# Request for comments: Community Science Data Interchange Format

*Document identifier: CSDIF-001-RFC*
*Revision: 2*
*Date: 2025-06-26*


*https://www.csdif.info*

## Table of Contents

# 1 Introduction

## 1.1 Document scope and status

This document makes the case for introducing a new standard format for data interchange between citizen and community science initiatives and institutional partners, building on top of an existing API and standard.

This document shows the background and need, and sketches a proposal for this new format. It does not propose a complete specification for this format, but has enough details to collect feedback and to start a prototype implementation to try out the concepts.

This document was created by Meet Je Stad Amersfoort together with SMAL Zeist, with some input from other parties. As a next step, these ideas will be shared and discussed more broadly.

It is expected that some new revisions of this document will be published, followed by a more complete and more formal specification of the proposed format.

The most recent version of this document can be found at www.csdif.info. For a revision history see chapter 4.

## 1.2 Community vs Citizen Science

The name of this proposal is “Community Science Data Interchange Format”. This intentionally uses the name “Community Science” instead of the commonly used “Citizen Science”.

We consider the name “Citizen Science” problematic because it emphasizes the distinction between institutes and citizens. It is also often used for institutional goals such as popularisation or participation. The name Community Science instead emphasizes doing science collectively, where everyone is involved as equal partners.

For more information, see also the Community Science Manifest.

## 1.3 Project and context

Many citizen science initiatives start as a one off effort to measure certain quantities, e.g. climate or air quality. A straightforward way for doing this is to design a piece of hardware that sends sensor data, and setup a database that is tailor made for the data sent by the sensor.

An introduction tutorial on data collection will show something like the following approach:

1. Measurement device firmware reads sensor and sends value periodically.
2. Server receives value and stores it in database together with a timestamp
3. Browser fetches time series data from server and displays it as a graph

This approach allows for rapid development of a working prototype and has the advantage of simplicity and transparency between what is going on in the measuring device, how data is stored in the database and eventually presented to an end user.

In this way many initiatives get started and design their own measurement devices, databases and web front ends.

However, this naive approach for setting up an initiative proves to be hard to scale.

Over time more sensors are adopted and the handling of ever more quantities has to be implemented in the firmware, transmission packets, database structure and application interfaces. Every new measurement experiment requires changes on all levels and the database becomes littered with empty fields not used in most cases.

Furthermore, in order to compare and exchange data between various DIY measuring initiatives and institutions a common language is needed to describe data as well as the context in which they are gathered. The naive approach often, applied in many initiatives, leaves this metadata implicit and inevitably lead to a Babylonian confusion of tongues.

In short: both the setting up of new experiments and the analysis of data from various sources are hampered by the same lack of a robust yet flexible data framework.

In this document we lay out the result of an exploration of existing standards, propose a practical subset to adopt for data interchange.

We will focus mainly on an interchange format which allows various initiatives to make use of each others’ datasets. The underlying software stack, firmware and protocols to actually collect these datasets is also relevant, but mostly left out of scope of this particular proposal.

We intend to find a language which is both concise and sparse in topic language, in order to keep the document relevant to both experts and beginners in software or data science.

## 1.4 Existing infrastructure and limitations

At the time of the writing of this document different initiatives and organizations use different ways to publish open data.

This data is often provided as downloadable (cold) data, and has a data schema that differs from initiative to initiative.

In addition, a lot of context around the data is implicit. For example, Meet je Stad keeps a “temperature” field in measurements, which refers to the temperature of outside air, in degrees Celsius, typically measured using a Si7021 (or HTU21D for older sensor stations). None of this context is explicit and it might not even be true for all data.

The data can be consumed, but the receiver needs to put a lot of effort in converting the data into a uniform format and making it comparable. In a lot of cases essential metadata is not available at all (and maybe not even known for certain anywhere).

## 1.5 Goals and usecases

The primary purpose of CSDIF is to make available data collected by reading sensors and making observations. This intends to allow sharing data with:

- Other community science initiatives
- Institutional partners: universities, environment agencies, municipalities etc.

CSDIF intends to support a wide range of data collection use cases, such as:

- Measurement station with a single sensor and a static location set by the station maintainer.
- Measurement station with multiple different sensors, moved occasionally with the location set by the station maintainer or based on periodic GPS readings.
- Mobile measurement sensor, location can vary from one measurement to the next (e.g. Meetjestad cityslam and Snuffelfiets).
- A measurement station that applies some calibration directly after reading the sensor.
- A data collection backend that applies calibration centrally at a later time, potentially based on analysis of a group of measurements.

- Measurements performed by human beings, such as periodic manual measurement of tree circumference, or observations of first bloom of plants.

CSDIF should offer measurement data along with relevant metadata that helps to interpret the measurement data. The focus here is on ensuring the relevant metadata is available or can be derived from other metadata. For example, exposing metadata about measurement accuracy is convenient, but less important than storing the type of sensor used to collect data, since the type of sensor can be later used to derive the measurement accuracy based on external documentation.

### 1.5.1 Non-goals

The scope of this document is to describe the interchange format that can be used to share open data between initiatives and organizations. Such parties collecting data can make their data available via this format, which can then be imported or used by any third party.

Some things are left out of scope of this proposal (but might be reconsidered in a future revision):

- Authentication and authorization: the data that is published is considered to be open data that can be queried anonymously.
- Support for streaming data: transfer of data is always initiated by the consumer of the data by bulk download or periodic polling.
- Fully automated data imports: ideally, importing data via CSDIF would be as simple as adding a URL to a list of datasets to import, but in practice some case-by-case setup might still be needed for each such dataset imported (because different datasets have different amounts of metadata, made different choices in representing metadata, etc).
- Using data for analysis and presentation directly: ideally, the CSDIF interface can be used directly by tooling that can be used to analyse data and present it (i.e. make maps and graphs), but fully supporting this would increase the complexity of the API needed. Some implementations can choose to support this, but CSDIF does not require this from all implementations.

Initiatives are free to add any of these to their implementation (being careful to keep their data meaningful for consumers that do not implement such extras), but CSDIF will not provide any means to standardize them.

### 1.5.2 Metadata concepts

In order to depart from the intuitive approach of a single purpose measurement and generalize these so that they become comparable we need to introduce a number of concepts.

**Number**

Data is typically formatted as a number. However, to understand a number we need to know its representation, precision, etc. Various standards can be chosen. The scientific notation uses Arabic numbers written in a decimal form with exponential notation. Adding a precision of 5 digits will yield something like $3.1415 \times 10^0$ for the value of π.

A number usually encodes for a physical phenomenon that is measured, and we need to know the specific quantity and the unit to make sense of a number, e.g. Temperature (the quantity) measured in degrees Celsius (the unit).

**Observer**

Data is the result of observations. This can be done by a human observer or by a sensor, an electronic device of a certain brand and type, sometimes with a serial number. When tools are being used for measuring they often need to be calibrated.

All this information is needed to better understand data e.g. to be able to trace down systematic measuring errors to their cause.

**Method**

In some cases a measurement is not a static affair but the result of a series of actions, each of which have an influence on the resulting data.

Calibration procedures too are described such a way.

Knowing the methods for measurement and calibration is essential to be able to replicate a certain result. These methods include also certain mathematical operations on (raw) data, the specific calibration parameters that were used etc.

**Environment**

In some cases we want to know the further circumstances in which a measurement or observation took place and that cannot be (entirely) controlled with the applied methods. E.g. the soil type in or on top of which a sensor was placed, its location, its orientation, its sun exposure directions, etc.

For each of above concepts standards exist, like the International System of Units (SI).

# 2 Proposal: Interchange format

For this proposal we made an inventory of existing standards to adopt instead of developing a completely new standard. Clear advantages are the possibility of connecting to existing datasets (already in such a format), making use of existing code for interpreting these data and being able to connect to, make use of and contribute to the communities that maintain these standards, datasets and code bases.

