UC3 - Describe a collection
- UC4 - Get features in a collection
- UC5 - Retrieve features in a collection intersecting a bounding box
- UC6 - Retrieve features in a collection within a temporal range
- UC7 - Retrieve a specific feature by identifier

8.2.4.1. UC1 - Get the landing page

The landing page of the OGC API - Features Service provides information about the API in YAML or JSON, conformance, and link to collections served in this service. Table 1 lists the path and possible encoding negotiation. Figure 10 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 1. Get the landing page of the OGC API - Features Service

Use Case	**UC1: Retrieve the landing page**	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3	
Path	/	
Header	"Accept"	"text/html"
Header	"Accept"	"application/json"



Web Feature Service v3

Web Feature Service 3 - OGC.

- GMU WFS 3.0 Server
- OpenAPI in YAML
- OpenAPI in JSON
- OpenAPI in HTML
- WFS 3.0 Conformance Classes implemented by this server
- Information about feature collections

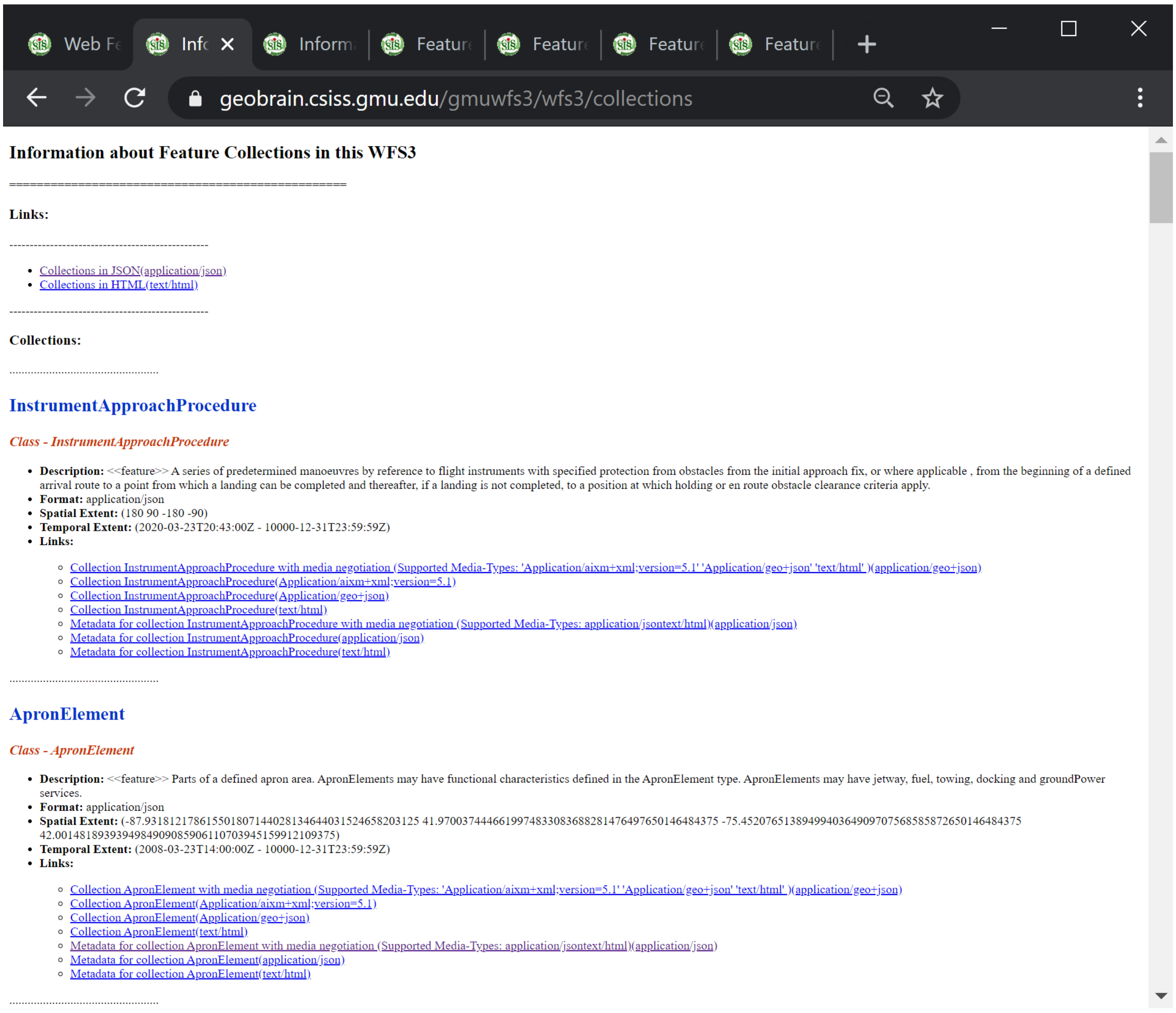
Figure 10. Landing Page of D100

8.2.4.2. UC2 - Get descriptions of collections

Table 2 lists the path for retrieving collections and possible encodings for negotiation. Figure 11 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 2. Get Collection Descriptions of the OGC API - Features Service

Use Case	**UC2: Get descriptions of collections**	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/	
Path	/collections	
Header	"Accept"	"text/html"
Header	"Accept"	"application/json"



Information about Feature Collections in this WFS3

===

Links:

- Collections in JSON(application/json)
- Collections in HTML(text/html)

Collections:

....

InstrumentApproachProcedure

Class - InstrumentApproachProcedure

- **Description:** <<feature>> A series of predetermined manoeuvres by reference to flight instruments with specified protection from obstacles from the initial approach fix, or where applicable , from the beginning of a defined arrival route to a point from which a landing can be completed and thereafter, if a landing is not completed, to a position at which holding or en route obstacle clearance criteria apply.
- **Format:** application/json
- **Spatial Extent:** (180 90 -180 -90)
- **Temporal Extent:** (2020-03-23T20:43:00Z - 10000-12-31T23:59:59Z)
- **Links:**
  - Collection InstrumentApproachProcedure with media negotiation (Supported Media-Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html')(application/geo+json)
  - Collection InstrumentApproachProcedure(Application/aixm+xml;version=5.1)
  - Collection InstrumentApproachProcedure(Application/geo+json)
  - Collection InstrumentApproachProcedure(text/html)
  - Metadata for collection InstrumentApproachProcedure with media negotiation (Supported Media-Types: application/jsontext/html)(application/json)
  - Metadata for collection InstrumentApproachProcedure(application/json)
  - Metadata for collection InstrumentApproachProcedure(text/html)

....

ApronElement

Class - ApronElement

- **Description:** <<feature>> 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.
- **Format:** application/json
- **Spatial Extent:** (-87.931812178615501807144028134644031524658203125 41.9700374446619974833083688281476497650146484375 -75.4520765138949940364909707568585872650146484375 42.0014818939394984909085906110703945159912109375)
- **Temporal Extent:** (2008-03-23T14:00:00Z - 10000-12-31T23:59:59Z)
- **Links:**
  - Collection ApronElement with media negotiation (Supported Media-Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html')(application/geo+json)
  - Collection ApronElement(Application/aixm+xml;version=5.1)
  - Collection ApronElement(Application/geo+json)
  - Collection ApronElement(text/html)
  - Metadata for collection ApronElement with media negotiation (Supported Media-Types: application/jsontext/html)(application/json)
  - Metadata for collection ApronElement(application/json)
  - Metadata for collection ApronElement(text/html)

....

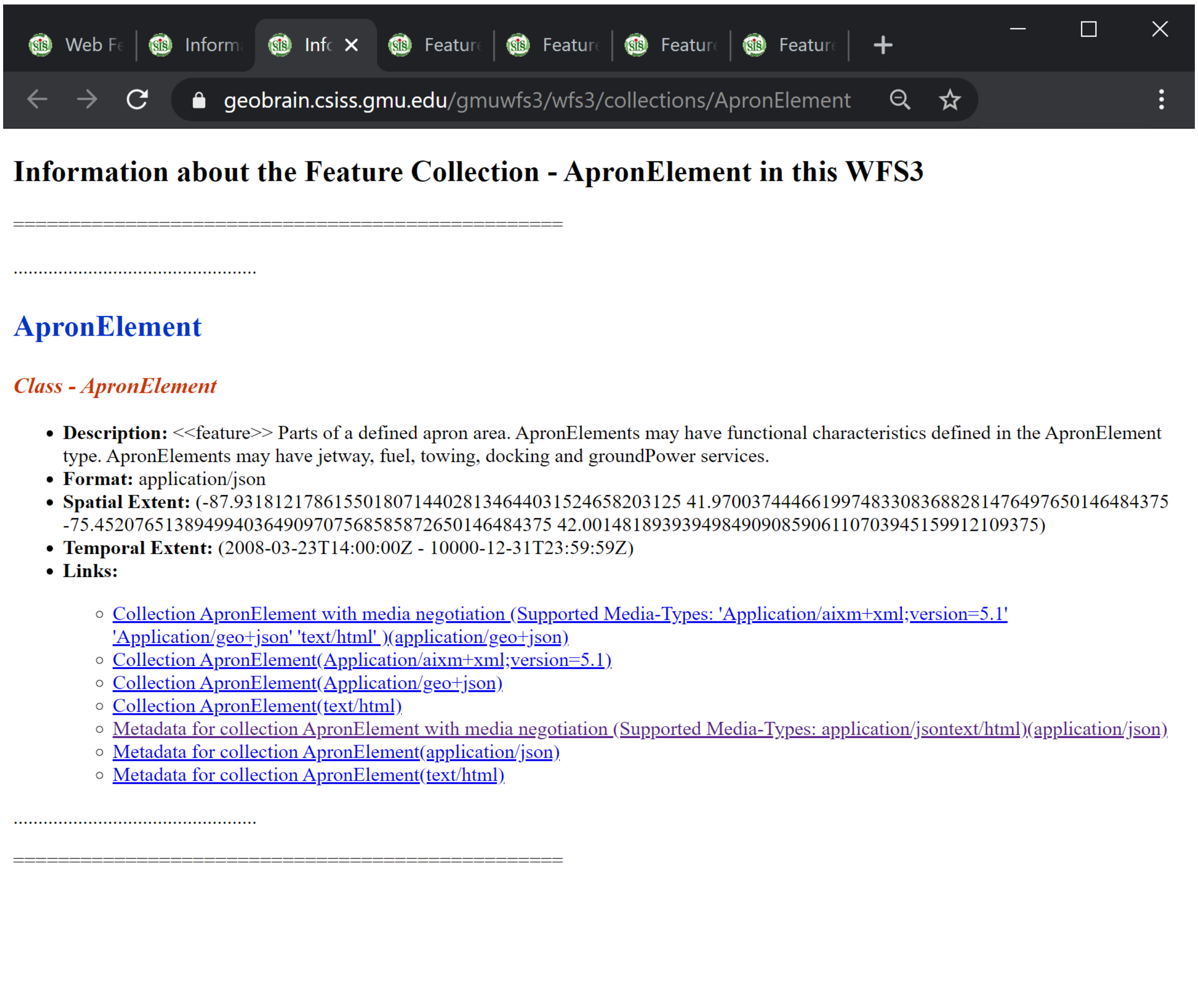
Figure 11. Getting the Description of all Collections in D100

8.2.4.3. UC3 - Describe a collection

The metadata for a specific collection can be retrieved by using a collection identifier. Table 3 lists the path for retrieving collection metadata and possible encodings for the negotiation. Figure 12 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 3. Get metadata for a specific collection in the OGC API - Features Service

Use Case	UC3: Get the metadata for a specific collections	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/	
Path	/collections/{CollectionIdentifier}	
Example Path	/collections/ApronElement	
Header	"Accept"	"text/html"
Header	"Accept"	"application/json"


Web Fe
Inform
Info
Feature
Feature
Feature
Feature
geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/ApronElement

Information about the Feature Collection - ApronElement in this WFS3

==

....

ApronElement

Class - ApronElement

- **Description:** <<feature>> 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.
- **Format:** application/json
- **Spatial Extent:** (-87.931812178615501807144028134644031524658203125 41.9700374446619974833083688281476497650146484375 -75.4520765138949940364909707568585872650146484375 42.0014818939394984909085906110703945159912109375)
- **Temporal Extent:** (2008-03-23T14:00:00Z - 10000-12-31T23:59:59Z)
- **Links:**
  - Collection ApronElement with media negotiation (Supported Media-Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html')(application/geo+json)
  - Collection ApronElement(Application/aixm+xml;version=5.1)
  - Collection ApronElement(Application/geo+json)
  - Collection ApronElement(text/html)
  - Metadata for collection ApronElement with media negotiation (Supported Media-Types: application/jsontext/html)(application/json)
  - Metadata for collection ApronElement(application/json)
  - Metadata for collection ApronElement(text/html)

....

==

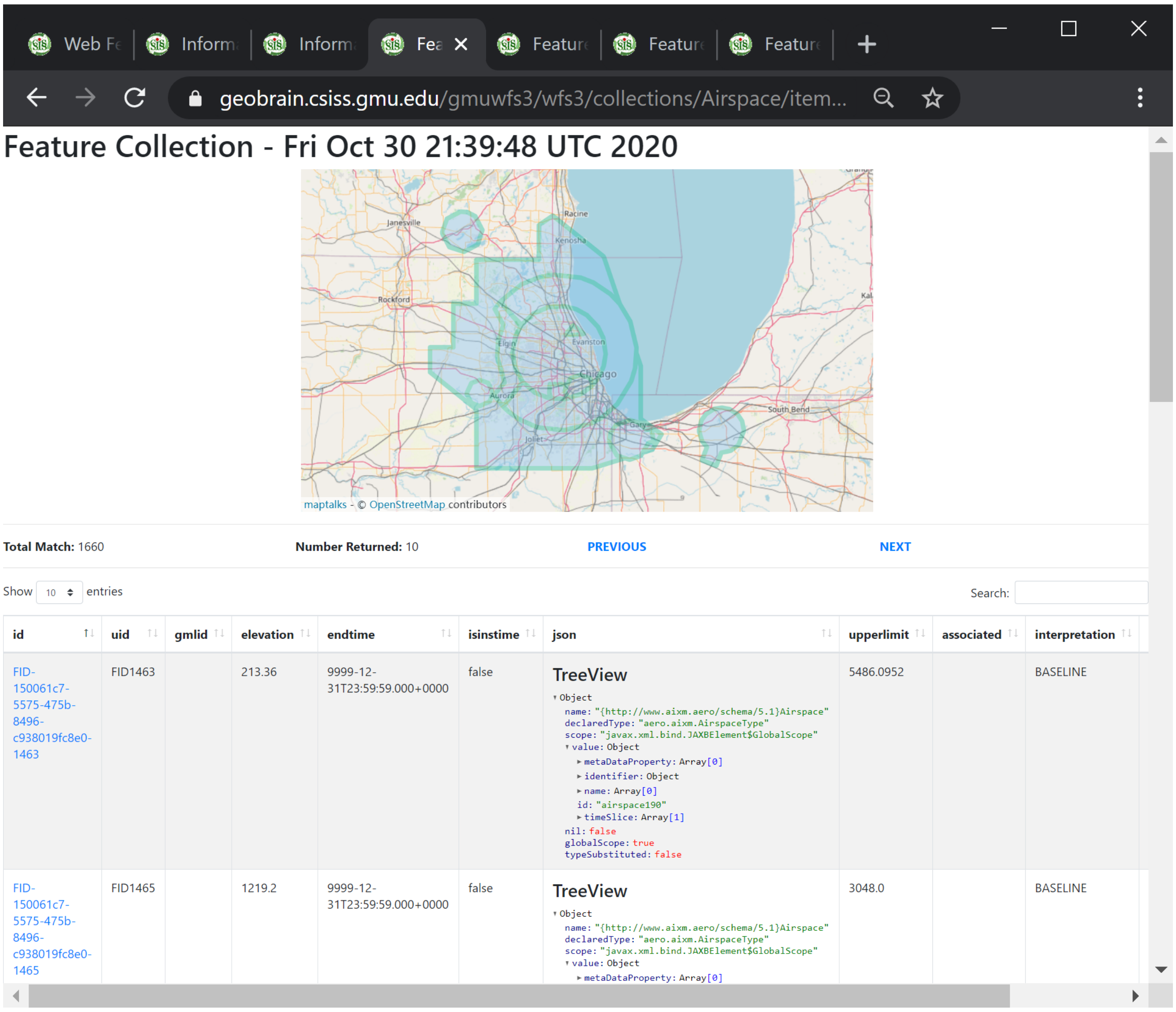
Figure 12. Viewing the Metadata of a Specific Collection in D100

8.2.4.4. UC4 - Get features in a collection

Features in a specific collection can be retrieved by using collection identifier. Table 4 lists the path for retrieving features in a collection and possible encodings for the negotiation. Figure 13 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html". Pagination is applied with default value of 10 items per page. The maximum per page is 10000.

Table 4. Get features of a specific collection in the OGC API - Features Service

Use Case	UC4: Get features of a specific collections	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/	
Path	/collections/{CollectionIdentifier}/items	
Example Path	/collections/Airspace/items	
Header	"Accept"	"text/html"
Header	"Accept"	"application/geo+json"
Header	"Accept"	"application/aixm+xml;version=5.1"



Feature Collection - Fri Oct 30 21:39:48 UTC 2020

Total Match: 1660 **Number Returned:** 10 PREVIOUS NEXT

Show 10 entries Search:

id	uid	gmlid	elevation	endtime	isinstime	json	upperlimit	associated	interpretation
FID-150061c7-5575-475b-8496-c938019fc8e0-1463	FID1463		213.36	9999-12-31T23:59:59.000+0000	false	TreeView ▾Object name: "{http://www.aixm.aero/schema/5.1}Airspace" declaredType: "aero.aixm.AirspaceType" scope: "javax.xml.bind.JAXBElement$GlobalScope" ▾value: Object ▸metaDataProperty: Array[0] ▸identifier: Object ▸name: Array[0] id: "airspace190" ▸timeSlice: Array[1] nil: false globalScope: true typeSubstituted: false	5486.0952		BASELINE
FID-150061c7-5575-475b-8496-c938019fc8e0-1465	FID1465		1219.2	9999-12-31T23:59:59.000+0000	false	TreeView ▾Object name: "{http://www.aixm.aero/schema/5.1}Airspace" declaredType: "aero.aixm.AirspaceType" scope: "javax.xml.bind.JAXBElement$GlobalScope" ▾value: Object ▸metaDataProperty: Array[0]	3048.0		BASELINE

Figure 13. Viewing all the Features of a Specific Collection in D100

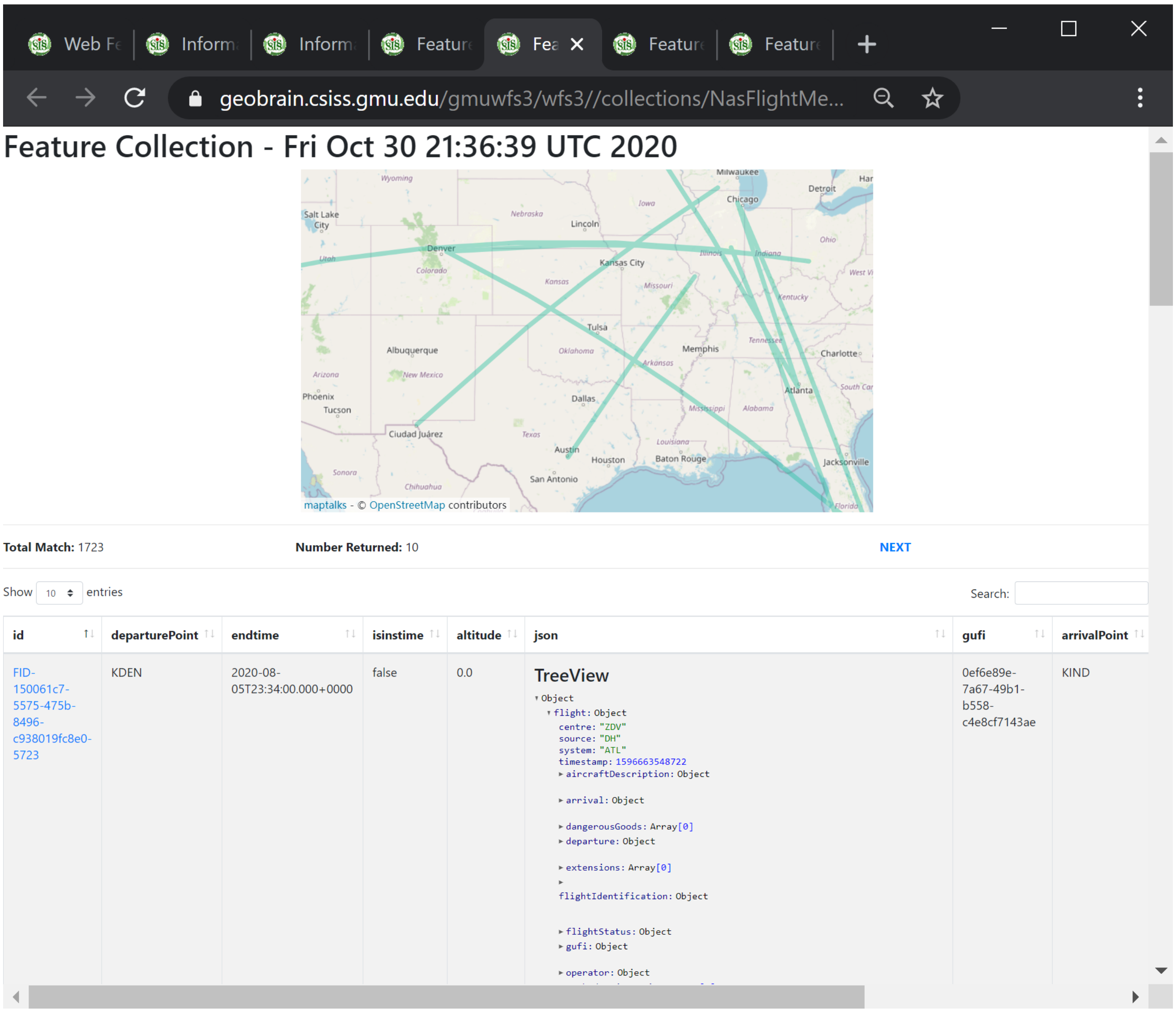
8.2.4.5. UC5 - Retrieve features in a collection intersecting a bounding box

Spatial filtering can be applied to features in a specific collection. Table 5 lists all parameters supported at feature retrieval and gives an example with bounding box restrictions. Figure 14 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 5. Get features of a specific collection in the OGC API - Features Service

Use Case	**UC5: Get features of a specific collections with bouding box**	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/	
Path	/collections/{CollectionIdentifier}/items	
Parameters	limit	limit per page
	bbox	bounding box
	datetime	temporal extent
Example Path	/collections/NasFlightMessage/items?bbox=-91.513079,36.970298,-87.494756,42.508481	

Use Case	UC5: Get features of a specific collections with bouding box	
Header	"Accept"	"text/html"
Header	"Accept"	"application/geo+json"
Header	"Accept"	"application/fixm+nas+xml;version=3.0"


Web Fe
Inform
Inform
Feature
Fea
Feature
Feature
geobrain.csiss.gmu.edu/gmuwfs3/wfs3//collections/NasFlightMe...
maptalks - © OpenStreetMap contributors
NEXT
Show 10 entries
Search:

Feature Collection - Fri Oct 30 21:36:39 UTC 2020

Total Match: 1723 **Number Returned:** 10

id	departurePoint	endtime	isinstime	altitude	json	gufi	arrivalPoint
FID-150061c7-5575-475b-8496-c938019fc8e0-5723	KDEN	2020-08-05T23:34:00.000+0000	false	0.0	TreeView ▾ Object ▾ flight: Object centre: "ZDV" source: "DH" system: "ATL" timestamp: 1596663548722 ▸ aircraftDescription: Object ▸ arrival: Object ▸ dangerousGoods: Array[0] ▸ departure: Object ▸ extensions: Array[0] ▸ flightIdentification: Object ▸ flightStatus: Object ▸ gufi: Object ▸ operator: Object	0ef6e89e-7a67-49b1-b558-c4e8cf7143ae	KIND

Figure 14. Viewing Features Intersecting a Bounding Box in D100

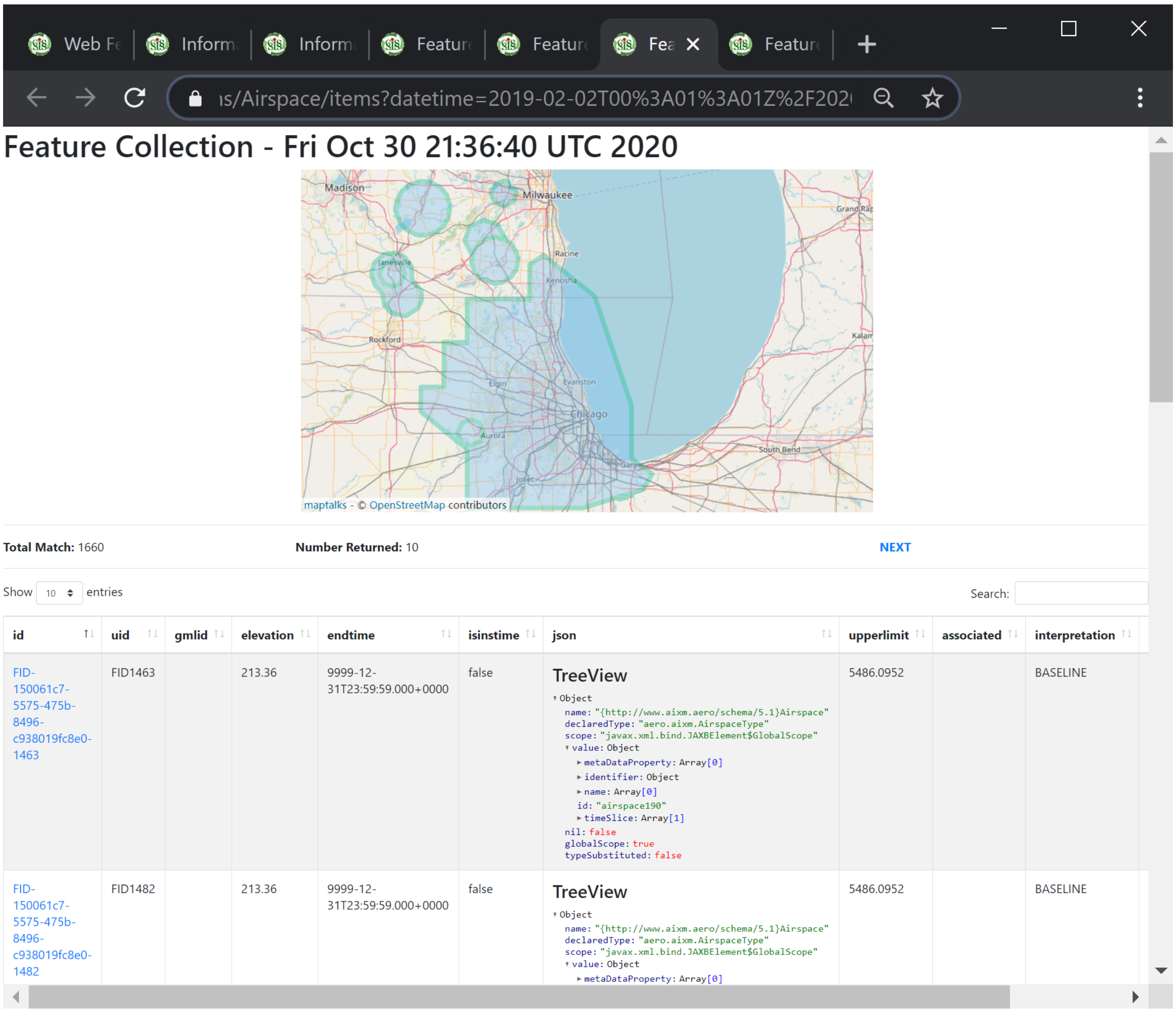
8.2.4.6. UC6 - Retrieve features in a collection within a temporal range

Temporal filtering can be applied to features in a specific collection. Table 6 lists all parameters supported at feature retrieval and gives an example with a temporal range between `2019-02-02T00:01:01Z` and `2020-08-08T23:59:59Z`. Figure 15 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 6. Get features of a specific collection in the OGC API - Features Service

Use Case	UC6: Get features of a specific collection within a temporal range
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/
Path	/collections/{CollectionIdentifier}/items

Use Case	UC6: Get features of a specific collection within a temporal range	
Parameters	limit	limit per page
	bbox	bounding box
	datetime	temporal extent
Example Path	/collections/Airspace/items?datetime=2019-02-02T00%3A01%3A01Z%2F2020-08-08T23%3A59%3A59Z	
Header	"Accept"	"text/html"
Header	"Accept"	"application/geo+json"
Header	"Accept"	"application/aixm+xml;version=5.1"



Feature Collection - Fri Oct 30 21:36:40 UTC 2020

Total Match: 1660 **Number Returned:** 10

id	uid	gmlid	elevation	endtime	isinstime	json	upperlimit	associated	interpretation
FID-150061c7-5575-475b-8496-c938019fc8e0-1463	FID1463		213.36	9999-12-31T23:59:59.000+0000	false	TreeView	5486.0952		BASELINE
FID-150061c7-5575-475b-8496-c938019fc8e0-1482	FID1482		213.36	9999-12-31T23:59:59.000+0000	false	TreeView	5486.0952		BASELINE

```
▾ Object
    name: "{http://www.aixm.aero/schema/5.1}Airspace"
    declaredType: "aero.aixm.AirspaceType"
    scope: "javax.xml.bind.JAXBElement$GlobalScope"
  ▾ value: Object
    ▸ metaDataProperty: Array[0]
    ▸ identifier: Object
    ▸ name: Array[0]
      id: "airspace190"
    ▸ timeSlice: Array[1]
    nil: false
    globalScope: true
    typeSubstituted: false
```

```
▾ Object
    name: "{http://www.aixm.aero/schema/5.1}Airspace"
    declaredType: "aero.aixm.AirspaceType"
    scope: "javax.xml.bind.JAXBElement$GlobalScope"
  ▾ value: Object
    ▸ metaDataProperty: Array[0]
```

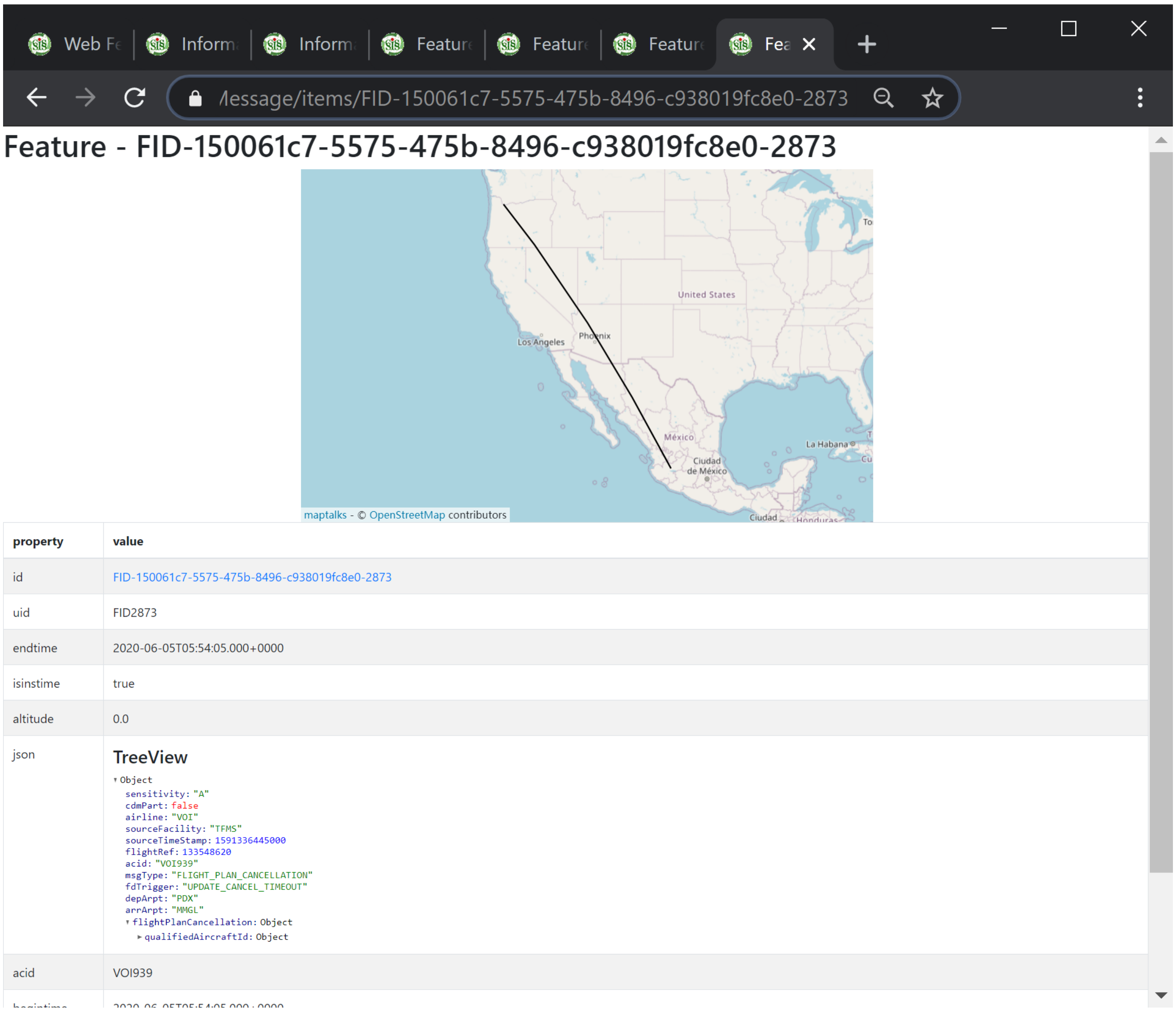
Figure 15. Viewing Features Within a Temporal Range in D100

8.2.4.7. UC7 - Retrieve a specific feature by identifier

If the identifier of a feature is known, it can be used to retrieve the feature. Table 7 lists the path to request a specific feature by its identifier and provides an example path for retrieving a `fltdMessage` (flight data message) feature with the identifier `FID-150061c7-5575-475b-8496-c938019fc8e0-2873`. Figure 16 shows a screen capture of the output in HTML when the request is negotiated as media type "text/html".

Table 7. Get a specific feature in the OGC API - Features Service

Use Case	**UC7: Get a specific feature**	
End point	https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/	
Path	/collections/{CollectionIdentifier}/items/{FeatureIdentifier}	
Example Path	/collections/fltdMessage/items/FID-150061c7-5575-475b-8496-c938019fc8e0-2873	
Header	"Accept"	"text/html"
Header	"Accept"	"application/geo+json"
Header	"Accept"	"application/tfmdataservice+xml;version=3.1"



Feature - FID-150061c7-5575-475b-8496-c938019fc8e0-2873

property	value
id	FID-150061c7-5575-475b-8496-c938019fc8e0-2873
uid	FID2873
endtime	2020-06-05T05:54:05.000+0000
isinstime	true
altitude	0.0
json	TreeView ▾ Object sensitivity: "A" cdmPart: false airline: "VOI" sourceFacility: "TFMS" sourceTimeStamp: 1591336445000 flightRef: 133548620 acid: "VOI939" msgType: "FLIGHT_PLAN_CANCELLATION" fdTrigger: "UPDATE_CANCEL_TIMEOUT" depArpt: "PDX" arrArpt: "MMGL" ▾ flightPlanCancellation: Object ▸ qualifiedAircraftId: Object
acid	VOI939

Figure 16. Viewing a Specific Feature in D100

NOTE

An identifier is only available in the `geojson` and `text/html` responses of this API. An identifier in native SWIM format may be difficult to define. The solution depends on the data format in the response. For example, for media type `"application/aixm+xml;verson=5.1"`, the `gmlid` of `TimeSlice` element is normally an acceptable identifier. The identifier is also added as an element in each time slice of features. Further details on the implementation are discussed in the Challenges section.

8.3. Challenges and Lessons Learned

The following summarize major challenges during the design and implementation of the OGC API - Features Service for SWIM data.

- **What is a feature in SWIM for the OGC API - Features service?** The SWIM cloud distribution service is implemented to deliver data through the Solace JMS (Java Message Service) messaging. Data needs to be mapped into features in order to make them available through an OGC API - Features, and this mapping is not a simple one-to-one match because a message may contain several features with different spatial and temporal ranges. The identification of mappable features from SWIM services needs to be considered carefully case by case.

  For AIXM features, as identified in previous Testbeds and initiatives, time slices may be treated as the smallest subset of features to be discovered and retrieved in the feature service. For FIXM features, individual message (for example, flight position) may be seen as features to be managed. [10] [11]

- **Unique identification of a feature**: Uniquely identifying a feature in the OGC API - Features Service is important to support search and retrieval. Previous work reported for AIXM features that the change of its identifier `gml:id` by using a format of `[▯uuid▯]. [UUID].[Timeslice Interpretation Code].[sequenceNumber].[correctionNumber]` could not guarantee uniqueness in the feature store [11]. For this Testbed, the original `gml:id` was kept intact and indexed since messages might have been cross-referenced using these IDs. This proved to be easier to implement since the sequence number `FID` was already used in uniquely identifying records in the back-end database.

