CSDIF Proof of Concept

Document identifier: CSDIF-002-POC
Revision: Draft1
Date: 2026-01-23

https://www.csdif.info

Table of Contents

1 Introduction

CSDIF (Community Science Data Interchange Format) aims to define a standard for exchanging observational data, along with rich metadata between institutional and community science organisations.

In 2025, a proposal (in the form of a request for comments) was published (Document CSDIF-001-RFC) that solicits comments on the proposed approach: Using the existing SensorThings API and SensorML metadata format, with additional recommendations, requirements and scope limitations defined by the (to-be-written) CSDIF specification.

Halfway 2025 work started on an initial proof-of-concept implementation (PoC) has been been made to evaluate the ideas in the RFC and provide a platform to try some of the parts of the proposal that are still uncertain.

This document reports on the experiences in creating and working with this PoC. The focus is to document things that came during the development. In some cases, this provided an answer or additional considerations to an open question from the RFC, but a lot of the insights in this document open up new questions (and sometimes answer them directly as well).

So far, the development has focused on getting the PoC in a working state. In the next phase of development, the PoC can be used to more systematically explore the issues and questions defined in the RFC and this document and support creating a final specification for CSDIF.

This is an early draft version of this document, which will be expanded and improved when more experience with the PoC is gathered.

1.1 Proof of Concept overview

For this proof of concept, two different approaches to a CSDIF data API were explored:

1. Meet Je Stad developed a solution based on an off-the-shelf SensorThings server (FROST-Server), which is intended to function as the primary storage for the observation data. Data is converted into CSDIF format before storage.
2. SMAL Zeist implemented the SensorThings API from scratch, with conversion from their existing ElasticSearch-based storage backend into the SensorThings model and SensorML metadata format on request.

These two approaches both seem reasonable for initiatives to adopt and are expected to produce a significantly different implementation experience and likely different demands of a CSDIF specification.

In addition, a web application that can consume data via the CSDIF protocol and allow interactive exploration of the observations of both systems on a map was built, to additionally obtain experience from the perspective of a data consumer.

The rest of this document contains more detailed description of these three parts, along with some specific issues or observations that were made while implementing these proof-of-concepts.

2 Overall considerations

2.1.1 Coarse location - use geohash or h3

Maybe inside GeoJSON as foreign member (https://datatracker.ietf.org/doc/html/rfc7946#section-6.1), combined with GeoJSON polygon (https://datatracker.ietf.org/doc/html/rfc7946#section-3.1.6)? GeoHash is probably easier to work with than GeoJSON polygon for consumers.

Or instead of GeoJSON (but there is no existing GeoHash mimetype, so we would need to invent our own).

Alternatively, a Thing can have multiple Location objects, which should be different representations of the same location, so we could also have one GeoHash Location and one GeoJSON-with-polygon location? However, this only applies to the Thing Location (and HistoricalLocation), an Observation FeatureOfInterest can only have one value.

2.1.2 Aggregating measurements in time

The STA departs from the intuitive device oriented approach to data that many initiatives will have adopted which ties several measurements to a single timestamp. This makes it easy to process time series as data have a strong conceptual tie (they were done at the same moment) and can be processed easily a graphs with a common time on the x axis. Perhaps a clustering in time (like h3 does for geolocation) is a workable solution. But is this something that should happen at the server or at the client? And do standards exist for temporal aggregation?

2.1.3 Multiple values from single sensor

How to model e.g. the Si7021 measuring both temperature and humidity, or an SPS030 to measure PM1, PM2.5 and PM10 (also in both mass/volume and particle counts)? Should this be separate datastreams (and should those refer to different sensors to allow per-output metadata, or to the same sensor object to clarify that it is physically the same sensor providing both outputs)? Or should this be a single (composite/multi) datastream providing multiple values?

One downside of a single multidatastream is that in STA 1.1, it contains a list of values. Each has its own ObservedProperty and UnitOfMeasurement, but does not have a name (only the ObservedProperty has a name, but that does not allow distinguishing multiple values that share the same kind of observed property).

STA 2.0 fixes this by using SWE Common encoding for all datastreams, which allows using a DataRecord type, which is a dictionary of key/name – value pairs, allowing to name each entry (see https://hylkevds.github.io/23-019/23-019.html#_d1f96f6d-5d3b-be1c-1e43-cd24a9b33293 section 8.6 Datastream, listing 12 and 13).