Many of the standards and systems we explored were either too simple (insufficient metadata or insufficient flexibility for heterogeneous and changing systems) or overly complex, imposing a steep learning curve only for the benefit of exotic use cases.

The result of this exploration is to use (a subset of) the OGC SensorThings API (STA) to offer read-only access to observation data (measurements), along with metadata about the systems and sensors used to generate that data. This metadata is encoded using (the JSON encoding of) the OGC Sensor Model Language (SensorML).

SensorThings is part of the OGC Sensor Web Enablement collection of standards, all of which share a similar base datamodel, some of which serve different usecases or are more modern replacements of older ones. Of particular mention is OGC API Connected Systems, a specification that is currently (mid 2025) being finalized. It is positioned as a possible followup to SensorThings, having a bit more clean, generic and complete API and data model. It was initially intended to be the basis of the CSDIF proposal, but because the API and specification were so generic and spread over different documents, they were also quite hard to understand. Since this conflicts with the goal of having a low barrier to entry, CSDIF now uses SensorThings instead. This poses some limitations (mostly in more advanced usecases that are not part of CSDIF itself), but such limitations also make the API more straightforward and accessible.

## 2.1 Approach

To prevent reinventing the wheel, this proposal builds on existing specifications. However, those existing specifications are more expressive than we need (and also can potentially express the same things in different ways). To prevent consumers of CSDIF data having to handle a lot of needlessly diverse data, CSDIF provides some additional restrictions and guidelines.

So, being compliant with CSDIF means:

1. Implementing the SensorThings API and data model (limited to the requirements listed in this document).
2. Any metadata added is encoded using the SensorML format, encoded in JSON, as shown in this document.
3. Where applicable, the SensorML descriptions use the vocabularies and terms defined by this document.

Implementations can support additional SensorThings features or add additional data (using SensorML or other formats, e.g. in the free-form “properties” field), but must take care that the data is still meaningful if such additions are not understood or supported.

### 2.1.1 Data model vs Access API

Conceptually, this proposal consists of two parts:

1. A data model for (meta)data, defining objects to store, what properties they have, what values those properties can have, etc.
2. An access API that defines how to this (meta)data can be queried and filtered and how the results are formatted.

To give some structure to this proposal, both parts are discussed separately, but note that in practice, both parts are intertwined, especially in the existing SensorThings API that CSDIF builds upon. This means that these parts cannot be implemented independently, but it might be that in a future version of CSDIF (or a separate, new, specification) the same data model could be used with a different access API as well, preserving some degree of interoperability.

### 2.1.2 Simple and complex implementations

Since different initiatives have different platforms, skill levels and data complexity, their (data producing) APIs might correspondingly vary.

To accommodate these different kinds of initiatives, some properties of simple or more complex implementations are defined in this section. These are not intended to be formalized in the interchange format itself (and are not strict categories), but are made explicit here to make the intended range of supported data producing applications more clear.

- A simple implementation might offer just observation data annotated with an ObservedProperty and unit of measurement and a location, with minimal information about the sensor (e.g. a reference to the datasheet PDF), no metadata about the measurement station, etc.

- A simple implementation might support just one or a limited set of measurement platforms. In this case, an implementation is still expected to offer such metadata via the API (as opposed to leaving this implied), to simplify data consumption and to make things explicit (but the metadata could be hard-coded internally, either when the data is stored, or when it is retrieved).
- A simple implementation can assume that a single measurement station (a “thing” in SensorThings terms) never changes (except for its physical location) and not deal with any validity time or other history provisions (and just create a new thing with a new identifier if something ends up changing anyway). A complex implementation could facilitate adding or removing sensors, changing metadata, etc. on a thing and export this with appropriate history.
- An implementation might add the SensorThings API endpoints onto an existing data collection platform (generating or converting some metadata on the fly) or might use an existing SensorThings data server. In the latter case, this could be the primary storage, or it could be a secondary storage intended just for publishing the data.
- A simple (or custom) implementation supports only very limited data querying, while a complex (or off-the-shelf) implementation might support complex queries and filtering.

It is to be determined if all of the above simple implementations should indeed be supported (to favor simple data suppliers), or if the minimal complexity should be raised (to favor simple data consumers), by e.g. requiring all sensors to have SensorML metadata and never allow a datasheet PDF.

## 2.2 Underlying standards

CSDIF builds on top of various standards, which will be briefly introduced in this section. How these standards relate to each other is indicated in Figure 1.

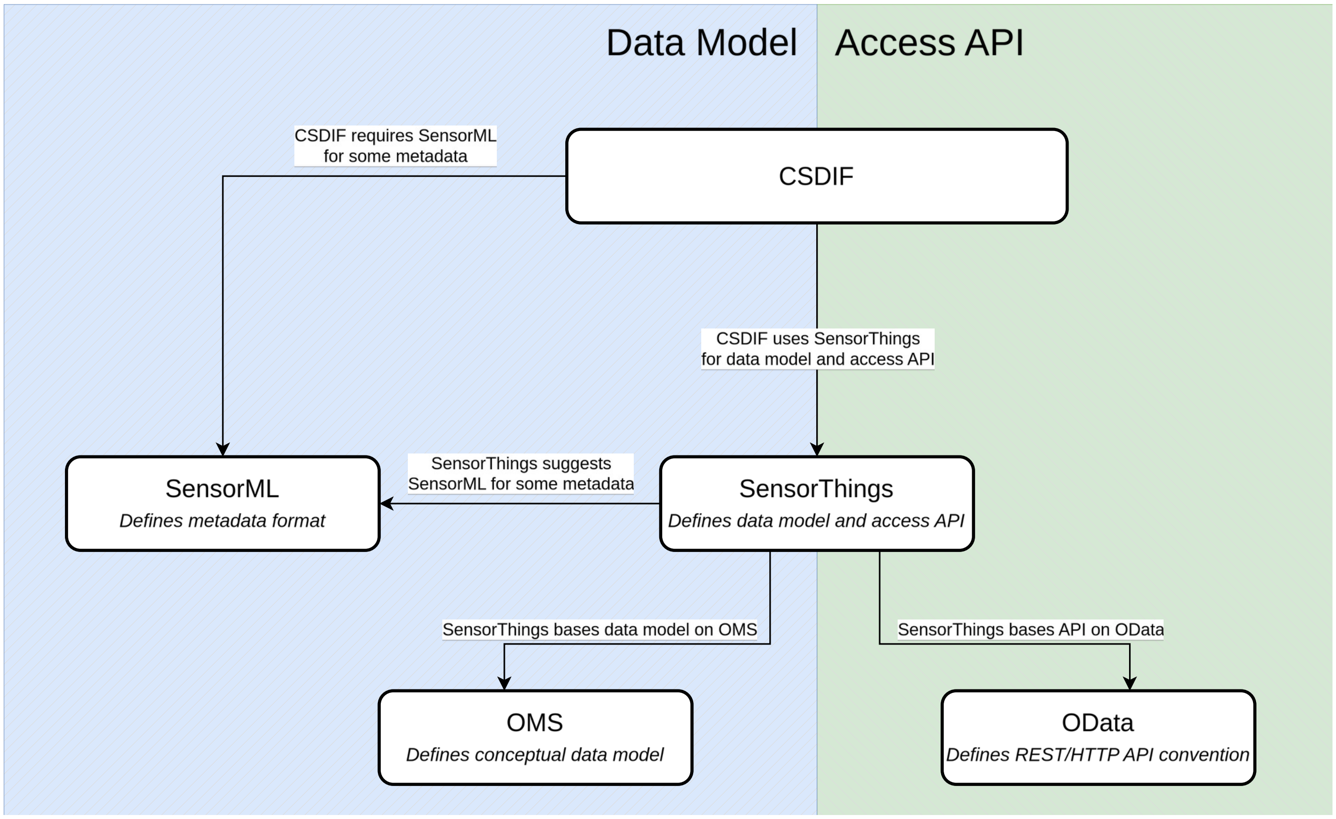


*Figure 1: Overview of standards used by CSDIF*

### 2.2.1 OGC Observations, Measurements & Samples (OMS)

"Observations, measurements & samples" (OMS, most recent version is [Observations, measurements and samples 3.0]) is an OGC specification that describes a conceptual data model for observational data. The model is conceptual and intended as a building block for other specifications (such as SensorThings and many other OGC specifications), in the sense that it does not specify any particular storage or exchange protocol or format.

