Hamburg | Landesbetrieb Geoinformation und Vermessung

UDP über APIs anzapfen

Jeder Datensatz hat mehrere Distributionswege, die von der UDP durch **standardisierte APIs** geebnet werden. Welche Wege ein Datensatz bietet, wird in dessen Metadaten aufgeschlüsselt.

> Meist verlinken die Metadaten auch einen Download als CSV und GeoJSON. Das ermöglicht einen besonders einfachen und schnellen Zugang zu den Daten. Für die *dauerhafte* Einbindung in Anwendungen, ETL-Pipelines oder Analysen empfehlen wir unsere standardisierten APIs.

API-Übersicht

Neben den verlinkten Dokus bieten wir auch einen übergreifenden Quickguide!

API-Typ	Beschreibung	Doku
OAF: OGC API Features	Das Schweizer Taschenmesser für urbane Daten! **REST-API mit JSON** und HTML. Zahlreiche Funktionen zum Filtern; standardisiertes Handling von Zeit und Geometrien. Einfach zu bedienen. Und ziemlich mächtig!	LINK
STA: Sensor Things API	Unsere Nummer Eins für städtische **Echtzeitdaten**! Ist eine Datenquelle via STA erschlossen, könnt ihr einzelne Datenströme via MQTT abonnieren (Push) oder über das REST-Interface Batches anfragen (Pull).	LINK
WMS: Web Mapping Service	Eine bewährte Schnittstelle zur Datenvisualisierung. Rendert Geodaten serverseitig und gibt **Bilddateien** zurück. Bei Bedarf auch Attribute. Eine robuste Alternative zum Rendering von Vektordaten im Client. Bei komplexeren Visualisierungen oft performanter.	TBA
WFS: Web Feature Service	Vorgänger von OAF mit ähnlichem Funktionsumfang. Das RPC-Interface (Remote-Procedure-Call) sowie das Standardformat GML bedürfen jedoch etwas Einarbeitung. Gut zu wissen: **Komplexe (hierarchische) Datenschemata** sind derzeit noch WFS-exklusiv. Z.B. INSPIRE. In der UDP jedoch eher die Ausnahme.	TBA
CSW: Catalogue Service for the Web	RPC-Interface für den **Austausch von Metadaten** nach ISO_19115/ISO_19119. Auch mit DCAT-Katalogen kompatibel. Perspektivisch durch OGC API Records abgelöst (nur das Interface, nicht die Datenmodelle).	TBA

Historisierung und Versionierung

Eine Historisierung ist derzeit über das Transparenzportal umgesetzt. Hierzu werden auf Basis vom WFS statische XML-Dateien (bzw. GML) nach gewissen Events und Abständen gespeichert.

Hingegen liefern die APIs der UDP in der Regel nur den aktuellen Stand der Daten. Das gilt auch für das Datenschema.

> ⚠ Aufgrund steigender Datenbedarfe und der enormen Dynamik bei der Erschließung neuer Daten können sich **Datenschemata gelegentlich ändern**. Eine Versionierung gibt es bisher nicht.

Wir wissen um den Bedarf, möglichst proaktiv zu Schemaänderungen informiert zu werden. Sobald sich eine praktikable Lösung findet, erfahrt ihr es als erstes über unseren Infoservice!

Quickguide

Wer ohne viel Erläuterung direkt ausprobieren möchte, findet hier den geeigneten Startpunkt.

OGC API Features (OAF)

Die meisten Datensätze der UDP sind über unsere OAF-Schnittstelle verfügbar. Jeder Datensatz hat eine Landingpage, die alle wichtigen Ressourcen verlinkt. Essenzielle Abfragemöglichkeiten:

https://api.hamburg.de/datasets/v1/radverkehrsnetz		
`/collections`	Link	Verfügbare Teildatensätze
`/collections/radwege_fahrradstrasse/schema`	Link	Attributnamen und -format
`/collections/radwege_fahrradstrasse/items?f=json`	Link	Datenabfrage
`/collections/radwege_fahrradstrasse/items?f=json&filter=richtung LIKE 'in beide Richtungen'`	Link	Filterabfrage

Hinter OAF verbirgt sich eine Menge Funktionalität. Unser **Tutorial** gibt euch einen vollständigen Überblick und verweist auf Besonderheiten im UDP-Kontext.

SensorThings API (STA)

Über die STA bekommt ihr unsere Echtzeitdaten. Unterstützt wird neben HTTPS auch MQTT! Das grundlegende Datenmodell ist hier erklärt.

https://iot.hamburg.de/v1.1		
`/Datastreams`	Link	Spezifischer Objekt Typ
`/Datastreams(251)`	Link	Spezifisches Objekt
`/Datastreams(251)/Thing`	Link	Zugehöriges Objekt
`/Datastreams?$filter=properties/serviceName eq 'HH_STA_StadtRad'`	Link	Filter nach serviceName
`/Datastreams?$filter=properties/serviceName eq 'HH_STA_StadtRad'&$expand=Observations`	Link	Erweitert auf andere Entitäten

Auch zur STA haben wir ein **Tutorial** erstellt, das euch durch zentrale STA-Konzepte und Spezifika der UDP_HH führt!

Web Feature Service (WFS)

Alle offenen Datensätze der UDP haben im Regelfall einen WFS-Endpunkt. Das ist die Vorgänger-Schnittstelle zu OAF. In seltenen Fällen ist ein Datensatz nur über WFS zugänglich. Essenzielle Abfragemöglichkeiten:

https://geodienste.hamburg.de/HH_WFS_Radverkehrsnetz?Service=WFS&Version=2.0.0		
`&Request=GetCapabilities`	Link	Service-Eigenschaften
`&Request=DescribeFeatureType`	Link	Attributnamen und -format
`&Request=GetFeature&typename=de.hh.up:radwege_fahrradstrasse`	Link	Datenabfrage

Weitere Request-Beispiele hier! (z.B. resulttype=hits, maxFeatures, sortBy, filter)

OAF: OGC API Features

Die meisten Datensätze der Urban Data Platform Hamburg (UDP_HH) könnt ihr bequem und flexibel über OGC API Features (OAF) beziehen.