  The OGC API service was planned to deliver the data in native, domain-specific encodings. For example, AIXM, FIXM, and traffic message data service format of FAA may be suitable for domain-specific applications. These encodings follow a well-defined schema that may restrict the generation of unique identifiers for the OGC API Service. The SWIM Data Relay API Component was built implementing the following strategies to maintain a unique identifier for each feature:

  - For outputs in `geojson` or `text/html`, a unique identifier was generated by using the UUID of the service plus the unique number within the service. For example, the service created in this Testbed has a service UUID as `150061c7-5575-475b-8496-c938019fc8e0`. As an example, a unique feature ID, `FID-150061c7-5575-475b-8496-c938019fc8e0-2859` is formed by using template `FID-{UUID}-{Seq#}` where the sequence number is `2859`.
  - For native formats, a unique identifier was added to each feature through an additional or

unused element under the schema.

  - For example, in AIXM features, the generated unique identifier was added to the generally-not-used `name` element without violating the schema. To distinguish it from other identifier/name, a special `codeSpace` attribute was defined as http://www.opengis.net/ogcapi-features-1/1.0/aixm/id. An example of an identifier is `<gml:name codeSpace="http://www.opengis.net/ogcapi-features-1/1.0/aixm/id">FID-150061c7-5575-475b-8496-c938019fc8e0-2859</gml:name>`.
  - For FIXM-NAS message, a `gumi` metadata element was added for each message. For example, `<metadata gumi="FID-150061c7-5575-475b-8496-c938019fc8e0-5728" xsi:type="nas:MessageMetadataType"/>` was added as a first child element to a FIXM-NAS message.

- **Spatial extent in a feature collection**: The spatial extent of a collection may increase as each message is received and data is accumulated in the database. By doing so, the spatial extent of each collection reflects the actual spatial coverage.

  Figure 17 shows an example of updating the spatial extent for the feature collection *Apron Element* during the harvesting stage. The collection is initiated with the first harvested element. The extent is assigned to be the same as that of element 1, i.e. `(-87.9011,41.9545,-87.8985,41.9568)`. When feature 2 in the collection *Apron Element* is added, the spatial extent is expanded to cover both feature 1 and 2, i.e. `(-87.9048,41.9545,-87.8985,41.9664)`. The spatial extent for feature collection Apron Element gets updated as new features are harvested.

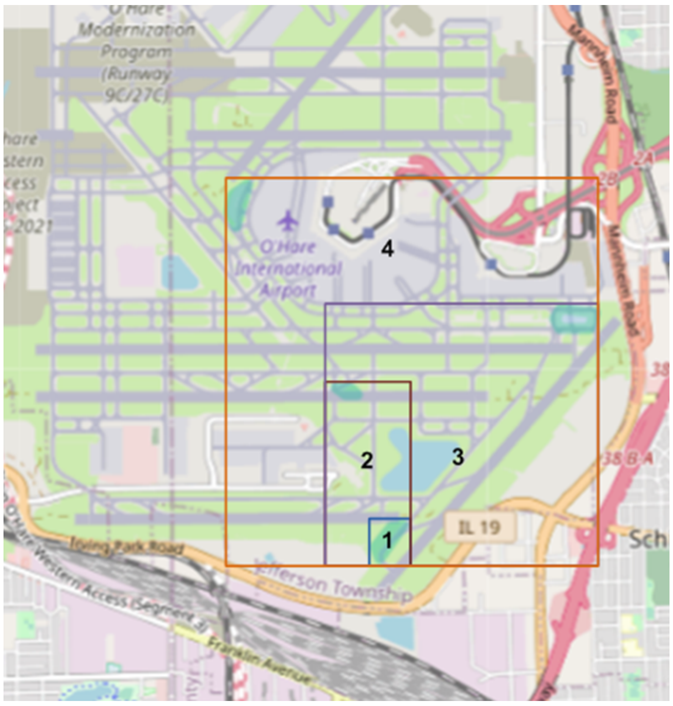


Figure 17. Spatial Extent Expansion During the Harvesting Stage

- **Temporal extent in a feature collection**: When a SWIM message is received, features are extracted and harvested into a backend geospatially-aware database for spatial/temporal indexing. A collection is also picked up when any feature from that collection is received. The temporal extent of a collection is first set based on the first feature in that collection received. When more features from that collection are received, the temporal ranges of the collection are expanded based on the minimum and maximum temporal extent values of the entire collection.

- **Spatial information in a feature**: Some SWIM data may not have explicit geometry properties but rather named places or locations, requiring geocoding to add or extract geometry properties for each feature. Two of these situations were encountered during this Testbed:
    - One geometry property of a flight feature may be represented as the trajectory from the departure airport to the destination airport (or the great circle if no waypoints are available). Flight data from SWIM data services may only include the codes of the departure and arrival airports, and not its coordinates. To generate the geometry property for two given departure and arrival airports, the Harvester takes three steps to geocode and interpolate the trajectory, before saving it in the database encoded in GeoJSON:
        1. First, one of the most frequently updated airport databases, ourairports [https://ourairports.com/data/airports.csv], is used to retrieve the codes and coordinates of over 57,000 airports worldwide.
        2. Second, a script maps inside the database the airport code coming from SWIM data with the IATA code, GPS code, or any recognized local code (including 3 letter ICAO code) of the database.
        3. Finally, geographic interpolation is used to generate the approximate trajectory using the *GeographicLib* library for Java.
    - Another type of named geometry property in SWIM data may be expressed as the affected area by a given radius. The location may be an airport or a location. As an example, the extension of features of type *Event* in AIXM often describe an event that affect an area with a given radius.

      The Harvester may use geocoding to extract the actual coordinates of the center. Additionally, because GeoJSON does not support circles or curves, the Harvester would use an interpolated circular area (polygon) to approximate the spatial coverage. GeoJSON encoding of curves is less precise than GML and may consume as much as ten times of storage space. The geocoding program first looked for the location to convert to latitude and longitude. If that failed, the program would check if there was any given coordinate in the original dataset. The reason to put the geocoding of a location as the preferred location was that the original coordinates seemed off due to precision issues.
- **Limited semantic descriptions of feature collections and features**: The semantic description of each collection and feature is limited. A proper ontology to describe each collection may not be available. The implementation of the OGC API - Features Service for SWIM data took the following measures to augment features semantically:
    1. Geocoding was used to add spatial contextual information;
    2. online descriptions and documents of features were used to add descriptions and schema for each collection; and
    3. units for distance and speed are unified by using ISO standard units.

Further work is needed to add more semantic descriptions to each collection and feature. Controlled vocabularies and ontologies should be further developed. Descriptions for each collection with a well-defined ontology should be further developed.

- **Large data volume**: Due to the large amount of flight, weather, airport and ground data transferred, more than 4 terabytes of data are estimated to be distributed daily through the FAA SWIM SCDS services [12]. The implementation of the SWIM Data Relay API Component supported a permanent ingestion and storage of SWIM data on its database. However, since the actual volume exchanged is beyond the capacity of the testing server used in this Testbed to store the vast amount of data. Therefore, for the purposes of testing the harvesting service, the service only ran for as long as two days to collect enough testing datasets to demonstrate the capabilities of the OGC API - Features Service.

A more scalable database may be used, such as *Accumulo*, *HBase*, or *GeoMesa*. Another aspect to be considered could be for how long the data needs to be kept. A considerable volume of SWIM data consists of status data that are valid only for certain periods. A time limit could be defined for this data to be removed after the limit is surpassed.

- **Automatic code generation is not perfect**: One of the benefits of adopting the OGC API approach is the capability of automatically generating code for both client and server applications in many popular frameworks and programming languages. This implementation used the OpenAPI Generator [https://github.com/OpenAPITools/openapi-generator] to generate server-side code stubs using the OpenAPI in YAML [https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.yaml]. The general Java JAX-RS RESTful framework was chosen to create the server. The code was generated successfully, adding convenience in the implementation of the server. However, the following downsides of this approach were detected:
    1. For JAX-RS code, the handling of the property `OneOf` is not supported and properly generated. A workaround and fix-up should be applied.
    2. The parameter of `bbox` was generated with a dimension constraint of either 4 or 6 elements. This constraint made the `bbox` parameter not to be optional, which was not desired. To make the parameter optional, the solution was either to remove the constraint or to specify the constraint with looser values like (min=0, max=6). In the final implementation, the constraint was removed.
    3. For `bbox` parameter, the generated code stub uses repeated parameter `bbox` to pass an array of `BigDecimal` values - 4 repeats for 2-D `bbox` and 6 repeats for 3-D `bbox`. This is not commonly preferred to pass `bbox`. The commonly preferred `bbox` is passed to the server using a comma-separated array. In the implementation, a string of comma-separated values is accepted and parsed to get bounding box.

8.4. Accomplishments

- Exposed features from SWIM data services: OGC API Feature Core was implemented in Testbed 16. Selected data and services from SWIM are proxied through the OGC API - Features service. These include AIXM Features from the AIM FNS for Notice to Airmen (NOTAMs), FIXM features from the SWIM Flight Data Publication Service (SFDPS), and the flight information from the SWIM TFMData Service. These provide information on flights, flight status, ground delay program (GDP), etc., which were necessary to answer the questions in the Testbed 16 demonstration scenarios.
- Spatial and temporal filtering: Features were populated with spatial and temporal properties. This enables the discovery of features by spatial and temporal constraints.
- Encoding: The OGC API - Features Service expands the data distribution encodings. Not only

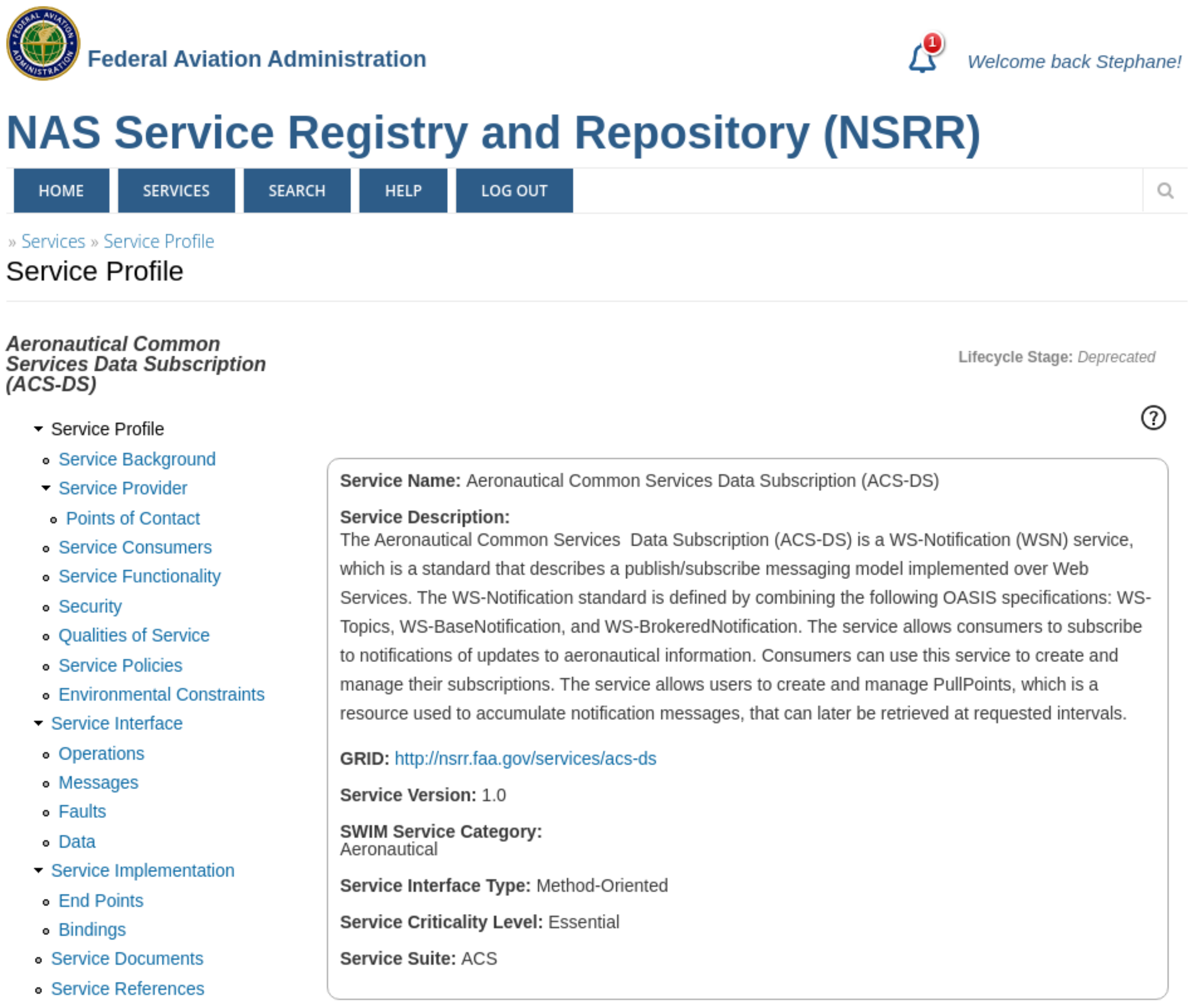
native aviation-domain-specific encodings (AIXM, FIXM, TFM Data Service encoding) are supported, but also well-accepted GeoJSON and HTML are supported. GeoJSON is good for data distribution. HTML is good for instantly previewing and examining data in depth.

Chapter 9. Semantic Registry

The Semantic Registry component was built to enable the search and discovery of Aviation-related assets (services, datasets, vocabularies, etc) using metadata about these assets. The Semantic Registry was designed to harvest semantic metadata information from SWIM Service Registries, as well as OGC APIs serving SWIM data. Assets such as Services and Datasets are searched and discovered by receiving search requests from the Triple Builder and the SWIM Data Client using either a REST API or a GraphQL API. The matching asset metadata records are sent back to the requesting component, which uses the access information to query and retrieve the content of the datasets. The development of Semantic Registry was carried out by Image Matters LLC.

9.1. Status Quo

In order to connect to aviation data services to extract information, users first need to know which services are available and what information each service provides. The NAS Service Registry and Repository (NSRR) is an online service for storing, sharing, and managing information about all SWIM-enabled services, both currently available and under development. As seen on Figure 18, the exploration of services is mainly performed manually through the NSRR website. Automation of this exploratory process has the potential of fostering the development and interconnectivity of applications based on the consumption of SWIM data. For this reason, this automation has been the subject of numerous research endeavors in the past.


Federal Aviation Administration
Welcome back Stephane!
NAS Service Registry and Repository (NSRR)
HOME
SERVICES
SEARCH
HELP
LOG OUT
» Services » Service Profile

Service Profile

Aeronautical Common Services Data Subscription (ACS-DS)

Lifecycle Stage: *Deprecated*

- Service Profile
  - Service Background
  - Service Provider
    - Points of Contact
  - Service Consumers
  - Service Functionality
  - Security
  - Qualities of Service
  - Service Policies
  - Environmental Constraints
- Service Interface
  - Operations
  - Messages
  - Faults
  - Data
- Service Implementation
  - End Points
  - Bindings
- Service Documents
- Service References

Service Name: Aeronautical Common Services Data Subscription (ACS-DS)

Service Description:
The Aeronautical Common Services Data Subscription (ACS-DS) is a WS-Notification (WSN) service, which is a standard that describes a publish/subscribe messaging model implemented over Web Services. The WS-Notification standard is defined by combining the following OASIS specifications: WS-Topics, WS-BaseNotification, and WS-BrokeredNotification. The service allows consumers to subscribe to notifications of updates to aeronautical information. Consumers can use this service to create and manage their subscriptions. The service allows users to create and manage PullPoints, which is a resource used to accumulate notification messages, that can later be retrieved at requested intervals.

GRID: http://nsrr.faa.gov/services/acs-ds

Service Version: 1.0

SWIM Service Category:
Aeronautical

Service Interface Type: Method-Oriented

Service Criticality Level: Essential

Service Suite: ACS

Figure 18. SWIM Service Description in the NSRR Website

Testbed-16 leveraged the Testbed-13 work that defined a draft Semantic Registry REST API specification and the Semantic Registry Information Model (SRIM). These specifications were based on a generalization of DCAT 1 [https://www.w3.org/TR/2014/REC-vocab-dcat-20140116/] to accommodate other types of assets such as Maps, Layers, Galleries, Services. DCAT 2 [https://www.w3.org/TR/vocab-dcat-2/] closed some of the gaps addressed in the SRIM specification (`dcat:Resource`, `dcat:DataService`). The implementation of this **Semantic Registry API** and the SRIM model was refined in the context of Geoplatform.gov effort that managed all geospatial assets from the US Government. The service is currently powering the Geoplatform.gov registry.

There is also an ongoing OGC effort to define, document, and approve the OGC API – Records. This API is a multi-part draft specification that offers the capability to create, modify, and query metadata on the Web. OpenAPI 3.0 is used to define and document that API. While the Records API has a lot of similarity with the Semantic Registry API, there are some significant differences. The OGC API - Records uses GeoJSON as the default data encoding while the Semantic Registry uses JSON-LD and Linked Data formats based on well-defined ontologies (DCAT, Dublin Core [https://www.dublincore.org/specifications/dublin-core/dces/], PROV-O [https://www.w3.org/TR/prov-o/]). The REST API endpoints are very similar though in term of path structure. During the period of performance for Testbed-16, the OGC API - Records specification was still in a draft version.

Therefore, the Testbed-16 participants could not investigate the adaptation of OGC API - Records for this testbed. Once the specification is finished and approved, this would be a subject of investigation for future testbeds.

REST principles have become the most common approach for designing web APIs. REST offers some great ideas, such as stateless servers and structured access to resources. REST APIs have been widely adopted in the industry as it lowers the bar of integration of services with clients using standards HTTP verbs (GET, POST, PUT, DELETE) for performing create, read, update and delete (CRUD) operations. Many of these APIs are now using JSON as the standard formats because it is less verbose than XML and integrate easily with JavaScript client based (web browsers). However, REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them.

One of the most common problems with REST is that of **overfetching** and **underfetching**. This happens because the only way for a client to download data is by hitting endpoints that return fixed data structures. It is very difficult to design the API in a way that is able to provide clients with their exact data needs. The issues of overfetching and underfetching resources is that the server side imposes specific data view point of the client, forcing clients to perform multiple calls to the server to get the data following the structure needed by client or getting too many information for displaying specific views, especially in mobile environment. Another issue with REST APIs is that there is no standard way to describe the schema of the data model to the client that allows the discovery of APIs without writing custom code.

To address these gaps, Facebook published in 2015 the GraphQL specification, which provides a query API for APIs. GraphQL not only provides a single endpoint to perform CRUD operations but also a publish/subscription mechanism for asynchronous communication. In addition, GraphQL provides a schema language (GraphQL Schema) that enables clients to discover the data model and operations allowed on the data model in a consistent way and then perform validation on client and server side.

GraphQL query language enables clients to define the exact structure of the response (including aliasing of field names) giving back control to clients to define what data needs to be returned. This solves the issue of overfetching and underfetching in a REST API. This significantly improves performance of the applications by minimizing the number of requests and payload size exchanged with the service. A large number of vendors have adopted the GraphQL standard (GitHub, AWS, Facebook, Twitter) and libraries exists for mostly all programming languages making it easy to integrate with any clients.

9.2. Functional Overview

The Semantic Registry consists of the following sub-components:

- *Harvester*: Designed to collect service and dataset information by crawling relevant links. For this Testbed, the Harvester only retrieved data from the NSRR.
- *Importer Plugins*: Each native format (OGC API - Features JSON or NSRR XML document) is associated with an importer plugin that parses the native metadata document to SRIM Asset model. The Importer for OGC API - Features Service was also designed to connect to the API and extract the information without going through the Harvester.

- *Semantic Registry Service*: This service stores the SRIM Asset Metadata generated by the Importers into a *NoSQL database*, along with the versioning history and associations to other assets. In addition, the assets are indexed in a search index to support lexical, geospatial, temporal and semantic search. The Service provides *GraphQL and REST APIs* for clients to perform search and discovery tasks and transactions.

Figure 19 breaks down the Semantic Registry, and outlines how data flows through its subcomponents. The workflow sequence of its elements, as well as their connections to external components, is outlined on Figure 20.

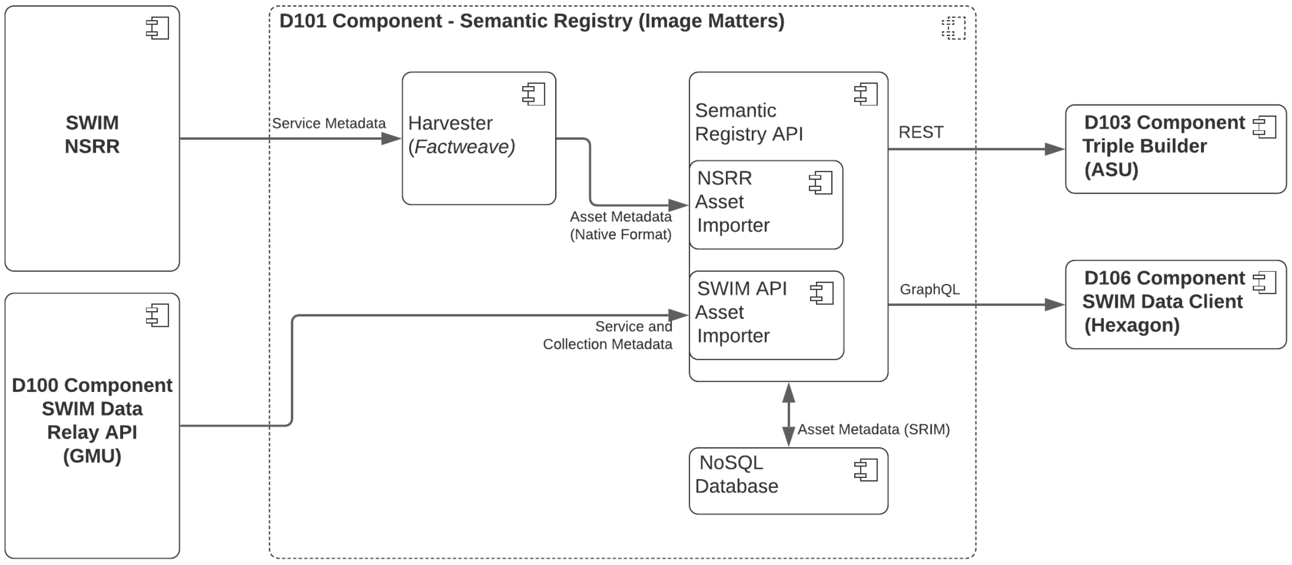

SWIM NSRR
D100 Component SWIM Data Relay API (GMU)
D101 Component - Semantic Registry (Image Matters)
Service Metadata
Harvester (Factweave)
Asset Metadata (Native Format)
Semantic Registry API
NSRR Asset Importer
SWIM API Asset Importer
Service and Collection Metadata
Asset Metadata (SRIM)
NoSQL Database
REST
GraphQL
D103 Component Triple Builder (ASU)
D106 Component SWIM Data Client (Hexagon)

Figure 19. Semantic Registry Component Breakdown and Data Flow

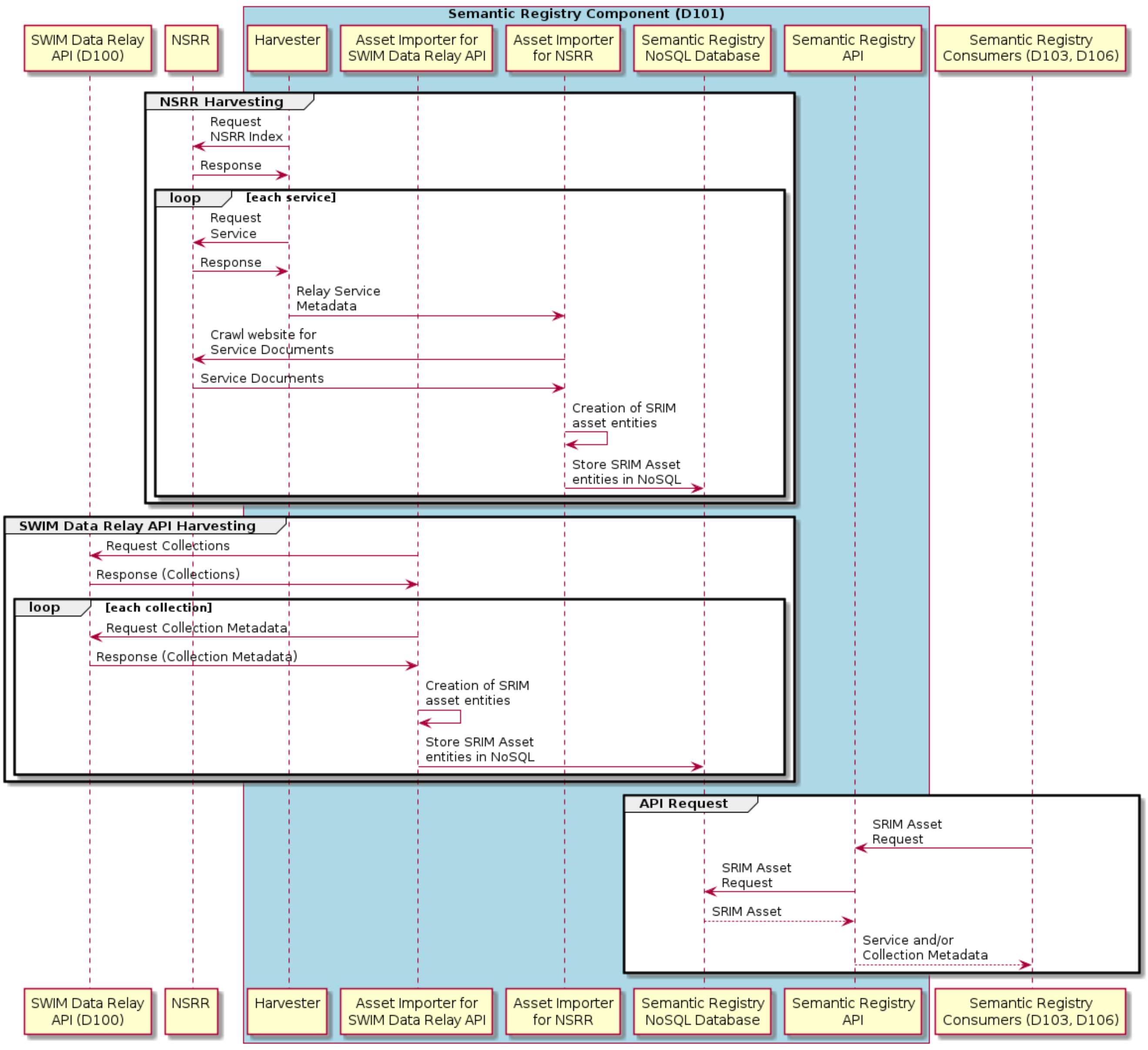

Semantic Registry Component (D101)
SWIM Data Relay API (D100)
NSRR
Harvester
Asset Importer for SWIM Data Relay API
Asset Importer for NSRR
Semantic Registry NoSQL Database
Semantic Registry API
Semantic Registry Consumers (D103, D106)
NSRR Harvesting
Request NSRR Index
Response
loop [each service]
Request Service
Response
Relay Service Metadata
Crawl website for Service Documents
Service Documents
Creation of SRIM asset entities
Store SRIM Asset entities in NoSQL
SWIM Data Relay API Harvesting
Request Collections
Response (Collections)
loop [each collection]
Request Collection Metadata
Response (Collection Metadata)
Creation of SRIM asset entities
Store SRIM Asset entities in NoSQL
API Request
SRIM Asset Request
SRIM Asset Request
SRIM Asset
Service and/or Collection Metadata

Figure 20. Semantic Registry Subcomponent Interaction

9.2.1. Harvester

To harvest metadata information from different sources, Image Matters used their Factweave Harvester [https://www.imagemattersllc.com/products-and-services/factweave/]. The Harvester connects to a data source, extracts data in a programmatic way, and handles the extracted information to an Asset Importer. The Importer is responsible for parsing the XML document and mapping the data to the SRIM model and storing the results in the registry.

The Harvester has a REST API that enables managing its capabilities. The configuration consists of the following elements:

- *Source configuration*: Consists of a source type (OGC CSW, Web Accessible Folder, RSS Feeds, OpenAPIs, etc.) and parameters bindings. Each source type requires a specific harvester plugin, which contains the logic required to extract that specific data source.
- *Handler configuration*: Depending on the source harvested, the Harvester relayed the retrieved data to a specific *Asset Importer Plugin* designed to process that type of data. The handler configuration consists of a handler type and parameter bindings.

- *Schedule*: Configuration of the scheduling of the harvesting consisting of trigger conditions, frequency and duration of the harvesting.
- *Log*: A log can be configured to provide status, errors, and amount of services retrieved.

For Testbed-16, an instance of the Semantic Registry Harvester was configured to harvest the root index page URL of the NSRR; this page contains only a brief description of the service (id, title, description) and a link to the XML document describing a full description of the service. The Harvester extracted these links from index page (located at https://nsrr.faa.gov/rim/index) and then accessed them in order to fetch the complete XML service description document. An example of the harvested NSRR Index and a full XML service description can be seen in Appendix B. The Harvester was not used to extract data from SWIM Data Relay API because there was just one service to connect to. Future work could see the Harvester being used to connect to multiple OpenAPI Services.

9.2.2. Asset Importer

Two Asset Importers were implemented for the Semantic Registry: One to process the imported service and collection metadata from SWIM Data Relay API, and another one to process the imported service metadata from the NSRR Service Description. These two Importers were responsible for converting the native asset metadata information retrieved from the service into SRIM asset entities.

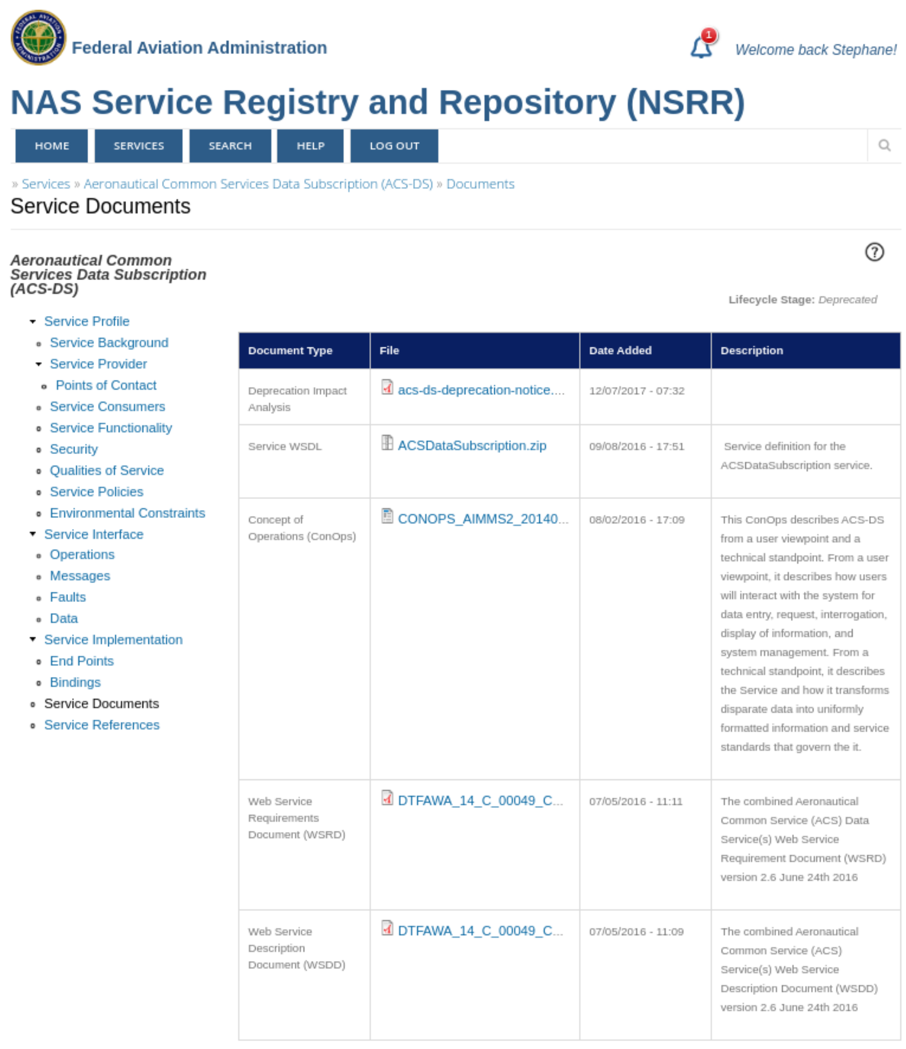
The importers parse the incoming native metadata documents and extract **only** the relevant information to support the search and discovery of assets, such as:

- The descriptive metadata (id, title, description, version).
- Attributions (publisher, contributor,…).
- Classification of assets (theme, function, status, subject, service category) using semantic-based taxonomies and keywords.
- Standard conformance.
- Access and distribution information and related resources.

A globally unique universal URI is generated if needed for each asset so they can be used to link to other assets using Linked Data principles. The assets extracted by the importers are persisted in a NoSQL database and indexed in a search index to support fast search and discovery requests from the REST API and GraphQL API. All the original native metadata is preserved separately in the database, and referred by the asset using Dublin Core source `(dct:source)` property.

9.2.3. Asset Importer for NSRR

The Asset Importer of the NSRR extracted Related Resources from each SWIM service by interpolating new URLs from the data using web page scraping techniques in order to complement information about the service that were not encoded in the XML document. These Service Documents are available through the web version of the NSRR, as seen on Figure 21.


Federal Aviation Administration

Welcome back Stephane!

NAS Service Registry and Repository (NSRR)

» Services » Aeronautical Common Services Data Subscription (ACS-DS) » Documents

Service Documents

Aeronautical Common Services Data Subscription (ACS-DS)

- Service Profile
  - Service Background
  - Service Provider
    - Points of Contact
  - Service Consumers
  - Service Functionality
  - Security
  - Qualities of Service
  - Service Policies
  - Environmental Constraints
- Service Interface
  - Operations
  - Messages
  - Faults
  - Data
- Service Implementation
  - End Points
  - Bindings
- Service Documents
- Service References

Lifecycle Stage: *Deprecated*

Document Type	File	Date Added	Description
Deprecation Impact Analysis	acs-ds-deprecation-notice....	12/07/2017 - 07:32	
Service WSDL	ACSDataSubscription.zip	09/08/2016 - 17:51	Service definition for the ACSDataSubscription service.
Concept of Operations (ConOps)	CONOPS_AIMMS2_20140...	08/02/2016 - 17:09	This ConOps describes ACS-DS from a user viewpoint and a technical standpoint. From a user viewpoint, it describes how users will interact with the system for data entry, request, interrogation, display of information, and system management. From a technical standpoint, it describes the Service and how it transforms disparate data into uniformly formatted information and service standards that govern the it.
Web Service Requirements Document (WSRD)	DTFAWA_14_C_00049_C...	07/05/2016 - 11:11	The combined Aeronautical Common Service (ACS) Data Service(s) Web Service Requirement Document (WSRD) version 2.6 June 24th 2016
Web Service Description Document (WSDD)	DTFAWA_14_C_00049_C...	07/05/2016 - 11:09	The combined Aeronautical Common Service (ACS) Service(s) Web Service Description Document (WSDD) version 2.6 June 24th 2016

Figure 21. Service Documents scraped from HTML encoded as Related Resources.

The result of the import of the previous example can be seen in Appendix B. The JSON encoding can be turned to a Linked Data using the implicit JSON-LD context serviced at the new endpoint /context.

9.2.4. Asset Importer for SWIM Data Relay API

This Asset Importer extracted metadata from the SWIM Data Relay API through a POST call to the import endpoint of the registry, specifying as parameters the version and the URL of the root of the service. The importer then traverses the links to collect metadata information about the service and the different feature collections managed by the service. Each feature collection metadata was converted to a DCAT Dataset with different distributions (one for each representation available in the API).

The result of the import of the previous example can be seen in Appendix B. The JSON encoding can be turned to Linked Data using the implicit JSON-LD context serviced at the new endpoint /context.

9.3. Semantic Registry API

For Testbed-16, Image Matters extended the Semantic Registry APIs implemented during Testbed-13 by:

- **Linked Data Support**: Adding support of JSON-LD context and Linked Data formats (TTL, RDF/XML, JSON-LD).
- **GraphQL Support**: Adding a GraphQL API to address some of the limitations of the REST API.

9.3.1. Linked Data support

To support Linked Data serialization, the item classes were annotated with Java RDF annotations. The Linked Data representation and JSON-LD context were generated by processing these annotations to produce the output. Content-negotiation by mime type or file extension on URL were added to support Turtle, RDF/XML, N-Triples and JSON-LD.

The JSON-LD encoding provides an easy way to integrate with web clients (compared to RDF/XML or Turtle [https://www.w3.org/TR/turtle/]), while preserving the semantic of the information, which can be harvested and processed by agents to perform information integration, linking and reasoning. An example in JSON-LD can be seen in Appendix B.

The same dataset can be returned in Turtle format using file extension (ttl) or HTTP Accept header. An example in TTL can be seen in Appendix B.

The previous examples demonstrate the equivalent representation of the JSON-LD in Turtle, which can be directly ingested in Triple Stores and used by semantic reasoners to perform logical consistency checking and inferences. The example shows that the SRIM model is superset of the well-established DCAT standard.