The specification was originally called "O&M" and is still referred to as such in various places, but in recent versions the full title also has "& samples" added.

Section 7.1 of the specification gives a good overview of the concepts used and is summarized here.

#### ***Features***

Features are a generic concept (A tree, a forest, the outside air, a sensor, a measurement procedure, etc). Features have properties that can be defined specifically (by an authority, like names) or observed (by measuring, estimating) with some error margin.

#### *Observations*

An observation is the act of observing a property at a specific time instant or over a period, to find a numeric value or other characterization for the property. An observation can be done automatically with a sensor, but also manually following some kind of procedure, with or without instruments, on-site or in a lab, etc.

#### *Measurements*

A measurement is an observation that assigns a numerical value to a property.

#### *Values*

Values can be simple numerical values, counts or categories, but also more complex values such as timestamps or ranges, location, geometries, etc.

#### *Location*

OMS does not assign a location to an observation directly, since this is not a property that is necessarily known, relevant or even sensical. Location information can be modeled as a property of the feature of interest, or provided by the observation procedure.

#### *Time*

In contrast, temporal data (*when* was the observation performed) is a direct property of every observation in the OMS model.

#### *Other concepts*

An observation can be further decomposed into its feature of interest, its observer (a sensor, chain of measurements, simulation, person, etc.), the observed property, the observation procedure and the value. In some cases, a distinction can be made between the ultimate-feature-of-interest (what are you trying to observe) and a proximate-feature-of-interest (what are you actually observing).

Sections 7.2 and 7.3 add the concept of sampling, where observations are made of a subset of (or proxy for) the actual feature of interest.

### 2.2.2 OASIS OData

OData (latest version OData v4.0) defines how to structure a REST/HTTP service. It defines HTTP URL conventions, headers, response codes and JSON formats for requesting, modifying and filtering objects, their properties and their relations.

The SensorThings specification takes some concepts from OData, but is not fully compliant. In practice, the SensorThings specification document repeats the relevant parts from OData and should be readable without referring to the OData specification at all.

### 2.2.3 OGC SensorThings API (STA)

The SensorThings API (STA, full name "OGC SensorThings API Part 1: Sensing Version 1.1") is an API specification to expose observations and information about the observers that made these observations.

There is also a second part (OGC SensorThings API Part 2: Tasking Core) to allow executing tasks (*i.e.* control actuators), which is not used at all for CSDIF (so references to SensorThings or STA in this document usually mean part 1).

The specification builds on top of other specifications: Observations, Measurements and samples (OMS, originally "Observations & Measurements") for the objects and their relations, OASIS OData v4.0 for the REST/HTTP API structure, OGC SensorML Encoding Standard v3.0 for the encoding of metadata about some of the objects.

In the SensorThings data model, a number of objects are defined, which are listed in the next sections. Because the names of these objects (like "thing") are also regularly used in other contexts, we will use italics and capitalization (like *Thing*) to indicate when a word refers to an object defined by SensorThings.

An overview of these objects, their relations and attributes is provided in Figure 2. It contains extra detail for those familiar with such diagrams, but is not needed for the understanding of this document.

#### ***Observation objects***

An *Observation* models an observation. This is usually quantitative (also referred to as a "measurement" in other contexts), but could also include more qualitative observations (such as a categorization or textual description). An O*bservation* is always made at a particular moment in time and observes a phenomenon at a particular (potentially different) moment in time.

An O*bservation* often contains one value, but can also contain a composite value (e.g. latitude/longitude or another sensor that measures multiple values).

An *Observation* also has an optional *FeatureOfInterest* (the place or thing that is being observed, typically a location or area) and (via its D*atastream*) a *ObservedProperty* (the property of the feature of interest that has been observed, like air temperature or particulate matter concentration).

### *Datastream and MultiDatastream objects*

A *Datastream* models a series of observations made by a single sensor, but at different moments in time. A *MultiDatastream* is the same, but containing multi-valued observations.

### *Sensor objects*

A *Sensor* models an instrument that can make observations, producing a single (M*ulti*)D*atastream*.

Each S*ensor* has associated metadata, which can take various forms, such as the make and model of the sensor, a link to its datasheet, a structured description of various characteristics and capabilities, a textual description of steps taken (useful for manually executed observations), etc.

For simple deployments, a composite measurement device can be represented as a single S*ensor* object (producing multi-valued O*bservations*), for more detail it can also be split in multiple S*ensor* objects each representing a single measurement instrument or module.

A Se*nsor* corresponds to the "Procedure" concept in the underlying OMS standard. Note that SensorThings leaves it undefined whether a *sensor* refers to a type of sensor, or a specific sensor (instance).

## Thing objects

A *Thing* models a physical or virtual thing as meant in “the internet of things”. It is a very generic concept, but in our context typically means something that produces D*atastreams* with observations using Se*nsors*. This would typically be a measurement device, but could also model for example a human making notes.

A *thing* also has an optional L*ocation* (and can keep a history of *HistoricalLocations*).