Hier machen wir euch mit allen wichtigen Funktionen vertraut. Auch wenn ihr die API bereits nutzt, kann sich die Lektüre lohnen! Ihr bekommt viele nützliche Hinweise zur OAF-Nutzung im UDP_HH-Kontext. Die wichtigste Botschaft vorab:

> Diese API erschließt sich **auch ohne Geo-Knowhow**!

Ihr bekommt ein unkompliziertes und langfristig stabiles Interface zum Download urbaner Daten - mit einer *Menge* Funktionalität unter der Haube! Die Spezifikation wurde vom Open Geospatial Consortium (OGC) in Kooperation mit dem W3C erarbeitet:

- handhabbar wie typische REST-APIs
- als HTML navigierbar inkl. interaktiver Doku (OpenAPI + Swagger UI)
- offen, herstellerunabhängig und modular implementierbar

3.1 Paging
3.2 Sortierung
3.3 Einfache Filter
3.4 Textfilter
4 Filtern mit CQL2
5 Handling von Zeit
5.1 Zeitliches Filtern via Datetime
5.2 Zeitliche Operatoren in CQL2
6 Handling von Geometrien
6.1 Räumliches Filtern via Bounding-Box
6.2 Räumliche Operatoren in CQL2
7 Weiterführende Ressourcen

1 API-Struktur

Die Spezifikation sieht vor, dass jeder Datensatz als eigenständige API zur Verfügung steht.

> Die **Ressourcen** der einzelnen APIs sind standardisiert und folglich **identisch strukturiert**.

Für den Großteil der Anleitung reicht daher ein Beispiel - Das *Straßenbaumkataster Hamburg*:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster
- https://api.hamburg.de/datasets/v1/strassenbaumkataster?f=json

Jede Ressource wird in mehreren Formaten repräsentiert: Aktuell HTML (Default) und JSON.

1.1 Landing Page

Der Beispiellink kann als OAF-interne Landingpage zum Datensatz verstanden werden. Dort findet ihr Informationen zum Datenangebot, darunter den räumlichen Abdeckungsbereich, Links zur API-Dokumentation und weiterführende Metadaten.

⚠ Unter https://api.hamburg.de/datasets/v1 gibt es einen Index aller verfügbaren APIs. Dieser ist **nicht gleichzusetzen mit dem Gesamtbestand der UDP_HH**! Bitte stets über die Metadatenkataloge suchen.

1.2 Collections

Ein Datensatz gliedert sich in beliebige Teilmengen: Die Collections. Oft repräsentieren sie unterschiedliche Entitäten. Im Datensatz ALKIS - Liegenschaften und Verwaltungseinheiten gibt es z.B. Flurstücke und Gebäude. Manchmal sind Collections eher als Views zu verstehen, wie in relationalen Datenbanken. Im Straßen- und Wegenetz Hamburg findet ihr neben dem gesamten Straßennetz auch Collections für einzelne Wegearten: Radwege, Fußwege etc. Das Straßenbaumkataster hingegen ist simpel gestrickt: Nur eine Collection, die alle Straßenbäume umfasst:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections
- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster

Jede Collection bietet einen praktischen Überblick: Allem voran die Zahl enthaltener Objekte, ihre räumliche und (falls vorhanden) auch zeitliche Ausdehnung. Auch Beschreibungen können hinterlegt sein.

⚠ Collections **können redundante Daten enthalten**! Im Wegenetz-Beispiel solltet ihr die Rad- und Fußwege lieber aus der Collection für das gesamte Straßennetz filtern. So habt ihr die größtmögliche Flexibilität und minimiert die Zahl angesprochener Endpunkte. Die redundanten Collections bedienen oft Spezialanforderung bei der Datenvisualisierung in bestimmten Anwendungen.

1.3 Items

Das Herzstück der API! Der Pfad führt zu den Daten einer Collection:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items

Als Items werden die Datenobjekte selbst bezeichnet, hier also die einzelnen Bäume. Per Default erhaltet ihr die ersten 10 Items. Praktisch für einen schnellen Blick in die Daten! Die HTML-Ansicht enthält sogar eine Karte. So könnt ihr die neben den Attributen auch die Geometrien inspizieren, ganz ohne Extra-Tools!

Einzelne Items sind auch über einen Pfad erreichbar: Hier der Aufruf eines Straßenbaums über seine ID:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items/226362

⚠ Momentan ist die **Persistenz von IDs nicht gewährleistet**! Diese hängt vom Fachverfahren ab, dem ein Datensatz entspringt.

2 API-Dokumentation

Jeder Datensatz hat eine eigenständige API-Dokumentation:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/api

Habt ihr die Funktionsweise für einen Datensatz durchschaut, lässt sich der Rest analog bedienen. Der primäre Unterschied liegt in den Datensatz-Attributen - diese könnt ihr in euren Anfragen als Parameter nutzen.

ℹ Die Konformitätsressource verrät euch, welche Teile aus welcher OGC-Spezifikation unterstützt werden. Die Konformitätsklasse *http://www.opengis.net/spec/ogcapi-features-1/1.0/conf/core* (eine URI, kein Link!) besagt z.B., dass die Funktionalität aus dem gleichnamigen Baustein der OAF-Spezifikation vorhanden ist. Auch hierbei gilt: Alle Datensätze bieten denselben Funktionsumfang.

2.1 Datenschema

Zu jeder Collection ist ein Schema hinterlegt:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/schema

> Zu dieser Ressource gibt es **momentan keine HTML-Repräsentation**.

Das Dokument folgt der Spezifikation von json-schema.org. Somit bekommt ihr eine standardisierte Quelle zur Einsicht und Validierung der Datenstruktur!

Zu jedem Attribut werden mindestens Datentyp und Titel ausgewiesen. In diesem Beispiel entspricht der Titel allerdings dem technischen Attributnamen. Das ist der Default, wenn keine Titel vergeben sind. Weiterhin können auch Einheiten, Beschreibungen sowie Enumerationen hinterlegt sein. Manche Datensätze reizen diese Möglichkeiten zunehmend aus. Etwa das Schema vom Radverkehrsnetz.

> Viele Datensätze der UDP_HH wurden vor der OAF-Einführung veröffentlicht. Mit OAF jedoch öffnet sich der direkte Datenzugriff für ein weites Publikum. Daher sind wir kontinuierlich dabei, die Dokumentation gemeinsam mit den vielzähligen Dateneigentümern auszubauen. Bitte habt Verständnis, wenn **die meisten Schemaressourcen noch minimalistisch** ausfallen. Bei inhaltlichen Fragen findet ihr fachliche Anlaufstellen im jeweiligen Metadatensatz.