In writing the data consumer, it turns out multiple values per datastream/observation is also inconvenient, when filtering for a single kind of measurement (based on ObservedProperty), the consumer still has to do significant client-side processing to determine which of the multiple values is the interesting one.

What could define when to group values or when to separate them?

- Group all values produced by the same physical sensor.
  This adds extra structure (e.g. two sensors of the same type in the same Thing could refer to the same Sensor object and the grouping determines which belongs to which). This structure could maybe better be added by adding a serial number of sensor index to the sensor metadata and grouping based on those.
- Group values that are (almost) always used together, for example latitude/longitude (when storing GPS output as an Observation).
- Group values that are essentially the same measurement, but with different parameters or qualifications. For example, particulate matter measurements for different sizes are the same measurement (such as mass-per-volume). Or an audio spectrum, where all measurements are amplitude, but derived at different frequencies.

- Do not group values at all.
  For values that are very much related/connected, such as latitude/longitude, could maybe be presented as a single complex value. STA supports this using the generic `OM_Observation` type (see table 12), which allows any data type (presumably including arbitrary objects, GeoJSON, etc.).

2.1.4 Units of measurement UCUM or QUDT

CSDIF currently suggests using UCUM or alternatively QUDT for units of measurement. Egon found that QUDT provided an off-the-shelf “µg/m³” unit that UCUM might not have (or maybe it is available but must be composed from parts?).

However STA 2.0 switches the SWE Common to describe the observation format, which includes an uom member, which has a “code” member that explicitly refers to a UCUM code. However, it can, additionally or alternatively, also refer to an explicit URI using the “href” member, so that still allows using QUDT when appropriate (one example also shows using both a UCUM code and QUDT url). See e.g. https://docs.ogc.org/DRAFTS/24-014.html#json_unitreference_obj

It does seem that UCUM does not associate URIs with units, just a code (case sensitive and case insensitive version), name and a print symbol. STA needs a name, symbol and and definition URI. STA also recommends following UCUM, but is not explicit how exactly. The examples (only for degree Celcius and percent, so no multi-part units) suggest it uses the UCUM name for name, the UCUM print symbol for symbol and a link to a section of the UCUM specification (e.g. http://unitsofmeasure.org/ucum.html#para-30, which is a dead link, https://ucum.org/ucum#para-30 is a working equivalement). There is no way to put the the machine-readable (non-print) symbol, and the rough URL pointing to the spec really is useless.

https://github.com/units-of-measurement/units-of-measurement makes an attempt to encode UCUM codes into w3id.org URIs, but that requires normalizing units (which might be useful for direct string comparison, but might be harder for humans, e.g. normalizing m/s to m.s-1 or N.m to m.N).

In the current CSDIF RFC example we used “ucum:%”, probably in a self-invented “URI” code. An alternative could be something like https://ucum.org/ucum?code=m/s. Or maybe we, or ucum.org can host some URL which you can pass a ucum code which will then be parsed, validated and explained (ideally using only client-side javascript to make it easily scaleable).

2.1.5 History of sensors

For things, CSDIF defines history can be kept by creating distinct things with the same uniqueId but non-overlapping validTimes. For Sensors, this is not specified. It is also meaningless if a Sensor refers to a *type* of sensor. However, if Sensor would refer to a specific physical sensor (e.g. identified by a serial number), then it might be relevant to keep history for it. This could then be done in the same way as for things?

Alternatively, a sensor can be seen as part of a Thing, which means versioning just the things is sufficient.

2.1.6 Additional qualification on measurement / observedProperty

How to specify the observed property is e.g. air temperature (as opposed to water temperature or soil temperature)? Technically this would maybe be a property of the FeatureOfInterest, but in STA that is defined as the place/location/geoshape, not more specific than that.
Also, how about e.g. different PM sizes? Now PM2.5 is distinguished from PM10 based on the observedProperty name, but that might not actually be accurate, or description, which is very much not machine-readable.

The ObservedProperty object does have a free-form “properties” member that could maybe store this, or maybe it could be appended as URL parameters to the “definition” URI (though QUDT does not specify any of this, and I have not seen examples of this anywhere).

2.1.7 Uniqueness of ObservedProperty definitions

It seems useful to do some deduplication on ObservedProperty – if multiple datastreams and things measure the same thing, reference the same ObservedProperty.

However, should this mean only a single ObservedProperty object should exists for each used “definition” value? In other words, does the “name”, “description” and any free-form “properties” follow deterministically from the “definition”? Or can those be used to create distinct ObservedProperties with the same definition? The above question (about PMx values) suggest the latter.