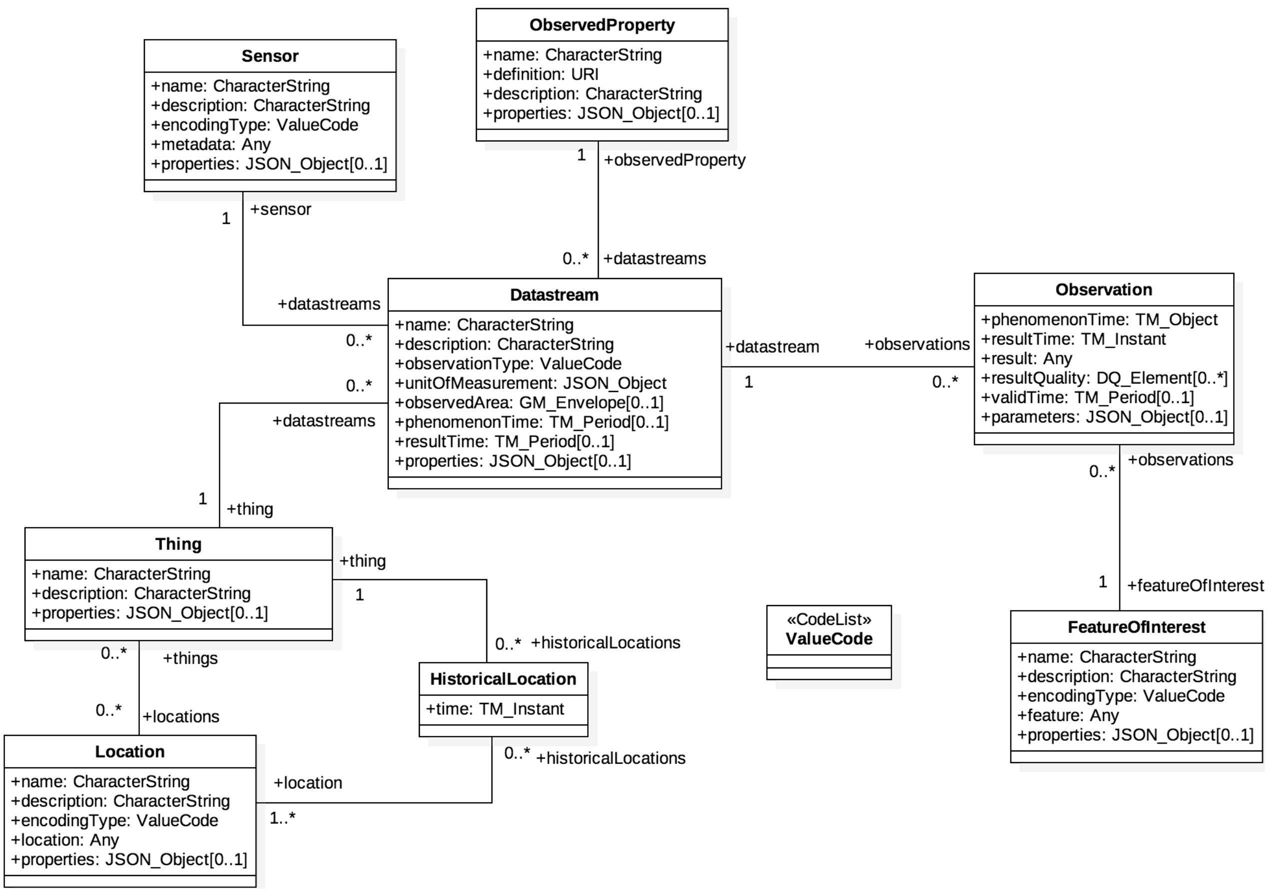


*Figure 2: SensorThings data model (image taken from STA v1.1, ©2021 OGC)*

## Object attributes

SensorThings defines a number of attributes for each of its objects, but this is a limited and fixed list of attributes (except for the “properties” attribute which is a free-form JSON object property that can be used for implementation-defined additional properties).

For *Sensor* objects, a free-form “metadata” attribute is available for expressing metadata in some externally defined format (with the “encodingType” attribute defining what format is used). SensorThings recommends storing a SensorML description in this field, which can be used to store any number of metadata attributes about these sensors.

#### *API Endpoints*

Each of the objects defined above has a corresponding HTTP endpoint (e.g. `/Things({id})` to access a specific thing), typically a list endpoint (e.g. `/Things` to get all known things, optionally filtered using query parameters) and can often also be accessed indirectly (e.g. `/Things({id})/Datastreams` to get all datastreams of a given thing).

The content of these endpoints is a description of the object and its attributes in JSON format.

### 2.2.4 OGC Sensor Model Language (SensorML)

SensorML is a language to model observatory systems, their properties and structure.

A SensorML description of a system can be serialized into an XML or JSON document, which can then be stored in for example the “metadata” attribute of a SensorThings *Sensor* object.

SensorML defines a hierarchical data model, of which CSDIF will use only the “AbstractProcess” class and its subclasses. Each of these has properties like identification, classification, capabilities, contacts, inputs and outputs. Each of its subclasses define extra properties. These are roughly divided into two groups:

1. AbstractPhysicalProcess, PhysicalComponent and PhysicalSystem, for defining processes that have a physical representation (measurement stations, sensors, etc.). These classes add properties like position, temporal and positional reference frames and subcomponents.
2. SimpleProcess and AggregateProcess for defining non-physical processes, typically computations (possibly as a subcomponent of a composite PhysicalSystem).

In CSDIF, SensorML descriptions are used to describe thing and sensor objects.

To describe objects, SensorML takes a generic approach. For example, the specification defines an “identifiers” property, which is a list of properties that help identify the object. For a sensor, this could be the manufacturer and model number of the sensor. The SensorML specification, however, does not define what identification properties are available, but instead relies on external ontologies (vocabularies) of possible properties.

As an example, a sensor description could contain:

```
"identifiers": [
  {
    "definition": "http://sensorml.com/ont/swe/property/Manufacturer",
    "label": "Manufacturer Name",
    "value": "Sensirion"
  },
  {
    "definition": "http://sensorml.com/ont/swe/property/ModelNumber",
    "label": "Model Number",
    "value": "SPS30"
  }
]
```

This defines two properties, which point to an external definition using a URI to define these properties. This allows relying on existing systems (i.e. DNS and HTTP) to handle allocation and uniqueness of these identifiers. Additionally, these URIs typically point to a webpage that contains (human or machine-readable) information of what the property means. In this case, the information is very consise (http://sensorml.com/ont/swe/property/Manufacturer says the property should contain *"The organization responsible for building the system."*), but such a webpage could contain a more precise definition of a property as well, possibly also providing (or referencing) a list of possible values (for example country codes) or a format (for example a time format).

Other properties are more precisely defined in the SensorML specification (sometimes building on top of other standards, such as ISO19115 for contacts and legalConstraints), or just simple values. Some properties are also more open ended, such as characteristics and capabilities, for which a bit of structure is defined by SensorML, but the actual value uses the generic SWE Common dataformat, requiring external agreement on what these values (which is something this proposal will provide in a future revision).

SensorML was originally specified to use an XML representation, but the (currently in development) version 3.0 of SensorML replaces that with a JSON representation instead (while keeping the underlying datamodel the same). For CSDIF, the draft version of the 3.0 JSON representation will be used.

## 2.3 CSDIF Example

To get an idea of the exchange format proposed, an example is first presented. The example is not explained in detail, but serves as a first impression to also make the upcoming sections with explanations a bit more tangible.

The example shown here is MJS2020 measurement station number 2000, which is a single board containing a number of sensors. For simplicity, this example only shows the Si7021 sensor attached to the board, omitting other sensors.

The station is modeled as a T*hing*, which has a single *Datastream* containing values produced by the Si7021 *Sensor*. The main JSON structure is specified by the SensorThings API specification, while the "metadata" fields contain information about the station and sensor according to the SensorML specification.

To query information about this station, one might make an HTTP GET request to a URL like:

```
https://sta.example.org/v1.1/Things(2)?$expand=MultiDatastreams,MultiDatastreams/
Sensor,MultiDatastreams/ObservedProperties
```

This requests metadata on the measurement station (modeled as a *Thing* in SensorThings) with ID 2, and includes (expands) any related *MultiDatastream* objects, plus any *Sensor* and *ObservedProperty* objects related to those *MultiDatastreams*.