2.2 Sortierbare und filterbare Attribute

Mit der Schemaressource verwandt sind folgende Ressourcen:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/sortables
- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/queryables

Sie listen sortierbare bzw. filterbare Attribute auf. In unserer Umsetzung gibt es diesbezüglich keine Einschränkungen. Alle Attribute einer Collection lassen sich filtern. Auch die Sortierung ist überall freigeschaltet. Nur bei Geometrien ist diese Operation nicht zulässig.

3 Basisfunktionen

Starten wir mit den wichtigsten Queryparametern. Diese sind schnell aufgezählt und selbstverständlich auch im OpenAPI-Dokument auffindbar:

Parameter	Erläuterung	Beispielrequest
f	Antwortformat [html / json]	Link
limit	Steuert die Zahl angefragter Objekte.	Link
offset	Überspringt die erste N Datenobjekte.	Link
crs	Definiert das Koordinatensystem zurückgegebener Geometrien (erklären wir später genauer).	Link
properties	Grenzt die Attribute ein, die im Antwortdokument ausgeliefert werden sollen	Link
skipGeometry	Die Geometrie wird nicht als “property” verstanden. Doch auch sie kann ausgelassen bzw. übersprungen werden.	Link

> Oft bilden die Geometrien das Schwergewicht der Daten. Werden sie mal nicht benötigt, kann **skipGeometry=true die übertragene Datenmenge beträchtlich reduzieren**. Das kommt der Performanz eurer Anwendungen und Analysen zu Gute!

3.1 Paging

Ein Paging lässt sich mit Hilfe der Parameter *limit* und *offset* sowie den Eigenschaften *numberMatched* und *links* im Antwortdokument umsetzen.

⚠ Per Default ist die Antwort auf die ersten 10 Items limitiert. Den **Limit-Parameter** könnt ihr auf **maximal 10.000** hochsetzen. Liegt *numberMatched* höher, ist Paging erforderlich!

3.2 Sortierung

Das funktioniert mit einem oder mehreren Attributen gleichzeitig (Geometrien ausgenommen). Beispielsweise aufsteigend nach Gattung und absteigend nach Pflanzjahr:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?sortby=+gattung_deutsch,-pflanzjahr

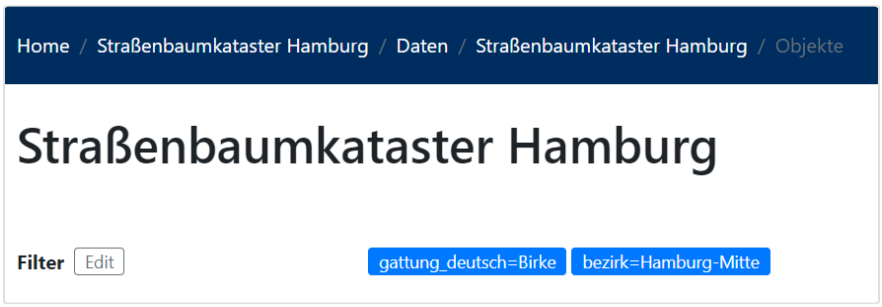
⚠ Einige Funktionen, darunter auch *sortby*, sind noch nicht abschließend standardisiert. In der OpenAPI-Doku findet ihr daher **Hinweise zum Reifegrad** (Maturity). Bei Änderungen informiert euch unser Infoservice.

3.3 Einfache Filter

Mit *einfach* meinen wir *Ist-Gleich* Filter auf Datensatz-Attribute, die ausschließlich *via AND-Operator verknüpft* werden. Nicht anders, als die Handhabung übergreifender Queryparameter. Hier die ersten 10 Bäume (Default-Limit) der Gattung Birke aus dem Bezirk Hamburg-Mitte, beschränkt auf ebendiese Attribute und ohne Geometrie, als JSON:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?gattung_deutsch=Birke&bezirk=Hamburg-Mitte&properties=gattung_deutsch,bezirk&skipGeometry=true&f=json

Im JSON ist einsehbar, wie viele Objekte eurer Anfrage entsprechen (*numberMatched*) und wie viele ihr im Antwortdokument zurückbekommt (*numberReturned*). Einfache Filteranfragen könnt ihr auch fix über die HTML-Ansicht der Items-Ressource zusammenklicken:



3.4 Textfilter

Der Parameter *q* erlaubt die einfache Textsuche über alle Attribute hinweg. Der folgende Request gibt alle Bäume zurück, die in min. einem Attribut des Substring “birke” enthalten:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?q=birke

Im Vergleich liefert der einfache (und case-sensitive) Filter über das Attribut *gattung_deutsch* weniger Treffer:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?gattung_deutsch=Birke

Ein großer Vorteil am Textfilter ist seine Kompaktheit. Explizite Filteranfragen gegen ausgewählte Attribute performen jedoch besser. Außerdem geben sie euch mehr Kontrolle. Folgender, *erweiterter* Filterausdruck sucht z.B. nach dem Substring *birke*, wahlweise im Attribut *gattung_deutsch* oder *sorte_deutsch*. Außerdem könnt ihr hier auch mit Wildcards arbeiten! Die Handhabung solcher Filterausdrücke erläutern wir im nächsten Abschnitt.

- case-sensitiv (Default): `filter=gattung_deutsch LIKE 'Birke' OR sorte_deutsch LIKE 'Birke'`
- mit wildcards: `filter=gattung_deutsch LIKE 'B%rke' OR sorte_deutsch LIKE 'B%rke'`
- case-insensitiv: `filter=CASEI(gattung_deutsch) LIKE CASEI('birke') OR CASEI(sorte_deutsch) LIKE CASEI('BIRKE')`

4 Filtern mit CQL2

OAF bringt eine ganze Abfragesprache mit: Die Common Query Language (CQL2):

- Neben `AND` sind `OR` und `NOT` zulässig - auch ineinander verschachtelt
- Einfache Vergleichsoperatoren (`=`, `<>`, `<`, `>`, `<=`, `>=`) sind an Board
- Ebenso erweiterte Vergleichsoperatoren: `LIKE`, `BETWEEN`, `IN`, `IS NULL`
- Garniert mit mächtigen räumlichen Operatoren (z.B. `S_INTERSECTS`, `S_WITHIN`) sowie zeitlichen (z.B. `T_AFTER`, `T_BEFORE`, `T_INTERSECTS`)