Also, if the values of these attributes are dictated by measurement station (and sent over the wire), then how to handle duplication (i.e. one state using name=”temperature” and another using name=”temp”)? Or should stations always just send out a definition and expect the server to figure out the

name/description? That is probably not generically possible (for qudt you could get the label and use that for name, though).

Maybe CSDIF could publish a list of commonly used ObservedProperties, with definintion, name and description, which could be preloaded by implementers to get some consistency in name/description used for these properties.

2.1.8 No FeatureOfInterest – use Thing Location

CSDIF states that the FOI of an Observation is optional and can default to the Thing Location, but on closer reading, this only applies when creating the Observation – A server implementation should copy the Location contents from the Thing into the FOI at that time, so queries always return one.

Unfortunately, a FOI contains a location embedded instead of linking to a Location object, so if the location of a Thing is updated retroactively, this also has to explicitly update appropriate FOI objects.

2.1.9 HistoricalLoation – how to interpret time

The HistoricalLocation object has a “time” property, which means “The time when the Thing is known at the Location”. In other words, it is not possible to express a validity range, only an instant. For GPS-supplied positions, this might be fine, since then you likely only have point-in-time samples, but for manually configured locations, you might have more detailed information on when a Thing was moved exactly.

Also, the current Location does not have a time attribute, so there is no way to tell when the current Location was sampled. The spec suggests auto-creation of HistoricalLocation when the Location attribute is written (https://docs.ogc.org/is/18-088/18-088.html#requirement-create-update-delete-historical-location-auto-creation) but without specifying the value to use for the HistoricalLocation “time” attribute (but it would follow that the only reasonable value available is the time at which the change took place, which would make “time” the time when the location *stopped* being valid, instead of when it was valid or sampled.

2.1.10 Validity of locations not well-defined in SensorThings

In SensorThings, a Thing has a location, which represents its current location. The data model does not specify explicitly (since) when a Thing’s Location is valid, other than that it is meant for the current location.

Then a Thing also has zero or more HistoricalLocations, each of which consist of a Location (position) and a timestamp. The meaning of the timestamp is not

exactly defined (is it the start or the end of the period that the thing was at this location), just that the thing was at this location at that particular time. For a location coming from GPS, this makes some sense, since between GPS readings the location is unknown anyway, so the time could indicate the moment that the GPS reading was made at this particular location. Assuming that any subsequent GPS readings the produce the same result (or close enough) are either not transmitted, or at least do not result in creating a new HistoricalLocation, it makes sense that timestamp indicates the *first* moment the given Location is correct, implying it is correct until the next HistoricalLocation.

For manually configured locations, you typically do not have a location valid at a particular moment in time, but a user can indicate the validity range explicitly, but then it is still possible to store the “valid from” in the location and be valid until the timestamp of the next Location (and for non-consecutive locations, i.e. gaps in the location history that could be the result of this, inserting unknown Location objects could be used to make these gaps explicit if needed).

However, SensorThings (req 8) also specifies that if the Location of a Thing is updated, then the server should automatically create a HistoricalLocation. Initially we interpreted this as creating a HistoricalLocation to contain the *old* Location (meaning that the full list of locations would be all HistoricalLocations followed by the current Location). The spec is not explicit about what timestamp to use, but only the current time would make sense here (it specifies a client can update the timestamp later). If this is done, then the timestamp stored with a HistoricalLocation would actually mean the timestamp *until* the given HistoricalLocation is valid, which is inconsistent with the above conclusion.

However, re-reading the spec it seems it is not so clear what HistoricalLocation should be created. It could also be taken to mean that whenever a Location is written to a thing, that Location should *also* be saved as a HistoricalLocation with the same time. This would mean that the timestamp for the HistoricalLocation would indeed indicate the first moment of validity for the HistoricalLocation and is consistent with the above. This is also supported by req 46 in the spec, which specifies the current Location of a Thing should be updated if a HistoricalLocation is added manually. Both of these would mean that the current Location is always duplicated by the most recent HistoricalLocation.

2.1.11 Location reliability and source

We probably need some metadata about where a location / featureofinterest comes from, and how reliable it is. This likely needs some additional fields in the geojson. Keeping a source (GPS / Manually specified / Network) makes sense.