The result would be a JSON-formatted response like below. The structure of this response is defined by SensorThings, except for the parts marked in purple which are defined by SensorML.


```
{
  "name": "station-2000",
  "description": "MJS2020 station built for MJS Amersfoort",
  "properties": {
    "encodingType": "application/vnd.ogc.sml+json",
    "metadata": {
      "type": "PhysicalSystem",
      "definition": "http://www.w3.org/ns/sosa/System",
      "uniqueId": "urn:fdc:meetjestad.nl:2024:thing:station-2000",
      "label": "MJS2020 station built for MJS Amersfoort",
      "validTime": [
        "2025-02-28T22:30:21.428175812Z",
        "now"
      ],
      "identifiers": [
        {
          "definition": "http://sensorml.com/ont/swe/property/Manufacturer",
          "label": "Manufacturer Name",
          "value": "Meet je stad"
        },
        {
          "definition": "http://sensorml.com/ont/swe/property/ModelNumber",
          "label": "Model Number",
          "value": "MJS2020"
        },
        {
```

```
          "definition": "http://sensorml.com/ont/swe/property/SerialNumber",
          "label": "Serial Number",
          "value": "2000"
        }
      ]
    }
  },
  "MultiDatastreams": [
    {
      "name": "Si7021 temperature and humidity data",
      "description": "",
      "observationType":
"http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_ComplexObservation",
      "multiObservationDataTypes": [
        "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
        "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement"
      ],
      "unitOfMeasurements": [
        {
          "name": "Degree Celcius",
          "symbol": "°C",
          "definition": "ucum:Cel"
        },
        {
          "name": "Percent RH",
          "symbol": "%",
          "definition": "ucum:%"
        }
      ],
      "Sensor": {
        "name": "Si7021",
        "description": "Si7021 temperature and humidity sensor",
        "encodingType": "application/vnd.ogc.sml+json",
        "metadata": {
          "type": "PhysicalComponent",
          "uniqueId": "urn:uuid:72cdcb94-86ae-4513-aada-3cf4d297aa52",
          "definition": "http://www.w3.org/ns/sosa/Sensor",
          "label": "Si7021 Temperature/Humidity Sensor",
          "identifiers": [
            {
              "definition": "http://sensorml.com/ont/swe/property/Manufacturer",
              "label": "Manufacturer Name",
              "value": "Silicon Labs"
            },
            {
              "definition": "http://sensorml.com/ont/swe/property/ModelNumber",
              "label": "Model Number",
              "value": "Si7021"
            }
          ]
        }
      },
      "ObservedProperties": [
        {
          "name": "temperature",
          "definition": "http://qudt.org/vocab/quantitykind/Temperature",
          "description": "Temperature"
```

```
        },
        {
          "name": "humidity",
          "definition": "http://qudt.org/vocab/quantitykind/RelativeHumidity",
          "description": "Humidity"
        }
      ]
    }
  ]
}
```

To actually get the data (*Observations*) for one of the *Datastreams* of this station, one could make an HTTP GET request to:

```
https://sta.example.org/v1.1/MultiDatastreams(1)/Observations
```

Which would result in:


```
{
  "value": [
    {
      "@iot.selfLink": "https://sta.example.org/v1.1/Observations(1)",
      "@iot.id": 1,
      "phenomenonTime": "2019-03-14T10:00:00Z",
      "resultTime": "2019-03-14T10:00:00Z",
      "result": [21.0, 30.3],
      "Datastream@iot.navigationLink":
"https://sta.example.org/v1.1/Observations(1)/Datastream",
      "FeatureOfInterest@iot.navigationLink":
"https://sta.example.org/v1.1/Observations(1)/FeatureOfInterest"
    },
    {
      "@iot.selfLink": "https://sta.example.org/v1.1/Observations(2)",
      "@iot.id": 2,
      "phenomenonTime": "2019-03-14T10:01:00Z",
      "resultTime": "2019-03-14T10:01:00Z",
      "result": [21.6, 28.9],
      "Datastream@iot.navigationLink":
"https://sta.example.org/v1.1/Observations(2)/Datastream",
      "FeatureOfInterest@iot.navigationLink":
"https://sta.example.org/v1.1/Observations(2)/FeatureOfInterest"
    }
  ]
}
```


## 2.4 Data model

### 2.4.1 Objects from SensorThings

In CSDIF, the object model from SensorThings is used (again, object names in italic refer to the corresponding objects in the SensorThings specification):

- A *Thing* refers to a measurement station that collects measurements (and typically transfers them to a central location). It has a *Location* and optionally *HistoricalLocations* to store its position.
- A *MultiDatastream* refers to a collection of multi-valued *Observations* (measurements) produced by a specific instance, using a given type of *Sensor* collected by a given *Thing* observing one or more *ObservedProperties*. For simplicity, the *Datastream* object is not used, modeling single-valued observations as a *MultiDatastream* containing (a list of) just one value instead.
- A *Sensor* refers to the type of sensor (so multiple *MultiDatastreams* can refer to the same *Sensor* object if they both use the same type of sensor).

The attributes of these objects are used as defined by the SensorThings specification, with the following additions:

- For metadata on *Things* and *Sensors* ("metadata", "encodingType" and "properties" attributes), SensorML content is used, as described below.
- For locations ("location", "feature" and "encodingType" attributes on *Location* and *FeatureOfInterest* objects), CSDIF follows SensorThings by using GeoJSON.
- For the *ObservedProperty* "definition" attribute and *MultiDatastream* "unitOfMeasurements" attribute, external ontologies are used, as described below.
- The "description" attribute on all objects can be empty (In SensorThings 1.1 the "description" attribute is required, but since a value is often redundant with the name or other properties, CSDIF explicitly allows it to be empty, looking forward to SensorThings 2.0 that is expected to make it optional).

### 2.4.2 Metadata using SensorML

Since SensorML is very expressive, some specific guidance will be defined in the final CSDIF proposal. In particular:

- What properties should be minimally defined, and (where appropriate) which definition URLs to use for them. This ensures that most data will have a minimal set of interchangeable metadata. This includes things like make and model for things and sensors.

- What properties and other structures to use for other common metadata. This ensures that if two implementations define the same thing, they will also use the same property and values.
- What parts of SensorML are not expected to be used (so compliant implementations can ignore them).
- What SensorML class to use for which object and which usecase.

### 2.4.3 Ontologies

Both SensorThings and SensorML use URLs for defining various properties and values, which allows relying on terms from external ontologies (repositories describing related concepts and terms and assigning a unique identifier to each).

To ensure interoperability, compliant implementations should ideally use the same terms. To facilitate this, the CSDIF specification will recommend particular ontologies to use, and for specific concepts recommend one specific term to use.

The ontologies that will be used for this are likely at least:

- http://sensorml.com/ont/swe/property for various thing and sensor metadata properties.
- https://www.qudt.org/doc/DOC_VOCAB-QUANTITY-KINDS.html for observed properties (quantities).
- https://ucum.org/ (or https://www.qudt.org/doc/DOC_VOCAB-UNITS.html) for units of measurement.

Where possible, alternative ontologies can also be referenced to establish which terms (from different ontologies) can be considered interchangeable. This could be done by explicitly listing terms, or referring to external lists (e.g. qudt.org already lists the matching ucum.org identifiers for its units where appropriate).

### 2.4.4 Identifiers

In SensorThings, every object has a single (local) identifier, which is an identifier that is local to the server using it and is unique only among all objects of the same type on the same server. No provisions are made to guarantee uniqueness across different servers and object types. This is the identifier that is used in the API endpoints.

SensorThings has no provision for a globally unique identifier on any of its objects. However, to simplify tracking objects across servers when data is exchanged, it is helpful if such a unique identifier is available.

*Things* and *Sensors* are described with SensorML metadata. In SensorML there is a mandatory "uniqueID" property (defined by the draft SensorML v3.0, section 9.1.4.1 "Unique Identifier") which would be convenient to use for this purpose. SensorML defines this to be URI (uniform resource identifier) that is globally unique and suggests it to be a URN (uniform resource name, which is a specific kind of URI using the "urn:" scheme). Some examples are given below.

CSDIF follows SensorML (and also Connected Systems) and requires every *Thing* and *Sensor* to have a globally unique identifier, which is stored as part of the SensorML metadata. Whenever possible, the same identifier should be used on all representations of the same physical object, but it is acceptable if multiple identifiers are used for the same physical object (This can happen in simple implementations that do not track history but instead assign a new identifier when an object changes, or when multiple servers/organizations each assign their own unique id for the same physical object). It is not acceptable for the same identifier to be used to represent different physical objects, not even on different servers.

Note that it is possible for the same (global) identifier to be used with multiple (server) objects (with different local identifiers), with non-overlapping "validTime" properties, when representing the history of a single physical object that changed over time (see also the section on history).

### *Unique identifier for observations*

As shown above, *Things* and *Sensors* have a unique identifier assigned (stored in their SensorML-encoded metadata). However, *Datastreams* and *Observations* have no SensorML metadata, so only have local identifiers.