CQL2 sieht sogar Rechenoperationen vor. Das wird von unserer Implementierung vorerst aber nicht unterstützt (s. Konfomitätsressource). Wir fokussieren uns daher auf das Filtern. Erweiterte Filterausdrücke werden über den Parameter *filter* übergeben. Hier eine Illustration (und die entsprechende Abfrage):

```
filter=
stadtteil IN ('Ottensen', 'Othmarschen') AND
(kronendurchmesser BETWEEN 20 AND 25 OR pflanzjahr >= 2020) AND
gattung_deutsch NOT IN ('Eiche', 'Buche', 'Linde')
```

Übrigens gibt es für CQL2 auch ein JSON-Encoding! Hier ein einfaches Beispiel:

```
{
  "op": "=",
  "args": [
    {"property": "gattung_deutsch" },
    "Birke"
  ]
}
```

Mit dem Parameter *filter-lang=cql-json* funktioniert das auch im GET-Request:

```
https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?f=json&filter-lang
```

Allerdings ist das umständlicher als das Textencoding. Letzteres ist daher der Default und wird in der OpenAPI-Doku auch empfohlen. Praktikabler wäre es, JSON-Bodies via POST-Request mitzugeben, allerdings:

> ⚠ Zur Datenabfrage ist momentan **nur die GET-Methode zulässig**! Wenn euch das cql-json Encoding im Verbund mit POST-Requests interessiert, kommt gerne auf uns zu!

Einen umfassenderen Eindruck von *cql-json* könnt ihr euch hier verschaffen. Die Ressource bezieht sich auf eine andere (mit OAF verwandte) API-Spezifikation, die ebenfalls CQL2 nutzt.

5 Handling von Zeit

Grundvoraussetzung ist der Datentyp *date* - das Schema der Straßenbäume listet jedoch kein entsprechendes Attribut. Das Pflanzjahr käme in Frage, ist jedoch als *integer* definiert. Diese Festlegung obliegt grundsätzlich den Datenhalter:Innen.

Daher ein alternatives Beispiel: Regionalstatistische Daten der Stadtteile Hamburgs. Der Datensatz enthält nur eine Collection, das Schema dazu listet drei zeitliche Attribute: *jahr*, *jahr_date* und *jahr_timestamp*. Davon hat nur *jahr_date* auch den Datentyp *date*.

> ⚠ Grundsätzlich können Datensätze **beliebig viele Attribute mit Zeitbezug** enthalten. Wie im obigen Beispiel sind die Informationen **manchmal redundant**, um unterschiedliche Formate oder Datentypen für diverse Clients/Anwendungsfälle abzudecken.

5.1 Zeitliches Filtern via Datetime

Im Schema findet sich unter *jahr_date* die Angabe *"x-ogc-role": "primary-instant"*. Sie markiert das Attribut als primären Zeitindex. Nicht alle Datensätze haben einen. Ist der Index vorhanden, seht ihr bereits auf der Übersichtsseite zur Collection den Minimal- und Maximalwert (*Zeitlicher Bereich*). Den Zeitindex könnt ihr dann über den Parameter *datetime* ansprechen:

- https://api.hamburg.de/datasets/v1/regionalstatistische_daten_stadtteile/collections/regionalstatistische_daten_stadtteile/items?datetime=2013-12-31T00:00:00Z

Unabhängig vom Zeitformat in den Quelldaten greift im Filter die Date/Time Syntax nach RFC 3339 (Sektion 5.6). Begrenzte sowie einseitig unbegrenzte Intervalle sind ebenfalls möglich (weitere Details in Open-API Doku und in der Spezifikation):

- Begrenztes Intervall: https://api.hamburg.de/datasets/v1/regionalstatistische_daten_stadtteile/collections/regionalstatistische_daten_stadtteile/items?datetime=2013-12-31T00:00:00Z/2015-12-31T00:00:00Z
- Einseitig begrenztes Intervall: https://api.hamburg.de/datasets/v1/regionalstatistische_daten_stadtteile/collections/regionalstatistische_daten_stadtteile/items?datetime=2022-12-31T00:00:00Z/..

> Interessieren euch nur die Statistiken, nicht die Geometrien, sei **nochmals auf skipGeometry=true verwiesen**. Im obigen Beispiel verringert sich die übertragene Datenmenge um ca. 90%!

5.2 Zeitliche Operatoren in CQL2

Unabhängig vom Zeitindex könnt ihr stets auf generische Zeitfilter in CQL2 zurückgreifen. Das zu filternde Zeitattribut wird dabei direkt in der Query adressiert. Euch stehen Daten oder Zeitstempel zur Verfügung:

Beispielanfrage	CQL2 Filterausdruck
Nach einem Datum/Zeitstempel	`filter=T_AFTER(jahr_date, DATE('2018-12-31'))`
Vor einem Datum/Zeitstempel	`filter=T_BEFORE(jahr_date, TIMESTAMP('2018-12-31T00:00:00Z'))`
Im Zeitintervall t, j	`filter=T_INTERSECTS(jahr_date, INTERVAL('2018-12-31','2020-12-31'))`
Außerhalb vom Zeitintervall t,j	`filter=T_DISJOINT(jahr_date, INTERVAL('2018-12-31T00:00:00Z','2020-12-31T00:00:00Z'))`

6 Handling von Geometrien

Die API-Pfade */items* und */item* liefern bei näherer Betrachtung kein generisches JSON, sondern GeoJSON. Dieser Community-Standard greift die Simple Feature Access Spezifikation der OGC auf. Sie definiert grundlegende Geometrietypen: `Point`, `LineString`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon`, `GeometryCollection` (übergeordnete Sammlung verschiedener Typen). Was es mit dem Begriff *Features* auf sich hat, wird hier erklärt.

Das Format ist simpel gestrickt und macht Geodaten sehr zugänglich. Die enthaltenen Geometrietypen können eine Vielzahl von Anwendungsfällen abdecken.

> GeoJSON schreibt WGS 84 als Koordinatensystem vor. In OAF ist das der Default. Zudem können auch **andere Koordinatensysteme** explizit angefragt werden (konform zur Spezifikation).