Maybe an “age” property to keep track when the position was last confirmed, but that does not allow expressing the extra confidence in a location that was confirmed (some time) both before *and* after the observation. Maybe a confidence to distinguish between a location for an observation that is confirmed on just one GPS reading, or confirmed by a reading before and after? Alternatively, if the GPS readings before and after are significantly different, all observations in between could be set to an unknown location (or use the “before” location and set a low confidence). Possible a TTN join can also be used as a heuristic in here – if a TTN join is received, it is likely that the station was moved at that point (and maybe it is acceptable to assign a low confidence to all observations shortly before and after a rejoin).

Note that confidence here is about whether the location was actually valid at the time of observation. How accurate the location is, would be a different subject and could be quantified differently.

2.1.12 Metadata: Datasheets and other documentation

SensorML defines a “Documentation” member for adding documentation for a described object, referring to the ISO 19115 CI_OnlineResource object to describe a specific document. To describe the role of the document (manual, brochure, datasheet, etc.), SensorML defines a xlink:arcrole attribute (in the XML version). The examples for this attribute use values like http://sensorml.com/ont/swe/role/UserManual or http://sensorml.com/ont/core/doc/MaintenanceManual, but neither of these ontologies (swe/role and core/doc) are still available online (other ontologies like swe/system are available on the same base url). It is not immediately clear if these were removed, or these were never officially published and the examples were written in anticipation of publishing these ontologies (which then never happened).

The later SensorML JSON specification uses roles like http://dbpedia.org/resource/Datasheet, so this seems good to use here as well.

Dbpedia seems a crowd-funded huge database of concepts. Searching it seems to be hidden a bit, but https://lookup.dbpedia.org/ works reasonably well.

2.1.13 Nomenclature

STA settled on a rather unintuitive nomenclature, with the consequence that human observers become a Thing. This seems an unnecessary barrier to entry. Can we propose a dialect and translate entities? E.g.

- Thing → Observer (who)
- FeatureOfInterest → Observee (where)

- ObservedProperty → Phenomenon (what)

3 Approach 1: existing STA implementation

API Accessible at: https://data-test.meetjestad.net/SensorThings/v1.1/
Code published at:
https://github.com/meetjestad/mjs_backend_design/tree/backend-prototype-frost-sta

This proof-of-concept centers around the use of FROST-Server (https://github.com/FraunhoferIOSB/FROST-Server), an existing open-source SensorThings v1.1 implementation, which is used as-is, together with a number of Python-based services that handle reception, conversion and insertion of the data.

In this proof-of-conept, data produced by measurement stations (transmitted via LoRaWAN) is converted into the SensorThings model and SensorML metadata directly when it is received and then stored inside the FROST-Server in that format.

Initially, this is done by converting the existing data format, by adding metadata and converting the data into the right format. This needs a fairly big converter / data ingester, which has a lot of out-of-band knowledge about stations and sensors used.

In the next step, a new protocol is developed for communication between the measurement station and the server which is aligned with the CSDIF data model, making the measurement station produce data already in (a heavily compressed) CSDIF format. This means the server side data ingester is very light-weight, mostly applying straightforward decompression and enriching of metadata according to predictable principles.

3.1.1 ValidTime queries

To find the currently valid (or harder: valid at a particular time) version of a Thing, you need to query based on the validTime property inside the SensorML. FROST-Server supports querying these nested values, but does not know the schema for the SensorML metadata, so cannot use datetime filtering on these fields.

However, since datetimes are stored as ISO formatted strings, they should be string-comparable as well, assuming they are all in the same timezone (and the consumer knows this timezone).

3.1.2 Metadata query performance

To find the Thing to add an observation to, the datastore must be queried based on the Thing's uniqueid and its validTime. These fields are not default STA fields, but embedded in the metadata field (with free-form JSON content). FROST-Server supports these queries (using postgres jsonb fields), but it is expected that if the dataset grows, such queries will become prohibitively expensive and slow. To fix this, indexes must be added to support such queries. It seems postgresql supports such indexes, but this needs further investigation (also to see if/how FROST-Server supports creating these).

4 Approach 2: STA on top of existing datastore

API Accessible at:
https://sensorthings-api.meten-natuurlijk.nl/sensorthings-api/SensorThingsService/v1.0
Code published at: TBD

The "Zeister implementation" is an implementation of the OGC SensorThingsAPI build using the Rust programming language and Axum. It does not store any data, but queries existing measurements from a data-access-api. The data-access-api acts as a Elasticsearch facade, and also does not store any data. The data-access-api is needed to filter out data, like privacy sensitive data and non-production data, but also allows for data anonimization (i.e. reduce the number of digits on a GPS coordinate).