This means that if data is synchronized from one server to the other, there is no easily available stable identifier to correlate them.

For *Datastreams*, this could be fixed by storing a unique identifier in its free-form "properties" field, but an *Observation* only has a "parameters" field which is free-form, but intended for process parameters used for this particular *Observation*).

In case such an identifier is needed, CSDIF could recommend to compose one made from the *Thing* unique identifier, *Datastream* name and the *Observation* phenomenonTime, which should together uniquely identify the *Observation*. Alternatively, the *Sensor* uniqueId could be used with the timestamp (if the sensor represents a sensor instance, not type).

A future revision will make this recommendation more specific.

### *Generating unique identifiers*

For the unique identifiers, SensorML requires using a Unique Resource Identifier (URI), and recommends using a Unique Resource Name (URN, which is a particular kind of URI). Such a name consists of a scheme, a colon and then a string whose structure depends on the schema.

There are a lot of different URN schemes defined, the IANA organization keeps of list of them here. Each of these schemes defines its own rules to ensure that any URN generated is unique (usually by referring to some existing external registry of names to identify an organization, that can then assign its own identifiers below that).

For data exchange with CSDIF, any URI can be used (as long as uniqueness is guaranteed, for example by using a URI that includes a DNS-derived name owned by the organization that generates the identifiers).

It is suggested that implementers use:

- The “fdc” URN scheme (defined in RFC4198), which is intended for “federated communities” where data is produced by different organizations and is shared. An example of such a URN would be “urn:fdc:example.org:2025:system/2000”. This consists of the “urn:fdc:” prefix, then a dns name controlled by the organization defining the identifiers, then a date (just a year in this case) on which the identifier scheme was established, followed by an arbitrary identifier. In this example, that identifier again follows a nested structure using a type and a system number, but any scheme that allows the organization to generate unique and stable identifiers works.
- The “uuid” URN scheme (defined in RFC9562) builds on the concept of the Universally Unique Identifier (UUID), which are generated from (a combination of) random numbers (big enough to assume uniqueness), timestamps, MAC addresses or hashes from other identifiers.

## 2.4.5 Storing Metadata

In SensorThings, a *Sensor* object has a “metadata” property that can store info about the *Sensor*. The “encodingType” property indicates the type of metadata stored. Defined values are pdf, html and sensorML (the 2.0 XML version), but other values are also allowed.

The “metadata” field is typically a URL to the metadata document (e.g. for a PDF), but is also allowed to contain the metadata content itself (if it is representable as JSON). See also OGC SensorThings API Part 1: Sensing Version 1.1, section 8.2.5.

This means that the newer SensorML JSON encoding can be used to store the SensorML metadata directly in the “metadata” field, as shown in the earlier example. For the “encodingType” the value “application/vnd.ogc.sml+json” should be used while the SensorML 3.0 specification is still in draft, switching to “application/sml+json” once the specification is final.

In SensorThings, a *Thing* does *not* have a metadata property. Since metadata might also be relevant for a thing (even though it might be less important), it can still be stored using the free-form “properties” field that SensorThings defines. In CSDIF, the “metadata” and “encodingType” fields of are defined as fields below the “properties” field of *Thing*, with the same meaning as the same fields defined for *Sensor*.

## 2.5 Access API

The access API for CSDIF is as defined by the SensorThings base specification (omitting all extensions, like Create/Update/Delete, Batch Requests and MQTT).

See the example at the start of this chapter for an initial example of the API, see the SensorThings specification for more details.

### 2.5.1 Query options

The SensorThings specification defines a “$filter” query parameter on all endpoints that allows extensive filtering options, including basic comparisons, some arithmetic, string processing or spatial comparisons using object attributes (see paragraph 9.3.3.5 of the SensorThings specification). Implementing all these query options would would be challenging when not using an existing SensorThings implementation. If possible, implementations should implement this, but to allow adding the SensorThings API on top of existing systems, this is optional.

If it turns out that some subset of these query options should be supported to allow practical data exchange, such a reduced set might be added as required in a future version.

## 2.6 Other considerations

### 2.6.1 History of *Things*

Over time, a *Thing* might be modified, which results in its metadata or its collection of associated *Sensors* changing. To ensure that each *Observation* can be linked to the appropriate metadata at the time of observation, multiple versions of the same *Thing* can exist. Each version has the same unique identifier (in the SensorML metadata), but a different local resource identifier (so a different URL). Each version also specifies a different (non-overlapping) validTime property (in the SensorML metadata).

For consumers that do not care about *Thing* history and just want the observations, this looks like the *Thing* that was measuring is removed and replaced by a new *Thing* that measures something (possibly) different.

Consumers that do care about correlating measurements coming from the same station, can link multiple versions of the same *Thing* based on their unique identifier.

As a special case, when the location of a *Thing* changes, but no other metadata, this should be recorded using the SensorThings *HistoricalLocation* objects (which essentially links a thing to its previous *Locations*, annotated with the time until that previous *Location* was applicable).

Of particular note is that it is possible (and likely will happen in practice) that a modification of a *Thing* (or its location) is recorded after the fact, by simply using an older timestamp for the validTime or *HistoricalLocation* timestamp. This could result in data consumers already having applied the old metadata to later data where it was not applicable. Consumers should keep this in mind, and possibly periodically recheck the validity of existing things and locations and update or re-import their data accordingly.

### 2.6.2 Thing location

In SensorThings, each *Thing* has an associated *Location* (plus a history of previous *Locations*). Whenever possible, these should be set to reflect the position of the *Thing*.

When such a position is manually configured, filling these fields is obvious.

For *Things* that contain a GPS or similar localization device, it is recommended that the output of such as a device is modeled as a separate *Datastream* to allow access to the raw location output as sent by the device. In addition, such data can be used (possibly combined with other data sources such as manual configuration or correction by a user) to fill the *Thing* location field. In the most simple approach, the GPS location is put into the *Thing Location* directly. This could result in a lot of *Location* changes (i.e. GPS noise that does not correspond to location changes in the real world), so for some implementations is could be better to consolidate multiple nearby locations (maybe taking into account the position fix accuracy info) into a single *Location* (and/or just not changing the *Location* when a new *Observation* is very close by).

It would be useful if the *Location* of a *Thing* would be annotated with its source (and possibly other metadata like accuracy). It seems that the GeoJSON specification that is used for *Locations* does not define such properties, but does allow additional properties to be added. The final CSDIF proposal could define such properties.

In addition, an *Observation* can have a *FeatureOfInterest*, which (when omitted) defaults to the *Location* of the *Thing* (at the time of the *Observation*). This can be useful when the *FeatureOfInterest* is not identical to the *Thing* location (such as a camera looking out over a wider area), or when no *Location* is stored for a *Thing* (which could be chosen for mobile systems where you would otherwise create many *HistoricalLocations*).

### 2.6.3 Coordinate reference frames

A *Location* (which can be a *Thing's Location*, or some measured position) is always measured in a particular frame of reference, that indicates how to interpret the *Location* value (coordinates).

For the *Thing's Location*, SensorThings refers to GeoJSON to specify the location. In GeoJSON, the coordiate reference system is always implicitly WGS84 (GPS), which is also applied to CSDIF. If any implementation ever needs to use a different system (and cannot convert to WGS84), it must use a different content type than GeoJSON (which will break CSDIF compatibility, which is better than silently misinterpreting data).

For *Datastreams* containing location data, GeoJSON is not applicable (instead one would typically use a *MultiDatastream* with latitude, longitude and altitude fields). This does not allow direct annotation with a reference system, but this could likely be done in the SensorML description of the associated *Sensor*. It is to be considered if such an annotation needs to be declared mandatory by CSDIF, or if coordinates should be assumed to be WGS84 (which would mean using any other system could still be annotated, but would be silently misinterpreted by consumers that do not support such annotation).