Wichtig zu wissen: *Das* Koordinatensystem für alle Anwendungsfälle gibt es nicht! Schließlich existiert keine verlustfreie Methode, den Globus auf zwei Dimensionen zu projizieren. Manche Methoden optimieren die originalgetreue Abbildung von Distanzen, andere von

Flächen - alle jedoch produzieren räumliche Verzerrungen.

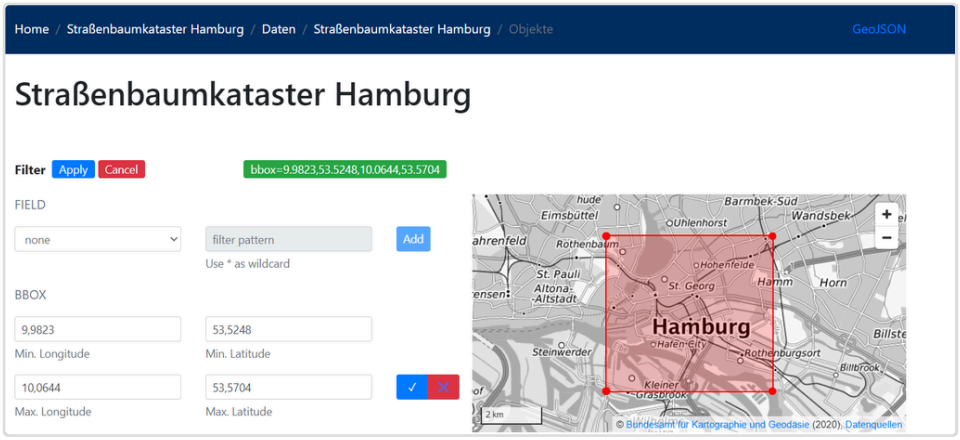
Für verschiedene Anwendungsfälle gibt es also besser und schlechter geeignete Projektionen. Mittels EPSG-Codes werden sie international einheitlich benannt. Derzeit haben wir folgende Optionen konfiguriert:

- EPSG:3857 - Aus Google Maps oder OpenStreetMap bekannt
- EPSG:4326 - Bekannt u.a. von GPS und äquivalent zu WGS84 (nur Latitude und Longitude vertauscht)
- EPSG:25832 - Standard in der Hamburger Landesvermessung

Als Default gilt WGS84. Statt eines EPSG-Codes ist im OGC-Kontext die Bezeichnung *CRS:84* geläufig.

6.1 Räumliches Filtern via Bounding-Box

Der einfachste Weg zur Definition räumlicher Ausschnitte ist die Bounding-Box (*bbox*). Inkludiert wird alles, was in diesem Rechteck liegt oder von den Rändern geschnitten wird. Über das interaktive Filterpanel einer Items-Ressource könnt ihr euch ganz intuitiv rantasten. Dazu kehren wir zu den Straßenbäumen zurück:



Bestätigt via Apply-Button und der Browser führt einen entsprechenden Request aus:

- https://api.hamburg.de/datasets/v1/strassenbaumkataster/collections/strassenbaumkataster/items?bbox=9.9823,53.524,10.0644,53.5704

Auf diese Weise kann z.B. ein Kartenclient die nötigen Datenobjekte (und Hintergrundkacheln) aufrufen, während ihr durch eine Karte navigiert.

> In der HTML-Ansicht wird das Default-Koordinatensystem des Servers genutzt. Im Request lässt sich die Bounding-Box jedoch unabhängig vom Koordinatensystem im Antwortdokument steuern. Siehe dazu die Parameter **crs vs. bbox-crs** in der OpenAPI-Doku.

6.2 Räumliche Operatoren in CQL2

Das Filtern via Bounding-Box ist nur die Spitze des Eisbergs! CQL2 bietet umfassende Möglichkeiten für räumliche Abfragen. Oftmals ist die Überschneidung zweier Geometrien von Interesse. Dazu müssen sie min. eine Koordinate gemeinsam haben. Der zugehörige Operator lautet `S_INTERSECTS`. Das logische Gegenteil lautet `S_DISJOINT`. Die Beziehung zwischen Geometrien lässt sich aber noch präziser abfragen. In diesem Abschnitt der CQL2-Spezifikation gibt es eine praktische Illustration dazu. Die wesentlichen Operatoren lauten:

- berühren (`S_TOUCHES`)
- kreuzen (`S_CROSSES`)
- überlappen (`S_OVERLAPS`)
- gegenseitig enthalten (`S_WITHIN` oder `S_CONTAINS` , je nach Perspektive)
- gleichen (`S_EQUALS`)

Zur Illustration begeben wir uns in das beliebte Portugiesenviertel. Über das Hamburger Geoportal rufen wir eine Luftbildkarte auf und überlagern sie mit den Straßenbäumen und Gebäudeumrissen (zur Ansicht). Nun stellen wir uns auf die Kreuzung der Dittmar-Koel- und Reimansstraße. Wie viele Straßenbäume befinden sich in einem Umkreis von 50 Metern?

Mit dem Zeichnen-Werkzeug und dessen CSV-Export generieren wir den Umkreis als Well-Known-Text (WKT) Dieses Encoding für grundlegende Geometrietypen kommt auch in CQL2 zum Einsatz! Weiterhin enthält die CSV das verwendete Koordinatensystem - EPSG:25832. Dieses müssen wir dem Filter als Parameter mitgeben, damit Input- und Quellkoordinaten richtig abgeglichen werden. Das probieren wir nun mit den Straßenbäumen sowie den Gebäudeumrissen aus.

Beispielanfrage	CQL2 Filterausdruck
Straßenbäume: S_INTERSECTS	`filter-crs=http://www.opengis.net/def/crs/EPSG/0/25832` `&filter=S_INTERSECTS(geom, POLYGON((564662.0463822997 5933443.1347707`
Straßenbäume: S_WITHIN	`filter-crs=http://www.opengis.net/def/crs/EPSG/0/25832` `&filter=S_WITHIN(geom, POLYGON((564662.0463822997 5933443.134770782,5`
Gebäudeumrisse: S_INTERSECTS	`filter-crs=http://www.opengis.net/def/crs/EPSG/0/25832` `&filter=S_INTERSECTS(geometrie, POLYGON((564662.0463822997 5933443.13`
Gebäudeumrisse: S_WITHIN	`filter-crs=http://www.opengis.net/def/crs/EPSG/0/25832` `&filter=S_WITHIN(geometrie, POLYGON((564662.0463822997 5933443.134770`

Zur Illustration vergleichen wir `S_INTERSECTS` mit `S_WITHIN`. Bei den Straßenbäumen macht das keinen Unterschied. Sie werden zwar als Kreise visualisiert, wurden jedoch als Punkte erfasst. Ein Punkt hat keine Fläche und kann daher nur in oder außerhalb eines Umkreises (bzw. Polygons) liegen. Anders bei den Gebäudeumrissen: Die Anfrage mit `S_WITHIN` filtert nur Gebäude, die sich *vollständig* im Umkreis befinden.