Each item in the semantic registry is addressable and can be serialized in JSON, JSON-LD, RDF/XML, Turtle, N-Triples. This does not mean that the data needs to be persisted in an RDF store. Relational database and NoSQL databases such as MongoDB can be used. The ontologies used in SRIM are based on a mixed of well-established standards such as Dublin Core Terms, PROV-O, DCAT, FOAF [http://xmlns.com/foaf/spec/]. Future work should investigate the implementation of a SPARQL endpoint to perform semantic query on the semantic registry.

9.3.2. Semantic Registry GraphQL API

REST APIs require declaring endpoints based on the resources they return. In GraphQL, you need to define a schema. This schema is used to:

- Declare the types available and their relationships.
- Declare how data can be mutated or queried.

While having *POST*, *GET*, *PUT*, *DELETE* and others as request methods in a REST API, for GraphQL there is just **Query** (equivalent of *GET* in REST) and **Mutation** (equivalent of *PUT*, *POST*, *PATCH* and *DELETE* in REST).

9.3.2.1. Semantic Registry GraphQL Schema

The *GraphQL Schema Definition Language (SDL)* is the schema language used for GraphQL. A schema is often seen as a contract between the server and client. SDL is simple and intuitive to use while being programming language-agnostic and extremely powerful, expressive and extensible. The syntax is defined in the official GraphQL specification [https://graphql.github.io/graphql-spec/draft/#

sec-Schema].

GraphQL query language is basically about defining the shapes of JSON objects to retrieve from the server by selecting fields on objects. GraphQL SDL describes the schema composed of the set of possible data that can be requested by providing a description of the object types and its fields and constraints. When GraphQL queries come in, they are validated and executed against that schema.

The main components of a schema definition are the types and their fields. Additional information can be provided as custom directives.

A **type** has a name and can implement one or more interfaces. This supports the inheritance of fields from the interfaces, allowing redefinition of the same field definition for each subtype.

```
type Dataset implements Item {
  # ...
}
```

GraphQL queries are hierarchical and composed, describing a tree of information. *Scalar types* describe the leaf values of these hierarchical queries.

The GraphQL specification defines some built-in scalar values but more can be defined by a concrete implementation.

Most types in the schema are just normal object types, but there are two types that are special within a schema:

```
schema {
  query: Query
  mutation: Mutation
}
```

Every GraphQL service has a *query type* and may or may not have a *mutation type*. These types are the same as a regular object type, but they are special because they define the entry point of every GraphQL query.

For the Semantic Registry GraphQL API, we define two queries: `getItem` and `searchItems`. A `deleteItem` mutation was defined to delete any registry item by identifier. Save mutation methods were defined for each item type (modeled as *input* instead of *type*). The save mutation methods are equivalent to the POST or PUT operation in REST API.

The GraphQL schema for the Semantic Registry is documented in Appendix C.

9.3.2.2. GraphQL Implementation

During Testbed-16, GraphQL Java library was used and a GraphQL schema for the SRIM model was defined first before writing the type resolver and data fetcher needed to assemble the data requested by the GraphQL query. A sandbox Graphiql [https://www.electronjs.org/apps/graphiql] was deployed with the registry to demonstrate the discovery of the schema and perform queries and mutations on the data.

The integration of the GraphQL endpoint with D106 was relatively straightforward without requiring to reference a how-to guide, as GraphQL schema allow clients to validate and discover the data model and operations available for the registry. The GraphQL also provides flexibility to evolve the API and support new operations and data model without redefining a new protocol.

9.3.2.3. Comparative analysis between REST and GraphQL

Table 8 compares REST API and GraphQL API based on the lessons learned from the implementation of the REST API and GraphQL API for the Semantic Registry.

Table 8. Comparison of GraphQL and REST API

	GraphQL	REST API
Architecture	Client-Driven	Server-Driven
Organized in terms of	Schema & Type System	Endpoints
Operations	Query, Mutation, Subscription	Create, Read, Update, Delete
Number of Endpoints	1	`n` (typically 1 per resource type).
Data Fetching	Specific Data with a single API call	Fixed Data with multiple API calls
Community	Fast-growing	Large
Performance	Fast	Multiple network calls take up more time
Development Speed	Rapid	Slower
Learning Curve	Moderate	Easy
Self-Documenting	Yes	No
File Uploading	Yes (Through Extension)	Yes
Web Caching	No (via libraries built on top)	Yes
Stability	Less error-prone: Automatic validation and type checking	Better choice for complex queries
Use Cases	Multiple microservices, Mobile apps	Simple apps, Resource-driven apps

9.4. Challenges and Lessons Learned

- **Limitations of the NSRR**: The FAA has defined the WSDOM ontology to describe their services. However, the NSRR registry was based on an XML encoding based on a XML schema. Each service description is classified using reference URIs to SKOS taxonomies. In addition, an index file provides a summary of each service required to make a call to get the full description of each service. There is no query API that allows performing a search and retrieving the list of services in full details. For this reason, custom codes are required to be written to harvest and convert data to Linked Data representation.

  Having the SWIM registry expose the services as linked data and providing a SPARQL endpoint to perform custom search of information would be simpler. The mix of data centric (XML based on XML Schema) and knowledge centric (SKOS) makes it impossible to leverage the WSDOM ontology to perform inference and linking services to other assets expressed in Linked data (such as Dataset, Map, Layer, etc.)

- **Support of different application profiles**: There are number of profiles for DCAT (DCAT-AP, GeoDCAT, DCAT-US, DCAT_FR, among others), each one defining different usage of the DCAT models using specific taxonomies or properties. SHACL has emerged as the standard way to validate Linked Data for these profiles. The current implementation of the Semantic Registry does not have a mechanism to publish the supported profiles and perform validation against specific profiles. This is an important topic to be investigated in the future. In addition, the import of RDF data and support of a GeoSPARQL endpoint on the Semantic Registry should be considered in future testbeds.
- **GraphQL and Linked Data**: Further investigation is needed to define a normative way to map GraphQL to Linked Data. In particular, investigating the role and usage of JSON-LD context with GraphQL schema to define the mapping to RDF model and SHACL to define the application profiles that translate to GraphQL Schema.

9.5. Accomplishments

- The descriptions of 94 services were harvested from the NSRR.
- The metadata from 33 collections were retrieved and converted to DCAT Datasets from the SWIM Data Relay API Component of this Testbed.
- The Semantic Registry was extended with a GraphQL API and Linked data serialization using JSON-LD, Turtle, RDF/XML and N-Triples.
- The Triple Store and OpenAPI Client Components accessed the Semantic Registry using the REST API and GraphQL API, respectively.
- The SRIM model was updated to be better aligned with DCAT 2 and taxonomies developed in previous testbeds (semantics.aero) were used to classify FAA services.

Chapter 10. Triple Builder and Triple Store

The Triple Builder was the component designed to build triples based on the retrieval and combination of both aviation data and aviation ontologies. A Triple Store component was included to store the aforementioned triples and make them available through SPARQL and GeoSPARQL endpoints.

The development of these two components was carried out by Arizona State University (ASU). The main goals were to demonstrate the process of semantic-enrichment of aviation data, and to unlock further demonstrations by providing linked data to the Semantic Web Client developed in this Testbed.

10.1. Status Quo

ASU's previous experience on semantic enrichment has been through research on the development of ontologies for land use and land cover change, as well as the development of a cyberinfrastructure portal that supports the semantic query and visualization of the time series data. To convert data into formal knowledge, ASU segmented remote sensing images into different land use patches, and then converted each of the polygonal features into triples and saved them into a triple store. Jena [https://jena.apache.org/] and Virtuoso [https://virtuoso.openlinksw.com/] were used for ontology building and triple building. Another experiment was the building of ontologies for a spatial decision support semantic portal where Protégé [https://protege.stanford.edu/] was used for ontology building and instance importing.

This was the first time ASU has used RDF4J [https://rdf4j.org/] for semantic work, and the first time ASU used the OGC API - Features Standard for the retrieval of data with the purpose of semantically enriching the data.

To the knowledge of this Testbed's participants, there is currently no SWIM service implemented as a service described using OpenAPI.

10.2. Functional Overview

The Triple Builder and Triple Store generated, stored, and relayed semantically enriched aviation data. The generation of triples was performed by the Triple Builder, which then handed over the triples to the Triple Store to store them and make them available for clients to consume.

The Triple Builder generated two sets of triples for this Testbed, one for flight data and another one for airport data. A third set of triples was generated by combining the two aforementioned sets of triples. Each set of triples was generated through a different process:

1. On one hand, the set of flight triples was generated by first retrieving flight data from the SWIM Data Relay API Component and then storing that data into a database. Finally, the Component used a script with the *Jena RDF API* to generate triples by combining the aforementioned data with a set of flight ontologies built specifically for this testbed.
2. On the other hand, the airport triples were manually generated using a tool named *Protégé* to combine a dataset of airports with a set of airport ontologies also built specifically for this

testbed.

3. The airport and flight triples were then loaded into a triple store built on a platform named RDF4J to build a federation repository. A federation repository can be built using the workbench UI or a repository *config* template deployed in the RDF4J Server. This repository serves as a virtual endpoint (SPARQL and GeoSPARQL) from which the Semantic Web Client Component consumed the triples.

The Triple Builder was also connected to the Semantic Registry Component to retrieve the flight datasets from SWIM Data Relay API to demonstrate the service discovery capabilities of the Semantic Registry.

Figure 22 breaks down the Triple Builder and Triple Store, and outlines how data flows through its subcomponents. The workflow sequence of its elements, as well as their connections to external components, is outlined on Figure 23.

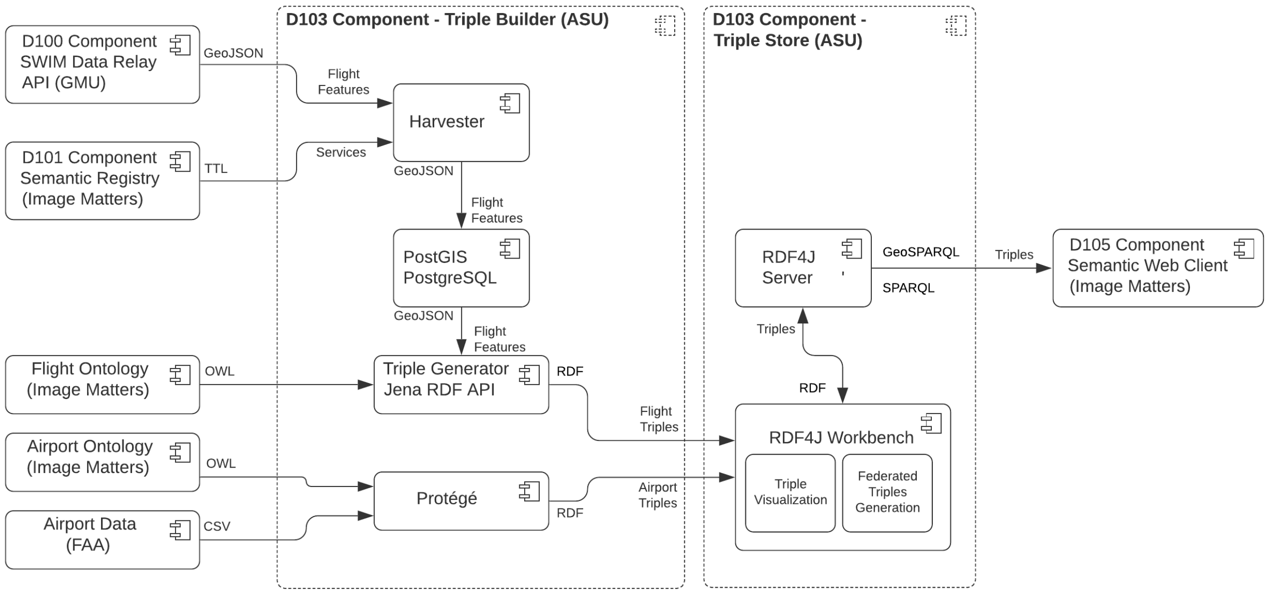

D100 Component SWIM Data Relay API (GMU)
GeoJSON
D101 Component Semantic Registry (Image Matters)
TTL
D103 Component - Triple Builder (ASU)
Flight Features
Services
Harvester
GeoJSON
Flight Features
PostGIS PostgreSQL
GeoJSON
Flight Features
Flight Ontology (Image Matters)
OWL
Triple Generator Jena RDF API
RDF
Airport Ontology (Image Matters)
OWL
Protégé
Airport Data (FAA)
CSV
RDF
Flight Triples
Airport Triples
D103 Component - Triple Store (ASU)
RDF4J Server
GeoSPARQL
SPARQL
Triples
Triples
RDF
RDF4J Workbench
Triple Visualization
Federated Triples Generation
D105 Component Semantic Web Client (Image Matters)

Figure 22. Triple Builder and Triple Store Component Breakdown and Data Flow

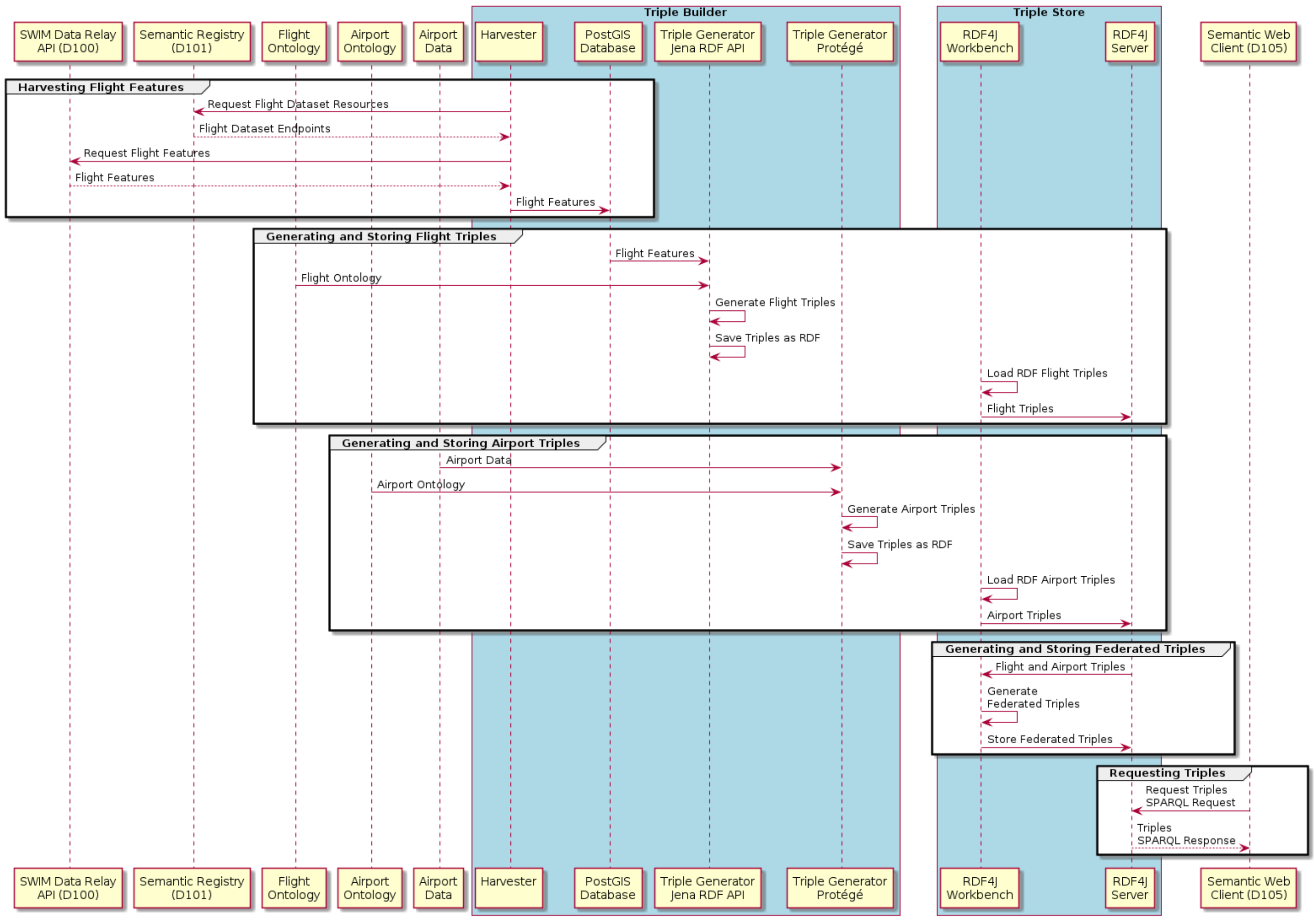


Figure 23. Triple Builder and Triple Store Subcomponent Interaction

10.2.1. Triple Builder

Each set of triples were generated through a different process, and later loaded into the Triple Store. The flight triples were generated through scripts, while the airport triples were manually generated through the user interface of a software application.

Flight Triples

The process for generating the flight triples consisted of three steps: Harvesting, storing, and triple generating. The creation process of the flight triples was entirely performed through scripts. Despite the fact ASU manually triggered these scripts for the purpose of this Testbed this process paved the way for future work to see these scripts being executed automatically.

A script written in Java, named "*Harvester*", was built to discover SWIM datasets and services through the Semantic Registry Component, retrieve SWIM data from the SWIM Data Relay API Component, and load the retrieved data into a database.

The Harvester communicated with the Semantic Registry to query for the flight datasets that were required to build the flight triples. The Harvester first makes an API query to the Semantic Registry to access a dataset that matched a specific collection being searched. Next, a Turtle (TTL) description of the metadata is retrieved. Finally, a SPARQL query on the TTL returns the exact endpoint of the SWIM Data Relay API to download the requested collection.

After acquiring the service and dataset information, the Harvester performed API calls to the SWIM Data Relay API Component to request the flight data that would ultimately be used for the creation of the flight triples. The SWIM Data Relay API returned flight data collections encoded as GeoJSON objects which implemented OGC’s simple feature specification.

For the purposes of this Testbed, two collections were retrieved using the GeoJSON format:

- `trackInformation`: Containing flight track information, both international and domestic.
- `NASFlightMessage`: Containing flight information with U.S. Extension in FIXM.

Once the flight features were retrieved, the Harvester stored them into a PostgreSQL database extended with PostGIS. PostgreSQL (version 9.5.11) was used to manage and archive indexes, while PostGIS (version 2.2.1) was used to maintain and index spatial properties.

A second Java script, named "*Triple Generator*", was coded to retrieve the flight features from the PostgreSQL and combine them with an ontology in order to generate the flight triples. The triples were generated using an open-source Semantic Web framework for Java named *Jena RDF API*. First, *Jena* was used to create a `Jena Ontology Model` object that included all the namespace references, ontology class definition, ontology properties, and description of relationships. The script then iterated all the feature objects and put all the attributes into newly created ontology pieces generated by the ontology model based on the schema. The output format of the triples was RDF.

A triple generation example is outlined in Appendix D.

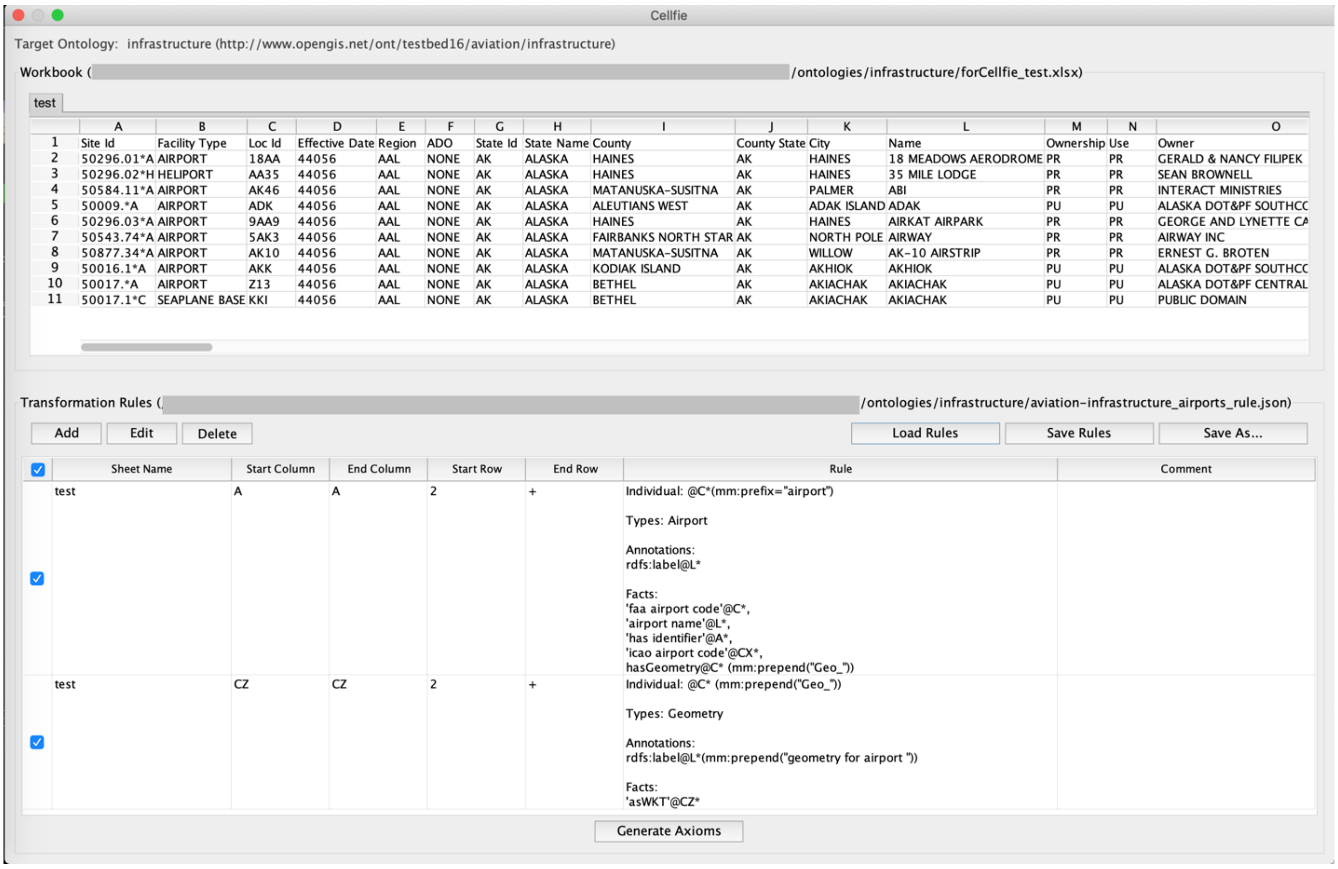
Airport Triples

Triples may also be generated based on external datasets. To demonstrate this, airport triples were generated based on data not retrieved from the SWIM Data Relay API.

The generation of the airport triples was performed manually through the user interface (UI) of Protégé [https://protege.stanford.edu/], a free and open-source ontology editor and framework for building intelligent systems.

Airport data were retrieved from an FAA database [https://www.faa.gov/airports/airport_safety/airportdata_5010/] as comma-separated values (CSV) files. The main tool used to generate the flight triples, Jena API, is effective at building triples through parsing rules coded in a script; in the case of processing CSV files (which have a table-like data structures), Jena does not provide a visual interface to interact with the data. Protégé, which has a friendly UI for both triple building and instance importing, was therefore chosen to generate the airport triples.

The airport datasets were imported as instances through a Protégé plugin for spreadsheet importing named Cellfie [https://github.com/protegeproject/cellfie-plugin]. As seen on Figure 24, the plugin provides a visualization of the imported data. The upper section of the window displays the spreadsheet as a reference for the construction of rules. The lower section of the window displays the importing rule editor. For reusability, mapping rules can be saved in a syntax named *Manchester* as a JSON file. A sample JSON file can be seen in Sample Cellfie Rule Mapping JSON Code.


Cellfie

Target Ontology: infrastructure (http://www.opengis.net/ont/testbed16/aviation/infrastructure)

Workbook ([redacted]/ontologies/infrastructure/forCellfie_test.xlsx)

test

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Site Id	Facility Type	Loc Id	Effective Date	Region	ADO	State Id	State Name	County	County State	City	Name	Ownership	Use	Owner
2	50296.01*A	AIRPORT	18AA	44056	AAL	NONE	AK	ALASKA	HAINES	AK	HAINES	18 MEADOWS AERODROME	PR	PR	GERALD & NANCY FILIPEK
3	50296.02*H	HELIPORT	AA35	44056	AAL	NONE	AK	ALASKA	HAINES	AK	HAINES	35 MILE LODGE	PR	PR	SEAN BROWNELL
4	50584.11*A	AIRPORT	AK46	44056	AAL	NONE	AK	ALASKA	MATANUSKA-SUSITNA	AK	PALMER	ABI	PR	PR	INTERACT MINISTRIES
5	50009.*A	AIRPORT	ADK	44056	AAL	NONE	AK	ALASKA	ALEUTIANS WEST	AK	ADAK ISLAND	ADAK	PU	PU	ALASKA DOT&PF SOUTHCC
6	50296.03*A	AIRPORT	9AA9	44056	AAL	NONE	AK	ALASKA	HAINES	AK	HAINES	AIRKAT AIRPARK	PR	PR	GEORGE AND LYNETTE CA
7	50543.74*A	AIRPORT	5AK3	44056	AAL	NONE	AK	ALASKA	FAIRBANKS NORTH STAR	AK	NORTH POLE	AIRWAY	PR	PR	AIRWAY INC
8	50877.34*A	AIRPORT	AK10	44056	AAL	NONE	AK	ALASKA	MATANUSKA-SUSITNA	AK	WILLOW	AK-10 AIRSTRIP	PR	PR	ERNEST G. BROTEN
9	50016.1*A	AIRPORT	AKK	44056	AAL	NONE	AK	ALASKA	KODIAK ISLAND	AK	AKHIOK	AKHIOK	PU	PU	ALASKA DOT&PF SOUTHCC
10	50017.*A	AIRPORT	Z13	44056	AAL	NONE	AK	ALASKA	BETHEL	AK	AKIACHAK	AKIACHAK	PU	PU	ALASKA DOT&PF CENTRAL
11	50017.1*C	SEAPLANE BASE	KKI	44056	AAL	NONE	AK	ALASKA	BETHEL	AK	AKIACHAK	AKIACHAK	PU	PU	PUBLIC DOMAIN

Transformation Rules ([redacted]/ontologies/infrastructure/aviation-infrastructure_airports_rule.json)

✓	Sheet Name	Start Column	End Column	Start Row	End Row	Rule	Comment
✓	test	A	A	2	+	Individual: @C*(mm:prefix="airport") Types: Airport Annotations: rdfs:label@L* Facts: 'faa airport code'@C*, 'airport name'@L*, 'has identifier'@A*, 'icao airport code'@CX*, hasGeometry@C* (mm:prepend("Geo_"))	
✓	test	CZ	CZ	2	+	Individual: @C* (mm:prepend("Geo_")) Types: Geometry Annotations: rdfs:label@L*(mm:prepend("geometry for airport ")) Facts: 'asWKT'@CZ*	

Figure 24. Cellfie Spreadsheet Importer

Sample Cellfie Rule Mapping JSON Code

```
{
   "Collections":[
      {
         "sheetName":"test", "startColumn":"A", "endColumn":"A", "startRow":"2",
"endRow":"+",
         "comment":"",
         "rule":"Individual: @C*(mm:prefix\u003d\"airport\")\n\nTypes: Airport\n
\nAnnotations: \nrdfs:label@L*\n\nFacts: \n\u0027faa airport code\u0027@C*,\n
\u0027airport name\u0027@L*,\n\u0027has identifier\u0027@A*,\n\u0027icao airport
code\u0027@CX*,\nhasGeometry@C* (mm:prepend(\"Geo_\"))",
         "active":true
      }, {
         "sheetName":"test", "startColumn":"CZ", "endColumn":"CZ", "startRow":"2",
"endRow":"+",
         "comment":"",
         "rule":"Individual: @C* (mm:prepend(\"Geo_\"))\n\nTypes: Geometry\n
\nAnnotations: \nrdfs:label@L*(mm:prepend(\"geometry for airport \"))\n\nFacts:
\n\u0027asWKT\u0027@CZ*",
         "active":true
      }
   ]
}
```

Aircraft Triples

A third set of aviation data was proposed for triple generation. These data were based on aircraft data. These data are useful not only as an independent triple repository, but also as an additional component of the federated triples that were already composed by flight and airport triples. Adding aircraft triples would have expanded the semantic-enrichment of the flight triples by allowing data consumers (i.e. users of Semantic Web Client) to explore additional information related to the aircraft associated with a flight.

Aircraft data was obtained from the FAA as CSV static files. Aircraft data were found to be available in SWIM feeds but under certain conditions:

1. Registration number is optional. In their distributed messages from SWIM Flight Data Publication Service (SFDPS), the *registration* is available under `aircraftDescription` for messages with `source`=`"HU"`, `"AH"`, or `"FH"`. These are FIXM messages related to flight plan.
2. Linking "registration" in `HU`, `AH`, or `FH` to flight is possible through Globally Unique Flight Identifier (GUFI) or flightPlan/identifier.

Due to time constraints in the Testbed, the creation of aircraft triples and its use within the federated triples was left for future work.

10.2.2. Triple Store

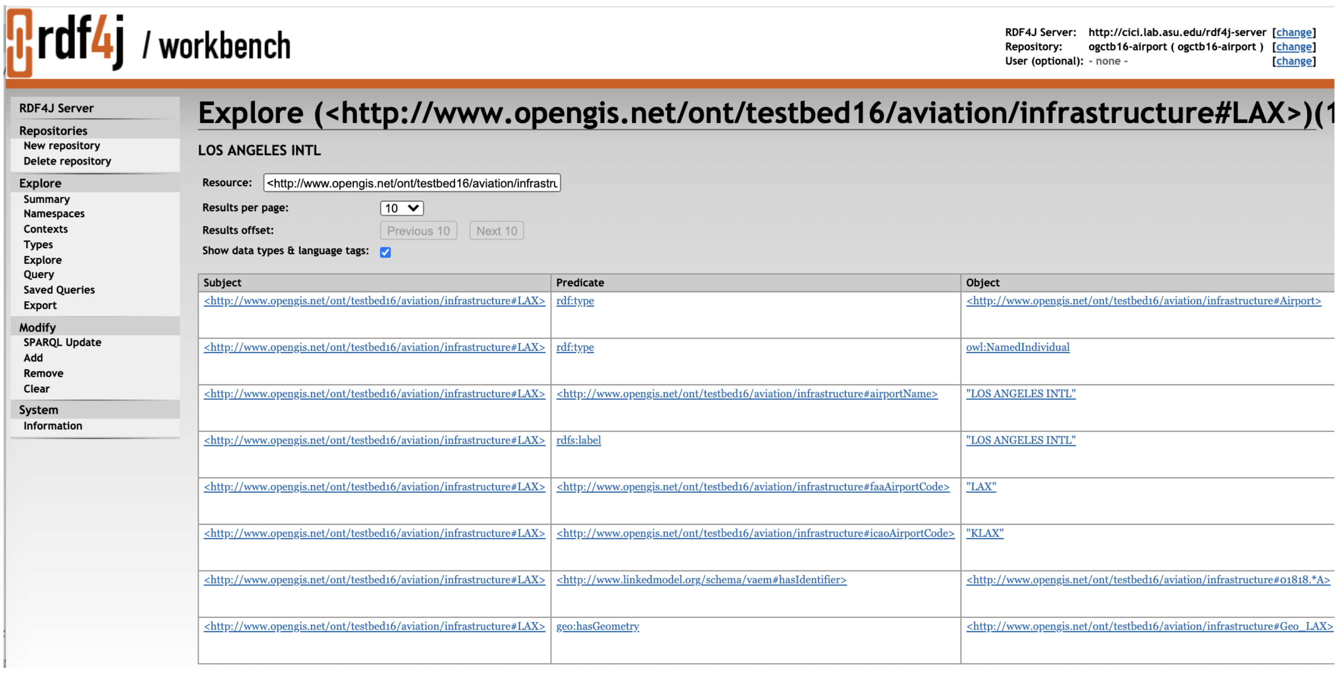
The triples generated in the Triple Builder were stored in a triple store named *RDF4J Server*. Eclipse RDF4J is an open-source modular Java framework for working with RDF data. RDF4J allows external software tools to connect with SPARQL endpoints and create applications that leverage the power of Linked Data and Semantic Web.

RDF4J provides two tools for users to load triples into the RDF4J Server: The *RDF4J Console*, which is a text console application for interacting with RDF4J, and the web-based UI named *RDF4J Workbench*. RDF4J Server is a servlet-based web application deployed to any standard servlet container. ASU used Tomcat as the container. A special Tomcat configuration was needed to enable UTF-8 Support for RDF4J Workbench to solve a character encoding issue.

For this testbed, the RDF4J Workbench was used to manually load the triples into the RDF4J Server. Figure 25 shows the Workbench being used to visualize airport triples after being loaded into the RDF4J Server. Future work could see this manual process be automated by using scripts to load triples into the triple store without user intervention.

RDF4J enables users to connect with SPARQL endpoints and provides support to the majority of GeoSPARQL queries. This was a critical reason for ASU selecting this triple store because the Semantic Web Client Component required both SPARQL and GeoSPARQL queries. Up to the beginning of Testbed 16, there was no triple store fully supporting GeoSPARQL. Furthermore, compared to other solutions, RDF4J had shown satisfying performance in both GeoSPARQL support as well as efficiency especially for spatial selection queries [13]. Finally, another reason for ASU to choose RDF4J in this Testbed was RDF4J is open source and well documented.

ASU decided to work with triple stores because these support ontologies. This allows for a formal description of the data: Triple stores specify both object classes and relationship properties, and their hierarchical order, while general graph databases do not. For this Testbed, the ability to infer and reason was considered more useful because various aviation datasets are not just linked but also semantically understandable.



RDF4J Server: http://cici.lab.asu.edu/rdf4j-server [change]
Repository: ogctb16-airport (ogctb16-airport) [change]
User (optional): - none - [change]

Explore (<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>)(1

LOS ANGELES INTL

Resource: <http://www.opengis.net/ont/testbed16/aviation/infrastru
Results per page: 10
Results offset:
Show data types & language tags: ✓

Subject	Predicate	Object
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	rdf:type	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#Airport>
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	rdf:type	owl:NamedIndividual
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#airportName>	"LOS ANGELES INTL"
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	rdfs:label	"LOS ANGELES INTL"
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#faaAirportCode>	"LAX"
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#icaoAirportCode>	"KLAX"
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	<http://www.linkedmodel.org/schema/vaem#hasIdentifier>	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#01818.*A>
<http://www.opengis.net/ont/testbed16/aviation/infrastructure#LAX>	geo:hasGeometry	<http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_LAX>

Figure 25. RDF4J Workbench Displaying Airport Triples

Federated Triples

After the airport and flight triples were loaded into RDF4J as separate repositories, a single federation repository was created as a combination of the aforementioned individual repositories. The federation repository consisted of triples made up of references from both flight and airport triples. These federated triples enabled the Triple Store to answer complex queries made in SPARQL, as seen on Sample SPARQL Query to the Triple Store, and whose response was in the form of linked data. The triples were generated using the RDF4J Workbench.

Table 9 presents a sample response to a SPARQL query. A regular flight element would not include detailed information about the airports, but linked data enables access to additional information about each element by following the links on each attribute. The subsequent tables outline the sample content returned when exploring each element of the response (Flight, Airport Code, Airport Name, Airport FAA Code, Geometry, and FWKT), providing an example of the semantic enrichment of aviation data.