Note that SensorThings 2.0 (as well as Connected Systems) is expected to use SWE Common encoding for values, which requires explicit annotation of a reference frame on all coordinate (vector) values, so maybe adopting such an approach might also be feasible.

### 2.6.4 Data modified for privacy reasons

For some usecases, the data that is published might be modified for privacy reasons (e.g. reducing location accuracy). If this is done, this should be somehow noted in the metadata so a data consumer can detect his, but it must still be determined what properties can be used for this (or maybe this must be modeled as a composite system with processing applied to the sensor output? But how about the *Thing's Location*?).

### 2.6.5 Timestamp precision

When embedded devices are involved in generating timestamps for measurements, these might not be as accurate as timestamps generated by time-synchronized internet-connected systems. It would be useful if the accuracy of observation timestamps can be specified, so applications that need to correlate multiple readings can do so with more confidence.

It is to be determined how to specify this. One option would be to specify clock accuracy as a property of the *Thing* or *Sensor*, or model the clock as a component of the *Thing* or *Sensor* and annotate that accordingly. However, in some cases, the accuracy might not be the same between all *Observations* by a single *Thing* or *Sensor* (i.e. the clock synchronization of a *Thing* might be improved over time, or when measurements are queued to be sent in a bundle, an observation timestamps could be a combination of an accurate “receive time” combined with different inaccurate “measured X ago” intervals). In such cases, the annotation might need to happen on each *Observation* separately.

### 2.6.6 Data ownership and licensing

The use of data is subject to various legal restrictions. This proposal is intended to be used for “open data”, but that is still a diverse concept that does not mean data is free of any restrictions.

To allow reuse of such data, a license should be applied to it. This makes it explicit to consumers what is allowed with the data (e.g. use, redistribute, modify, etc.) and under what conditions (e.g. by providing credit).

In implementations where all data is made available by a single party under the same license, that license could be communicated out-of-band (e.g. on the website linking to the data). However, it is recommended to always add a license in-band, as part of the exposed metadata to ensure the license is always up-to-date. This is especially important when data from multiple sources is aggregated.

For annotating license info, the STAPlus specification can be used. This defines an additional *License* object (with properties like license name/definition, logo and attribution text) that can be associated with a *Datastream* to attach a license to all *Observations* in that *Datastream*.

Alternatively (e.g. if STAPlus turns out to be not well-supported by existing implementations), the attributes defined by the STAPlus *License* object could also be stored in the "properties" field of a *Datastream* (resulting in some duplication of content).

Note that SensorML also has a "legalconstraints" property (referencing pay-to-view ISO19115, see this workbook for some public info about that), which could be used to model constraints on data usage, but its values are a not well-defined, and also the scope (i.e. do the constraints apply to the thing metadata, or to the observations produced by it) is not very explicit. This means these legalconstraints are probably not useful here.

A related concept is that of data ownership, which is also often mentioned in the context of citizen and community science data. However, the legal status of data owership is not very well-defined. STAPlus defines a *Party* object as the owner for *Things* and *Datastreams*, but also does not clearly define its meaning (in examples it is used for both authorizing write access to the data and for documenting the origin of data). For this reason, the STAPlus *Party* concept is probably best left alone. If registration of some sort of data ownership is needed, it should be made very specific what such ownership actually means (e.g. user who owns or operates the data generating hardware, initiative that collected that data, party that has particular rights to the data, party that offers the license, etc.).

### 2.6.7 Object names, labels and descriptions

In various places, objects can have a name, description and/or label. All of these are intended to be human-readable strings with varying levels of details. Using all of them can lead to some duplication in the data.

For example, a *Sensor* has a "name" and "description" attribute (as defined by SensorThings), but the associated SensorML metadata also has a (mandatory) "label" attribute that will typically encode the same thing as either the "name" or "description". It might make sense to leave the "label" empty (omitting it would violate the SensorML spec) and mandate that the "description" should be used instead. However, for usecases where existing software is used to work with the SensorML description, which might not be aware of SensorThings and/or CSDIF, it might be beneficial to preserve this duplication to make the SensorML description self-contained.

Additionally, the “name” and “description” properties are very similar, so CSDIF explicitly allows the description to be empty. An implementation might not even have a meaningful human-readable name configured for every *Sensor*, but then it should probably generate one anyway (for example based on other properties it has, such as model name), so consumers can just use the name to display the data to a user.

### 2.6.8 SensorThings 2.0

At the time this document was prepared, the OGC is also working on an updated 2.0 version of the SensorThings specification. It is not expected to be finished in time to be usable for CSDIF initially, but it a future revision of CSDIF might switch to using it.

Notable changes that are planned are:

- Description field is made optional everywhere. A description is not always available, especially when data is generated and collected automatically. In the current version, this can already be achieved by allowing the description to be empty.
- *Observation* values are encoded using datastructures from the SWE Common standard (another standard published by the OGC), which are more expressive than the limited set of datatypes supported by SensorThings 1.1. Since SWE Common can encode (among others) datarecords or scalar values, this allows merging *DataStream* and *MultiDataStream* into a single object, which is also simpler by using SWE Common to describe the value type (which integrates the structure, observed properties and units of measurement into a single nested structure).
  In theory, using this SWE Common encoding for values can already be done, but this would break strict compatibility with SensorThings 1.1 (by using more complex representations for values than specified, and omitting the regular observed property and unit of measurement definition).

### 2.6.9 SensorThings requirement classes

The SensorThings API consists of multiple parts, has some extensions and is further divided into requirement classes that contain specific requirements.

The CSDIF specification is based on SensorThings API Part 1: Sensing version 1.1. In section 2 of the standard specification, a number of “requirements classes” are defined to allow partially conforming implementations and extensions. All requirements classes defined there are mandatory except:

- Query options (req/request-data/filter and its dependencies), which defines extensive query and filtering options, but would be challenging to implement when not using an existing SensorThings implementation. If possible, implementations should implement this, but to also allow adding the SensorThings API on top of existing systems, this is optional.
- Creating, updating, and deleting entities (req/create-update-delete), which defines how to modify data. The focus of CSDIF is offering read-only access to data for data exchange. Implementation that want to also support receiving data pushed by other parties are encouraged to implement this requirement, but should then separately advertise, negotiate and authorize such write access.
- Processing multiple requests with a single request (req/batch-request), which defines ways to pack multiple requests in a single request to simplify things for resource-constrained devices. Since the intended uses of CSDIF should have no issues making multiple HTTP requests, this is left out to keep the API simpler.
- Data array extension (req/data-array), which is a more resource-efficient way to encode observations. If possible, this should be implemented, but consumers should be prepared to fall back to accept the basic one-observation-per-JSON-object encoding.
- Sensing MQTT extension (req/create-observations-via-mqtt and req/receive-updates-via-mqtt), which defines ways to publish observations and subscribe to updates to various objects via MQTT. This might be a useful addition to facilitate streaming data exchange, but is not mandatory.

In addition, the Sensor Things API Part 2: Tasking core, which allows sending commands to *Things*, is left out of scope for CSDIF.

## 2.7 Open questions

This section lists some significant questions that are still unanswered, in addition to some of the smaller questions posed in the rest of this document (most of them in the “Other Considerations” section).

Answering these questions requires both more research into the SensorThings and SensorML capabilities, as well as a better view of the requirements.