Den **Namen des Geometrie-Attributs** findet ihr **nur in den Schema-Ressourcen** (Straßenbäume; Gebäudeumrisse). Oft lautet er *geom, geometry oder geometrie*. Doch nicht alle Datensätze folgen dieser Konvention!

Der **Detailgrad in CQL2 genutzter Geometrien** - die Anzahl ihrer Koordinaten und Nachkommastellen - ist durch die serverseitige Zeichenbegrenzung eingehender Anfragen **limitiert**. Meldet euch, wenn euere Anwendungsfälle mehr Spielraum brauchen!

7 Weiterführende Ressourcen

Mehr konzeptionellen Hintergrund zu OAF findet ihr im offiziellen OGC-Tutorial.

STA: SensorThings API

Die **SensorThings API** (STA) ist ein offener und internationaler Standard des Open Geospatial Consortiums (OGC). Er ermöglicht die interoperable Bereitstellung von Echtzeitdaten, insbesondere im IoT-Bereich (Internet of things). Über die STA stellt auch die Urban Data Platform Hamburg (UDP_HH) ihre Echtzeit-Datensätze zur Verfügung.

Im Folgenden wollen wir euch mit den wichtigsten Funktionen vertraut machen. Dabei gehen wir auch auf Spezifika unserer Plattform ein. Die zentralen Vorzüge der STA vorab:

- **REST-API Struktur**: Einfach zu bedienen wie andere Web-APIs.
- **Datenformate**: JSON, GeoJSON und CSV
- **Standardisiertes Datenmodell**: Ob Ladesäule oder Feinstaub-Messstation, ihr bekommt ein einheitliches Interface!
- **Protokolle**: Unterstützt neben HTTPs auch das Abonnement von Echtzeitdaten via MQTT.
- **Herstellerunabhängig und modular implementierbar**: eine sinnvolle Ergänzung für jede Datenplattform, die mit Echtzeitdaten hantiert!

3 CSV-Download
4 Paging
5 Einbindung via MQTT
6 Einbindung in Geoinformationssysteme
7 Weiterführende Ressourcen

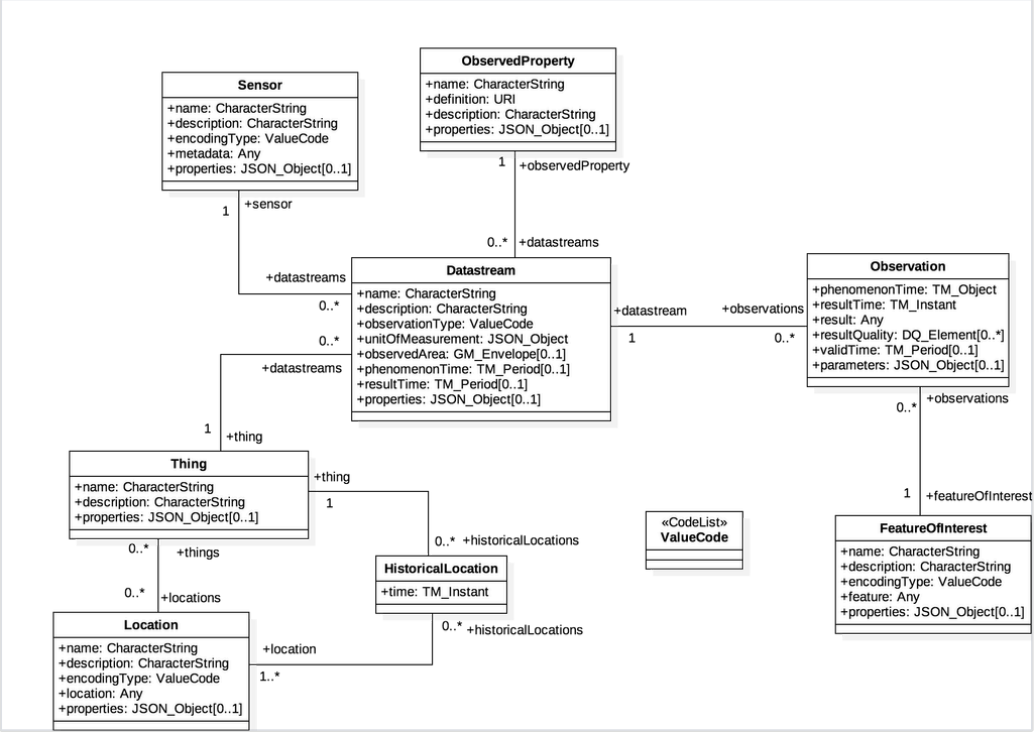
1 API-Struktur

Die STA gliedert sich in mehrere Endpunkte zu folgenden Entitäten:

- **Thing**: Ein physisches oder virtuelles Objekt.
- **Location**: aktueller Standort eines Thing.
- **Datastream**: Repräsentiert eine Gruppe von Messungen mit dem gleichen Sensor und der gleichen ObservedProperty.
- **Sensor**: Ein reales oder gedachtes Gerät, das Beobachtungen oder Messwerte erfasst.
- **ObservedProperty**: Die Größe (physikalisch oder kategorisch), die beobachten bzw. gemessen wird.
- **FeaturesOfInterest:** Der konkrete Ort und der konkrete Zeitpunkt einer Observation
- **Observation**: Eine Beobachtung oder ein Messwert.

1.2 Datenmodell

Die Eigenschaften und Beziehungen der Entitäten werden in diesem Schaubild zusammengefasst (Quelle):



1.2 Beispiel: Elektroladestationen in Hamburg

Vielleicht seid ihr schonmal auf die Elektroladestationen im Hamburger Geoportal gestoßen. Ein gutes Beispiel, um den generischen Aufbau der STA zu veranschaulichen.