Sample SPARQL Query to the Triple Store

```
PREFIX flight: <http://www.opengis.net/ont/testbed16/aviation/activities/flight#>
PREFIX infrastructure: <http://www.opengis.net/ont/testbed16/aviation/infrastructure#>
PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>

SELECT ?Flight ?Arrival_Airport_Code ?airportName ?FAA_code ?geo ?fWKT
WHERE {
  ?Flight a flight:Flight;
    flight:arrival ?Airport_Arrival.
  ?Airport_Arrival flight:airport ?Arrival_Airport.
  ?Arrival_Airport infrastructure:icaoAirportCode ?Arrival_Airport_Code.

  ?airport a infrastructure:Airport;
    infrastructure:icaoAirportCode ?Arrival_Airport_Code;
    infrastructure:airportName ?airportName;
    infrastructure:faaAirportCode ?FAA_code;
    geosparql:hasGeometry ?geo.

  ?geo geosparql:asWKT ?fWKT .
}
```

Table 9. RDF4J Response to the Sample SPARQL Query

Flight	Arrival_Airport_Code	AirportName	FAA_code	Geo	FWKT
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	"KIAD"	"WASHINGTON DULLES INTL"	"IAD"	http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	Point(-77.4599444444444 38.9474444444444)"

Table 10. Result of Exploring the "Flight" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.w3.org/2000/01/rdf-schema#label	"Flight FID2898"
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.opengis.net/ont/testbed16/equipment#operatedBy	http://www.opengis.net/ont/testbed16/aviation/activities/flight#AircraftOperator_SKQ
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.opengis.net/ont/testbed16/aviation/activities/flight#arrival	http://www.opengis.net/ont/testbed16/aviation/activities/flight#Arrival_FID2898

http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.opengis.net/ont/testbed16/aviation/activities/flight#departure	http://www.opengis.net/ont/testbed16/aviation/activities/flight#Departure_FID2898
http://www.opengis.net/ont/testbed16/aviation/activities/flight#Flight_FID2898	http://www.opengis.net/ont/testbed16/aviation/activities/flight#flightNumber	"SKQ74"

Table 11. Result of Exploring the "Arrival_Airport_Code" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/data/testbed16/aviation/infrastructure/us_airports#IAD	http://www.opengis.net/ont/testbed16/aviation/infrastructure#icaoAirportCode	"KIAD"

Table 12. Result of Exploring the "AirportName" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/data/testbed16/aviation/infrastructure/us_airports#IAD	http://www.opengis.net/ont/testbed16/aviation/infrastructure#airportName	"WASHINGTON DULLES INTL"

Table 13. Result of Exploring the "FAA_code" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/data/testbed16/aviation/infrastructure/us_airports#IAD	http://www.opengis.net/ont/testbed16/aviation/infrastructure#faaAirportCode	"IAD"

Table 14. Result of Exploring the "Geo" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#NamedIndividual
http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.opengis.net/ont/sf#Geometry
http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	http://www.w3.org/2000/01/rdf-schema#label	"geometry for airport WASHINGTON DULLES INTL"
http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	http://www.opengis.net/ont/geosparql#asWKT	"Point(-77.4599444444444 38.9474444444444)"
http://www.opengis.net/data/testbed16/aviation/infrastructure/us_airports#IAD	http://www.opengis.net/ont/geosparql#hasGeometry	http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD

Table 15. Result of Exploring the "FWKT" Element in the RDF4J Response

Subject	Predicate	Object
http://www.opengis.net/ont/testbed16/aviation/infrastructure#Geo_IAD	http://www.opengis.net/ont/geosparql#asWKT	"Point(-77.4599444444444 38.9474444444444)"

10.3. Challenges and Lessons Learned

- **Benefits of using OpenAPI to define APIs**: The OGC API - Feature Service was straight forward to use. This is in contrast with previous uses of WFS 2.0. ASU has used WFS 2.0 in the past. For WFS 2.0, a web application/machine needs to parse a complex XML document to understand the capabilities of a specific WFS service. Moreover, XML is also difficult for developers to interpret its content. The complexity has hindered the widespread use of WFS 2.0 as an open standard for sharing feature data. Using OpenAPI to define the Feature API was seen by ASU as a modernized service architecture. The XML-based capability document is no longer needed. Further, the new feature collection document supports multiple encoding methods, such as GeoJSON, which are more “friendly” for data comprehension and retrieval for this Testbed work.
- **Benefits of the Semantic Registry**: The Semantic Registry was implemented without major drawbacks. The Service has two query building processes: One is the RESTful query to search the resources of dataset triples, and the other is the SPARQL query on dataset triples to get the feature collection download links. The RESTful query was straightforward to use because of the simple syntax of API. The SPARQL query was deemed friendly for the Triple Builder because triples in TTL format are simple to load and query.

  For the purposes of this Testbed, the Semantic Registry provided the Triple Builder with information that was already known by ASU. The purpose of this exercise was to demonstrate the service discovery workflow. Future triple builders could query for datasets and services without knowing about them beforehand, unlocking further triple creation capabilities.
- **Lack of WKT Support**: Cellfie did not support the well-known text (WKT) literal. ASU solved this by assigning the `asWKT` property as the default `XSD.xstring`, and then replacing it by `geo:wktLiteral` (`geo` is the prefix representing `http://www.opengis.net/ont/geosparql#`). This relatively minimum modification ensured the geospatial information to be encoded in WKT format which was accepted in the RDF4J triple store.
- **Geometry Encoding in Triples using WKT**: Geospatial queries compliant with GeoSPARQL require the text-based serializations of geometry data such as well-known text (WKT) and Geography Markup Language (GML). RDF4J only supports GeoSPARQL functions on top of geospatial data represented as a WKT literal. The geometry information had to be parsed into the WKT format before storing it in the triple. Some other triple stores like GraphDB and GeoSPARQL-Jena support GML as well, which could be explored in future work.
- **Lack of Ontologies**: Semantic-enrichment requires converting geospatial data into RDF triples under a well-defined ontological schema. GeoJSON-LD is a good candidate to transmit and store linked data because it contains both the ontology part for the data and the data itself. Because GeoJSON-LD is an extension of JSON-LD to deal with geometry, it can be directly stored in triple stores such as RDF4J. One of ASU’s experiments was to create an ontology based on the message structure: part of the context section of JSON-LD was challenging to comprehend because no

definitions and explanations were found on numerous message tags. To address this issue, ASU collaborated with Image Matters to reconstruct a set of aviation-related ontologies based on existing well-known ontologies. Future work should address this gap in semantics.

- **Lack of aviation ontologies and lack of availability of aviation Linked Data**: Considerable time was spent on identifying a subset of data that could be demonstrated and then developing ontologies to represent them. The integration of non-RDF data (FIXM and AIXM) was initially done by generating an ontology from the XML schema. The issue was that the linked data representation of the information was not coherent semantically and no inference could be performed (using subclass, sub properties, transitivity, and inverse relationship). This showed having an automated way to convert data to knowledge representation is not possible. Ontology design requires subject experts and careful crafting of the logical axioms to represent the information.

  Modularization of the ontologies and reuse of existing ontologies (GeoSPARQL features, OWL Time, QUDT) are also crucial for reusability of the ontologies. Due to the lengthy process, the triple stores were not able to store enough information to show a compelling scenario leveraging the full power of the semantics. The goal was to show flight information (including trajectory) which are 4D points (geospatial+temporal) along with airport and aircraft information coming from other sources. There is no well-defined ontology to represent trajectory (OGC Moving Features [https://www.ogc.org/standards/movingfeatures] is not actually an ontology). This may be a topic for improving GeoSPARQL or define some best practices to represent spatial-temporal geometry and moving features.

- **Source of Datasets for Triple Generation**: The flight triples were generated based on SWIM data retrieved from the SWIM Data Relay API. In contrast, the team was unable to retrieve airport data through SWIM nor the SWIM Data Relay API. Aircraft data was ultimately found to be optional in certain SWIM data services. Instead, the team was able to retrieve aircraft data from static datasets (as was done with the airport data). This challenge could possibly reduce the potential of fostering semantic enrichment of aviation data. Measures to boost availability should be considered to foster semantic enrichment endeavors in the future.

10.4. Accomplishments

The following are the key accomplishments of this Testbed-16 task:

- Triples were generated for flight data retrieved from an OGC API - Features service.
- Triples were generated for airport data coming from a data source external to SWIM.
- Federated triples were generated out of the combination of flight and airport triples.
- The aforementioned triples were made available through a SPARQL endpoint to the Semantic Web Client Component to further demonstrate the use of semantically-enriched aviation data.

Chapter 11. Aviation Ontologies

To support the creation of aviation triples through the Triple Builder and Triple Store component, a set of aviation ontologies had to be used. This section describes the process of searching for existing aviation ontologies, and describes the ontologies that were ultimately built. These ontologies were created by Image Matters LLC.

11.1. Status Quo

Previous testbeds focused on developing taxonomies encoded in SKOS to support different types of classifications of SWIM services (see semantics.aero). The concepts of these taxonomies were referred in XML documents specified by XML Schemas. Two different data representations are used in the same document: Linked Data and XML document based on XML Schema. Because of this, it is not possible to leverage the semantic web stack tool to perform linkage of these concepts to other semantic concepts such as services or datasets, neither leverage the inferences related to logical axioms defined in ontologies. To enable the integration of different data silos representing different information such as services, datasets containing flights, airport infrastructure, weather, aircraft information, a common representation based on Linked Data standards was needed.

In Testbed-10, a set of micro ontologies (also called *microtheories*) were donated by Image Matters as a way to "bootstrap" the geospatial semantic web. The set of microtheories provided a set of core cross-domain ontologies including *mereology* (part-whole), collections, geometries, spatial relations, events, topologies, quantities and unit of measures, temporal entities and relations. Since then, a number of ontologies have been published that overlap with these early microtheories such as DCAT, OWL Time and QUDT. Testbed 14 included a review of the existing data models currently used by FAA and the aviation community (OGC 18-035). Testbed-16 was used as an opportunity to explore the usage of these ontologies to represent a subset of aviation domain (due to time constraints).

11.2. Requirements Statement

The initial focus of the ontology modeling work was on modeling aircraft, airport infrastructure and flight information, but also services and datasets from the NSRR Registry.

The starting point for the construction of the ontologies was a set of *competency questions* (CQ) that need to be answered by the ontologies created. These CQs helped define the scope of the ontologies. Conceptually, a good ontology is when what we represent has been represented in the ontology and answer the competency questions. Yet what is actually represented is very close or only slightly more than the intention, that is we have maximum coverage and low precision. A less good ontology is when the ontology represents much more than it should. In this case there is maximum coverage but bad precision. A bad ontology occurs when it does not contain all the concepts that it should represent and cannot answer all the competency questions (low precision and low coverage) [14].

In the Testbed-16 CFP, the goal of the aviation scenario was to address the integration of SWIM data from various sources to answer complex queries, such as:

- Which flights from Washington Dulles Airport (IAD) to any airport in Europe have not been subject of Ground Delay Program (GDP) Advisories in the last 2 hours?
- What is the closest airport in Florida to land a flight from IAD, given a Temporary Flight Restriction due to a hurricane?

Due to Testbed time constraints and data availability, a decision was made to initially focus on representing flight data with information about departure, arrival, airports and aircraft information.

The following competency questions were outlined to define the scope of the ontologies:

- CQ1: Which flights are between two airports on a given day?
- CQ2: Which aircraft is used for a given flight?
- CQ3: Which aircraft model is used on a given flight?
- CQ4: Who is the manufacturer of a given aircraft?
- CQ5: What is the trajectory of a given flight?
- CQ6: Where is the location of a given airport?
- CQ7: What are the different identifiers for a given airport
- CQ8: What is the duration of a given flight?
- CQ9: What are the runways on a given airport?
- CQ10: How many aircraft are operated by a given air operator?
- CQ11: How many airports are in a given area?
- CQ12: Which airport is the closest to a current position.

Each of these competency questions could be associated with one or several SPARQL queries so they could then be used to check that the datasets using the ontologies were actually answering the questions.

11.3. Functional Overview

The first approach was to identify and explore the capabilities of current aviation ontologies. The NASA Air Traffic Management (ATM) ontologies were explored for this purpose. However, a series of issues detected (described in Challenges and Lessons Learned) required the creation of the ontologies from scratch. Overall, a set of six ontologies were developed for this Testbed:

- Equipment Ontology
- Aircraft Ontology, an extension of the Equipment Ontology
- Flight Ontology
- Routing Ontology
- Facility Ontology
- Aviation Infrastructure Ontology

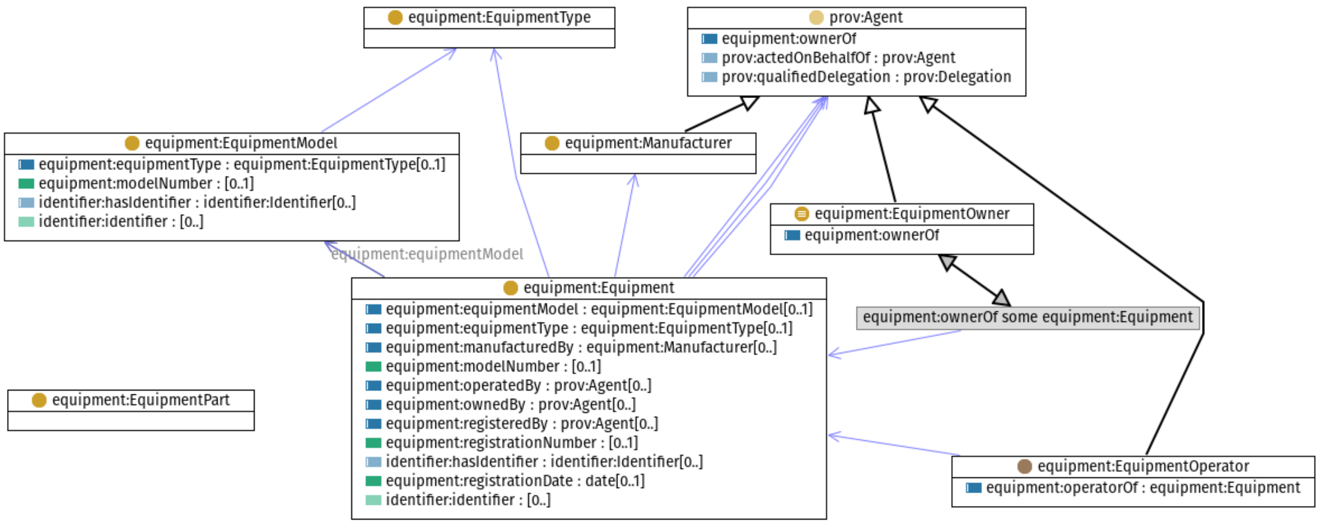
To build these ontologies, a number of standards were looked at. First, FIXM and AIXM were explored as these two data models define the main business objects (flight, airport, airspace) to be represented by the ontologies. Object names and properties were extracted from these models, and were formalized as ontologies using linked data standards (OWL and RDF) in order to have machine-understandable models that can be reasoned on.

Image Matters also looked at the NASA ATM ontologies to find out if there were any reusable concepts; these ontologies were found not to separate the general equipment model from the aviation specific equipment. To enhance reusability, a general equipment ontology was defined. This ontology could be used and extended in other domain such as modeling vessels for water navigation or cars for road traffic management. The equipment ontology uses some core common ontologies such as W3C Organization Ontology [https://www.w3.org/TR/vocab-org/] and the W3C Prov-O ontology. The equipment ontology also uses the Mereology and Identifier Ontologies (provided in Testbed 10).

NOTE	These ontologies can be found as TTL files in the Aviation Task section of the Testbed-16 GitLab Repository.

11.3.1. Equipment Ontology

The Equipment Ontology, as represented on Figure 26, describes base information about equipment and can be further extended into specific types of equipment. This ontology could be used and extended in other domain such as modeling vessels for water navigation or cars for road traffic management. For the purposes of this Testbed, this ontology was extended into an Aircraft Ontology.



equipment:EquipmentType
prov:Agent
equipment:ownerOf
prov:actedOnBehalfOf : prov:Agent
prov:qualifiedDelegation : prov:Delegation
equipment:EquipmentModel
equipment:equipmentType : equipment:EquipmentType[0..1]
equipment:modelNumber : [0..1]
identifier:hasIdentifier : identifier:Identifier[0..]
identifier:identifier : [0..]
equipment:Manufacturer
equipment:EquipmentOwner
equipment:ownerOf
equipment:equipmentModel
equipment:Equipment
equipment:equipmentModel : equipment:EquipmentModel[0..1]
equipment:equipmentType : equipment:EquipmentType[0..1]
equipment:manufacturedBy : equipment:Manufacturer[0..]
equipment:modelNumber : [0..1]
equipment:operatedBy : prov:Agent[0..]
equipment:ownedBy : prov:Agent[0..]
equipment:registeredBy : prov:Agent[0..]
equipment:registrationNumber : [0..1]
identifier:hasIdentifier : identifier:Identifier[0..]
equipment:registrationDate : date[0..1]
identifier:identifier : [0..]
equipment:ownerOf some equipment:Equipment
equipment:EquipmentPart
equipment:EquipmentOperator
equipment:operatorOf : equipment:Equipment

Figure 26. Equipment Ontology Module

11.3.2. Aircraft Ontology

The Aircraft Ontology, as represented on Figure 27, defines more specific classes and restrictions on the equipment model related to aircraft, aircraft model, aircraft types, aircraft manufacturer, operator and owner.

This ontology was intended to model static data from the FAA Aircraft Registration Database into aircraft triples in the Triple Builder. The Aircraft Ontology could be further refined by adding information such as number of seats or wake category.

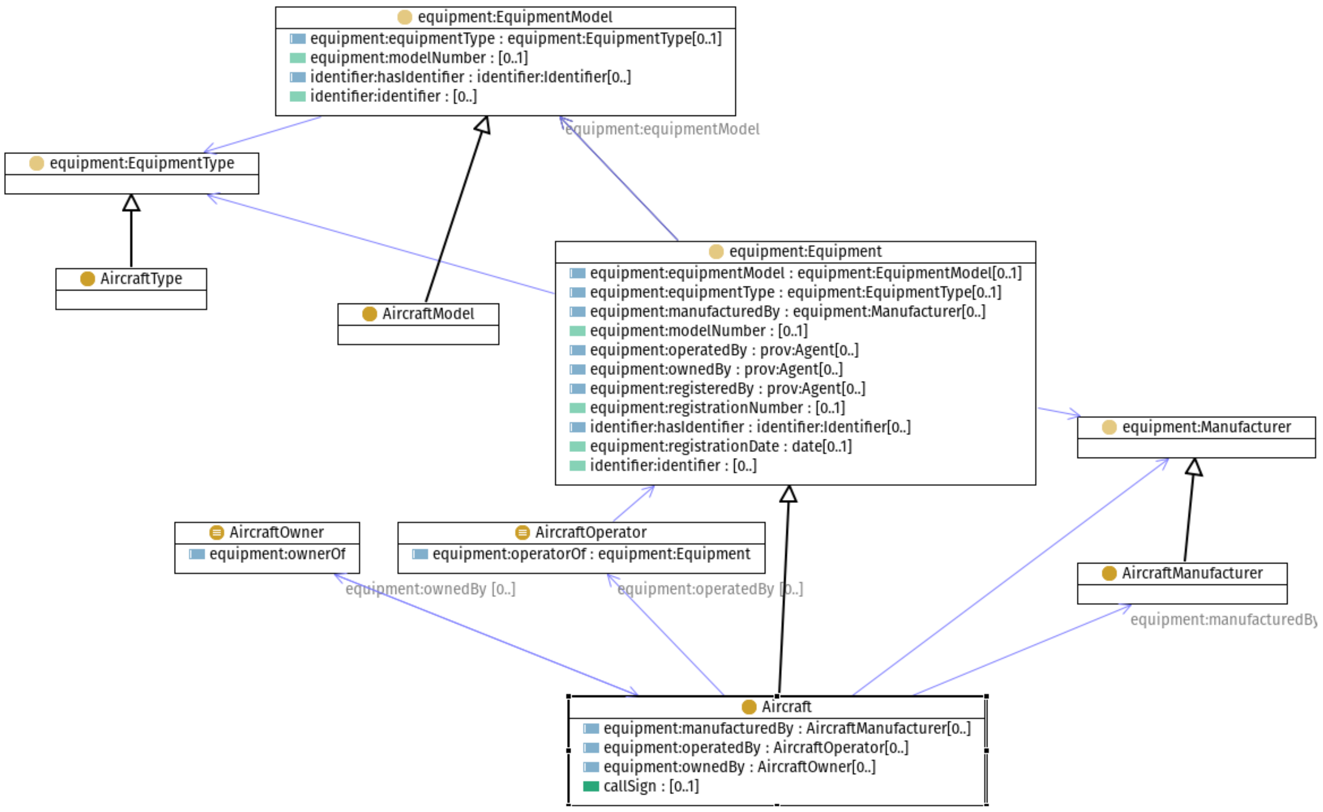

equipment:EquipmentModel
equipment:equipmentType : equipment:EquipmentType[0..1]
equipment:modelNumber : [0..1]
identifier:hasIdentifier : identifier:Identifier[0..]
identifier:identifier : [0..]
equipment:EquipmentType
equipment:equipmentModel
AircraftType
AircraftModel
equipment:Equipment
equipment:equipmentModel : equipment:EquipmentModel[0..1]
equipment:equipmentType : equipment:EquipmentType[0..1]
equipment:manufacturedBy : equipment:Manufacturer[0..]
equipment:modelNumber : [0..1]
equipment:operatedBy : prov:Agent[0..]
equipment:ownedBy : prov:Agent[0..]
equipment:registeredBy : prov:Agent[0..]
equipment:registrationNumber : [0..1]
identifier:hasIdentifier : identifier:Identifier[0..]
equipment:registrationDate : date[0..1]
identifier:identifier : [0..]
equipment:Manufacturer
AircraftOwner
equipment:ownerOf
AircraftOperator
equipment:operatorOf : equipment:Equipment
AircraftManufacturer
equipment:ownedBy [0..]
equipment:operatedBy [0..]
equipment:manufacturedBy
Aircraft
equipment:manufacturedBy : AircraftManufacturer[0..]
equipment:operatedBy : AircraftOperator[0..]
equipment:ownedBy : AircraftOwner[0..]
callSign : [0..1]

Figure 27. Aircraft Ontology Module

11.3.3. Flight Ontology

The Flight Ontology was built for the Triple Builder to model FIXM flight data into triples. A Flight can be seen as an activity bounded by two major events, departure and arrival, which are both temporal entities. This ontology (as represented on Figure 28) uses `W3C OWL Time` to model Temporal Entity, `W3C Prov-O` to model `prov-o:Activity`. The latter is refined and extended by the Activity and Event ontologies provided by Image Matters in Testbed 10.

Leveraging standard ontologies can enable performing temporal and event-based reasoning on any activities (in this case flight activity). The Flight concept was also related to Aircraft using the Aircraft Ontology.

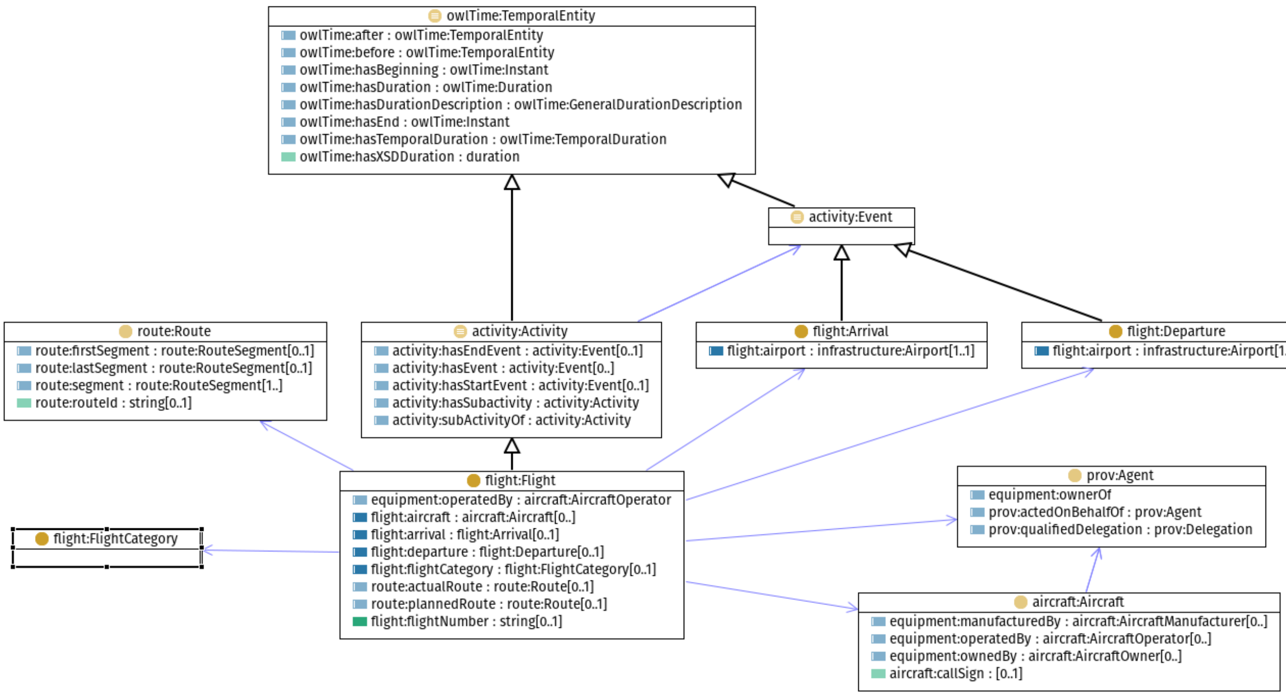

owlTime:TemporalEntity
owlTime:after : owlTime:TemporalEntity
owlTime:before : owlTime:TemporalEntity
owlTime:hasBeginning : owlTime:Instant
owlTime:hasDuration : owlTime:Duration
owlTime:hasDurationDescription : owlTime:GeneralDurationDescription
owlTime:hasEnd : owlTime:Instant
owlTime:hasTemporalDuration : owlTime:TemporalDuration
owlTime:hasXSDDuration : duration
activity:Event
route:Route
route:firstSegment : route:RouteSegment[0..1]
route:lastSegment : route:RouteSegment[0..1]
route:segment : route:RouteSegment[1..]
route:routeId : string[0..1]
activity:Activity
activity:hasEndEvent : activity:Event[0..1]
activity:hasEvent : activity:Event[0..]
activity:hasStartEvent : activity:Event[0..1]
activity:hasSubactivity : activity:Activity
activity:subActivityOf : activity:Activity
flight:Arrival
flight:airport : infrastructure:Airport[1..1]
flight:Departure
flight:airport : infrastructure:Airport[1..1]
flight:FlightCategory
flight:Flight
equipment:operatedBy : aircraft:AircraftOperator
flight:aircraft : aircraft:Aircraft[0..]
flight:arrival : flight:Arrival[0..1]
flight:departure : flight:Departure[0..1]
flight:flightCategory : flight:FlightCategory[0..1]
route:actualRoute : route:Route[0..1]
route:plannedRoute : route:Route[0..1]
flight:flightNumber : string[0..1]
prov:Agent
equipment:ownerOf
prov:actedOnBehalfOf : prov:Agent
prov:qualifiedDelegation : prov:Delegation
aircraft:Aircraft
equipment:manufacturedBy : aircraft:AircraftManufacturer[0..]
equipment:operatedBy : aircraft:AircraftOperator[0..]
equipment:ownedBy : aircraft:AircraftOwner[0..]
aircraft:callSign : [0..1]

Figure 28. Flight Ontology

11.3.4. Route Ontology

The Route Ontology (as represented on Figure 29) was developed mainly to be used by the Flight Ontology. A Route describes the path consisting of a collection of nodes and ordered segments connecting them; as an example, nodes could represent airports or waypoints, and segments could represent airways. Routing information was modularized in order to be used in different domains such as road routing or vessel navigation routing. The classes `Route`, `RouteSegment` and `RouteNode` were defined as a sub class of a GeoSPARQL `SpatialObject` class.

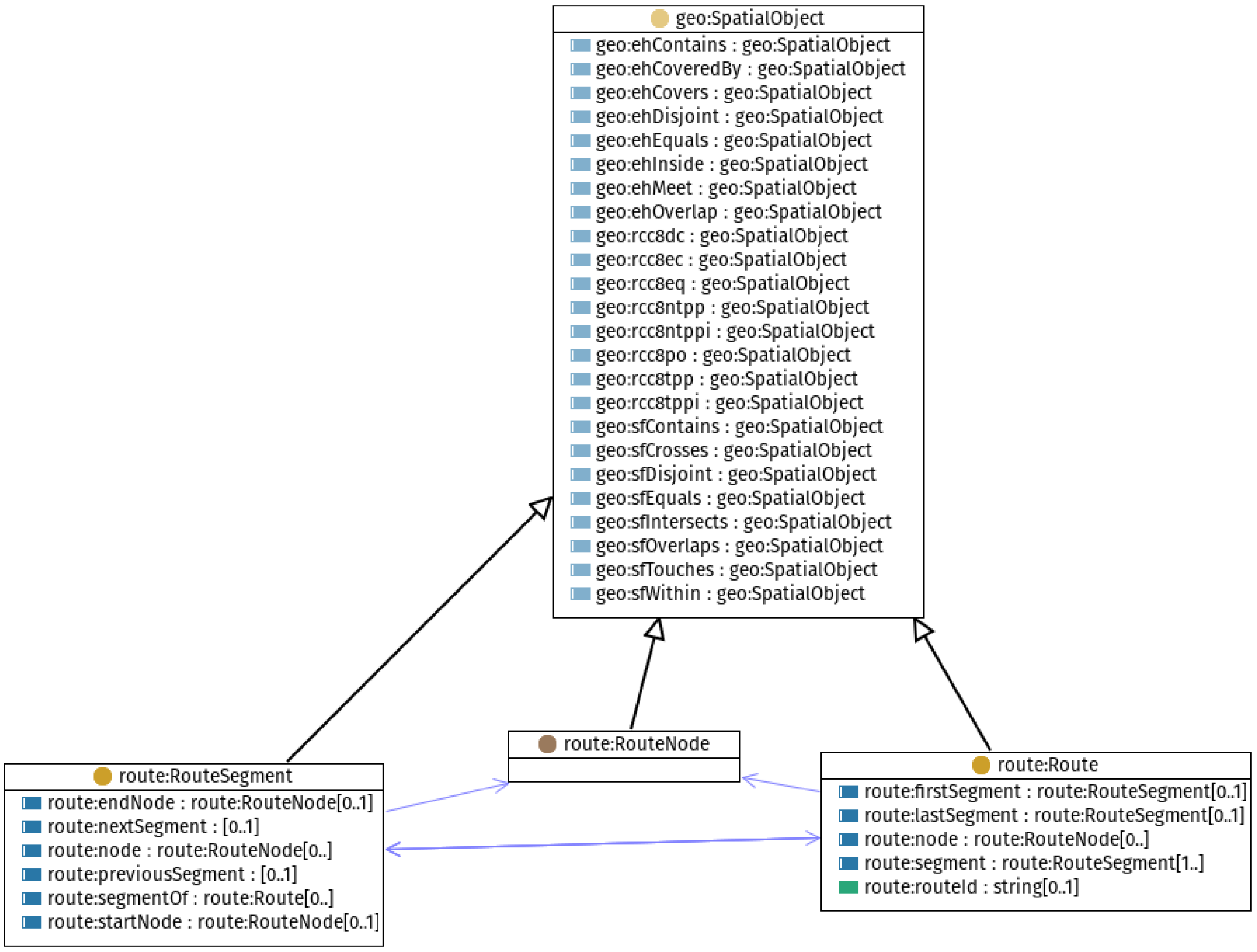

geo:SpatialObject
geo:ehContains : geo:SpatialObject
geo:ehCoveredBy : geo:SpatialObject
geo:ehCovers : geo:SpatialObject
geo:ehDisjoint : geo:SpatialObject
geo:ehEquals : geo:SpatialObject
geo:ehInside : geo:SpatialObject
geo:ehMeet : geo:SpatialObject
geo:ehOverlap : geo:SpatialObject
geo:rcc8dc : geo:SpatialObject
geo:rcc8ec : geo:SpatialObject
geo:rcc8eq : geo:SpatialObject
geo:rcc8ntpp : geo:SpatialObject
geo:rcc8ntppi : geo:SpatialObject
geo:rcc8po : geo:SpatialObject
geo:rcc8tpp : geo:SpatialObject
geo:rcc8tppi : geo:SpatialObject
geo:sfContains : geo:SpatialObject
geo:sfCrosses : geo:SpatialObject
geo:sfDisjoint : geo:SpatialObject
geo:sfEquals : geo:SpatialObject
geo:sfIntersects : geo:SpatialObject
geo:sfOverlaps : geo:SpatialObject
geo:sfTouches : geo:SpatialObject
geo:sfWithin : geo:SpatialObject
route:RouteNode
route:RouteSegment
route:endNode : route:RouteNode[0..1]
route:nextSegment : [0..1]
route:node : route:RouteNode[0..]
route:previousSegment : [0..1]
route:segmentOf : route:Route[0..]
route:startNode : route:RouteNode[0..1]
route:Route
route:firstSegment : route:RouteSegment[0..1]
route:lastSegment : route:RouteSegment[0..1]
route:node : route:RouteNode[0..]
route:segment : route:RouteSegment[1..]
route:routeId : string[0..1]

Figure 29. Route Ontology

11.3.5. Facility Ontology

A general ontology for Facilities, as represented on Figure 30, was modularized in order to support its use in different domains, such as port facilities or train stations. The `Facility` class extends GeoSPARQL `Feature`, and introduces a subclass of `skos:Concept` called `FacilityType` which allows the classification of `Facility` using different taxonomies (a modeling technique called soft typing).

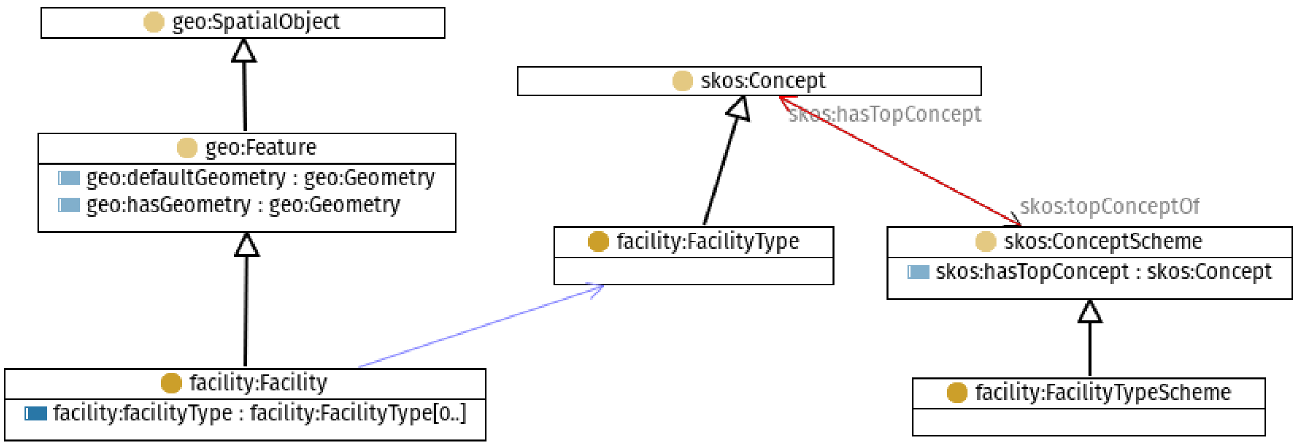

geo:SpatialObject
geo:Feature
geo:defaultGeometry : geo:Geometry
geo:hasGeometry : geo:Geometry
facility:Facility
facility:facilityType : facility:FacilityType[0..]
skos:Concept
skos:hasTopConcept
skos:topConceptOf
facility:FacilityType
skos:ConceptScheme
skos:hasTopConcept : skos:Concept
facility:FacilityTypeScheme

Figure 30. Facility Ontology

11.3.6. Aviation Infrastructure Ontology

The Aviation Infrastructure Ontology defines two types of infrastructures: Airspaces and airport infrastructures (including apron, taxiway, runway and heliport). Both types extended the GeoSPARQL `Feature` class, allowing users to use standard GeoSPARQL query extension to perform spatial queries on instances of these classes.

This ontology was used by the Triple Builder to generate the airport triples out of the airport data retrieved from the FAA website.

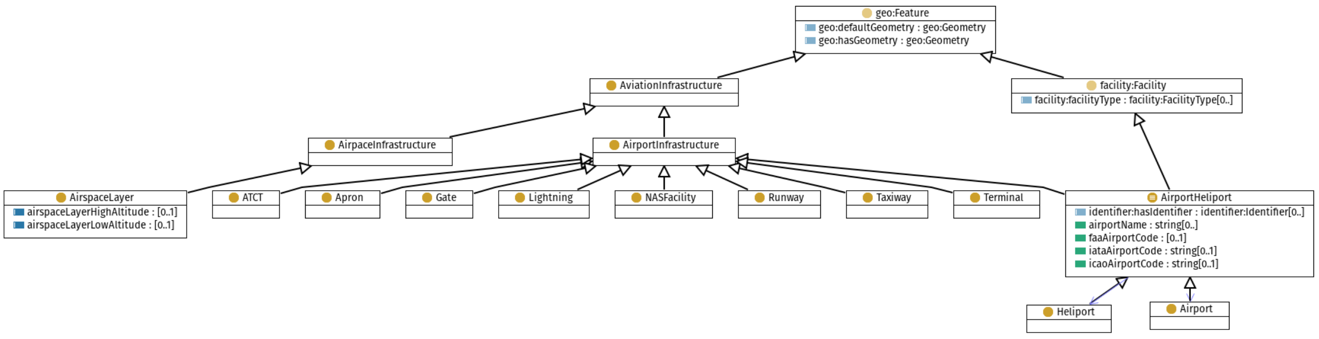

geo:Feature
geo:defaultGeometry : geo:Geometry
geo:hasGeometry : geo:Geometry
AviationInfrastructure
facility:Facility
facility:facilityType : facility:FacilityType[0..]
AirpaceInfrastructure
AirportInfrastructure
AirspaceLayer
airspaceLayerHighAltitude : [0..1]
airspaceLayerLowAltitude : [0..1]
ATCT
Apron
Gate
Lightning
NASFacility
Runway
Taxiway
Terminal
AirportHeliport
identifier:hasIdentifier : identifier:Identifier[0..]
airportName : string[0..]
faaAirportCode : [0..1]
iataAirportCode : string[0..1]
icaoAirportCode : string[0..1]
Heliport
Airport

Figure 31. Aviation Infrastructure Ontology

11.4. Challenges and Lessons Learned

- **Limitations of NASA ATM Ontologies**: The NASA Air Traffic Management (ATM) ontology was identified as a candidate for representing information related to air traffic. While this ontology covers a broad range of concepts and is built in a modular way, a deeper analysis found several issues that led to the creation of new ontologies:
    - ATM ontologies are not modularized enough so as to favor reusability and maintainability. As an example, a route ontology could be used for both air and maritime navigation.
    - ATM ontologies do not follow a layered approach of building ontologies, where each layer represent different levels of abstraction. This improves the quality of ontology modeling by better structuring the ontologies.
    - ATM ontologies do not leverage existing ontologies that are already well established such as OWL-Time, GeoSPARQL, and QUDT for quantity and unit of measures. This is a problem because existing tools that understand these ontologies cannot be leveraged (such as GeoSPARQL clients).
- **Topology support in GeoSPARQL**: Topological objects descriptions, such as nodes or edges, are not supported by GeoSPARQL. Having GeoSPARQL support topology would facilitate requests based on the topology and not just geometry thus providing a standardized way of querying topological features.