This has several advantages:

- No need for data duplication to a second data-store
- No extra costs for data-storage
- No data inconsistencies (the queried data is always accurate)
- Low maintenance solution (it can only be up or down, not inconsistent)
- Code ownership. Full control over the architecture and code quality. Adding features and fixing issues can be done quickly.
- The implementation has a replaceable backend. The implementation can be reused for other data-sources. This does not change the architecture of the

application. Only a specific backend-adapter and data mapper need to be developed.

It also has some disadvantages:

- A custom implementation of the OGC STA needs to be developed
- Responsibility for maintaining the source-code and fixing (security) issues.

The measurements in the Elasticsearch index are “rich measurements”. They contain measurement information and metadata like the time, location, device, sensor and measurement_type of the measurement.

This allows the OGC STA implementation to return Observations, Locations, Things, Datastreams, FeaturesOfInterest and ObservedProperties from a single data source.

The OGC STA implementation generates synthetic ID’s for @iot.id which are round-trip proof.

The implementation consists of an API-layer (frontend), Routing-layer, Mapping-layer and Model layer. The software uses Dependency Injection for increased testability.

4.1.1 Rust/Axum routing

Rust/Axum by default uses a routing pattern that is based on /path_a/:id and **/path_a/:id/path_b**
This conflicts with the routing pattern that OGC STA requires. The OGC STA requires parentheses around an ID: **/type_a/(:id)/type_b**

The default Axum router does not handle the brackets nicely, so a special form of routing needs to be implemented.

5 STA data consumer – map with sensitive locations

Application accessible at: https://meetjestad.net/spuksla/map

For exploring the data interactively, a map application was created. This application runs entirely in the browser, using direct HTTP requests to any CSDIF or SensorThings server.

In addition to working with the two systems built for this proof-of-concept, the application was also made to work with the RIVM Samen Meten platform at

https://api-samenmeten.rivm.nl/v1.0/. This platform also uses SensorThings, and even though it lacks the additional (SensorML) metadata that CSDIF uses, the basic observational data can still be obtained via the regular SensorThings interface.

5.1.1 Datastream abstraction

When writing the consumer, the datastream abstraction (especially combined with multiple values in the same observation/datastream) turned out to be somewhat tricky to query and comprehend.

Whether or not to combine values is already discussed elsewhere.

On additional observation is that (multi)datastream is essentially an abstraction that has no direct relation to a real-world concept. This is unlike other concepts like Thing, Observation, FeatureOfInterest, etc.

Essentially a datastream is just a way to group similar observations, where the similarity includes at least having the same ObservedProperty, UnitOfMeasurement, Sensor and Thing, but additional grouping could also be made. If no such additional grouping is made, one can also ignore datastreams semantically and consider Observations as a big unsorted pile of them, and only see the datastreams as a syntactical concept via which Observation properties like Sensor or ObservedProperty are stored/accessed.

Possibly one big reason why datastreams exist at all, is to reduce data duplication and make Observation objects as small as possible.

5.1.2 Verbosity and purpose of names and description

In various places, the data model has name and description properties, such as for a Thing, Datastream, Sensor or ObservedProperty. The SensorThings specification does not provide specific guidance about the purpose and format of these properties, other than that they are intended for human consumption.

In an initial version of the proof of concept, the ObservedProperty name was filled with very compact names (which could be used as variable names), but those are not very friendly to display to users. For this reason, the description was used to display, but that turned out to be too verbose.

In a later version, the names were changed to be human-readable but concise, with the description being more verbose.

The CSDIF specification should provide some guidance and examples on these properties.

5.1.3 Multiple APIs for different use cases?

Discussing data interchange we need to discriminate between data storage, data exchange, data consumption. These each come with different tradeoffs between verbosity vs (footprint)efficiency, universality vs accessibility, queryablilty vs (processing)efficiency. To what extent can these uses be accommodated into one API?

Another way to cut this may be to discriminate between power users and first timers and develop two API's: one for conviviality and another one for data nerds.

5.1.4 Minimizing data traffic

How to minimize data traffic: supporting $export=CVS or $export=geoJSON would cut some overhead and also address some of the concerns stated above (5.1.3).

6 Revision history

- 2026-01-23: Draft 1
  - Initial version.