- Can a single *Sensor* model a complex system? SensorML can support this by defining a system with subcomponents, each with their own properties. Should we forbid this (require splitting into multiple *Sensor* objects which is harder to publish but easier to consume)? Or is it sometimes necessary to express relations between components (for calibration steps, accuracy reduction due to intermediate processing, values derived from multiple sensor values, etc.).
- How to annotate calibrated data? Separate *Datastream*? *Sensor* with SensorML subcomponents that process the output of the sensor?
- Do *Datastreams* (and/or elements of a *MultiDatastream*) need some kind of machine-readable name (could be useful when storing multiple related *Observations* in an object, then each property needs a name). What should the name of a *Datastream* be? Something machine-readable to correspond with a SensorML output name? Should it be unique among a *Thing's Datastream*?
- Can we add per-output metadata (e.g. on accuracy?) in sensorML? These are identified by name, but items in a *MultiDatastream* do not have names? Or do we need to add SensorML metadata to a *Datastream*?
- SensorThings 2.0 uses SWE Common for modeling *Observation* values (and possibly other things), which makes values a bit more expressive (in particular allows datarecords and vectors). Should/could we add a convention to do that here already (keeping in mind compatibility with existing implementations)?
- Can we encode measurement accuracy/precision/granularity in the SensorML metadata? Also for timestamp accuracy?
- Should CSDIF define a (living) list of metadata descriptions for specific values? E.g. for specific sensor modules, define which attributes to use with what values. To prevent e.g. one project from using “Si7021” and another “SI-7021” as a model number?
- Is a SensorThings *Sensor* a specific sensor instance, or a type of sensor? SensorThings is vague about this (see [https://github.com/opengeospatial/sensorthings/issues/203](https://github.com/opengeospatial/sensorthings/issues/203)), RIVM SamenMeten seems to use a *Sensor* as a type of sensor (with many things referencing the same sensor). Downside of the latter is that there is no obvious place to store e.g. calibration info or a serial number for a sensor

instance (though that could be stored as metadata in the “properties” attribute of a datastream?).

- Should we allow SensorML “typeOf” to model is-a relations for *Things* and *Sensors*? This could allow modeling both the “sensor type” and “sensor instance” concepts. But what URL should this point to? Another *Thing* or *Sensor*, or something out-of-band?
- When data is imported into another system (for correlations or aggregations), can we trace the origin of that data? Maybe with custom attributes on the *Datastream* or *Thing*?
- Do we need to define metadata on the *Location* object (it has a “properties” field)? For example “mounted on the (north-facing) wall” vs “mounted on a lamp post” could be seen as metadata on the *Location* instead of the *Thing* itself (which has the side effect of not creating a new version of the *Thing* when just its *Location* and e.g. mounting orientation changes).

## 2.8 Risks

### 2.8.1 Using non-final specifications

The SensorML (JSON) Encoding standard is a fairly new specification, which is still being finalized (January 2025). This might mean some parts of the specification might still be unclear or might change in the near future.

Because of the newness of this specification, there is also not so much tooling support and off-the shelf implementations.

However, since this specification builds on top of a much older (and functionally/semantically equivalent) XML-based SensorML standard, these risks should be minimal.

### 2.8.2 Specification complexity

The SensorThings and SensorML specifications are not simple specifications, consisting of multiple parts, referencing external specifications, being very generic, etc. Even though SensorThings was chosen as a standard that is less complex than alternatives (such as Connected Systems), it is still not simple.

This means a considerable effort is needed to understand and implement these specifications, which might provide a barrier for people to adopt CSDIF.

This risk could be partially mitigated by providing (in a future revision of CSDIF) more detailed explanations and examples combined with pointers to the relevant parts of the specifications that define details for these examples.

Additionally, providing reference implementations that can be used as-is would also be helpful.

### 2.8.3 Privacy leaks

All (community science) initiatives are responsible for their own data, and the protection of privacy sensitive data.

Privacy sensitive data can be split into two categories

1. Personal data (name, address, birthday, telephone number, etc)
2. Behavioral data (measurement location, environment, indoor temperatures, etc)

CSDIF does not intend to support usecases where personal data is exchanged. However, the exchanged data may include behavioral data.

Each initiative is responsible for ensuring that the right measures are taken for the processing of privacy-sensitive data, and the way in which this is shared (or not shared) with other initiatives. The principle is that data is stripped of privacy-sensitive elements before it is shared.

# 3 Existing systems and solutions

## 3.1 Other protocols and systems that we considered

- SenML is a simple JSON/CBOR-based system for encoding observations and units of measurements, but nothing else. Too limited in scope.
- OpenIOT framework is a system for data collection and streaming, but it is not currently maintained.
- CKAN is a system for publishing open data, but it is mostly focused on governments publishing open data, so focuses static and regular datasets without much variation in systems and metadata.
- NetCDF is a generic system to encode data, which does not seem to have a well-defined specification for metadata and is not very easy to work with.

- ODM2 is a datamodel for earth observations that operates in a similar space as the various OGC standards (and also touches some of them). It is a similarly comprehensive standard, that was not investigated well.
- SOSA/SSN by w3c is a RDF specification matching the OGC observations and measurements. It seems limited to just an RDF datamodel, without an API specification and without existing implementations to build on.
- OGC SOS is a predecessor of SensorThings, which occupies a similar space. It is more expressive than SensorThings but uses XML instead of JSON REST API.
- OGC Connected Systems is a successor of SensorThings (being finalized in 2025) which has a bigger feature set and has a more expressive (and a bit more consistent) data model, but is also significantly more complex (both in complexity of the model as well as complexity in the way the specification is written, referencing also many external specifications). This complexity, combined with limited existing tooling support would likely hinder adoption too much.

## 3.2 Existing implementations of SensorThings

### 3.2.1 RIVM Samen Meten

The RIVM (Dutch National Institute for Public Health and the Environment) runs “Samen Meten”, which is a data platform that collects measurement data (primarily aimed at air quality and sound) and publishes this using the SensorThings API v1.0 (unclear what implementation is used).

However, the metadata that is made available is limited. Some observations about their use:

- Information on the sensor used is a bit vague. The “metadata” property points to the same PDF for all *Sensors*, which vaguely defines 10 sensortypes (numbered 1-10). The “description” of the *Sensor* seems to be an index into that list, but a lot of *Sensors* use higher numbers (so maybe the PDF is out of date, or the name of the *Sensor* is the defining characteristic). In any case, it seems only a single sensor type/model attribute is encoded, no room for additional sensor properties.
- Raw output vs calibrated output is done with multiple *Datastreams*. The distinction seems to be encoded in the name of the *Datastream* (“_kal” suffix), but also in the definition property of the related *ObservedProperty*,

which references e.g. a eea-glossary url with a “?calibration=” url parameter that references a textual description of the calibration applied.

- The *Datastream* name is composed of the *Thing* name and the property measured, with an optional suffix for the calibrated version.

https://www.samenmeten.nl/

### 3.2.2 OpenSensorHub

OpenSensorHub (OSH) is a data collection server, written in java, is fully open-source and modular. It contains modules for different OGC APIs that use a shared database, allowing accessing the same data through different APIs (with some limitations).

OSH implements SensorThings v1.0, but the module is disabled by default and the code seems to have been stale for a couple of years, so this might not be the easiest way to start using STA.

https://opensensorhub.org/

### 3.2.3 FROST server

The FRaunhofer Opensource SensorThings (FROST) Server is a data collection server, written in Java. Its primary API is the SensorThings API (v1.1), but it supports (or at least allows) also other data models or access methods using plugins. It is actively maintained and probably an easy way to get started with SensorThings.

https://github.com/FraunhoferIOSB/FROST-Server

# 4 Revision history

- 2025-05-05: Revision 1
  - Initial version.
- 2025-06-26: Revision 2
  - Restructured chapter 2 to be more readable and discuss the data model separate from the access API.
  - Added section “Object names, labels and descriptions” under “Other considerations”.
  - Added section “Specification complexity” under “Risk”.

- Clarified the relation between SensorThings and SensorML in various places.
- Other minor improvements.