11.5. Accomplishments

- A set of ontologies was created that allowed the Triple Builder Component to build triples based on the aviation data retrieved from the SWIM Data Relay API (FIXM and AIXM) and from the static data related to aircrafts and airports.

Chapter 12. Semantic Web Client

The Semantic Web Client was the component designed to allow end users to request semantically-enriched information from the Triple Store in the form of complex queries, and display this information on a graphic interface.

The development of the Semantic Web Client was carried out by Image Matters. The main goals were to demonstrate the consumption of semantically-enriched aviation data, and to explore the utilization of GeoSPARQL for this purpose.

12.1. Status Quo

The SPARQL Protocol and RDF Query Language (SPARQL) are the W3C standard query language and protocol for Linked Data (RDF data). SPARQL 1.0 was published in 2008 and was later extended in SPARQL 1.1 in 2013 with support of updates and federated query in SPARQL 1.1. The standard has been widely adopted by RDF database vendors (RDF4J, AllegroGraph, GraphDB, AnzoGraph, Stardog, Jena, etc.) and a number of SPARQL APIs in different programming languages are readily available.

In 2011, the OGC published the GeoSPARQL standard, which supports representing and querying geospatial data on the Semantic Web. GeoSPARQL defines a vocabulary for representing geospatial data in RDF and defines an extension to the SPARQL query language for processing geospatial data. In addition, GeoSPARQL is designed to accommodate systems based on qualitative spatial reasoning and systems based on quantitative spatial computations. GeoSPARQL has also been adopted by a number of RDF database vendors. Recently, the open source RDF4J (previously named Sesame) has added GeoSPARQL support to their RDF database This technology was used in this Testbed as the Triple Store Solution.

A number of OGC Testbeds used GeoSPARQL in the past. For example, in Testbed 10 (OGC 14-029r2), federated gazetteers (Geonames, GNIS, and WFS-G) were virtualized as Linked Data using semantic mapping techniques from data to ontologies. GeoSPARQL was used to query these different databases and queries were rewritten toward the native data store (SQL, WFS Query). Results were returned following the SPARQL standard. Testbed-11 (OGC 15-066r1) explored the use of Semantic Linked Data and GeoSPARQL with RDF for National Map NHD and Gazetteer Data. In Testbed 13, GeoSPARQL was used in the Semantic Portrayal Service to define the rules of application of styling in feature data.

For this Testbed the use of GeoSPARQL was explored for aviation data stored in an RDF Triple Store (using RDF4J). A client was built to edit and perform GeoSPARQL queries and showing results in different visual representations (table, map, graph, charts).

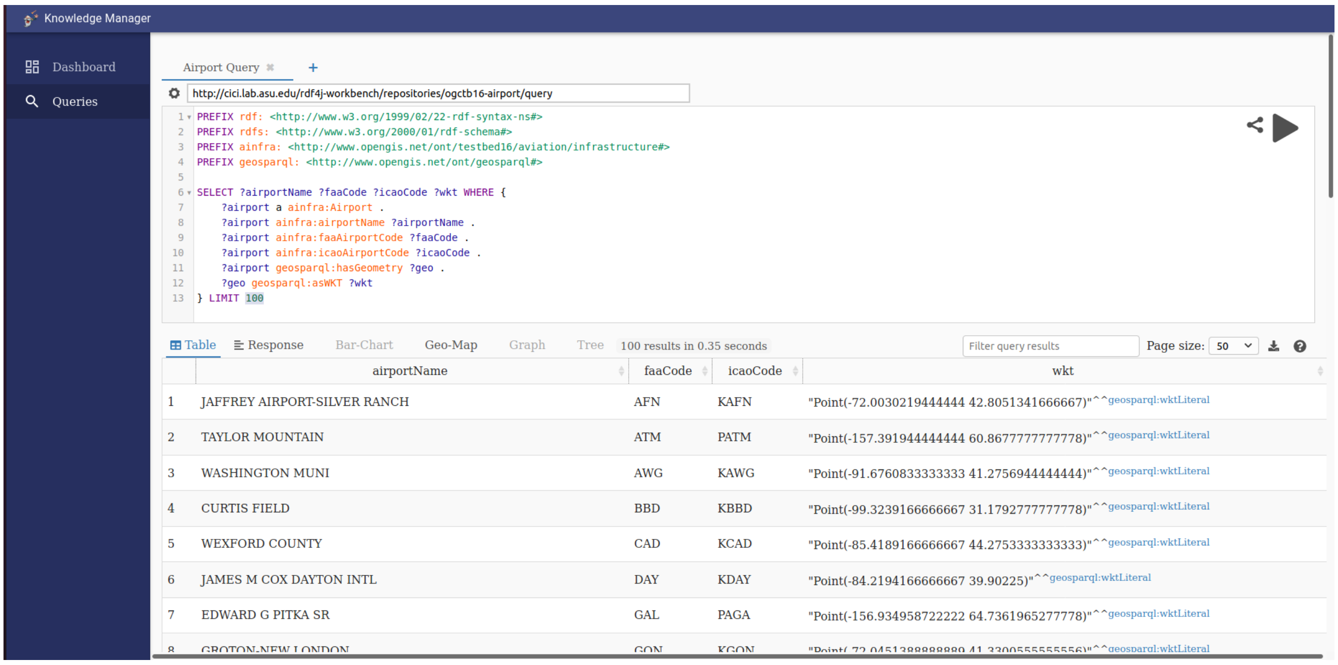
12.2. Functional Overview

The Semantic Web Client provides a SPARQL query editor with syntax highlighter and autocompletion. The client accesses the endpoint by using the SPARQL protocol. The response of the request can be SPARQL Results (XML or JSON, CSV, TSV) or RDF/XML (in case of SPARQL Construct or SPARQL Describe operation).

The editor was built integrating the open source YasGUI [https://github.com/TriplyDB/Yasgui], an API which provides various advanced features for creating, sharing, and visualizing SPARQL queries and their results. The visualization modes were extended using Apache EChart to render data as charts, maps and graph relationship views.

The results can be displayed in different modes:

- For SPARQL SELECT response, the data can be visualized as a table (as shown on Figure 32), as charts (bar charts, pie chart) when applicable, or maps (as shown on Figure 33) if data contains some geospatial data.



http://cici.lab.asu.edu/rdf4j-workbench/repositories/ogctb16-airport/query

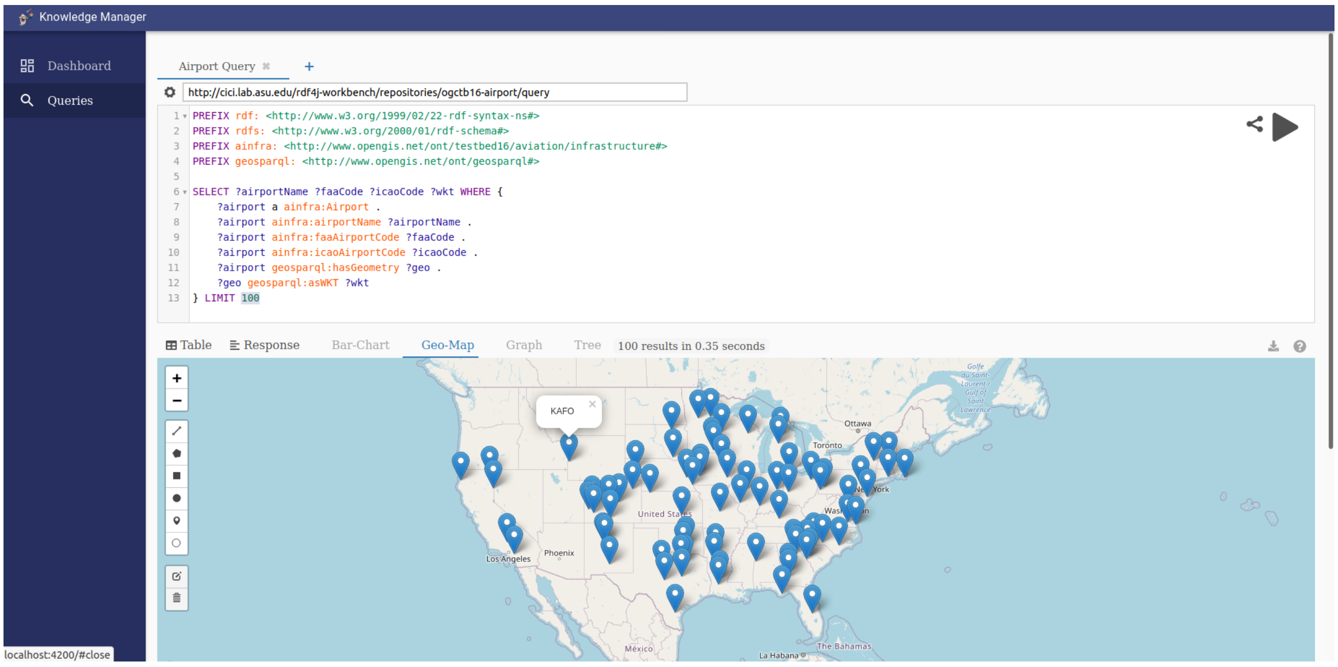
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ainfra: <http://www.opengis.net/ont/testbed16/aviation/infrastructure#>
PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>

SELECT ?airportName ?faaCode ?icaoCode ?wkt WHERE {
    ?airport a ainfra:Airport .
    ?airport ainfra:airportName ?airportName .
    ?airport ainfra:faaAirportCode ?faaCode .
    ?airport ainfra:icaoAirportCode ?icaoCode .
    ?airport geosparql:hasGeometry ?geo .
    ?geo geosparql:asWKT ?wkt
} LIMIT 100
```

100 results in 0.35 seconds

	airportName	faaCode	icaoCode	wkt
1	JAFFREY AIRPORT-SILVER RANCH	AFN	KAFN	"Point(-72.0030219444444 42.8051341666667)"^^geosparql:wktLiteral
2	TAYLOR MOUNTAIN	ATM	PATM	"Point(-157.391944444444 60.8677777777778)"^^geosparql:wktLiteral
3	WASHINGTON MUNI	AWG	KAWG	"Point(-91.6760833333333 41.2756944444444)"^^geosparql:wktLiteral
4	CURTIS FIELD	BBD	KBBD	"Point(-99.3239166666667 31.1792777777778)"^^geosparql:wktLiteral
5	WEXFORD COUNTY	CAD	KCAD	"Point(-85.4189166666667 44.2753333333333)"^^geosparql:wktLiteral
6	JAMES M COX DAYTON INTL	DAY	KDAY	"Point(-84.2194166666667 39.90225)"^^geosparql:wktLiteral
7	EDWARD G PITKA SR	GAL	PAGA	"Point(-156.934958722222 64.7361965277778)"^^geosparql:wktLiteral
8	GROTON-NEW LONDON	GON	KGON	"Point(-72.0451388888889 41.3300555555556)"^^geosparql:wktLiteral

Figure 32. Semantic Web Client Table View



http://cici.lab.asu.edu/rdf4j-workbench/repositories/ogctb16-airport/query

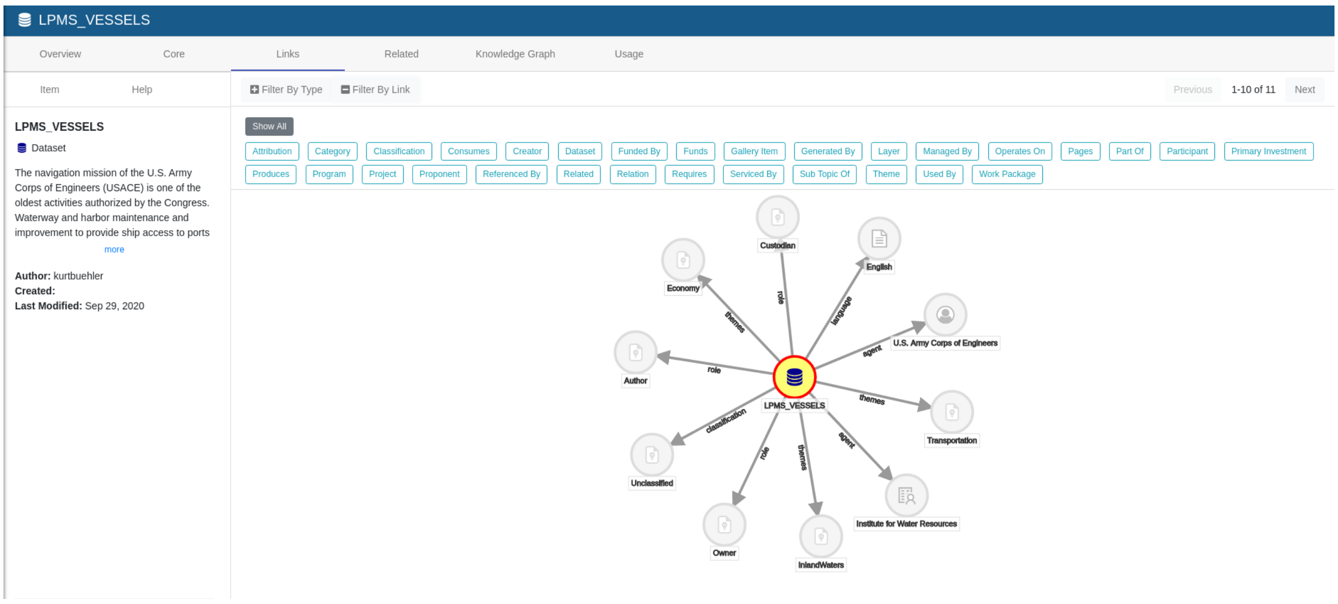
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ainfra: <http://www.opengis.net/ont/testbed16/aviation/infrastructure#>
PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>

SELECT ?airportName ?faaCode ?icaoCode ?wkt WHERE {
    ?airport a ainfra:Airport .
    ?airport ainfra:airportName ?airportName .
    ?airport ainfra:faaAirportCode ?faaCode .
    ?airport ainfra:icaoAirportCode ?icaoCode .
    ?airport geosparql:hasGeometry ?geo .
    ?geo geosparql:asWKT ?wkt
} LIMIT 100
```

100 results in 0.35 seconds

Figure 33. Semantic Web Client Map View

- In the case of RDF data response (SPARQL Construct or Describe request), a graph visualization is provided (as shown on Figure 34) that supports visualizing the relationship between the nodes of the RDF model.



LPMS_VESSELS

Dataset

The navigation mission of the U.S. Army Corps of Engineers (USACE) is one of the oldest activities authorized by the Congress. Waterway and harbor maintenance and improvement to provide ship access to ports

Author: kurtbuehler
Created:
Last Modified: Sep 29, 2020

Figure 34. Semantic Web Client Graph View

12.3. Challenges and Lesson Learned

- **Integration with GeoSPARQL endpoint**: This testbed demonstrated the ease of integration of an off-the-shelf open-source web component with a standard GeoSPARQL endpoint to retrieve semantically-enriched aviation data.
- **Lack of sufficient data for demonstration**: In order to perform a comprehensive and appealing demonstration of usage and integration of linked data, the amount and diversity of semantically-enriched aviation data is necessary. Unfortunately for this Testbed the data that was available for the Semantic Web Client was reduced due to the fact that the Triple Builder lacked sufficient ontologies to build larger sets of triples.
- **User-friendliness working with SPARQL**: While the SPARQL Editor seemed too advanced for a non-expert to use, the visualization of results seemed to be understandable by these end-users. Alternative ways to build queries could be explored in future Testbeds (for example by using parameterized queries), or a Linked Data REST API focused on features of interests).
- **Lack of GeoSPARQL service description**: The coordination between ASU and Image Matters made the existence of a GeoSPARQL service description unnecessary. These descriptions provide a mechanism by which a client or end user can discover information about the SPARQL service. There are a number of standards that already exists to describe SPARQL endpoints such as the W3C SPARQL 1.1 Service Description, VoiD and DCAT. Future work could demonstrate the feasibility and robustness of adapting one of these standards to accommodate descriptions of GeoSPARQL endpoints, by defining profiles and best practices.

12.4. Accomplishments

The Semantic Web Client was able to:

- Request from the Triple Store aviation Linked Data using GeoSPARQL queries.
- Display aviation Linked Data on a graphic user interface using different visualizations.

Chapter 13. SWIM Data Client

The SWIM Data Client was the component designed to allow end users to request, retrieve and visualize aviation data coming from SWIM Data Services through an OGC API - Features.

The development of the SWIM Data Client was carried out by Hexagon. The main goals of this development were to demonstrate the discovery of aviation data services, the connection and data fetching from an OpenAPI-based API, and the usage of the retrieved SWIM data.

13.1. Status Quo

The SWIM Data Client was based on LuciadRIA [https://www.hexagongeospatial.com/products/luciad-portfolio/luciadria], a solution for the development of geospatial situational awareness applications through the display of information in 3D maps running on browser-based environments. LuciadRIA is part of the Luciad suite of GIS applications focused on aviation, defense and security, developed by Hexagon. Hexagon has participated in previous OGC innovation initiatives exploring and demonstrating solutions for the consumption of SWIM data and the support of OGC services through their suite of Luciad applications.

The desktop version of the Luciad suite, Luciad Lightspeed, is capable of visualizing aviation data. Currently, LuciadRIA does not officially support AIXM and FIXM standards. Prototypes designed to decode AIXM and FIXM data directly in the browser were created for LuciadRIA, but are still not fully compliant with these standards. Previous LuciadRIA versions already supported connecting to OGC services such as WMS, WMTS, and WFS 1.0/2.0. This was Hexagon's first experience with OGC API - Features.

13.2. Functional Overview

To build this component, Hexagon used LuciadRIA version 2020.0. The main LuciadRIA API was not changed. The functionalities developed for this Testbed were built adding custom code on top of the API.

13.2.1. Visualization

The application displays aviation data through the 3D map environment of LuciadRIA. When loaded, collections are listed in a window where users can look them up, focus the map on them, toggle their visibility or remove them from the application.

Aviation data can be visualized combined with other non-aviation geospatial data. The capabilities developed in this Testbed were built compatible with LuciadRIA's map and feature visualization through its integration with other OGC services. Users are able to connect to a Web Map Service (WMS) or Web Feature Service (WFS) to retrieve and visualize a preferred base map (such as satellite imagery or road maps), as well as additional non-aviation data such as administrative jurisdictions or natural geographic features.

Users can visualize two types of aviation data:

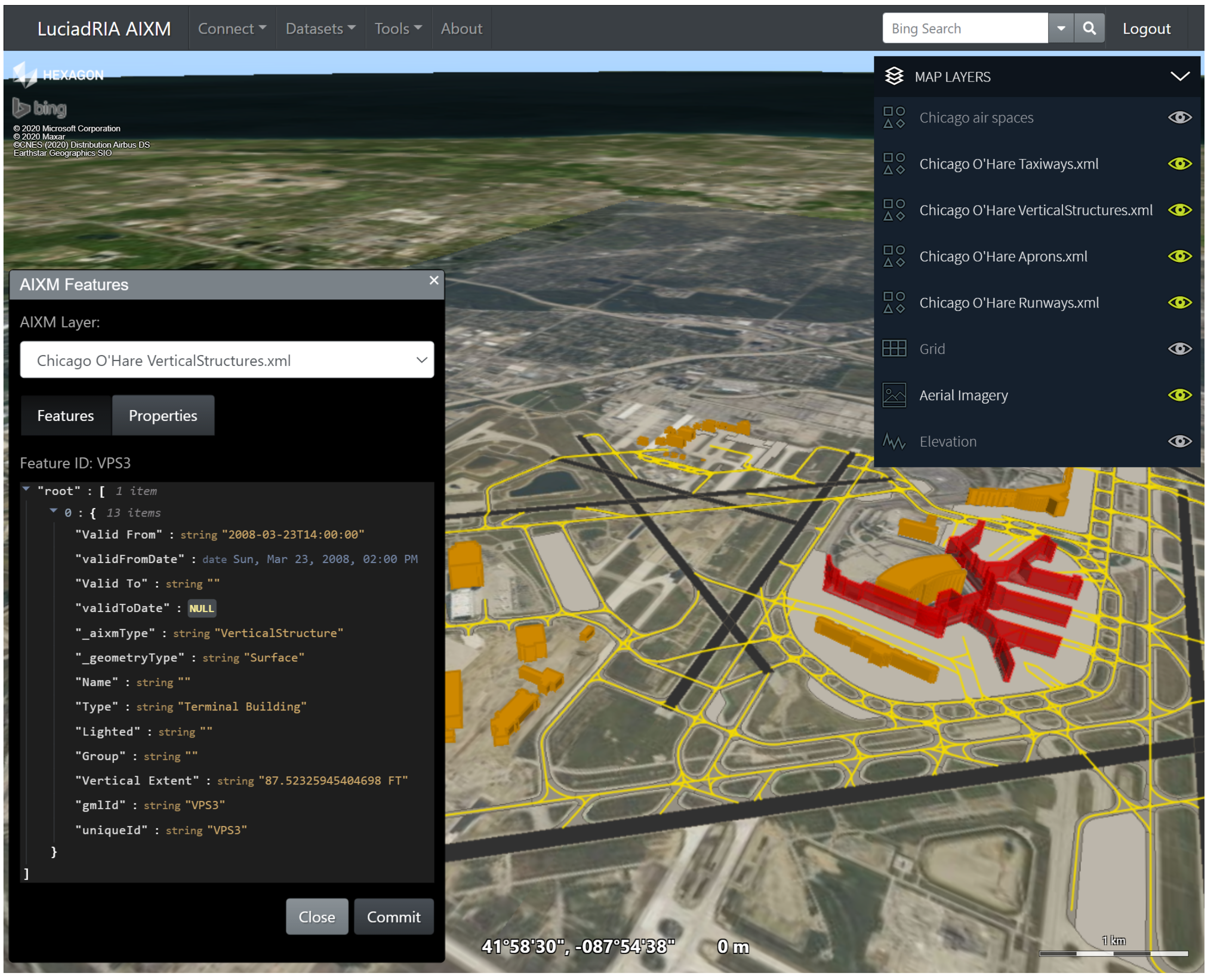
- Aeronautical information supported by AIXM such as airport structures and airspaces

- Flight information supported by FIXM such as flight information, location, and trajectory

Prototypes designed to decode AIXM and FIXM data directly in the browser were created for LuciadRIA, but are still not fully compliant with these standards. For Testbed 16, these prototypes were converted into reusable modules to facilitate the integration of this code into the SWIM Data Client Component. This modularization will also help Hexagon to reuse this code in future projects, as the company is considering adding full support for AIXM and FIXM in future LuciadRIA versions.

Since the SWIM Data Relay API delivered fully compatible AIXM data, the prototype for AIXM decoding was fully reused in this Testbed without changes. However, the prototype for FIXM decoding was not used because the SWIM Data Relay API delivered a FIXM version that is exclusive to the United States and for which Hexagon's codec was not compatible. Due to time constraints, this FIXM version was not used for this Testbed. As a work around, FIXM data was retrieved in GeoJSON format.

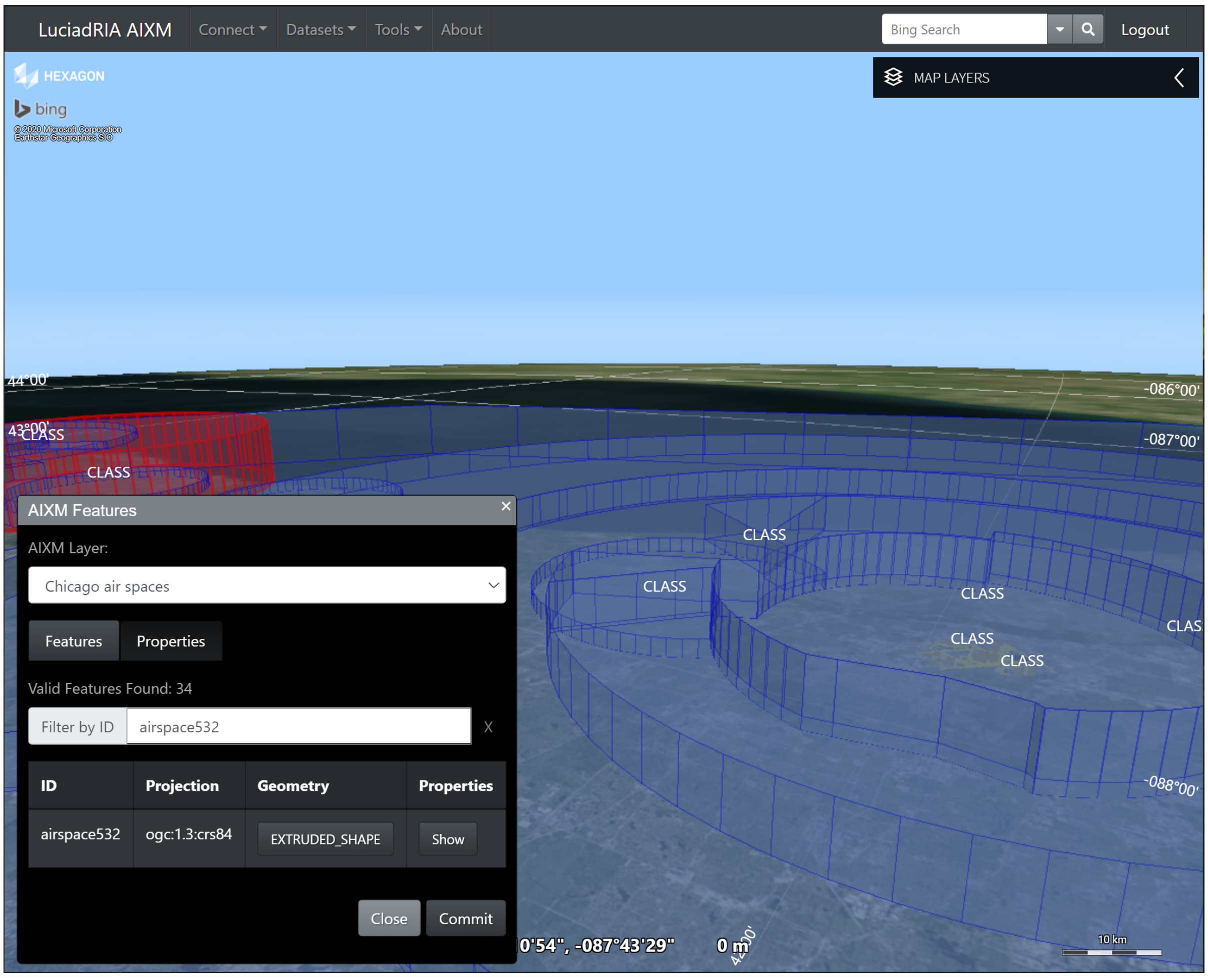
AIXM data is loaded natively in vector format. Features are visualized in the 3D map environment of LuciadRIA, and its properties can be explored by selecting them on the map and opening a tool named *AIXM Explorer*, as seen on Figure 35. This tool displays the feature ID, projection, geometry type and properties. The AIXM Explorer also supports searching features by layer or feature ID; this tool also allows users to focus the map on a feature when selecting it from the search results. When AIXM 5.1 format is used, proper aviation styling is applied following aviation standards, as seen on Figure 36.


LuciadRIA AIXM
Connect
Datasets
Tools
About
Bing Search
Logout
HEXAGON
bing
© 2020 Microsoft Corporation
© 2020 Maxar
©CNES (2020) Distribution Airbus DS
Earthstar Geographics SIO
MAP LAYERS
Chicago air spaces
Chicago O'Hare Taxiways.xml
Chicago O'Hare VerticalStructures.xml
Chicago O'Hare Aprons.xml
Chicago O'Hare Runways.xml
Grid
Aerial Imagery
Elevation
AIXM Features
AIXM Layer:
Chicago O'Hare VerticalStructures.xml
Features
Properties
Close
Commit
41°58'30", -087°54'38"
0 m
1 km

Feature ID: VPS3

```
"root" : [ 1 item
   0 : { 13 items
      "Valid From" : string "2008-03-23T14:00:00"
      "validFromDate" : date Sun, Mar 23, 2008, 02:00 PM
      "Valid To" : string ""
      "validToDate" : NULL
      "_aixmType" : string "VerticalStructure"
      "_geometryType" : string "Surface"
      "Name" : string ""
      "Type" : string "Terminal Building"
      "Lighted" : string ""
      "Group" : string ""
      "Vertical Extent" : string "87.52325945404698 FT"
      "gmlId" : string "VPS3"
      "uniqueId" : string "VPS3"
   }
]
```

Figure 35. LuciadRIA Display of Airport Structures and its Properties



AIXM Features

AIXM Layer: Chicago air spaces

Features | Properties

Valid Features Found: 34

Filter by ID: airspace532

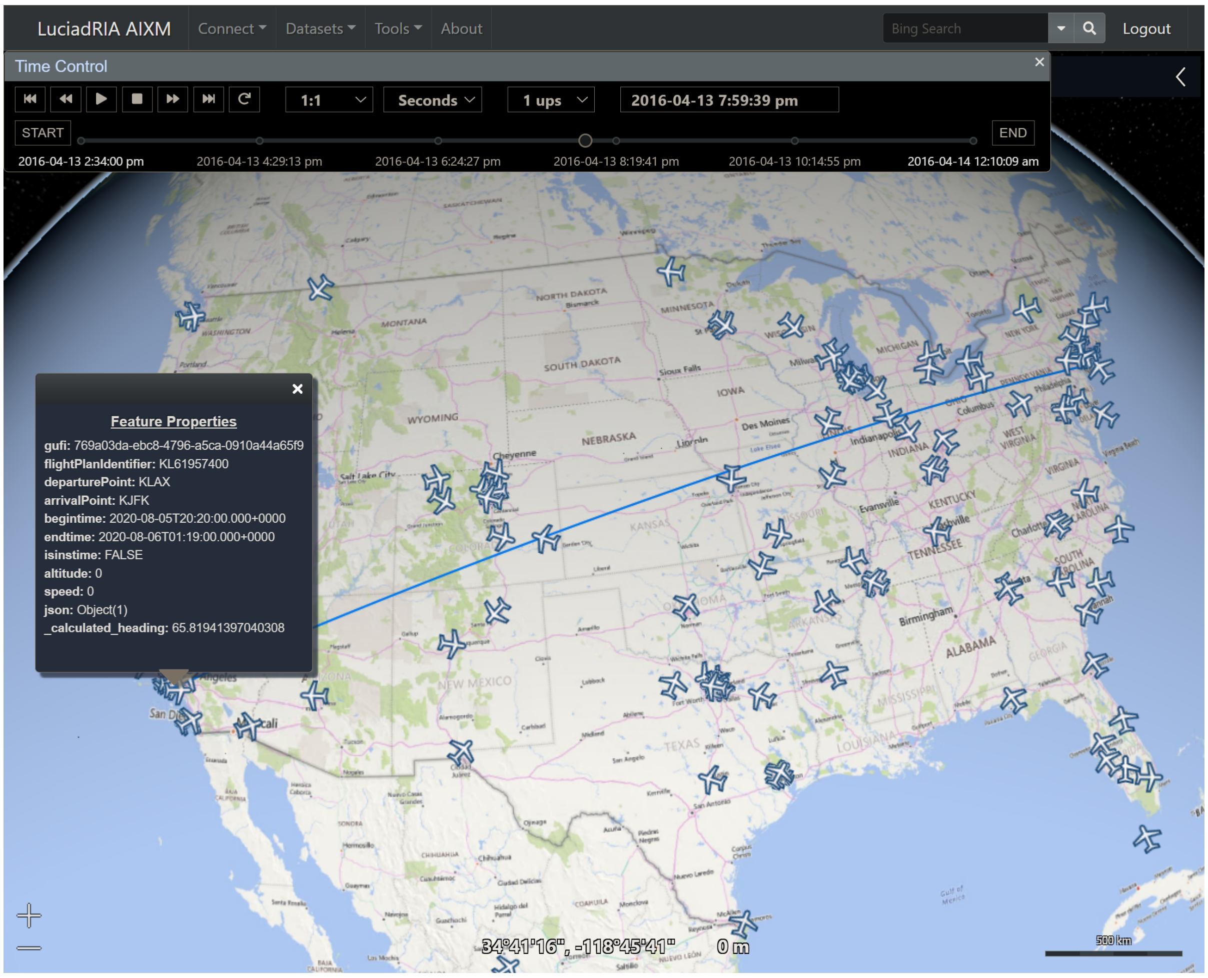
ID	Projection	Geometry	Properties
airspace532	ogc:1.3:crs84	EXTRUDED_SHAPE	Show

Close | Commit

Figure 36. LuciadRIA Search and Display of Airspaces

Visualizing FIXM data loaded with GeoJSON format required adapting the client application: When an end user flagged a loaded GeoJSON collection as containing FIXM data, the client applied a specific logic to interpret the properties contained inside those GeoJSON features as FIXM data and not as generic geographic elements.

Flight data is displayed on the map in the form of flight trajectories and an icon representing the location of the aircraft at a given time of the flight, as seen on Figure 37. A time control tool allows users to navigate throughout a given timespan: users are allowed to move forward or backward in time at different speeds, and aircrafts move accordingly on the map based on the trajectory included in the loaded FIXM data. Users can also explore feature properties of each flight by right clicking an aircraft and selecting "show properties" from the context menu.



Feature Properties

gufi: 769a03da-ebc8-4796-a5ca-0910a44a65f9
flightPlanIdentifier: KL61957400
departurePoint: KLAX
arrivalPoint: KJFK
begintime: 2020-08-05T20:20:00.000+0000
endtime: 2020-08-06T01:19:00.000+0000
isinstime: FALSE
altitude: 0
speed: 0
json: Object(1)
_calculated_heading: 65.81941397040308

Figure 37. LuciadRIA Display of FIXM Data

13.2.2. Connection and Retrieval of SWIM Data

For the purposes of this Testbed, LuciadRIA fetched aviation data from the SWIM Data Relay API Component created for this Testbed. The formats retrieved were JSON, FIXM, and AIXM, which were the ones made available by the SWIM Data Relay API. Two connection and retrieval methodologies were demonstrated:

- Connecting to an OGC API - Features Endpoint, and downloading collections published from a the service.
- Connecting to a Semantic Registry to discover services and collections, and downloading collections from the services discovered by the Semantic Registry.

13.2.2.1. Connecting to an OGC API - Features Endpoint

LuciadRIA allows users to load aviation data through a form designed to connect and retrieve data from OGC API - Features endpoints. The application also supports the capability to load aviation data from files by selecting repositories listed in a menu or by drag-&-drop AIXM files into the map (file needs to have extension .aixm).

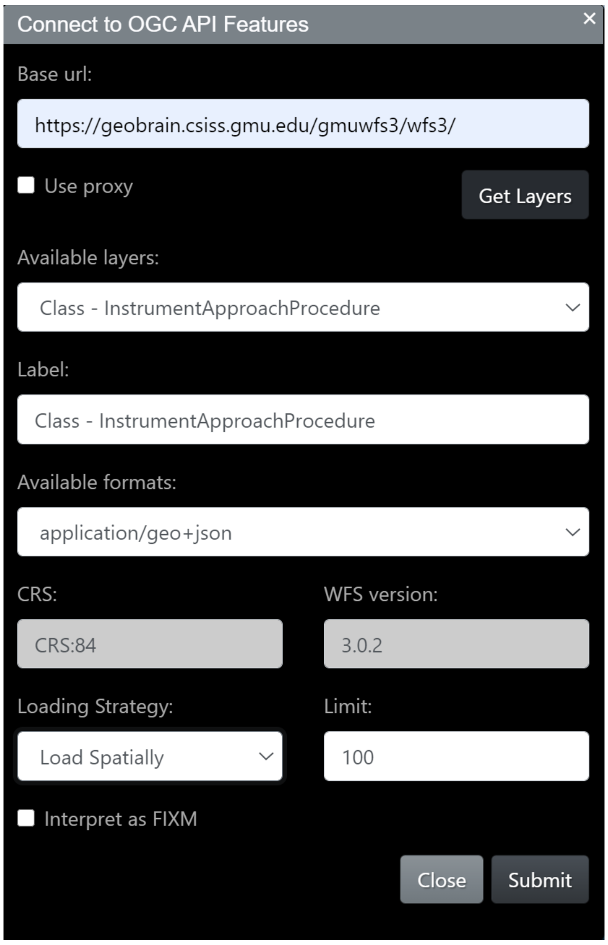
Testbed 16 was Hexagon’s first experience with OGC API - Features. The code to connect to this API was packed in a reusable module to facilitate future implementations that may require this service. Hexagon is considering officially supporting OGC API - Features in a future release of LuciadRIA once the standard matures and has wider implementation within the industry.

To load from an OGC API - Features endpoint, the user first types the server URL in order to load the available collections and formats published by the server (Figure 38). The user then selects from a dropdown menu the collection they want to load and the preferred collection format. Supported formats include GeoJSON, GML, and AIXM 5.1.

Finally, the form displays the CRS and WFS version, and provides the user with two feature loading strategies: *Load-All* and *Load-Spatially*. *Load-All* makes the application load all the features available in the requested collection. *Load-Spatially* makes the application load only the features visible on the screen at a given moment. As the user zooms or pans the map, the application loads new features to make them visible and removes from the memory those not visible. The *Limit* field determines the maximum number of features to be loaded at the same time. *Loading-Spatially* can be important since certain collections may consist of hundreds of thousands of features, and loading them concurrently could overload the application memory and crash the application.