Etwas Kontext vorab: Eine Elektroladesäule kann 1 bis 3 Ladepunkte besitzen. Für jeden Ladepunkt wird die Verfügbarkeit gesondert beobachtet und übertragen. Ihr könntet nun beispielsweise ermitteln, ob an einer bestimmten Ladesäule noch ein Platz zum Laden frei ist.

Endpunkt	Beispielaufruf	Erläuterung und weitere Beispiele	Beziehung zu anderen Entitäten
`/Things`	Elektroladesäule	Ein Thing repräsentiert ein physisches oder virtuelles Objekt *(z.B. eine Fahrradzählstelle oder Messstelle für Feinstaubkonzentration)* Es kann einen oder mehrere Sensoren beherbergen und einen konkreten Standort haben.	Kann eine oder mehrere *Locations* (Standorte) besitzen. Ist in der Regel mit einem Datastream verbunden.
`/Locations`	Standort einer Elektroladesäule mit geografischen Koordinaten	Die Location beschreibt den aktuellen physischen Standort eines Thing *(z.B. Geo-Koordinaten einer Feinstaub-Messstelle, eines fahrenden Autos, der Standort eines Satellits)* Ist das Thing beweglich, ändert sich die Location mit der Zeit.	Die Location *kann* mit einem Thing verknüpft sein. Es können HistoricalLocations existieren.
`/HistoricalLocations`	Vergangene Standorte einer Ladesäule	Gibt historische Standortinformationen eines Thing aus, wenn sich dessen Position über die Zeit ändert. *(z.B. vergangene Standorte eines Satellits oder Messfahrzeugs)* Bei der exemplarischen Ladesäule und sonstigen unbeweglichen Things ist die Angabe i.d.R. identisch zur aktuellen Location.	Verknüpft mit einem *Thing* und mindestens einer *Location*.

`/Datastreams`	Anschlüsse einer Elektroladesäule	Ein Datastream repräsentiert eine Gruppe von Messwerten, die vom gleichen Sensor erfasst werden und die das gleiche ObservedProperty messen bzw. beobachten *(z.B. ein Datastream einer KFZ-Zählstelle für PKWs und ein weiterer für LKWs)*	Ein Datastream hat immer genau ein *Thing*, einen *Sensor* und eine *ObservedProperty*. Ein Datastream repräsentiert eine Gruppe von Observations.
`/Sensors`	Verfügbarkeitssensor eines Anschlusses an einer Elektroladesäule	Ein Sensor kann ein real existierendes Gerät sein *(z.B. ein optischer Partikelsensor zur Feinstaubmessung, ein Temperatursensor, eine Infrarotkamera)* oder ein “Gedachtes”, wie der Verfügbarkeitssensor eines Anschlusses an einer Elektroladesäule. Die Verfügbarkeit wird hierbei nur indirekt ermittelt, u.a. über den Stromfluss.	Verknüpft mit keinem, einem oder mehreren *Datastream(s)*.
`/ObservedProperties`	Verfügbarkeit eines Ladepunktes an einer E-Ladestation	Die *ObservedProperty* beschreibt, welche physikalische oder kategorische Größe bzw. welches Phänomen erfasst wird *(z.B. die Anzahl von Verkehrsteilnehmern an einem Zählfeld, die Verfügbarkeit von Fahrrädern an einer Ausleihstation, die Konzentration von Feinstaubpartikeln an einem Straßenquerschnitt)*	Kann mit keinem oder einem, oder mehreren *Datastream(s)* verknüpft sein
`/Observations`	aktuelle und historische Verfügbarkeit eines Ladepunktes an einer E-Ladesäule	*Eine Observation* repräsentiert einen einzelnen Messwert, der von einem Sensor zu einem bestimmten Zeitpunkt, an einem bestimmten Ort erfasst wurde und zu einer Gruppe von Messwerten gehört *(z.B. die Stromspannung, Feinstaubpartikel-Konzentration oder Anzahl der passierenden KFZ)*	Eine Observation ist verknüpft mit genau einem Datastream und mit genau einem FeatureOfInterest
`/FeaturesOfInterest`	Standorte der Beobachtungen an den Ladepunkten	Da Things beweglich sein können, wird jede Messung stets mit ihrem Entstehungsort und dem Erfassungszeitpunkt gespeichert *(z.B. ein fahrendes Auto mit Temperatur-Sensor für die Oberflächentemperatur der Straße mit Messungen im 10-Minuten Intervall)* Bei Things mit festem Standort entspricht das FeatureOfInterest einer Observation stets der aktuellen Location des Things.	Ein FeatureOfInterest *kann* mit einer Observation verknüpft sein.

2 Hamburger Echtzeitdaten in Aktion

Fast alle Echtzeitdatensätze der UDP_HH sind über unter https://iot.hamburg.de abrufbar. Eine Ausnahme bilden die Ampeldaten. Aufgrund ihres Umfangs gibt es einen separaten Endpunkt: https://tld.iot.hamburg.de.

> Beide Endpunkte geben auch ein **Open API** Dokument aus (vgl. z.B. https://iot.hamburg.de/v1.1/api). Ihr könnt das u.a. im Swagger Editor öffnen und alle Funktionen darüber ausprobieren.

Zur Hamburger STA gibt es einen zentralen Metadateneintrag. Neben einer generischen API-Beschreibung findet ihr dort eine Auflistung verfügbarer Echtzeitdatensätze. Diese werden auch im Hamburger Geoportal visualisiert:

- HAW-Energie Campus Daten

- Elektroladestandorte Hamburg
- Stadtrad-Stationen Hamburg
- Verkehrsdaten Rad (Infrarotdetektoren) Hamburg
- Verkehrsdaten KFZ (Infrarotdetektoren) Hamburg
- Traffic Lights Data Hamburg

2.1 Zuordnung von Datenströmen zu Datensätzen

Das Konzept “Datensatz” ist im STA Datenmodell nicht vorgesehen. Die Zuordnung haben wir dennoch in der API hinterlegt. Vorab ist wichtig zu wissen:

> Jede Entität besitzt ein generisches Attribut Namens `properties` (vgl. Datenmodell-Schaubild). Dies ist ein Platzhalter für beliebige Schlüssel-Wertpaare, die über das standardisierte Datenmodell hinausgehen.