The checkbox labeled *Interpret as FIXM* is used when FIXM data is loaded in GeoJSON format. It instructs the application to interpret GeoJSON features as FIXM data.

When loaded, the collections are listed in the Map Layers Window, where their visibility can be toggled on or off.



Connect to OGC API Features

Base url: https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/

☐ Use proxy

Available layers: Class - InstrumentApproachProcedure

Label: Class - InstrumentApproachProcedure

Available formats: application/geo+json

CRS: CRS:84

WFS version: 3.0.2

Loading Strategy: Load Spatially

Limit: 100

☐ Interpret as FIXM

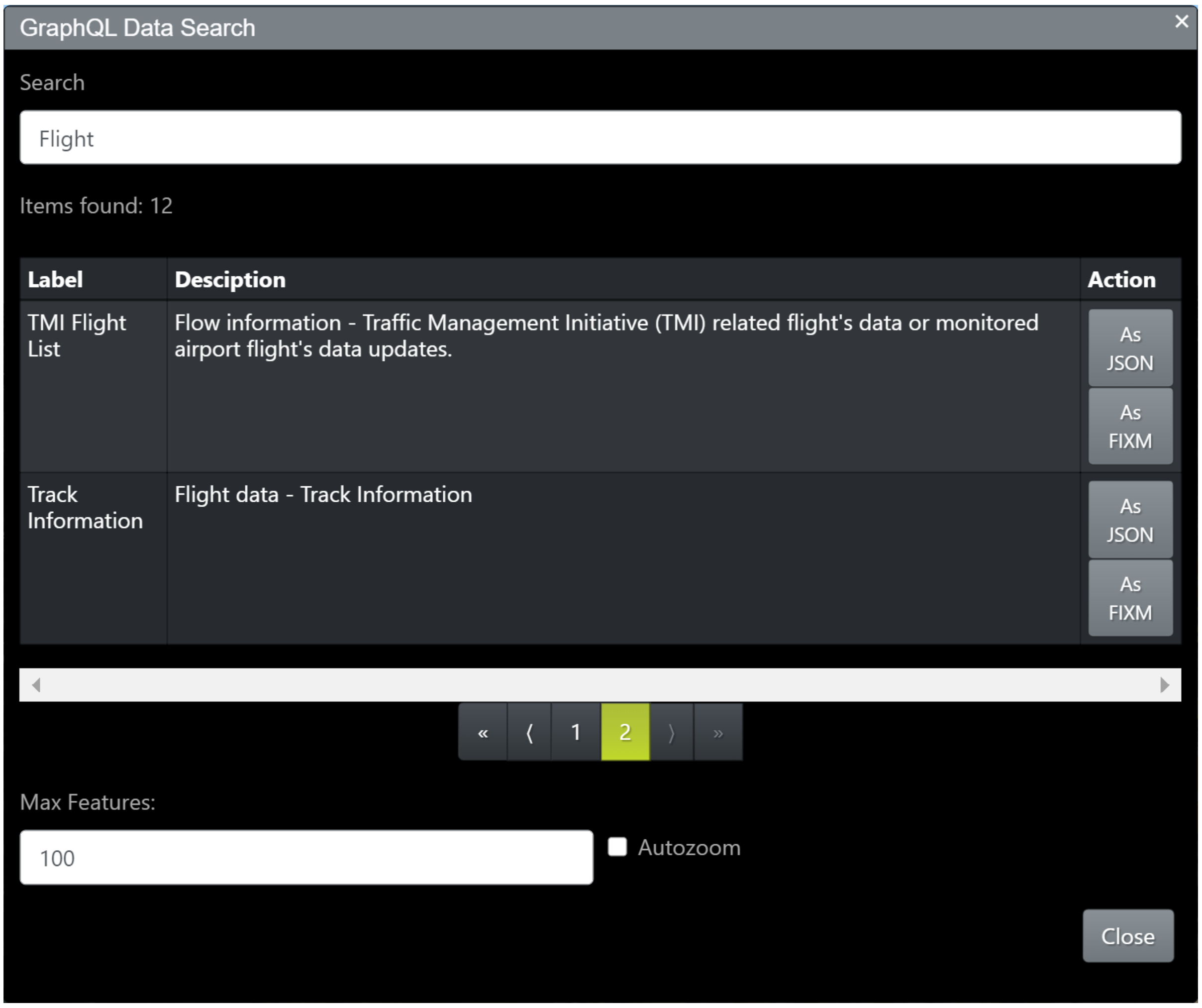
Figure 38. Connection Form for OGC API - Features

13.2.2.2. Connecting to a Semantic Registry

In order to demonstrate the service discovery capabilities of the Semantic Registry developed in this Testbed, a Resource Discovery Module was built into LuciadRIA, including a User Interface (UI).

The module connects to the Semantic Registry through a GraphQL interface, retrieves information about the collections discovered by the Semantic Registry, displays them on the UI (as seen on Figure 39), and allows the user to select which collections to retrieve.

For the purposes of this Testbed, the GraphQL queries were made partially dynamic: The endpoint was hardcoded to the Semantic Registry, the page size was set to 10, and the `type` filter was set to only `datasets`. The UI gives the user pagination and a textbox to query resources based on a given title or description.


GraphQL Data Search

Search

Flight

Items found: 12

Label	Desciption	Action
TMI Flight List	Flow information - Traffic Management Initiative (TMI) related flight's data or monitored airport flight's data updates.	As JSON / As FIXM
Track Information	Flight data - Track Information	As JSON / As FIXM

Max Features: 100

Autozoom

Figure 39. Service Discovery Module

The Resource Discovery Module displays on a UI the name and description of the discovered collections. The end user can then load the collections into LuciadRIA by clicking one of the action buttons included per collection, as seen on Figure 39. There is one action button per supported data format: users might be able to load collections as FIXM or JSON, based on availability. When loaded, the collections are listed in the Map Layers Window, where their visibility can be toggled on or off.

A sample GraphQL call and its response has been included in Appendix E.

13.3. Challenges and Lessons Learned

- **Benefits of not Loading AIXM as GeoJSON**: GeoJSON is a generic format that is not only used in aviation but also in a large number of other domains. This makes GeoJSON, in principle, an appropriate standard for the exchange of geospatial information. Nevertheless, during this

Testbed a decision was made to use the AIXM format for two main reasons:

  - AIXM is the native format used in SWIM, which means that once a certain dataset is identified as AIXM the consumer can safely assume the data (airspaces, runways, navaids) can be handled and styled following aviation standards. GeoJSON, on the contrary, consists of abstract geometries that require clients to access additional information in order to interpret the data. Therefore, using AIXM helped make the component easier and faster to implement.
  - AIXM uses GML to define the geometries. Since GML supports more geometries than GeoJSON, any conversion from AIXM to GeoJSON might require simplifying geometries in order to make a dataset "fit" the GeoJSON specification. Therefore, using AIXM allowed the component to work with more complex geometries.

- **Downsides of Loading FIXM as GeoJSON**: Work on this Testbed demonstrated the downsides of loading FIXM data as GeoJSON format. When data is loaded as GeoJSON, features are normally displayed in the map as geometries and no further interpretations are made by the client application. FIXM data consists of different types of information - including flight trajectories, flight attributes, and aircraft location based on a timestamp – not supported by GeoJSON.

Since for this Testbed FIXM data was retrieved in GeoJSON format, the client application required additional customizations to correctly interpret certain GeoJSON features as actual FIXM data. As an example, a GeoJSON polyline would normally be displayed on an application without further interpretations; in the case of FIXM data as GeoJSON, certain polylines actually represented the flight trajectories and the client application required a customization to handle this.

- **Value of the Semantic Registry**: Normally, LuciadRIA would have to make a call to one or several instances of a data service such as SWIM Data Relay API in order to retrieve metadata before making the calls required to retrieve the actual data. The Semantic Registry provided enough metadata information to LuciadRIA so as to directly retrieve collections from D100. The resource discovery module methodology prevented calling the SWIM Data Relay API to request metadata and instead called D100 only to retrieve the items from specific collections. Future iterations of this resource discovery module could see several additional capabilities:

  1. The Semantic Registry could be connected to multiple instances of APIs such as D100, each with different data, and instead of interrogating each instance of D100 separately a user could send a single query to the Semantic Registry in order to return matches from multiple servers.
  2. The UI could have more filtering capabilities and the GraphQL could be more dynamic, thus allowing users to filter discovered collections and services by more parameters.

- **Benefits of using GraphQL**: Hexagon decided to make the SWIM Data Client communicate with Semantic Registry through its GraphQL endpoint and not through its REST endpoint for several reasons:

  - The GraphQL interface was found to be well defined and easy to use.
  - GraphQL queries can specify multiple search parameters making more complex queries than a REST interface.
  - GraphQL was found to be more efficient for requesting complex queries to the Semantic Registry by making them on one HTTP call where a REST endpoint would have required

several HTTP calls.

* **Ease of use of OGC API - Features**: Hexagon was able to easily implement the connection to the SWIM Data Relay API, thanks in part to the many improvements that OGC API - Features has over WFS 2.0:
  * Better and easy-to-find online documentation.
  * Easiness of connection requiring a lower amount of code.
  * The human-readable HTML display of data which makes it easier for developers to test API calls and understand the data being returned.

13.4. Accomplishments

A LuciadRIA application was upgraded to successfully perform the following tasks:

* Connecting to a Semantic Registry through a GraphQL endpoint to retrieve information about aviation datasets and the endpoints providing it.
* Retrieving AIXM and FIXM data from an OGC API - Features endpoint. FIXM data was received as GeoJSON.
* Displaying on a 3D map environment both FIXM and AIXM data.

Chapter 14. Recommendations and Future Work

OGC Testbed 16 served as a practical approach to the subject of semantic enrichment and interoperability of aviation data. Concepts that were described in past OGC Testbeds from a conceptual perspective were built and interconnected in order to begin understanding how a semantically-enriched system would operate. Future work should build upon the findings that emerged from the development and testing of these components and to answer questions that were left out of scope.

14.1. Fostering the use of OGC API and Linked Data in aviation

Work in Testbed-16 demonstrated the benefits of using current and emerging OGC APIs but also outlined several downsides that may require adapting the perspective on how the aviation industry pretends to leverage their power of interoperability. The generation and consumption of aviation linked data in this Testbed laid the foundation for future activities that could facilitate the adoption of linked data in the aviation community.

14.1.1. Implementing OGC API - Features within SWIM Data Services

An aspect of the work in Testbed-16 consisted of demonstrating the value of OGC API - Features for consuming and relaying SWIM data by acting as a proxy between SWIM Data Services and SWIM Data Consumers. While this architecture did provide the benefits of using OpenAPI, the SWIM Data Relay API Component of this Testbed has demonstrated two major disadvantages: On one hand, the complexities involved in mapping the SWIM pub/sub messages into features, and on the other hand, the large volume of data being exchanged daily combined with the need to storage this information in a database inside the proxy.

Future work should explore using OpenAPI-based API descriptions within SWIM Data Services. This new architecture paradigm, represented in Figure 40, would eliminate the need for the massive additional storage and transformation logic required on an OpenAPI-based proxy. At the same time this paves the way for the standardization of SWIM Data Providers. An initial demonstration could be limited to implementing OGC API - Features on a SWIM Data Service currently based on a WFS endpoint that distributes a relatively reduced amount of data.

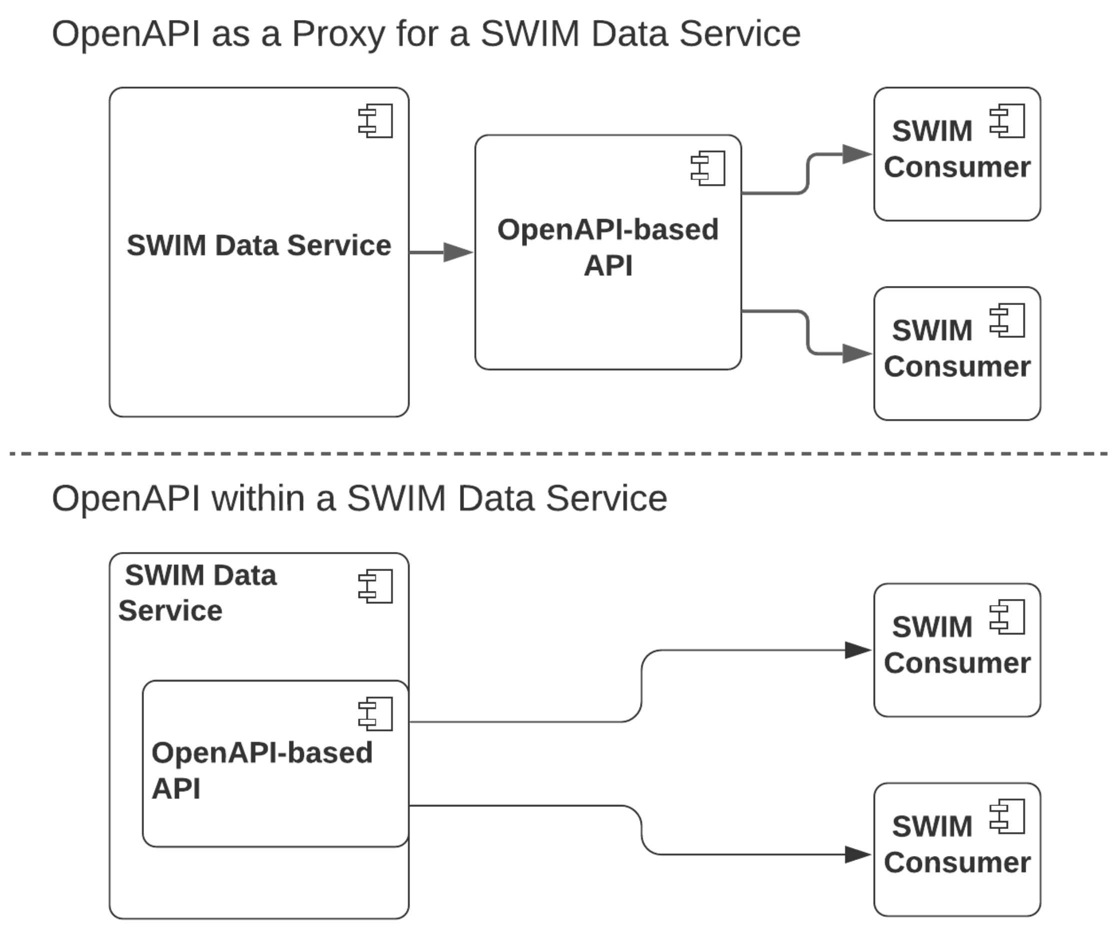


Figure 40. Linked Data Value Demonstration Experiment

14.1.2. Demonstrating The Value of Linked Data in Aviation

Transitioning SWIM into using Linked Data is a time - and cost -intensive endeavor. Demonstrating the benefits that would result from this transition has the potential of speeding up this process by making the industry understand the return of investment of such a transition. The components demonstrated in Testbed-16 could be used to perform experiments that evidence the benefits of linked data within the aviation domain from a practical point of view.

One possible experiment could consist of having two nearly-identical SWIM data clients consuming the same SWIM dataset, but where one of the datasets is enriched with semantics. By comparing the resulting capabilities of each client, a case could be made on the exact capabilities that arise from client applications implementing linked data.

The architecture of such an experiment would resemble part of the architecture of this Testbed. As seen in Figure 41, *OpenAPI-based API With Linked Data Support* could resemble Triple Builder and Triple Store, *SWIM Linked Data Client* could resemble Semantic Web Client, *OpenAPI-based API* could resemble SWIM Data Relay API, *SWIM Data Client* could resemble SWIM Data Client. The major differences in the new architecture would consist of making the clients virtually identical, having both APIs consume the same SWIM dataset, and performing the experiments in a controlled way.

The research question in this experiment could be: What problem does Client Application A solve compared to Client Application B?

To ensure unambiguity, the only differences the two client applications should have are those capabilities added to enable the use of linked data. Any external information added through the semantic enrichment process should also be made available to the client consuming the non-semantically-enriched data. As an example, the flight triples built throughout Testbed-16 were enriched with airport information. A comparison between the consumption of these flight triples and the consumption of the non-enriched flight data would only make sense if the client consuming the non-enriched flight data also has access to the airport information.

Close work with the industry would help ensure any experiment reflects an actual real industry case.

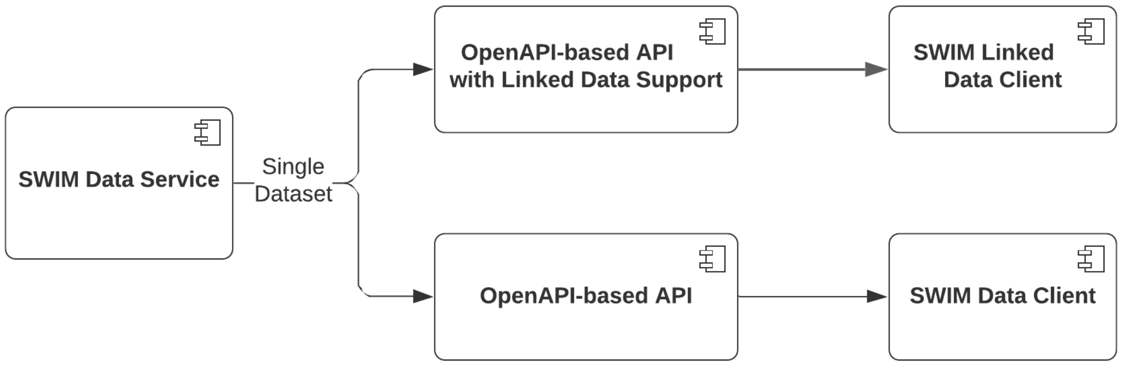


Figure 41. Linked Data Value Demonstration Experiment

14.1.3. Exploring Alternatives for a Seamless Transition to Linked Data

SWIM is made up of a large number of servers and clients exchanging aviation information. In order to transition from a data-centric to a semantics-centric paradigm, these aforementioned clients and servers would require a transformation which, as mentioned on the previous recommendation, would be time and cost intensive. Servers should be re-engineered, and clients would need to adapt to the re-engineered Services in order to keep compatibility and maintain their operations uninterrupted.

Finding alternatives to make this transition as seamless as possible could provide the aviation industry with the tools needed to shift into a semantically-enriched paradigm without necessarily leaving the current data-centric paradigm in the short term. SWIM data providers would be allowed to steadily increase the availability of semantically-enriched data while continuing to support the availability of the current data-centric datasets. This would allow existing SWIM data clients to adopt this new paradigm at their own pace. Furthermore, new SWIM data consumers would have the option of building their new clients with the new paradigm

This topic is currently the subject of exploration by the Geosemantics Community. The transition from JSON to JSON-LD by means of an added `@context` object has been the subject of discussions and studies to understand the potential of this technology as the one responsible of a smooth transition into a semantically-enriched paradigm [15]. A similar transition from XML to XML/RDF has also been subject to demonstrations. Future work could demonstrate Feature Servers advertising SWIM data through both JSON and JSON-LD or XML and XML/RDF.

14.1.4. Demonstrating Interoperability Between Diverse APIs

Testbed-16 demonstrated the behavior of a single OGC API - Features Service as a proxy between SWIM data providers and SWIM data consumers. In order to demonstrate interoperability, future work could see two or more diverse APIs be set up to deliver similar data, while one or more clients seamlessly consume data from the APIs.

This demonstration would test the ability of different organizations, using different technologies, to deliver similar content that clients can use seamlessly. Interoperability will require OGC API standards and also standardized data models. This demonstration would help resolve conflicts by comparing the interactions of the diverse APIs with the diverse API Clients, and provide further insight on the need to extend the OGC API - Features standard.

Furthermore, a similar experiment could explore in parallel different ways to search, discover and access linked data (SPARQL endpoint, REST, GraphQL, Linked Data Platform). Contrasting the gains and pains from these experiments would help the community leverage the value of linked data technologies.

14.1.5. Exploring Linked Data Support Alternatives for OGC APIs

The new OGC APIs being defined using OpenAPI are based on REST principles and thus can provide multiple representations for a given resource. Further work should investigate the best practices to make OGC APIs support Linked Data representations. A number of standards could be looked at, such as Linked Data Fragments (https://linkeddatafragments.org/), Linked Data Platform (LDP) (https://www.w3.org/TR/ldp/) and SPARQL Service description (https://www.w3.org/TR/sparql11-service-description/) and investigate how they can be aligned with OGC APIs.

14.2. Ontology Development

Work in Testbed-16 demonstrated the critical role of ontologies to support information integration and reasoning by means of search and discovery of assets and support of aviation linked data. The following recommendations focus on the expansion and improvement of the SRIM Model, Aviation Ontologies, and GeoSPARQL.

14.2.1. Expanding the Scope of Aviation Ontologies

Work in Testbed-16 demonstrated the need of defining ontologies to work with the different aviation data objects. In order to advance the process of semantic-enablement of aviation data, future work should expand the scope of these ontologies. The ontologies used for this testbed and the one provided by Image Matters during Testbed 10 could be used as a starting point.

The development of the ontologies should be done in a modular and layered way starting with *cross-domain ontologies* such as topology, quantity and unit of measures, mereology, geometry, spatial feature. A set of mid-level ontologies would then need to be developed such as facilities, equipment, infrastructure, facilities, routing. Finally, domain specific ontologies building on top of the cross-domain and mid-level ontologies should be developed for aviation such as aircrafts, flights, traffic, air infrastructure. This approach has been demonstrated during this testbed, but more work is needed to have a sufficient coverage to demonstrate the value of ontologies for aviation domain.

14.2.2. Standardizing the SRIM Model

Currently, there is no well-established standard to describe semantic assets, such as services, maps, layers, vocabularies, or portrayal information. Since OGC Testbed-12, the data catalog standards DCAT 2 and GeoDCAT have both gained wide-adoption. The SRIM model, being a superset of DCAT, has been proposed and applied in three different OGC Testbeds (12, 13, 16). To favor adoption of the Semantic Registry, the SRIM model should be standardized to represent registry items and assets. Guidance should be provided on how to define and use application profiles using the W3C SHACL standard, so registry items can be validated when submitted to the Semantic Registry.

14.2.3. Improving the GeoSPARQL Standard

OGC Testbed-10 identified the need to modularize and simplify GeoSPARQL. So far, advancements in this subject have been limited. As an example, in Testbed-16 the Route Ontology required defining topological elements from scratch. Topological objects descriptions (such as nodes or edges) and topological relations are not supported by GeoSPARQL. Having GeoSPARQL support topology would have facilitated requests to these topological elements.

The Testbed-10 geospatial ontologies addressed many of these aspects (for example modularization of spatial relations), and could be used as a starting point for this task. Additions could include:

- Topological objects descriptions and topological relations.
- Extensions to accommodate 2.5D and 3D geometries to represent airspace sections.
- Support for modeling moving features is also needed to model traffic information such as flights.
- A definition of a more compact representation of bounding boxes with CRS so that metadata can be used in other standards such as GeoDCAT to represent, for example, the spatial extent of a dataset.

14.3. Better Understanding the Industry by Engaging with SWIM Consumers

The SWIM Discovery Service (SDS) Workshop, carried out by the FAA near the end of this Testbed, provided not only an overview of the status quo on SWIM service discovery, but also served as a forum for interested members to express their points of view on the subject of consumption of SWIM data. When discussing the addition of semantics to the SDS, a point was made on the need to better understand the current requirements of the industry.

A plethora of activities could enhance our understanding of the industry’s needs. Follow-on FAA workshops could focus on:

- Discussing other SWIM components or concepts, such as semantics itself;
- Industry experts could be invited to a *Plugfest* where API and client developers explore solutions to SWIM challenges and experts provide their feedback.

A traditional but comprehensive survey could provide a clear understanding of basic questions about SWIM data consumers such as:

1. What technologies are end users ready to implement?
2. How is the lack of semantics affecting the day-to-day operations of end users?
3. Where do users see bottlenecks or inefficiencies in the retrieval and use of SWIM data?
4. What challenges did newcomers had to go through to implement their systems, and what would’ve smoothened their implementation processes?

Appendix A: SWIM Data Relay API: Sample FIXM to GeoJSON Mapping

This Annex describes an example of a FIXM message converted into a GeoJSON feature and its encoding of `application/fixm+nas+xml;version=3.0` in OGC API - Features output. Table 16 describes the origin of the mapped elements.

Table 16. FIXM to GeoJSON Property Mapping

Property	Origin
Geospatial position	`/flight/enRoute/position/position/location/pos`
Temporal extent	`/flight/enRoute/position/@positionTime`. When `isinstime` is true, `begintime` and `endtime` are identical.
`gufi`	`/flight/gufi/@codeSpace`
`flightPlanIdentifier`	`/flight/flightPlan/@identifier`
`departurePoint`	`/flight/departure/@departurePoint`
`arrivalPoint`	`/flight/arrival/@arrivalPoint`
`altitude`	`/flight/enRoute/position/targetAltitude`
`speed`	`/flight/enRoute/position/actualSpeed/surveillance`

The following snippets represent the same FIXM data encoded in three different formats:

1. The FIXM-NAS (original), which is the original encoding the data comes as from the SWIM Data Provider
2. The MessageCollection, which is the response in media type of "application/fixm+nas+xml;version=3.0" from the OGC API - Features Service. The response uses `MessageCollection` to wrap up the collection of features (messages) in encoding of FIXM NAS 3.0. An unique identifier is added as child element <metadata> of each feature (message) for uniquely identifying each feature.
3. The GeoJSON, which is one of the available outputs provided by SWIM Data Relay API

A.1. FIXM-NAS (original)

```
<flight centre="ZME" source="TH" system="ATL" timestamp="2020-08-05T21:40:29.319Z"
xsi:type="nas:NasFlightType">
    <arrival arrivalPoint="KDFW" xsi:type="nas:NasArrivalType">
        <runwayPositionAndTime>
            <runwayTime>
                <estimated time="2020-08-05T23:06:00Z"/>
```

```xml
            </runwayTime>
        </runwayPositionAndTime>
    </arrival>
    <controllingUnit sectorIdentifier="91" unitIdentifier="ZID" xsi:type=
"fb:IdentifiedUnitReferenceType"/>
    <departure departurePoint="KIND" xsi:type="nas:NasDepartureType">
        <runwayPositionAndTime>
            <runwayTime>
                <actual time="2020-08-05T21:23:00Z"/>
            </runwayTime>
        </runwayPositionAndTime>
    </departure>
    <enRoute xsi:type="nas:NasEnRouteType">
        <boundaryCrossings xsi:type="nas:NasUnitBoundaryType">
            <handoff xsi:type="nas:NasHandoffType">
                <receivingUnit sectorIdentifier="33" unitIdentifier="ZME" xsi:type=
"fb:IdentifiedUnitReferenceType"/>
            </handoff>
        </boundaryCrossings>
        <position positionTime="2020-08-05T21:40:18Z" reportSource="SURVEILLANCE"
targetPositionTime="2020-08-05T21:40:16Z" xsi:type="nas:NasAircraftPositionType">
            <actualSpeed>
                <surveillance uom="KNOTS">450.0</surveillance>
            </actualSpeed>
            <altitude uom="FEET">36000.0</altitude>
            <position xsi:type="fb:LocationPointType">
                <location srsName="urn:ogc:def:crs:EPSG::4326">
                    <pos>38.166389 -87.449444</pos>
                </location>
            </position>
            <targetAltitude uom="FEET">36000.0</targetAltitude>
            <targetPosition srsName="urn:ogc:def:crs:EPSG::4326">
                <pos>38.168889 -87.446111</pos>
            </targetPosition>
            <trackVelocity>
                <x uom="KNOTS">-324.0</x>
                <y uom="KNOTS">-312.0</y>
            </trackVelocity>
        </position>
    </enRoute>
    <flightIdentification aircraftIdentification="FDX3764" computerId="426"
siteSpecificPlanId="109" xsi:type="nas:NasFlightIdentificationType"/>
    <flightStatus fdpsFlightStatus="ACTIVE" xsi:type="nas:NasFlightStatusType"/>
    <gufi codeSpace="urn:uuid">060ef6c5-38c4-4582-9c30-e4e11044be15</gufi>
    <operator>
        <operatingOrganization>
            <organization name="FDX"/>
        </operatingOrganization>
    </operator>
    <supplementalData xsi:type="nas:NasSupplementalDataType">
        <additionalFlightInformation>
```

```
            <nameValue name="MSG_SEQ_NO" value="13875813"/>
            <nameValue name="FDPS_GUFI" value="us.fdps.2020-08-
05T17:53:54Z.000/08/400"/>
            <nameValue name="FLIGHT_PLAN_SEQ_NO" value="3"/>
        </additionalFlightInformation>
    </supplementalData>
    <assignedAltitude>
        <simple uom="FEET">36000.0</simple>
    </assignedAltitude>
    <flightPlan identifier="KI64434400"/>
</flight>
```

A.2. MessageCollection

```
<nas:MessageCollection xmlns:nas="http://www.faa.aero/nas/3.0" xmlns:fb=
"http://www.fixm.aero/base/3.0" xmlns:ff="http://www.fixm.aero/foundation/3.0"
xmlns:fx="http://www.fixm.aero/flight/3.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
    <message xmlns:fb="http://www.fixm.aero/base/3.0" xmlns:ff=
"http://www.fixm.aero/foundation/3.0" xmlns:fx="http://www.fixm.aero/flight/3.0"
xmlns:nas="http://www.faa.aero/nas/3.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="nas:FlightMessageType">
        <metadata gumi="FID-150061c7-5575-475b-8496-c938019fc8e0-5733" xsi:type=
"nas:MessageMetadataType"/>
        <flight centre="ZME" source="TH" system="ATL" timestamp="2020-08-
05T21:40:29.319Z" xsi:type="nas:NasFlightType">
            <arrival arrivalPoint="KDFW" xsi:type="nas:NasArrivalType">
                <runwayPositionAndTime>
                    <runwayTime>
                        <estimated time="2020-08-05T23:06:00Z"/>
                    </runwayTime>
                </runwayPositionAndTime>
            </arrival>
            <controllingUnit sectorIdentifier="91" unitIdentifier="ZID" xsi:type=
"fb:IdentifiedUnitReferenceType"/>
            <departure departurePoint="KIND" xsi:type="nas:NasDepartureType">
                <runwayPositionAndTime>
                    <runwayTime>
                        <actual time="2020-08-05T21:23:00Z"/>
                    </runwayTime>
                </runwayPositionAndTime>
            </departure>
            <enRoute xsi:type="nas:NasEnRouteType">
                <boundaryCrossings xsi:type="nas:NasUnitBoundaryType">
                    <handoff xsi:type="nas:NasHandoffType">
                        <receivingUnit sectorIdentifier="33" unitIdentifier="ZME"
xsi:type="fb:IdentifiedUnitReferenceType"/>
                    </handoff>
                </boundaryCrossings>
                <position positionTime="2020-08-05T21:40:18Z" reportSource=
"SURVEILLANCE" targetPositionTime="2020-08-05T21:40:16Z" xsi:type=
"nas:NasAircraftPositionType">
                    <actualSpeed>
                        <surveillance uom="KNOTS">450.0</surveillance>
                    </actualSpeed>
                    <altitude uom="FEET">36000.0</altitude>
                    <position xsi:type="fb:LocationPointType">
                        <location srsName="urn:ogc:def:crs:EPSG::4326">
                            <pos>38.166389 -87.449444</pos>
                        </location>
                    </position>
                    <targetAltitude uom="FEET">36000.0</targetAltitude>
                    <targetPosition srsName="urn:ogc:def:crs:EPSG::4326">
                        <pos>38.168889 -87.446111</pos>
                    </targetPosition>
                    <trackVelocity>
                        <x uom="KNOTS">-324.0</x>
                        <y uom="KNOTS">-312.0</y>
                    </trackVelocity>
                </position>
            </enRoute>
            <flightIdentification aircraftIdentification="FDX3764" computerId="426"
siteSpecificPlanId="109" xsi:type="nas:NasFlightIdentificationType"/>
            <flightStatus fdpsFlightStatus="ACTIVE" xsi:type="nas:NasFlightStatusType
"/>
            <gufi codeSpace="urn:uuid">060ef6c5-38c4-4582-9c30-e4e11044be15</gufi>
            <operator>
                <operatingOrganization>
                    <organization name="FDX"/>
                </operatingOrganization>
            </operator>
            <supplementalData xsi:type="nas:NasSupplementalDataType">
                <additionalFlightInformation>
                    <nameValue name="MSG_SEQ_NO" value="13875813"/>
                    <nameValue name="FDPS_GUFI" value="us.fdps.2020-08-
05T17:53:54Z.000/08/400"/>
                    <nameValue name="FLIGHT_PLAN_SEQ_NO" value="3"/>
                </additionalFlightInformation>
            </supplementalData>
            <assignedAltitude>
                <simple uom="FEET">36000.0</simple>
            </assignedAltitude>
            <flightPlan identifier="KI64434400"/>
        </flight>
    </message>
</nas:MessageCollection>
```

A.3. GeoJSON

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [
      38.1664,
      -87.4494
    ]
  },
  "properties": {
    "gufi": "060ef6c5-38c4-4582-9c30-e4e11044be15",
    "flightPlanIdentifier": "KI64434400",
    "departurePoint": "KIND",
    "arrivalPoint": "KDFW",
    "begintime": "2020-08-05T21:40:18.000+0000",
    "endtime": "2020-08-05T21:40:18.000+0000",
    "isinstime": true,
    "altitude": 10972.8,
    "speed": 833.4,
    "json": {
      "flight": {
        "centre": "ZME",
        "source": "TH",
        "system": "ATL",
        "timestamp": 1596663629319,
        "arrival": {
          "arrivalAerodromeAlternate": [],
          "runwayPositionAndTime": {
            "runwayTime": {
              "estimated": {
                "time": 1596668760000
              }
            }
          },
          "arrivalPoint": "KDFW"
        },
        "controllingUnit": {
          "sectorIdentifier": "91",
          "unitIdentifier": "ZID"
        },
        "dangerousGoods": [],
        "departure": {
          "runwayPositionAndTime": {
            "runwayTime": {
              "actual": {
                "time": 1596662580000
              }
            }
          },
          "takeoffAlternateAerodrome": [],
          "departurePoint": "KIND"
        },
        "enRoute": {
          "alternateAerodrome": [],
          "boundaryCrossings": [
            {
              "handoff": {
                "receivingUnit": {
                  "sectorIdentifier": "33",
                  "unitIdentifier": "ZME"
                }
              }
            }
          ],
          "controlElement": [],
          "position": {
            "actualSpeed": {
              "surveillance": {
                "value": 450,
                "uom": "KNOTS"
              }
            },
            "altitude": {
              "value": 36000,
              "uom": "FEET"
            },
            "position": {
              "location": {
                "pos": [
                  38.166389,
                  -87.449444
                ],
                "srsName": "urn:ogc:def:crs:EPSG::4326"
              }
            },
            "positionTime": 1596663618000,
            "reportSource": "SURVEILLANCE",
            "targetAltitude": {
              "value": 36000,
              "uom": "FEET"
            },
            "targetPosition": {
              "pos": [
                38.168889,
                -87.446111
              ],
              "srsName": "urn:ogc:def:crs:EPSG::4326"
            },
            "trackVelocity": {
              "x": {
                "value": -324,
                "uom": "KNOTS"
              },
              "y": {
                "value": -312,
                "uom": "KNOTS"
              }
            },
            "targetPositionTime": 1596663616000
          }
        },
        "extensions": [],
        "flightIdentification": {
          "marketingCarrierFlightIdentifier": [],
          "aircraftIdentification": "FDX3764",
          "computerId": "426",
          "siteSpecificPlanId": 109
        },
        "flightStatus": {
          "fdpsFlightStatus": "ACTIVE"
        },
        "gufi": {
          "value": "060ef6c5-38c4-4582-9c30-e4e11044be15",
          "codeSpace": "urn:uuid"
        },
        "operator": {
          "operatingOrganization": {
            "organization": {
              "name": "FDX"
            }
          }
        },
        "rankedTrajectories": [],
        "specialHandling": [],
        "supplementalData": {
          "additionalFlightInformation": {
            "nameValue": [
              {
                "name": "MSG_SEQ_NO",
                "value": "13875813"
              },
              {
                "name": "FDPS_GUFI",
                "value": "us.fdps.2020-08-05T17:53:54Z.000/08/400"
              },
              {
                "name": "FLIGHT_PLAN_SEQ_NO",
                "value": "3"
              }
            ]
          }
        },
        "assignedAltitude": {
          "simple": {
            "value": 36000,
            "uom": "FEET"
          }
        },
        "flightPlan": {
          "identifier": "KI64434400"
        }
      }
    }
  },
  "id": "FID-150061c7-5575-475b-8496-c938019fc8e0-5733",
  "links": [
    {
      "href":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items/FID-
150061c7-5575-475b-8496-c938019fc8e0-5733.fixm.nas.xml",
      "rel": "alternate",
      "type": "application/fixm+nas+xml;version=3.0",
      "title": "Feature FID-150061c7-5575-475b-8496-c938019fc8e0-5733"
    },
    {
      "href":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items/FID-
150061c7-5575-475b-8496-c938019fc8e0-5733.geo.json",
      "rel": "self",
      "type": "application/geo+json",
      "title": "Feature FID-150061c7-5575-475b-8496-c938019fc8e0-5733"
    },
    {
      "href":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items/FID-
150061c7-5575-475b-8496-c938019fc8e0-5733.html",
      "rel": "alternate",
      "type": "text/html",
      "title": "Feature FID-150061c7-5575-475b-8496-c938019fc8e0-5733"
    }
  ]
}
```

Appendix B: Semantic Registry: Sample Resource and Collection Mapping

This Annex describes an example of how a resource harvested from the NSRR was transformed into JSON, JSON-LD and TTL using the Semantic Registry.

B.1. Harvesting the NSRR

An instance of the Semantic Registry Harvester was configured to harvest the root index page URL of the NSRR; this page contains only a brief description of the service (id, title, description) and a link to the XML document describing a full description of the service. The Harvester extracted these links from index page (located at https://nsrr.faa.gov/rim/index) and then accessed them in order to fetch the complete XML service description document.

The following XML contains only two out of all the Services that can be found in the NSRR index page. The link to the detailed description of the service is found in the `xlink:href` attribute of the `RegisteredService` tag.

NSRR Index XML (Only Two Services Shown)

```
<Index
    xmlns="http://swim.aero/rim/1.0.0"
    xmlns:xlink="http://www.w3.org/1999/xlink" xlink:title="registryId" xlink:type=
"locator" xlink:href="http://nsrr.faa.gov">

    <RegisteredService xlink:href="http://nsrr.faa.gov/services/acs-dq" xlink:title=
"GRID" xlink:type="locator">
        <Name>Aeronautical Common Services Data Query (ACS-DQ) </Name>
        <Description>The ACS Data Query Service allows a service consumer to submit
queries and returns Special Use Airspace (SUA), SUA Schedules that match the query.
</Description>
        <Created>2020-03-04T08:23:36-05:00</Created>
        <LastModified>2020-03-04T08:23:36-05:00</LastModified>
        <ServiceProductCategory>
            <Taxonomy>http://semantics.aero/service-product</Taxonomy>
            <Code>http://semantics.aero/service-product#aeronautical</Code>
        </ServiceProductCategory>
        <AvailabilityStatusCategory>
            <Taxonomy>http://semantics.aero/availability-status</Taxonomy>
            <Code>http://semantics.aero/availability-status#operational</Code>
        </AvailabilityStatusCategory>
        <InterfaceTypeCategory>
            <Taxonomy>http://semantics.aero/interface-type</Taxonomy>
            <Code>http://semantics.aero/interface-type#method-oriented</Code>
        </InterfaceTypeCategory>
    </RegisteredService>
    <RegisteredService xlink:href="http://nsrr.faa.gov/services/acs-ds" xlink:title=
"GRID" xlink:type="locator">
```

```
        <Name>Aeronautical Common Services Data Subscription (ACS-DS)</Name>
        <Description>The Aeronautical Common Services Data Subscription (ACS-DS) is a
WS-Notification (WSN) service, which is a standard that describes a publish/subscribe
messaging model implemented over Web Services. The WS-Notification standard is defined
by combining the following OASIS specifications: WS-Topics, WS-BaseNotification, and
WS-BrokeredNotification. The service allows consumers to subscribe to notifications of
updates to aeronautical information. Consumers can use this service to create and
manage their subscriptions. The service allows users to create and manage PullPoints,
which is a resource used to accumulate notification messages, that can later be
retrieved at requested intervals. </Description>
        <Created>2020-03-04T08:23:36-05:00</Created>
        <LastModified>2020-03-04T08:23:36-05:00</LastModified>
        <ServiceProductCategory>
            <Taxonomy>http://semantics.aero/service-product</Taxonomy>
            <Code>http://semantics.aero/service-product#aeronautical</Code>
        </ServiceProductCategory>
        <AvailabilityStatusCategory>
            <Taxonomy>http://semantics.aero/availability-status</Taxonomy>
            <Code>http://semantics.aero/availability-status#operational</Code>
        </AvailabilityStatusCategory>
        <InterfaceTypeCategory>
            <Taxonomy>http://semantics.aero/interface-type</Taxonomy>
            <Code>http://semantics.aero/interface-type#method-oriented</Code>
        </InterfaceTypeCategory>
    </RegisteredService>
    ....
</Index>
```

The following XML contains the complete service description for the *ACS-DS* service. The Harvester extracted this information by following the link found in the NSRR index page.

Complete ACS-DS Service XML Description

```
<Profile xmlns="http://swim.aero/rim/1.0.0" xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:title="GRID" xlink:type="locator" xlink:href="http://nsrr.faa.gov/services/acs-
ds">
    <ServiceName xmlns="http://swim.aero/sdm-x/1.0.0">Aeronautical Common Services
Data Subscription (ACS-DS)</ServiceName>
    <ServiceVersion xmlns="http://swim.aero/sdm-x/1.0.0">1.0</ServiceVersion>
    <ServiceDescription xmlns="http://swim.aero/sdm-x/1.0.0">The Aeronautical Common
Services  Data Subscription (ACS-DS) is a WS-Notification (WSN) service, which is a
standard that describes a publish/subscribe messaging model implemented over Web
Services. The WS-Notification standard is defined by combining the following OASIS
specifications: WS-Topics, WS-BaseNotification, and WS-BrokeredNotification. The
service allows consumers to subscribe to notifications of updates to aeronautical
information. Consumers can use this service to create and manage their subscriptions.
The service allows users to create and manage PullPoints, which is a resource used to
accumulate notification messages, that can later be retrieved at requested intervals.
    </ServiceDescription>
    <ServiceCategories xmlns="http://swim.aero/sdm-x/1.0.0">
        <Category>
```

```
            <CategoryName>ATM Service Category</CategoryName>
            <Value>Flight Planning</Value>
        </Category>
        <Category>
            <CategoryName>SWIM Service Product Category</CategoryName>
            <Value>Aeronautical</Value>
        </Category>
        <Category>
            <CategoryName>Lifecycle Status</CategoryName>
            <Value>Deprecated</Value>
        </Category>
    </ServiceCategories>
    <Provider xmlns="http://swim.aero/sdm-x/1.0.0">
        <Name>Aeronautical Information Management Modernization (AIMM)</Name>
        <Description>AIMM's mission is to enhance the safety and efficiency of the NAS
by establishing a single trusted access point of digital Aeronautical Information
(AI). AIMM uses internationally adopted open standards, bringing AI into the FAA's
Service Oriented Architecture (SOA) via System Wide Information Management (SWIM)
compliant services and infrastructure.
        </Description>
        <WebPage>https://www.faa.gov/air_traffic/flight_info/aimm/</WebPage>
        <PointsOfContact>
            <POC>
                <Name>Davy andrew</Name>
                <Function>General Engineer</Function>
                <Phone>(202) 267-9582</Phone>
                <Email>davy.andrew@faa.gov</Email>
            </POC>
        </PointsOfContact>
    </Provider>
    <Functions xmlns="http://swim.aero/sdm-x/1.0.0">
        <Function>
            <Description>Querying and retrieving AI Data</Description>
            <RealWorldEffect>The ACS-DS has operations that aid the consumer in
querying and retrieving AI Data. Services allow the ability to query for AI data
features like SAA, SUA and integrated SAA/SUA data. In addition, this service  also
provides a publish/subscribe function that notifies subscribers when there is a change
to the AI Data.
            </RealWorldEffect>
        </Function>
    </Functions>
    <Security xmlns="http://swim.aero/sdm-x/1.0.0">
        <SecurityMechanism>
            <Name>Access Control Mechanisms</Name>
            <Description>The Service will use industry-standard, RBAC. User
authentication is required for each Service call. A username/password combination will
be contained in the SOAP header in each request to the Service. The Service will not
allow for any modification of AI data.
            </Description>
            <RegulatingProtocol>
                <Title></Title>
            </RegulatingProtocol>
        </SecurityMechanism>
    </Security>
    <Policies xmlns="http://swim.aero/sdm-x/1.0.0">
        <Policy>
            <Title>Web Services Policy 1.5 - Framework</Title>
            <Location>https://www.w3.org/TR/ws-policy/ws-policy-
framework.pdf</Location>
        </Policy>
    </Policies>
    <QualitiesOfService xmlns="http://swim.aero/sdm-x/1.0.0">
        <QualityOfServiceParameter>
            <Name>Availability</Name>
            <Value>.999  Essential Service</Value>
            <Definition>Probability that the service is present or ready for immediate
use.</Definition>
            <CalculationMethod>100 * ((24 Total Outage Time) / 24).Measurements are
taken daily and apply to the preceding 24-hour period.</CalculationMethod>
            <UnitOfMeasure>Percentage, accurate to 3 decimal places</UnitOfMeasure>
        </QualityOfServiceParameter>
        <QualityOfServiceParameter>
            <Name>Capacity</Name>
            <Value>The system will allow a minimum of 3000 concurrent ACS
consumers.</Value>
            <Definition>Number of service requests that the service can accommodate
within a given time period</Definition>
            <CalculationMethod>Simple count</CalculationMethod>
            <UnitOfMeasure>Whole number</UnitOfMeasure>
        </QualityOfServiceParameter>
        <QualityOfServiceParameter>
            <Name>Capacity</Name>
            <Value>The system will maintain subscription services with at least 30,000
users</Value>
            <Definition>Number of service requests that the service can accommodate
within a given time period </Definition>
            <CalculationMethod>Simple count</CalculationMethod>
            <UnitOfMeasure>Whole Number</UnitOfMeasure>
        </QualityOfServiceParameter>
        <QualityOfServiceParameter>
            <Name>ResponseTime</Name>
            <Value>The system will respond to airspace conflict detection requests for
a single airspace in 10 seconds or less</Value>
            <Definition>Maximum time required to complete a service
request.</Definition>
            <CalculationMethod>Measured from the time the service provider agent
receives the request to the time the service provider transmits the
response.</CalculationMethod>
            <UnitOfMeasure>Seconds</UnitOfMeasure>
        </QualityOfServiceParameter>
        <QualityOfServiceParameter>
            <Name>ResponseTime</Name>
            <Value>The system will meet an end-toend, one-way message latency within a
500 millisecond time range. The message to be tested will consist of a simple query
(e.g. based on a UUID) for a single AIXM feature element.</Value>
            <Definition>Maximum time required to complete a service
request.</Definition>
            <CalculationMethod>Measured from the time the service provider agent
receives the request to the time the service provider transmits the
response.</CalculationMethod>
            <UnitOfMeasure>Milliseconds</UnitOfMeasure>
        </QualityOfServiceParameter>
    </QualitiesOfService>
    <EnvironmentalConstraints xmlns="http://swim.aero/sdm-x/1.0.0">
        <Constraint>
            <Description>AIMM operates within the FAA Telecommunications
Infrastructure (FTI) and is subject to its performance constraints</Description>
        </Constraint>
    </EnvironmentalConstraints>
</Profile>
```

The Asset Importer for NSRR Data of the Semantic Registry converted the complete service description from XML format to JSON. A scrapper built within the Asset Importer extracted the Related Resources found on the NSRR HTML webpage.

The following JSON Service Representation resulted from the conversion of the harvested *ACS-DS* service description from XML to JSON, and the addition of its corresponding scrapped Related Resources.

ACS-DS JSON Service Representation

```
{
  "_created": 1602549647665,
  "_modified": 1602549647665,
  "_createdBy": "regp-admin",
  "_lastModifiedBy": "regp-admin",
  "_versionId": 1,
  "_status": "current",
  "_visibility": "public",
  "type": "regp:Service",
  "id": "b8685040798ddeb678d10f6c4b6dec7b",
  "uri": "https://ogctb16.usersmarts.com/id/service/e2b93bb45a645a02feead3229e836a39",
  "modified": 1602549647665,
  "label": "Aeronautical Common Services Data Subscription (ACS-DS) (v2.0)",
  "title": "Aeronautical Common Services Data Subscription (ACS-DS) (v2.0)",
  "description": "The Aeronautical Common Services  Data Subscription (ACS-DS) is a
WS-Notification (WSN) service, which is a standard that describes a publish/subscribe
messaging model implemented over Web Services. The WS-Notification standard is defined
by combining the following OASIS specifications: WS-Topics, WS-BaseNotification, and
WS-BrokeredNotification. The service allows consumers to subscribe to notifications of
updates to aeronautical information. Consumers can use this service to create and
manage their subscriptions. The service allows users to create and manage PullPoints,
which is a resource used to accumulate notification messages, that can later be
```

```
retrieved at requested intervals.\n",
  "publisher": [
    {
      "_created": 1602549647665,
      "_modified": 1602549647665,
      "_createdBy": "regp-admin",
      "_lastModifiedBy": "regp-admin",
      "id": "bd431254e13607887365030f0c40bef2",
      "uri":
"https://ogctb16.usersmarts.com/id/organization/9b80f98051d04aae7edc70872d4a3c35",
      "type": "org:Organization",
      "label": "Aeronautical Information Management Modernization (AIMM)",
      "name": "Aeronautical Information Management Modernization (AIMM)",
      "description": "AIMM's mission is to enhance the safety and efficiency of the
NAS by establishing a single trusted access point of digital Aeronautical Information
(AI). AIMM uses internationally adopted open standards, bringing AI into the FAA's
Service Oriented Architecture (SOA) via System Wide Information Management (SWIM)
compliant services and infrastructure. \n",
      "status": "submitted",
      "resourceType": [
        "foaf:Agent",
        "org:Organization"
      ],
      "_visibility": "public",
      "modified": 1602549647665,
      "_status": "current",
      "_versionId": 1
    }
  ],
  "status": "testing",
  "relatedResource": [
    {
      "type": "AuxiliaryResource",
      "uri": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/CDRL_E15_ACSDataSubscription_R3_WSDD_v3.5.pdf",
      "href": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/CDRL_E15_ACSDataSubscription_R3_WSDD_v3.5.pdf",
      "label": "CDRL_E15_ACSDataSubscription_R3_WSDD_v3.5.pdf",
      "title": "CDRL_E15_ACSDataSubscription_R3_WSDD_v3.5.pdf"
    },
    {
      "type": "AuxiliaryResource",
      "uri": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/ACSDataSubscription.zip",
      "href": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/ACSDataSubscription.zip",
      "label": "ACSDataSubscription.zip",
      "title": "ACSDataSubscription.zip"
    },
    {
      "type": "AuxiliaryResource",
      "uri": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/DTFAWA_14_C_00049_CDRL_E16_FINAL_ACS_R3_Web_Service_Requirements_Document_V3.4.pd
f",
      "href": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/DTFAWA_14_C_00049_CDRL_E16_FINAL_ACS_R3_Web_Service_Requirements_Document_V3.4.pd
f",
      "label":
"DTFAWA_14_C_00049_CDRL_E16_FINAL_ACS_R3_Web_Service_Requirements_Document_V3.4.pdf",
      "title":
"DTFAWA_14_C_00049_CDRL_E16_FINAL_ACS_R3_Web_Service_Requirements_Document_V3.4.pdf"
    },
    {
      "type": "AuxiliaryResource",
      "uri": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/CONOPS_AIMMS2_20140529_2%200.docx",
      "href": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/CONOPS_AIMMS2_20140529_2%200.docx",
      "label": "CONOPS_AIMMS2_20140529_2 0.docx",
      "title": "CONOPS_AIMMS2_20140529_2 0.docx"
    },
    {
      "type": "AuxiliaryResource",
      "uri": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/ACS_Web_Service_Faults.pptx",
      "href": "https://nsrr.faa.gov/sites/default/files/acs-
dsv2/ACS_Web_Service_Faults.pptx",
      "label": "ACS_Web_Service_Faults.pptx",
      "title": "ACS_Web_Service_Faults.pptx"
    }
  ],
  "theme": [
    {
      "_created": 1602549647665,
      "_modified": 1602549647665,
      "_createdBy": "regp-admin",
      "_lastModifiedBy": "regp-admin",
      "_visibility": "public",
      "id": "03dfe8bb1cf2a2b12544a2efa0b2465e",
      "uri": "http://semantics.aero/atm-service-category#Flight_Planning",
      "type": "skos:Concept",
      "label": "ATM Service Category",
      "prefLabel": "Flight Planning",
      "altLabel": [
        "ATM Service Category"
      ],
      "title": "ATM Service Category",
      "status": "submitted",
      "transitivePath": "http://semantics.aero/atm-service-category#Flight_Planning",
      "modified": 1602549647665,
      "_enriched": true,
      "_status": "current",
        "_versionId": 1
      },
      {
        "_created": 1602549647665,
        "_modified": 1602549647665,
        "_createdBy": "regp-admin",
        "_lastModifiedBy": "regp-admin",
        "_visibility": "public",
        "id": "a1eb5a3c6f7e16f1e0181c4ccc805e74",
        "uri": "http://semantics.aero/service-product#Aeronautical",
        "type": "skos:Concept",
        "label": "SWIM Service Product Category",
        "prefLabel": "Aeronautical",
        "altLabel": [
          "SWIM Service Product Category"
        ],
        "title": "SWIM Service Product Category",
        "status": "submitted",
        "transitivePath": "http://semantics.aero/service-product#Aeronautical",
        "modified": 1602549647665,
        "_enriched": true,
        "_status": "current",
        "_versionId": 1
      }
    ],
    "category": [
      {
        "_created": 1602549647665,
        "_modified": 1602549647665,
        "_createdBy": "regp-admin",
        "_lastModifiedBy": "regp-admin",
        "_visibility": "public",
        "id": "03dfe8bb1cf2a2b12544a2efa0b2465e",
        "uri": "http://semantics.aero/atm-service-category#Flight_Planning",
        "type": "skos:Concept",
        "label": "ATM Service Category",
        "prefLabel": "Flight Planning",
        "altLabel": [
          "ATM Service Category"
        ],
        "title": "ATM Service Category",
        "status": "submitted",
        "transitivePath": "http://semantics.aero/atm-service-category#Flight_Planning",
        "modified": 1602549647665,
        "_enriched": true,
        "_status": "current",
        "_versionId": 1
      },
      {
        "_created": 1602549647665,
        "_modified": 1602549647665,
        "_createdBy": "regp-admin",
        "_lastModifiedBy": "regp-admin",
        "_visibility": "public",
        "id": "a1eb5a3c6f7e16f1e0181c4ccc805e74",
        "uri": "http://semantics.aero/service-product#Aeronautical",
        "type": "skos:Concept",
        "label": "SWIM Service Product Category",
        "prefLabel": "Aeronautical",
        "altLabel": [
          "SWIM Service Product Category"
        ],
        "title": "SWIM Service Product Category",
        "status": "submitted",
        "transitivePath": "http://semantics.aero/service-product#Aeronautical",
        "modified": 1602549647665,
        "_enriched": true,
        "_status": "current",
        "_versionId": 1
      }
    ],
    "source": {
      "id": "1f0cc9a8993cb72f71df529bc2d842cd",
      "uri": "https://ogctb16.usersmarts.com/id/source/1cb10a67497705f34108c961562a6091
",
      "type": "Source",
      "ingestionDate": 1602549643610,
      "mediaType": "text/xml",
      "schema": "nsrr",
      "providerName": "Test NSRR",
      "providerUrl": "https://nsrr.faa.gov/rim/index",
      "providerType": "nsrr",
      "tool": "FW Harvester",
      "method": "harvest",
      "md5": "68f68fd0cdef400c7989dbe7e09f51cf",
      "filename": "[B@3fddaea2.xml",
      "byteSize": 3642
    },
    "serviceTypeVersions": [
      "2.0"
    ],
    "_links": {
      "self": {
        "href":
"https://ogctb16.usersmarts.com/registry/items/b8685040798ddeb678d10f6c4b6dec7b"
      },
      "reg:commits": {
        "href":
"https://ogctb16.usersmarts.com/registry/items/b8685040798ddeb678d10f6c4b6dec7b/versio
ns"
      },
      "reg:source": {
```

```
      "href":
"https://ogctb16.usersmarts.com/registry/sources/1f0cc9a8993cb72f71df529bc2d842cd"
    },
    "up": {
      "href": "https://ogctb16.usersmarts.com/registry/items"
    },
    "reg:registry": {
      "href": "https://ogctb16.usersmarts.com/registry"
    },
    "curies": [
      {
        "href": "http://www.factweave.com/rels/{rel}",
        "name": "reg",
        "templated": true
      }
    ]
  }
}
```

B.2. Harvesting the SWIM Data Relay API

The Asset Importer for SWIM Data Relay API Metadata retrieved service and collection metadata from the SWIM Data Relay API Component and transformed it into JSON, JSON-LD and Turtle (TTL) Representations.

The following JSON Service Representation resulted from the conversion of the harvested Heliport Class description of the SWIM Data Relay API Component.

Sample Collection Metadata Retrieved from OGC API

```
{
    "_created": 1600792592101,
    "_modified": 1600792592101,
    "_createdBy": "regp-admin",
    "_lastModifiedBy": "regp-admin",
    "_versionId": 1,
    "_status": "current",
    "_visibility": "public",
    "type": "dcat:Dataset",
    "id": "f6ec05301289230a02c844b341eb936e",
    "uri": "http://ogctb16.usersmarts.com/id/dataset/8822953c20b3fca92d14dd4b782ceb09
",
    "modified": 1600792592101,
    "label": "Class - AirportHeliport",
    "title": "Class - AirportHeliport",
    "description": "&lt;&lt;feature&gt;&gt; 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.",
    "servicedBy_id": [
        "af7ef59349ae6fdeed8213baa95c7857"
    ],
    "servicedBy": [
        {
            "_created": 1600792592101,
            "_modified": 1600792592101,
            "_createdBy": "regp-admin",
            "_lastModifiedBy": "regp-admin",
            "_versionId": 1,
            "_status": "current",
            "_visibility": "public",
            "type": "regp:Service",
            "id": "af7ef59349ae6fdeed8213baa95c7857",
            "uri":
"http://ogctb16.usersmarts.com/id/service/aa332d686432ca245e93c7095da01d66",
            "modified": 1600792592101,
            "label": "A sample API conforming to the draft standard OGC API - Features
- Part 1: Core",
            "title": "A sample API conforming to the draft standard OGC API - Features
- Part 1: Core",
            "description": "This is a sample OpenAPI definition that conforms to the
conformance\nclasses \"Core\", \"GeoJSON\", \"HTML\" and \"OpenAPI 3.0\" of the
draft\nstandard \"OGC API - Features - Part 1: Core\".\n\nThis example differs from
the [other example](ogcapi-features-1-example1.yaml)\nin that each feature collections
is specified explicitly in its own path, not using a path\nparameter. This API
definiton is more verbose, but provides information about the feature\ncollection
'buildings' (paths `/collections/buildings`), the schema of the building features
\n(schema `buildingGeoJSON`) and a filter parameter for building features (parameter
`function`).",
            "accessURL": "https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.json",
            "status": "submitted",
            "keyword": [
                "Capabilities",
                "Data"
            ],
            "source": {
                "id": "eaf029676b7345596a488b0412f14106",
                "uri":
"http://ogctb16.usersmarts.com/id/source/ec4005e77d53c56a58c2ce01eee9bf96",
                "type": "Source",
                "ingestionDate": 1600792587037,
                "schema": "wfs",
                "md5": "cc2804ba9cc8ffc455ebf92b2035499f",
                "sourceUrl":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.json",
                "byteSize": 59086
            },
        }
    ],
    "geographicBoundingBox": {
        "northBoundLatitude": 64.8151111111111,
        "southBoundLatitude": 40.7287777777778,
        "westBoundLongitude": -147.856444444444,
        "eastBoundLongitude": -68.8281388888889
    },
    "identifier": [
        "AirportHeliport"
    ],
    "status": "submitted",
    "source": {
        "id": "eaf029676b7345596a488b0412f14106",
        "uri":
"http://ogctb16.usersmarts.com/id/source/ec4005e77d53c56a58c2ce01eee9bf96",
        "type": "Source",
        "ingestionDate": 1600792587037,
        "schema": "wfs",
        "md5": "cc2804ba9cc8ffc455ebf92b2035499f",
        "sourceUrl": "https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.json",
        "byteSize": 59086
    },
    "temporal": {
        "startDate": 1591065000000,
        "endDate": 1591794000000
    },
    "distribution": [
        {
            "type": "dcat:Distribution",
            "label": "Collection AirportHeliport with media negotiation (Supported
Media-Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html' )",
            "title": "Collection AirportHeliport with media negotiation (Supported
Media-Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html' )",
            "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items",
            "mediaType": "application/geo+json"
        },
        {
            "type": "dcat:Distribution",
            "label": "Collection AirportHeliport",
            "title": "Collection AirportHeliport",
            "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.aixm.xm
l",
            "mediaType": "application/aixm+xml;version=5.1"
        },
        {
            "type": "dcat:Distribution",
            "label": "Collection AirportHeliport",
            "title": "Collection AirportHeliport",
            "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.geo.jso
n",
            "mediaType": "application/geo+json"
        },
        {
            "type": "dcat:Distribution",
            "label": "Collection AirportHeliport",
            "title": "Collection AirportHeliport",
            "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.html",
            "mediaType": "text/html"
        }
    ],
    "_links": {
        "self": {
            "href":
"https://ogctb16.usersmarts.com/registry/items/f6ec05301289230a02c844b341eb936e"
        },
        "reg:commits": {
            "href":
"https://ogctb16.usersmarts.com/registry/items/f6ec05301289230a02c844b341eb936e/versio
ns"
        },
        "reg:source": {
            "href":
"https://ogctb16.usersmarts.com/registry/sources/eaf029676b7345596a488b0412f14106"
        },
        "up": {
            "href": "https://ogctb16.usersmarts.com/registry/items"
        },
        "reg:registry": {
            "href": "https://ogctb16.usersmarts.com/registry"
        },
        "curies": [
            {
                "href": "http://www.factweave.com/rels/{rel}",
                "name": "reg",
                "templated": true
            }
        ]
    }
}
```

Retrieved collection metadata was encoded in JSON-LD. The following is the JSON-LD representation of the collection retrieved in the previous example. Note the JSON-LD is inline at the end of the document. An additional endpoint /context was added on the registry to produce the JSON-LD context for all the items managed by the registry.

Collection Metadata Encoded in JSON-LD

```
{
  "@id": "https://ogctb16.usersmarts.com/id/dataset/8822953c20b3fca92d14dd4b782ceb09",
  "@type": "dcat:Dataset",
  "distribution": [
    {
      "@type": "dcat:Distribution",
      "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items",
      "mediaType": "application/geo+json",
      "type": "dcat:Distribution",
      "label": "Collection AirportHeliport with media negotiation (Supported Media-
Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html' )",
      "title": "Collection AirportHeliport with media negotiation (Supported Media-
Types: 'Application/aixm+xml;version=5.1' 'Application/geo+json' 'text/html' )"
    },
    {
      "@type": "dcat:Distribution",
      "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.aixm.xm
l",
      "mediaType": "application/aixm+xml;version=5.1",
      "type": "dcat:Distribution",
      "label": "Collection AirportHeliport",
      "title": "Collection AirportHeliport"
    },
    {
      "@type": "dcat:Distribution",
      "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.geo.jso
n",
      "mediaType": "application/geo+json",
      "type": "dcat:Distribution",
      "label": "Collection AirportHeliport",
      "title": "Collection AirportHeliport"
    },
    {
      "@type": "dcat:Distribution",
      "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.html",
      "mediaType": "text/html",
      "type": "dcat:Distribution",
      "label": "Collection AirportHeliport",
      "title": "Collection AirportHeliport"
    }
  ],
  "source": {
    "@id": "https://ogctb16.usersmarts.com/id/source/ec4005e77d53c56a58c2ce01eee9bf96
",
    "@type": "rdfs:Resource",
    "type": "Source"
  },
  "servicedBy": [
    {
      "@id":
"https://ogctb16.usersmarts.com/id/service/aa332d686432ca245e93c7095da01d66",
      "@type": "regp:Service",
      "keyword": [
        "Capabilities",
        "Data"
      ],
      "source": {
        "@id":
"https://ogctb16.usersmarts.com/id/source/ec4005e77d53c56a58c2ce01eee9bf96",
        "@type": "rdfs:Resource",
        "type": "Source"
      },
      "accessURL": "https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.json",
      "status": "submitted",
      "type": "regp:Service",
      "label": "A sample API conforming to the draft standard OGC API - Features -
Part 1: Core",
      "title": "A sample API conforming to the draft standard OGC API - Features -
Part 1: Core",
      "description": "This is a sample OpenAPI definition that conforms to the
conformance\nclasses \"Core\", \"GeoJSON\", \"HTML\" and \"OpenAPI 3.0\" of the
draft\nstandard \"OGC API - Features - Part 1: Core\".\n\nThis example differs from
the [other example](ogcapi-features-1-example1.yaml)\nin that each feature collections
is specified explicitly in its own path, not using a path\nparameter. This API
definiton is more verbose, but provides information about the feature\ncollection
'buildings' (paths `/collections/buildings`), the schema of the building features
\n(schema `buildingGeoJSON`) and a filter parameter for building features (parameter
`function`)."
    }
  ],
  "temporal": {
    "@type": "extent:TemporalExtent",
    "startDate": 1591065000000,
    "endDate": 1591794000000
  },
  "geographicBoundingBox": {
    "@type": "extent:GeographicBoundingBox",
    "northBoundLatitude": 64.8151111111111,
    "southBoundLatitude": 40.7287777777778,
    "westBoundLongitude": -147.856444444444,
    "eastBoundLongitude": -68.8281388888889
  },
  "identifier": [
    "AirportHeliport"
  ],
  "status": "submitted",
  "type": "dcat:Dataset",
  "label": "Class - AirportHeliport",
  "title": "Class - AirportHeliport",
  "description": "&lt;&lt;feature&gt;&gt; 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.",
  "@context": {
    "extent": "http://www.opengis.net/ont/spatial/extent#",
    "schema": "http://schema.org/",
    "dct": "http://purl.org/dc/terms/",
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "regp": "http://www.factweave.com/ont/regp#",
    "eo": "http://www.factweave.com/profile/eo#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "dcat": "http://www.w3.org/ns/dcat#",
    "distribution": {
      "@id": "dcat:distribution",
      "@type": "dcat:Distribution"
    },
    "downloadURL": { "@id": "dcat:downloadURL", "@type": "@id" },
    "mediaType": { "@id": "dcat:mediaType", "@type":
"http://www.w3.org/2001/XMLSchema#string" },
    "type": { "@id": "rdf:type", "@type": "rdfs:Class" },
    "label": { "@id": "rdfs:label", "@type": "http://www.w3.org/2001/XMLSchema#string"
},
    "title": { "@id": "dct:title", "@type": "http://www.w3.org/2001/XMLSchema#string"
},
    "source": { "@id": "dct:source", "@type": "rdfs:Resource" },
    "servicedBy": { "@id": "regp:servicedBy", "@type": "regp:Service" },
    "keyword": {
      "@id": "dcat:keyword",
      "@type": "http://www.w3.org/2001/XMLSchema#string"
    },
    "accessURL": {
      "@id": "dcat:accessURL",
      "@type": "@id"
    },
    "status": {
      "@id": "regp:status",
      "@type": "http://www.w3.org/2001/XMLSchema#string"
    },
    "description": {
      "@id": "dct:description",
      "@type": "http://www.w3.org/2001/XMLSchema#string"
    },
    "temporal": {
      "@id": "dct:temporal",
      "@type": "dct:PeriodOfTime"
    },
    "startDate": {
      "@id": "schema:startDate",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "endDate": {
      "@id": "schema:endDate",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "geographicBoundingBox": {
      "@id": "extent:geographicBoundingBox",
      "@type": "extent:GeographicBoundingBox"
    },
    "northBoundLatitude": {
      "@id": "extent:northBoundLatitude",
      "@type": "http://www.w3.org/2001/XMLSchema#double"
    },
    "southBoundLatitude": {
      "@id": "extent:southBoundLatitude",
      "@type": "http://www.w3.org/2001/XMLSchema#double"
    },
    "westBoundLongitude": {
      "@id": "extent:westBoundLongitude",
      "@type": "http://www.w3.org/2001/XMLSchema#double"
    },
    "eastBoundLongitude": {
      "@id": "extent:eastBoundLongitude",
      "@type": "http://www.w3.org/2001/XMLSchema#double"
    },
    "identifier": {
      "@id": "dct:identifier",
      "@type": "http://www.w3.org/2001/XMLSchema#string"
    }
  }
}
```

Retrieved collection metadata was also encoded in TTL. The following is the TTL representation of the collection retrieved in the previous example. The following example demonstrates the equivalent representation of the JSON-LD in Turtle, which can be directly ingested in Triple Stores and used by semantic reasoners to perform logical consistency checking and inferences. The example shows that the SRIM model is superset of the well-established DCAT standard.

Collection Metadata Encoded in TTL

```
@prefix schema: <http://schema.org/> .
@prefix extent: <http://www.opengis.net/ont/spatial/extent#> .
@prefix adms:  <http://www.w3.org/ns/adms#> .
@prefix pav:   <http://purl.org/pav/> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix org:   <http://www.w3.org/ns/org#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix skos:  <http://www.w3.org/2004/02/skos/core#> .
@prefix regp:  <http://www.factweave.com/ont/regp#> .
@prefix prm:   <http://www.factweave.com/ont/prm#> .
@prefix eo:    <http://www.factweave.com/profile/eo#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .
@prefix gp-regp: <http://www.geoplatform.gov/ont/regp#> .
@prefix dct:   <http://purl.org/dc/terms/> .
```

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dcat:  <http://www.w3.org/ns/dcat#> .
@prefix prov:  <http://www.w3.org/ns/prov#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix sdo:   <http://schema.org/> .

<https://ogctb16.usersmarts.com/resources/datasets/195cc903903d66781b57f34afdace4fc>
        a                              dcat:Dataset ;
        rdfs:label                     "Class - AirportHeliport" ;
        dct:description                "&lt;&lt;feature&gt;&gt; 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." ;
        dct:identifier                 "AirportHeliport" ;
        dct:source                     <https://ogctb16.usersmarts.com/resources
/sources/d736654cd8cae042e0fb901639bd9170> ;
        dct:temporal                   [ a                extent:TemporalExtent ;
                                         sdo:endDate    "Wed Jun 10 13:00:00 UTC
2020"^^<java:java.util.Date> ;
                                         sdo:startDate  "Tue Jun 02 02:30:00 UTC
2020"^^<java:java.util.Date>
                                       ] ;
        dct:title                      "Class - AirportHeliport" ;
        regp:servicedBy                <https://ogctb16.usersmarts.com/resources
/services/e93daabdb04e81f45aee1967db6ca7c9> ;
        regp:status                    "submitted" ;
        extent:geographicBoundingBox  [ a
extent:GeographicBoundingBox ;
                                         extent:eastBoundLongitude  "-
68.8281388888889"^^xsd:double ;
                                         extent:northBoundLatitude
"64.8151111111111"^^xsd:double ;
                                         extent:southBoundLatitude
"40.7287777777778"^^xsd:double ;
                                         extent:westBoundLongitude  "-
147.856444444444"^^xsd:double
                                       ] ;
        owl:sameAs                     <https://ogctb16.usersmarts.com/id/dataset
/8822953c20b3fca92d14dd4b782ceb09> ;
        dcat:distribution              [ a                dcat:Distribution ;
                                         rdfs:label       "Collection AirportHeliport"
;
                                         dct:title        "Collection AirportHeliport"
;
                                         dcat:downloadURL  <https:/
/geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.aixm.xml> ;
                                         dcat:mediaType
"application/aixm+xml;version=5.1"
                                       ] ;
        dcat:distribution              [ a                dcat:Distribution ;
                                         rdfs:label       "Collection AirportHeliport"
;
                                        dct:title         "Collection AirportHeliport"
;
                                        dcat:downloadURL  <https:/
/geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.geo.json> ;
                                        dcat:mediaType    "application/geo+json"
                                      ] ;
        dcat:distribution             [ a                 dcat:Distribution ;
                                        rdfs:label        "Collection AirportHeliport
with media negotiation (Supported Media-Types: 'Application/aixm+xml;version=5.1'
'Application/geo+json' 'text/html' )" ;
                                        dct:title         "Collection AirportHeliport
with media negotiation (Supported Media-Types: 'Application/aixm+xml;version=5.1'
'Application/geo+json' 'text/html' )" ;
                                        dcat:downloadURL  <https:/
/geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items> ;
                                        dcat:mediaType    "application/geo+json"
                                      ] ;
        dcat:distribution             [ a                 dcat:Distribution ;
                                        rdfs:label        "Collection AirportHeliport"
;
                                        dct:title         "Collection AirportHeliport"
;
                                        dcat:downloadURL  <https:/
/geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/AirportHeliport/items.html> ;
                                        dcat:mediaType    "text/html"
                                      ] .

<https://ogctb16.usersmarts.com/resources/services/e93daabdb04e81f45aee1967db6ca7c9>
        a                 regp:Service ;
        rdfs:label        "A sample API conforming to the draft standard OGC API -
Features - Part 1: Core" ;
        dct:description   "This is a sample OpenAPI definition that conforms to the
conformance\nclasses \"Core\", \"GeoJSON\", \"HTML\" and \"OpenAPI 3.0\" of the
draft\nstandard \"OGC API - Features - Part 1: Core\".\n\nThis example differs from
the [other example](ogcapi-features-1-example1.yaml)\nin that each feature collections
is specified explicitly in its own path, not using a path\nparameter. This API
definiton is more verbose, but provides information about the feature\ncollection
'buildings' (paths `/collections/buildings`), the schema of the building
features\n(schema `buildingGeoJSON`) and a filter parameter for building features
(parameter `function`)." ;
        dct:source        <https://ogctb16.usersmarts.com/resources/sources
/d736654cd8cae042e0fb901639bd9170> ;
        dct:title         "A sample API conforming to the draft standard OGC API -
Features - Part 1: Core" ;
        regp:status       "submitted" ;
        owl:sameAs        <https://ogctb16.usersmarts.com/id/service
/aa332d686432ca245e93c7095da01d66> ;
        dcat:accessURL    <https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/api/wfs3.json> ;
        dcat:keyword      "Data" , "Capabilities" .
```

```
<https://ogctb16.usersmarts.com/resources/sources/d736654cd8cae042e0fb901639bd9170>
        a           rdfs:Resource ;
        owl:sameAs  <https://ogctb16.usersmarts.com/id/source
/ec4005e77d53c56a58c2ce01eee9bf96> .
```

Appendix C: Semantic Registry: GraphQL Schema

This section provides the GraphQL schema implemented for the Semantic Registry during Testbed 16. The schema can be easily extended by adding new item types in the schema by extending the interface Item. The API contract defined by GraphQL specification remains unchanged.

GraphQL Schema for Semantic Registry

```
scalar Date
scalar Url
scalar DateTime

type Query {
    getItem(id: ID): Item
    searchItems(query: SearchQuery, page: Int, size: Int, orderBy: [SortInput]):
SearchResults
}

type Mutation {
    saveAgent(input: AgentInput): Item
    savePerson(input: AgentInput): Item
    saveOrganization(input: AgentInput): Item
    saveDataset(input: DatasetInput): Item
    saveService(input: ServiceInput): Item
    saveLocation(input: LocationInput): Item
    saveStandard(input: StandardInput): Item
    saveConcept(input: ConceptInput): Item
    saveLinguisticSystem(input: LinguisticSystemInput): Item
    saveConceptScheme(input: ConceptSchemeInput): Item
    deleteItem(id: ID): Boolean
}

interface Item {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label: String
    title :  String
    description:String
}

input ItemInput {
    id : ID
    uri : Url
    type: String
}
```

```
type Asset implements Item {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    alternativeTitle: [String]
    description:String
    version:String
    versionNotes:String
    currentVersion:Item
    previousVersion: Item
    versions: [Item]
    isRelease: Boolean
    memberOf: [Item]
    hasMember: [Item]
    partOf: [Item]
    hasPart: [Item]
    creator: [Agent]
    publisher: [Agent]
    contributor:[Agent]
    rightsHolder: [Agent]
    usedBy: [Agent]
    attribution: [Attribution]
    credit: [String]
    hasIdentifier: [Identifier]
    identifier: [String]
    keyword:[String]
    contactPoints: [VCard]
    servicedBy: [Service]
    theme: [Concept]
    hasTopic:[Concept]
    productType:[Concept]
    category: [Concept]
    purpose:String
    thumbnail : Thumbnail
    classifiers: Classifiers
    landingPage: Url
    accessURL :  Url
    spatial: [Location]
    geographicBoundingBox: GeographicBoundingBox
    temporalExtent: TemporalExtent
    conformsTo: Standard
    license: LicenseDocument
    rights: [RightsStatement]
    accessRights: [RightsStatement]
    languages: [LinguisticSystem]
}
```

```
type Dataset implements Item {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    alternativeTitle: [String]
    description:String
    version:String
    versionNotes:String
    currentVersion:Item
    previousVersion: Item
    versions: [Item]
    isRelease: Boolean
    contactPoints: [VCard]
    creator: [Agent]
    publisher: [Agent]
    contributor:[Agent]
    rightsHolder: [Agent]
    usedBy: [Agent]
    attribution: [Attribution]
    credit: [String]
    hasIdentifier: [Identifier]
    identifier: [String]
    keyword:[String]
    theme: [Concept]
    hasTopic:[Concept]
    productType:[Concept]
    category: [Concept]
    purpose:String
    thumbnail : Thumbnail
    classifiers: Classifiers
    landingPage: Url
    accessUrl :  String
    spatial: [Location]
    geographicBoundingBox: GeographicBoundingBox
    temporalExtent: TemporalExtent
    license: LicenseDocument
    conformsTo: Standard
    rights: [RightsStatement]
    accessRights: [RightsStatement]
    languages: [LinguisticSystem]
    distribution: [Distribution]
}

input DatasetInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    alternativeTitle: [String]
    description:String
    version:String
    versionNotes:String
    currentVersion:ItemInput
    previousVersion: ItemInput
    versions: [ItemInput]
    isRelease: Boolean
    memberOf: [ItemInput]
    hasMember: [ItemInput]
    partOf: [ItemInput]
    hasPart: [ItemInput]
    creator: [AgentInput]
    publisher: [AgentInput]
    contributor:[AgentInput]
    rightsHolder: [AgentInput]
    usedBy: [AgentInput]
    attribution: [AttributionInput]
    credit: [String]
    hasIdentifier: [IdentifierInput]
    identifier: [String]
    keyword:[String]
    contactPoints: [VCardInput]
    servicedBy: [ServiceInput]
    theme: [ConceptInput]
    hasTopic:[ConceptInput]
    productType:[ConceptInput]
    category: [ConceptInput]
    purpose:String
    thumbnail : ThumbnailInput
    classifiers: ClassifiersInput
    landingPage: String
    accessUrl :  Url
    spatial: [LocationInput]
    geographicBoundingBox: GeographicBoundingBoxInput
    temporalExtent: TemporalExtentInput
    conformsTo: StandardInput
    license: LicenseDocumentInput
    rights: [RightsStatementInput]
    accessRights: [RightsStatementInput]
    languages: [LinguisticSystemInput]
    distribution: [ DistributionInput ]
}

input ServiceInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    alternativeTitle: [String]
    description:String
    version:String
    versionNotes:String
    currentVersion:ItemInput
    previousVersion: ItemInput
    versions: [ItemInput]
    isRelease: Boolean
    memberOf: [ItemInput]
    hasMember: [ItemInput]
    partOf: [ItemInput]
    hasPart: [ItemInput]
    creator: [AgentInput]
    publisher: [AgentInput]
    contributor:[AgentInput]
    rightsHolder: [AgentInput]
    usedBy: [AgentInput]
    attribution: [AttributionInput]
    credit: [String]
    hasIdentifier: [IdentifierInput]
    identifier: [String]
    keyword:[String]
    contactPoints: [VCardInput]
    operatesOn: [ItemInput]
    theme: [ConceptInput]
    hasTopic:[ConceptInput]
    productType:[ConceptInput]
    category: [ConceptInput]
    purpose:String
    thumbnail : ThumbnailInput
    classifiers: ClassifiersInput
    landingPage: String
    accessUrl :  String
    spatial: [LocationInput]
    geographicBoundingBox: GeographicBoundingBoxInput
    temporalExtent: TemporalExtentInput
    conformsTo: StandardInput
    license: LicenseDocumentInput
    rights: [RightsStatementInput]
    languages: [LinguisticSystemInput]
    accessRights: [RightsStatementInput]
}

type Service implements Item {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    alternativeTitle: [String]
    description:String
    version:String
    versionNotes:String
    currentVersion:Item
    previousVersion: Item
    versions: [Item]
    isRelease: Boolean
    memberOf: [Item]
    hasMember: [Item]
    partOf: [Item]
    hasPart: [Item]
    creator: [Agent]
    publisher: [Agent]
    contributor:[Agent]
    rightsHolder: [Agent]
    usedBy: [Agent]
    attribution: [Attribution]
    credit: [String]
    hasIdentifier: [Identifier]
    identifier: [String]
    keyword:[String]
    contactPoints: [VCard]
    operatesOn: [Item]
    theme: [Concept]
    hasTopic:[Concept]
    productType:[Concept]
    category: [Concept]
    purpose:String
    thumbnail : Thumbnail
    classifiers: Classifiers
    landingPage: Url
    accessUrl :  Url
    spatial: [Location]
    geographicBoundingBox: GeographicBoundingBox
    temporalExtent: TemporalExtent
    conformsTo: Standard
    license: LicenseDocument
    rights: [RightsStatement]
    accessRights: [RightsStatement]
}

type VCard {
    organization_name: String
    address: Address
    fn:String
    hasEmail: String
    tel: String
    fax:String
    landingPage: Url
    positionTitle: String
}

input VCardInput {
    organization_name: String
    address: AddressInput
    fn:String
    hasEmail: String
    tel: String
    fax:String
    landingPage: Url
    positionTitle: String
}

type Address {
    locality:String
    region: String
    country_name:String
    postal_code: String
}

input AddressInput {
    locality:String
    region: String
    country_name:String
    postal_code: String
}

type Classifiers {
    community: [Concept]
    topic:[Concept]
    primarySubject:[Concept]
    secondarySubject: [Concept]
    primaryTopic: [Concept]
    secondaryTopic: [Concept]
    purpose: [Concept]
    place : [Concept]
    audience: [Concept]
    category: [Concept]
    function: [Concept]
    event: [Concept]
    assumption: [Concept]
}

input ClassifiersInput {
    community: [ConceptInput]
    topic:[ConceptInput]
    primarySubject:[ConceptInput]
    secondarySubject: [ConceptInput]
    primaryTopic: [ConceptInput]
    secondaryTopic: [ConceptInput]
    purpose: [ConceptInput]
    place : [ConceptInput]
    audience: [ConceptInput]
    category: [ConceptInput]
    function: [ConceptInput]
    event: [ConceptInput]
    assumption: [ConceptInput]
}

type Attribution {
    type: String
    agent: Agent
    role: Url
}

input AttributionInput {
    type: String
    agent: AgentInput
    role: Url
}

type Thumbnail {
    url: Url
    label: String
    description: String
    mediaType: String
    title: String
}

input ThumbnailInput {
    url: Url
    label: String
    description: String
    mediaType: String
    title: String
}

type Identifier {
    name:String
    codeSpace:String
}

input IdentifierInput {
    name:String
    codeSpace:String
}

type GeographicBoundingBox {
    northBoundLatitude : Float
    southBoundLatitude : Float
    westBoundLongitude : Float
    eastBoundLongitude : Float
}

input GeographicBoundingBoxInput {
    northBoundLatitude : Float!
    southBoundLatitude : Float!
    westBoundLongitude : Float!
    eastBoundLongitude : Float!
}

type TemporalExtent {
    startDate : Date
    endDate : Date
}

input TemporalExtentInput {
    startDate : Date
    endDate : Date
}

type Location implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    prefLabel: String
    altLabel: [String]
    inScheme : ConceptScheme
    placeName: String
    geographicBoundingBox : [GeographicBoundingBox]
}

input LocationInput {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    prefLabel: String
    altLabel: [String]
    inScheme : ConceptSchemeInput
    placeName: String
    geographicBoundingBox : [GeographicBoundingBoxInput]
}

type Standard implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    availableVersions: [String]
}

input StandardInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    availableVersions: [String]
}

type LicenseDocument {
    uri : Url
    type: String

    label :  String
    title :  String
    description:String
}

input LicenseDocumentInput {
    uri : Url
    type: String
    label :  String
    title :  String
    description:String
}

type RightsStatement {
    id : ID!
    uri : Url
    type: String
    label :  String
    title :  String
    description: String
    thumbnail: Thumbnail
    juridictiom: [Concept]
    category: [Concept]
}

input RightsStatementInput {
    id : ID
    uri : String
    type: String
    label :  String
    title :  String
    description: String
    thumbnail: ThumbnailInput
    juridiction: [ConceptInput]
    category:[ConceptInput]
}

type Distribution {
    type: String
    contentId : ID
    label: String
    title :  String
    description:String
    accessURL : Url
    downloadURL: Url
    mediaType:String
    byteSize: Int
    conformsTo:Standard
    license: LicenseDocument
    rights: RightsStatement
    accessRights: RightsStatement
    accessLevel: String
    page: Document
    format: FileFormat
    language: LinguisticSystem
    representationTechnique: Url
}

input DistributionInput {
    type: String
    contentId : ID
    label: String
    title :  String
    description:String
    accessURL : Url
    downloadUrl: Url
    mediaType:String
    byteSize: Int
    conformsTo:StandardInput
    license: LicenseDocumentInput
    rights: RightsStatementInput
    accessRights: RightsStatementInput
    accessLevel: String
    page: DocumentInput
    format: FileFormatInput
    language: LinguisticSystemInput
    representationTechnique: Url
```

```
}

type Document {
    uri: Url
    label: String
    title :  String
    description:String
    conformsTo: Standard
    format:String
}



input DocumentInput {
    uri: Url
    label: String
    title :  String
    description:String
    conformsTo: StandardInput
    format:String
}

type FileFormat {
    uri : Url
    type: String
    label :  String
    title :  String
    description: String
    version: String
}

input FileFormatInput {
    uri : Url
    type: String
    label :  String
    title :  String
    description: String
    version: String
}

type LinguisticSystem implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
}

input LinguisticSystemInput {
    id : ID!
    uri : Url
    type: String
    label :  String
    title :  String
    description:String
}

type Agent implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label:String
    title :  String
    name: String
    description:String
}

type Organization implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label: String
    title :  String
    name: String
    description:String
}

type Person implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label: String
    title :  String
    name: String
    description:String
}

input AgentInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    title :  String
    name: String
    description:String

}
```

```
type Concept implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    prefLabel: String
    altLabel: [String]
    inScheme : ConceptScheme
}

input ConceptInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title :  String
    description:String
    prefLabel: String
    altLabel: [String]
    inScheme : ConceptSchemeInput
}

type ConceptScheme implements Item {
    id : ID!
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title: String
    description : String
}

input ConceptSchemeInput {
    id : ID
    uri : Url
    type: String
    resourceType: [String]
    label :  String
    title: String
    description : String
}

enum Op {
    "Equal operator" EQ,
    "Greater than operator" GT,
    "Greater than or equal operator" GTE,
    "Less than operator" LT,
    "Less than or equal operator" LTE,
    "Exists operator" EXISTS,
    "Not operator" NOT,
    "From operator" FROM,
    "To operator" TO
}

enum Aggregator {
    "Count Aggregator" COUNT,
    "Minimum Aggregator" MIN,
    "Maximum Operator" MAX,
    "Average Operator" AVG
}

input SearchQuery {
    "Text search"
    q:String
    "Filter constraints"
    filter: [FieldConstraintInput]
    aggregations: [AggregationInput]
}

input FieldConstraintInput {
    "Field path"
    field:String!
    "value constraints"
    constraints: [ConstraintInput]
}

input ConstraintInput {
    "operator"
    op:Op
    "values of constraints"
    values: [String]
}

input AggregationInput {
    "field path (use . notation) to aggregate"
    field:String!
    "aggregator operator"
    aggregator: Aggregator
    "Size of the aggregation"
    size: Int
    "Minimum count for aggregation to select"
    minDocCount: Int
}

"""
Search Results
"""
```

```
type SearchResults {
    items: [Item]
    aggregations: [AggregationResults]
    orderBy: [Sort]
    page: PageInfo
}

type AggregationResults {
    name : String!
    metrics: Metrics
    buckets: [Bucket]
}

type Bucket {
    key:String
    label: String
    metrics: Metrics
    aggregations: [AggregationResults]
}

type Metrics {
    count : Int
    sum : Float
    max :Float
    min :Float
    mean :Float
    sumOfSquares :Float
    variance :Float
    stdDeviation :Float
}

type PageInfo {
    number : Int
    pageSize : Int
    totalElements : Int
    totalPages : Int
}

type Sort {
    "field to sort by"
    by: String!
    "direction of sorting"
    dir: SortDirection,
    "Null handling hints"
    nullHandling: NullHandling
}

input SortInput {
    "field to sort by"
    by: String!
    "direction of sorting"
    dir: SortDirection,
    "Null handling hints"
    nullHandling: NullHandling
}

enum SortDirection {
    "Ascending order"
    ASC,
    "Descending order"
    DESC
}

enum NullHandling {
    NULLS_FIRST,
    NULLS_LAST
    NATIVE
}
```

Appendix D: Triple Builder: Sample Triple Generation

This annex describes the data pieces of a sample triple generation carried out by the Triple Builder. For a description of the triple creation process, please refer to Triple Builder and Triple Store.

This annex describes a sample flight feature retrieved from SWIM Data Relay API, the ontologies combined with the flight features to build the flight triples, and the resulting generated triples.

D.1. Sample Flight Feature

The following is a sample flight feature retrieved from SWIM Data Relay API.

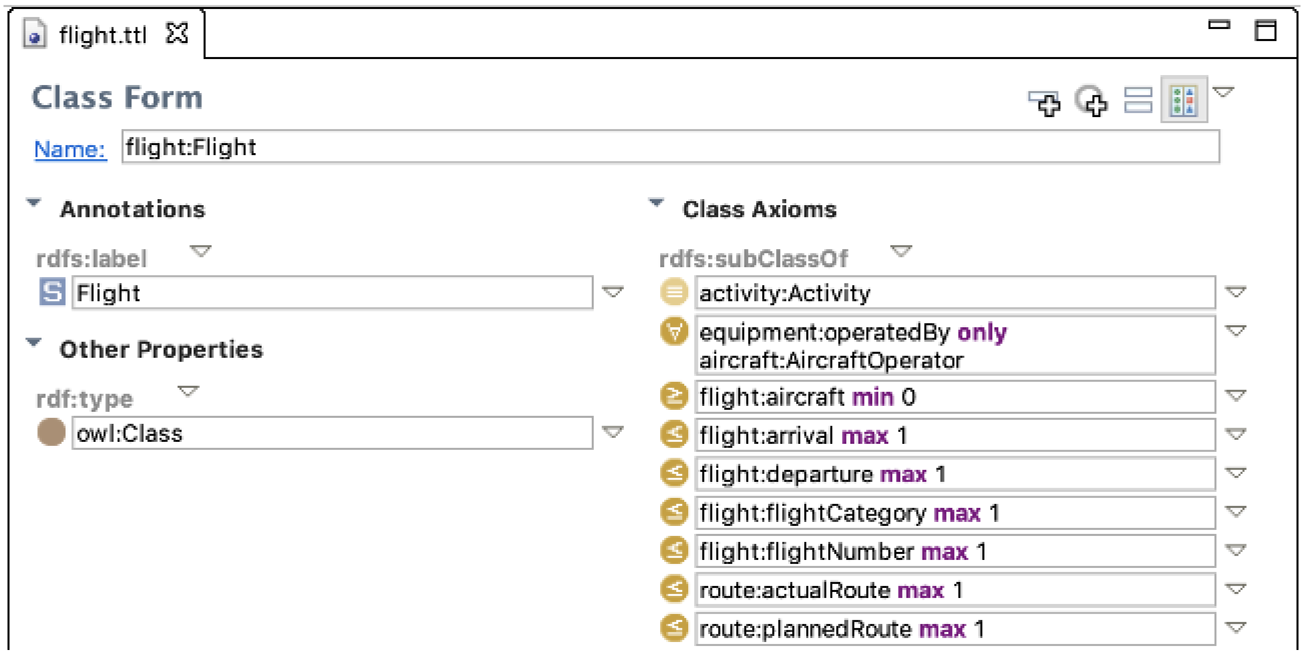
```
{
         "type":"Feature",
         "geometry":{
            "type":"Point",
            "coordinates":[
               -78.1819,
               38.07
            ]
         },
         "properties":{
            "uid":"FID2898",
            "acid":"SKQ74",
            "begintime":"2020-06-05T05:54:00.000+0000",
            "endtime":"2020-06-05T05:54:00.000+0000",
            "isinstime":true,
            "altitude":2133.6,
            "json":{
               "sensitivity":"A",
               "cdmPart":false,
               "airline":"SKQ",
               "sourceFacility":"KZDC",
               "sourceTimeStamp":1591336440000,
               "flightRef":133602467,
               "acid":"SKQ74",
               "msgType":"TRACK_INFORMATION",
               "fdTrigger":"HCS_TRACK_MSG",
               "depArpt":"KBUY",
               "arrArpt":"KIAD",
               "trackInformation":{
                  "qualifiedAircraftId":{
                     "aircraftId":"SKQ74",
                     "computerId":{
                        "facilityIdentifier":"KZDC",
                        "idNumber":"078"
                     },
   "gufi":"KT03496600",
   "igtd":1591333200000,
   "departurePoint":{
      "airport":"KBUY"
   },
   "arrivalPoint":{
      "airport":"KIAD"
   }
},
"speed":265,
"reportedAltitude":{
   "assignedAltitude":{
      "simpleAltitude":{
         "value":"070C"
      }
   }
},
"position":{
   "latitude":{
      "latitudeDMS":{
         "degrees":38,
         "minutes":4,
         "seconds":12,
         "direction":"NORTH"
      }
   },
   "longitude":{
      "longitudeDMS":{
         "degrees":78,
         "minutes":10,
         "seconds":55,
         "direction":"WEST"
      }
   }
},
"timeAtPosition":1591336440000,
"ncsmTrackData":{
   "eta":{
      "timeValue":1591337416000,
      "etaType":"ESTIMATED"
   },
   "rvsmData":{
      "equipped":true,
      "currentCompliance":true,
      "futureCompliance":true
   },
   "nextEvent":{
      "latitudeDecimal":38.42958,
      "longitudeDecimal":-78.00816
   }
}
```

```
                }
            }
        }
    }
```

D.2. Flight Ontology

A Flight Ontology, as seen on Figure 42, was used to build flight triples. The ontology has several subclasses, such as the Arrival and Departure classes, as seen on Figure 43 and Figure 44, respectively. For a detailed overview of these ontologies, please review the [Ontologies] section of this ER.

Flight Ontology, Visualized with a TTL Viewer



flight.ttl

Class Form

Name: flight:Flight

Annotations

rdfs:label

Flight

Other Properties

rdf:type

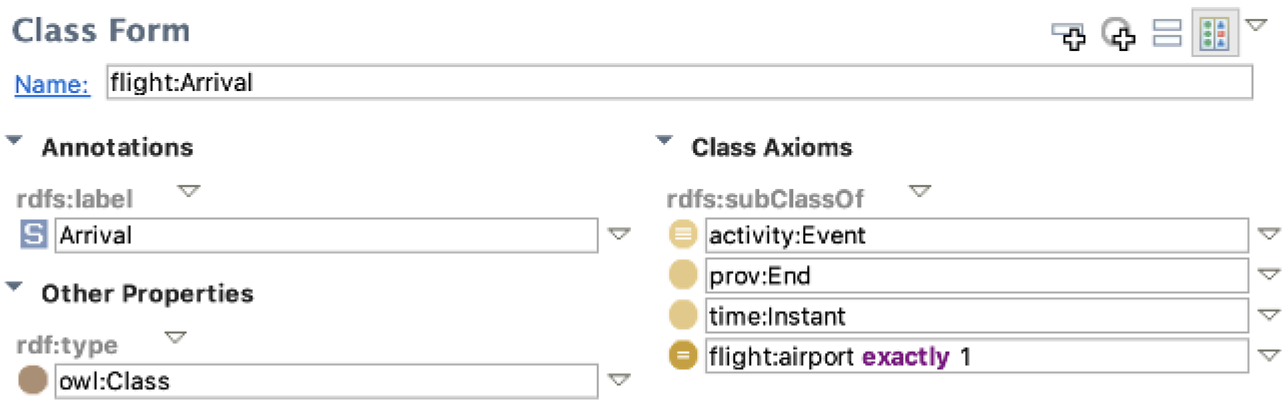
owl:Class

Class Axioms

rdfs:subClassOf

activity:Activity
equipment:operatedBy **only** aircraft:AircraftOperator
flight:aircraft **min** 0
flight:arrival **max** 1
flight:departure **max** 1
flight:flightCategory **max** 1
flight:flightNumber **max** 1
route:actualRoute **max** 1
route:plannedRoute **max** 1

Arrival Class From Flight Ontology, Visualized with a TTL Viewer



Class Form

Name: flight:Arrival

Annotations

rdfs:label

Arrival

Other Properties

rdf:type

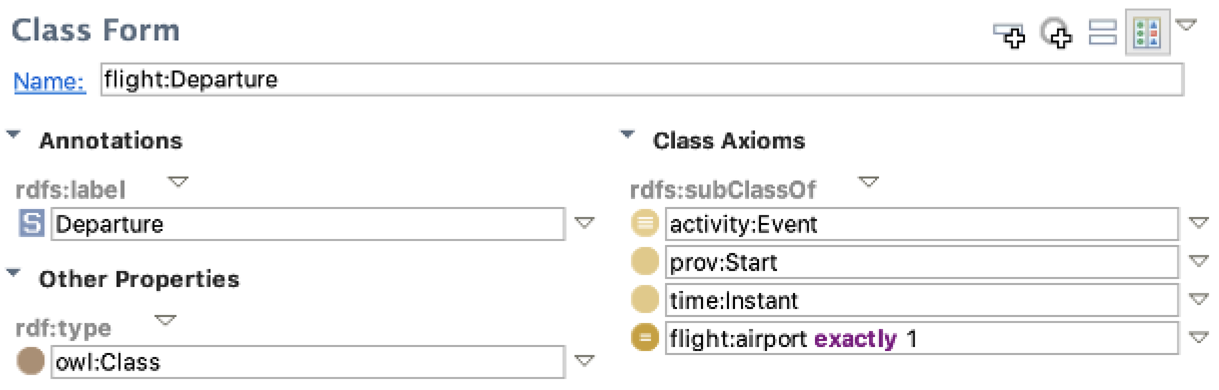
owl:Class

Class Axioms

rdfs:subClassOf

activity:Event
prov:End
time:Instant
flight:airport **exactly** 1

Departure Class From Flight Ontology, Visualized with a TTL Viewer


Class Form

Name: flight:Departure

Annotations

rdfs:label

Departure

Other Properties

rdf:type

owl:Class

Class Axioms

rdfs:subClassOf

activity:Event

prov:Start

time:Instant

flight:airport exactly 1

D.3. Flight Triples

The Triple Builder created flight triples by combining flight features with the flight ontology. The following is a sample generated flight triple:

Sample Flight Triple

```
# baseURI: http://www.opengis.net/ont/testbed16/aviation/activities/flight
# imports: http://www.opengis.net/ont/activity
# imports: http://www.opengis.net/ont/common/identifier
# imports: http://www.opengis.net/ont/testbed16/aircraft
# imports: http://www.opengis.net/ont/testbed16/aviation/infrastructure
# imports: http://www.opengis.net/ont/testbed16/route
# prefix: flight
@prefix : <http://www.opengis.net/ont/testbed16/aviation/activities/flight#> .
@prefix activity: <http://www.opengis.net/ont/activity#> .
@prefix aircraft: <http://www.opengis.net/ont/testbed16/aircraft#> .
@prefix equipment: <http://www.opengis.net/ont/testbed16/equipment#> .
@prefix flight: <http://www.opengis.net/ont/testbed16/aviation/activities/flight#> .
@prefix identifier: <http://www.opengis.net/ont/common/identifier#> .
@prefix infrastructure: <http
://www.opengis.net/ont/testbed16/aviation/infrastructure#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix route: <http://www.opengis.net/ont/testbed16/route#> .
@prefix spin: <http://spinrdf.org/spin#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.opengis.net/ont/testbed16/aviation/activities/flight>
  a owl:Ontology ;
  spin:imports <http://topbraid.org/spin/owlrl-all> ;
  spin:imports <http://topbraid.org/spin/rdfsplus> ;
  owl:imports <http://www.opengis.net/ont/activity> ;
  owl:imports <http://www.opengis.net/ont/common/identifier> ;
  owl:imports <http://www.opengis.net/ont/testbed16/aircraft> ;
  owl:imports <http://www.opengis.net/ont/testbed16/aviation/infrastructure> ;
  owl:imports <http://www.opengis.net/ont/testbed16/route> ;
  owl:versionInfo "Created with TopBraid Composer" ;
.
```

```
flight:AircraftOperator_SKQ
  a aircraft:AircraftOperator ;
  rdfs:label "Aircraft operator SKQ" ;
.
flight:Airport_KIAD
  a infrastructure:Airport ;
  rdfs:label "Airport KIAD" ;
.
flight:Airport_arrArpt_FID2898
  a infrastructure:Airport ;
  infrastructure:icaoAirportCode "KIAD" ;
  rdfs:label "Airport arr arpt FID2898" ;
.
flight:Airport_depArpt_FID2898
  a infrastructure:Airport ;
  infrastructure:icaoAirportCode "KBUY" ;
  rdfs:label "Airport dep arpt FID2898" ;
.
flight:Arrival
  a owl:Class ;
  rdfs:label "Arrival" ;
  rdfs:subClassOf activity:Event ;
  rdfs:subClassOf <http://www.w3.org/2006/time#Instant> ;
  rdfs:subClassOf <http://www.w3.org/ns/prov#End> ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:cardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:airport ;
    ] ;
.
flight:Arrival_FID2898
  a flight:Arrival ;
  flight:airport flight:Airport_arrArpt_FID2898 ;
  rdfs:label "Arrival FID2898" ;
.
flight:Arrival_KIAD
  a flight:Arrival ;
  rdfs:label "Arrival KIAD" ;
.
flight:Departure
  a owl:Class ;
  rdfs:label "Departure" ;
  rdfs:subClassOf activity:Event ;
  rdfs:subClassOf <http://www.w3.org/2006/time#Instant> ;
  rdfs:subClassOf <http://www.w3.org/ns/prov#Start> ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:cardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:airport ;
    ] ;
.
```

```
flight:Departure_FID2898
  a flight:Departure ;
  flight:airport flight:Airport_depArpt_FID2898 ;
  rdfs:label "Departure FID2898" ;
.
flight:Flight
  a owl:Class ;
  rdfs:label "Flight" ;
  rdfs:subClassOf activity:Activity ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:allValuesFrom aircraft:AircraftOperator ;
      owl:onProperty equipment:operatedBy ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:arrival ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:departure ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:flightCategory ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:flightNumber ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty route:actualRoute ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:maxCardinality "1"^^xsd:nonNegativeInteger ;
      owl:onProperty route:plannedRoute ;
    ] ;
  rdfs:subClassOf [
      a owl:Restriction ;
      owl:minCardinality "0"^^xsd:nonNegativeInteger ;
      owl:onProperty flight:aircraft ;
    ] ;
.
flight:FlightCategory
```

```
  a owl:Class ;
  rdfs:label "Flight category" ;
  rdfs:subClassOf <http://www.w3.org/2004/02/skos/core#Concept> ;
.
flight:Flight_FID2898
  a flight:Flight ;
  flight:arrival flight:Arrival_FID2898 ;
  flight:departure flight:Departure_FID2898 ;
  flight:flightNumber "SKQ74" ;
  equipment:operatedBy flight:AircraftOperator_SKQ ;
  rdfs:label "Flight FID2898" ;
.
flight:aircraft
  a owl:ObjectProperty ;
  rdfs:label "aircraft" ;
  rdfs:range aircraft:Aircraft ;
  rdfs:subPropertyOf <http://www.w3.org/ns/prov#entity> ;
.
flight:airport
  a owl:ObjectProperty ;
  rdfs:label "airport" ;
  rdfs:range infrastructure:Airport ;
  rdfs:subPropertyOf <http://www.w3.org/ns/prov#atLocation> ;
.
flight:arrival
  a owl:ObjectProperty ;
  rdfs:label "arrival" ;
  rdfs:subPropertyOf activity:hasEndEvent ;
.
flight:departure
  a owl:ObjectProperty ;
  rdfs:label "departure" ;
  rdfs:subPropertyOf activity:hasStartEvent ;
.
flight:flightCategory
  a owl:ObjectProperty ;
  rdfs:label "flight category" ;
  rdfs:range flight:FlightCategory ;
.
flight:flightNumber
  a owl:DatatypeProperty ;
  rdfs:label "flight number" ;
  rdfs:range xsd:string ;
  rdfs:subPropertyOf identifier:identifier ;
.
```

Appendix E: SWIM Data Client: Sample GraphQL Call to the Semantic Registry

This section provides a sample GraphQL call made by the SWIM Data Client to the Semantic Registry to discover SWIM datasets and services.

Sample GraphQL Call to the Semantic Registry

```
query SearchItemFunction($searchText: String!, $page: Int!, $pageSize: Int!) {
  searchItems(
    query: {
      q: $searchText
      filter: [
        { field: "type", constraints: [{ op: EQ, values: ["dcat:Dataset"] }] }
      ]
    }
    page: $page
    size: $pageSize
    orderBy: { by: "title" }
  ) {
    page {
      number
      pageSize
      totalElements
      totalPages
    }
    items {
      id
      uri
      label
      title
      description
      type
      ... on Dataset {
        identifier
        distribution {
          label
          title
          mediaType
          downloadURL
        }
        geographicBoundingBox {
          northBoundLatitude
          southBoundLatitude
          westBoundLongitude
          eastBoundLongitude
        }
      }
    }
  }
}

variables: {page: 0, pageSize: 10, searchText: "fixm"}
```

Sample GraphQL Response from the Semantic Registry

```
{
  "data": {
    "searchItems": {
      "page": {
        "number": 0,
        "pageSize": 10,
        "totalElements": 1,
        "totalPages": 1,
        "__typename": "PageInfo"
      },
      "items": [
        {
          "id": "d8b154b545d85b5981ecfda628626165",
          "uri": "http://localhost/id/dataset/1ac83546f5621dae3190f847fa95765e",
          "label": "Flight Message in FIXM US Extensiion",
          "title": "Flight Message in FIXM US Extensiion",
          "description": "FIXM - Flight information with US Extension. Shcema are
available at https://www.fixm.aero/download.pl?view=e .",
          "type": "dcat:Dataset",
          "identifier": [
            "NasFlightMessage"
          ],
          "distribution": [
            {
              "label": "Collection NasFlightMessage with media negotiation (Supported
Media-Types: 'application/fixm+nas+xml;version=3.0' 'application/geo+json' 'text/html'
)",
              "title": "Collection NasFlightMessage with media negotiation (Supported
Media-Types: 'application/fixm+nas+xml;version=3.0' 'application/geo+json' 'text/html'
)",
              "mediaType": "application/geo+json",
              "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items",
              "__typename": "Distribution"
            },
            {
              "label": "Collection NasFlightMessage",
              "title": "Collection NasFlightMessage",
              "mediaType": "application/fixm+nas+xml;version=3.0",
              "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items.fixm.n
as.xml",
              "__typename": "Distribution"
            },
            {
              "label": "Collection NasFlightMessage",
              "title": "Collection NasFlightMessage",
              "mediaType": "application/geo+json",
              "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items.geo.js
on",
                "__typename": "Distribution"
              },
              {
                "label": "Collection NasFlightMessage",
                "title": "Collection NasFlightMessage",
                "mediaType": "text/html",
                "downloadURL":
"https://geobrain.csiss.gmu.edu/gmuwfs3/wfs3/collections/NasFlightMessage/items.html",
                "__typename": "Distribution"
              }
            ],
            "geographicBoundingBox": {
              "northBoundLatitude": 78.89470006182776,
              "southBoundLatitude": -122.929722,
              "westBoundLongitude": -179.40216862479724,
              "eastBoundLongitude": 50.088333,
              "__typename": "GeographicBoundingBox"
            },
            "__typename": "Dataset"
          }
        ],
        "__typename": "SearchResults"
      }
    }
}
```

Appendix F: Revision History

Table 17. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
May 28, 2020	S. Taleisnik	.1	all	Initial Version
August 6, 2020	S. Taleisnik	.4	all	Outline, Background, Task Architecture
October 21, 2020	S. Taleisnik	.8	all	Draft Engineering Report
November 6, 2020	S. Taleisnik	.9	all	Review

Appendix G: Bibliography

[1] Fellah, S.: OGC Testbed-14: Semantically Enabled Aviation Data Models Engineering Report. Open Geospatial Consortium, http://docs.opengeospatial.org/per/18-035.html (2019).

[2] Fellah, S.: Testbed-12 Semantic Portrayal, Registry and Mediation Engineering Report. Open Geospatial Consortium, https://docs.ogc.org/per/16-059.html (2017).

[3] Kok, E., Fellah, S.: OGC Testbed-15: Semantic Web Link Builder and Triple Generator. Open Geospatial Consortium, http://docs.opengeospatial.org/per/19-021.html (2019).

[4] https://www.w3.org/standards/semanticweb/data, (2015).

[5] Berners-Lee, T.: Linked Data, https://www.w3.org/DesignIssues/LinkedData.html, (2009).

[6] Percivall, G.: OGC Open Geospatial APIs - White Paper. Open Geospatial Consortium, http://docs.opengeospatial.org/wp/16-019r4/16-019r4.html (2017).

[7] Simmons, S.: OGC APIs and the evolution of OGC standards, https://www.ogc.org/blog/2996, (2019).

[8] OGC API - Features, https://ogcapi.ogc.org/features/.

[9] OGC API - Records, https://ogcapi.ogc.org/records/.

[10] Chen, C.: Testbed-12: Aviation Architecture ER. Open Geospatial Consortium, http://docs.opengeospatial.org/per/16-018.html (2017).

[11] Wilson, D., Painter, I.: OWS-8 Aviation: Guidance for Retrieving AIXM 5.1 data via an OGC WFS 2.0. Open Geospatial Consortium, https://portal.opengeospatial.org/files/?artifact_id=46666 (2012).

[12] Matthews, M.: SWIM Cloud Distribution Service, https://www.faa.gov/air_traffic/flight_info/aeronav/atiec/media/Presentations/Day%201%20PM%20009%20Melissa%20Matthews%20SCDS.pdf, (2019).

[13] Assessment and benchmarking of spatially enabled RDF stores for the next generation of spatial data. Presented at the (2019).

[14] Keet, M.: An Introduction to Ontology Engineering. (2018).

[15] Spatial Data on the Web Best Practices - W3C Working Group Note, https://www.w3.org/TR/sdw-bp/, (2017).