Angenommen, ihr stoßt in unseren Metadatenkatalogen auf die Elektroladestandorte. Die Datensatzbeschreibung verweist bzgl. STA auf den Schlüssel `serviceName` mit dem Wert "HH_STA_E-Ladestationen". Dieses Schlüssel-Wertpaar findet ihr in besagten `properties` wieder. Und zwar bei den Datastreams. Sie sind die führenden Objekte, sobald die zugehörigen Messdaten (Observations) und damit das “Herzstück” der STA abgerufen werden. Die UDP_HH gliedert ihre STA-Endpunkte also konzeptionell in mehrere Services. Ein “STA-Service” gruppiert alle Datenströme, die einem spezifischen Datensatz zugeordnet sind.

- https://iot.hamburg.de/v1.1/Things(8490)/Datastreams

<details>
<summary>Auszug aus dem Antwortdokument zu obiger Anfrage</summary>


```
{
  "@iot.id": 18942,
  "name": "Ladepunkt DE*HHM*E1011*01",
  "description": "Availability status of connector at a charge-station",
  "observationType": "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CategoryObservation",
  "unitOfMeasurement": {...},
  "observedArea": {...},
  "phenomenonTime": "2024-07-18T13:55:07.024Z/2024-10-14T05:43:57.9Z",
  "properties": {
    "topic": "Transport und Verkehr",
    "current": "AC",
    "metadata": "https://registry.gdi-de.org/id/de.hh/84b8b24e-300d-48d3-a711-aadc472f886d",
    "layerName": "Status_E-Ladepunkt",
    "ownerData": "Freie und Hansestadt Hamburg",
    "serviceName": "HH_STA_E-Ladestationen",
    "resultNature": "primary",
    "connectorType": "S_TYPE_2",
    "infoLastUpdate": "2024-07-18T13:55:07.094716Z",
    "mediaMonitored": null,
    "chargingProtocol": "MODE3"
  },
  "resultTime": "2024-07-18T13:55:07.094832Z/2024-10-14T05:43:57.964478Z"
}
```


</details>

2.2 Pfade und URL-Parameter

Neben den Entitäten selbst lassen sich auch ihre Beziehungen als URL-Pfad aufrufen:

Beispiel	Erläuterung
https://iot.hamburg.de/v1.1/Things(8490)/Locations	Alle Locations, die zu einem Thing gehören

https://iot.hamburg.de/v1.1/Locations(26520331)/Things	Alle Things, die sich eine Location teilen

Darüber hinaus gibt es zahlreiche Query-Parameter zum Filtern, Sortieren oder Erweitern der Ergebnismenge. Die Schlüssel-Werte-Paare werden wie folgt mit dem Pfad verknüpft:

`/pfad?parameter1=wert1&parameter2=wert2`.

Hier ein paar gängige Beispiele:

Beispiel	Erläuterung
`$count=true`	Anzahl der Ergebnisse für eine bestimmte Abfrage (z.B. Anzahl der Things)
`$skip=10`	Die ersten N Ergebnisse überspringen
`$top=100`	N Ergebnisse zurückgeben (Maximal 1000 pro Request)
`$select=name,description`	Eingrenzung der Ergebnisse auf einzelne Attribute
`$orderby=phenomenonTime+desc`	Ab- oder aufsteigendes Sortieren bspw. nach einem Attribut.
`$expand=Datastreams,Locations`	Erweiterung einer Entität um verknüpfte Entitäten (hier z.B. die verbundenen Datastreams zu einem Thing)
`$filter=name+eq+'Fahrradaufkommen an Verkehrszählstelle 0109932 im 15-Min-Intervall'`	Filtern der Ergebnisse - Die STA unterstützt vielfältige Filteroperationen!

2.3 Abfragemöglichkeiten an einem Beispiel

Alle Ladestationen bzw. Things aus dem Datensatz “H_STA_E-Ladestationen“ lassen sich wie folgt abfragen (Paging beachten) :

- https://iot.hamburg.de/v1.1/Things?$filter=Datastreams/properties/serviceName+eq+'HH_STA_E-Ladestationen'&$count=true

Nun greifen wir eine Ladesäule heraus und reduzieren die Datenmenge durch Selektion relevanter Attribute:

- https://iot.hamburg.de/v1.1/Things(8490)/Datastreams?$select=name,id,properties

Die jüngsten 3 Observations einer Ladestation bekommen wir mit folgender Abfrage. Dabei lassen wir die Metadaten weg und verschlanken so das Antwortdokument:

- https://iot.hamburg.de/v1.1/Things(8490)/Datastreams/Observations?$top=3&$orderby=phenomenonTime+desc&$resultMetadata=off

⚠ Alle Zeitstempel sind in UTC!

Zu welchem Ladepunkt gehören die jüngsten 3 Observations an der ausgewählten Ladestation? Hierzu erweitern wir obige Abfrage mit `$expand=Datastream` und `$select=name`:

- https://iot.hamburg.de/v1.1/Things(8490)/Datastreams/Observations?$top=3&$orderby=phenomenonTime+desc&$resultMetadata=off&$expand=Datastream($select=name)

Wie ihr seht, lässt sich die API flexibel bedienen. Für einen vollständigen Überblick über eure Interaktionsmöglichkeiten empfehlen wir einen Blick in die Spezifikation.

3 CSV-Download

Mit dem Parameter `$format=csv` liefert unsere STA auch CSV-Dateien aus.

⚠ Das hierarchische Datenmodell der STA lässt sich nicht vollständig als CSV abbilden. Das Ausgabeformat ist nur zulässig, wenn die Anzahl verknüpfter Entitäten in eurer Anfrage maximal 1 beträgt.

Das folgende Beispiel liefert eine entsprechende Fehlermeldung, da es immer mehrere Datastreams zu einem Thing gibt, aber nur eines für die Erstellung einer CSV-Ausgabe abgerufen werden kann:

- https://iot.hamburg.de/v1.1/Things?$expand=Datastreams&$format=csv

Erst durch Reduktion der Datastreams auf eines pro Thing (via `$top`-Parameter) bekommt ihr eine valide CSV zurück:

- https://iot.hamburg.de/v1.1/Things?$expand=Datastreams($top=1)&$format=csv

4 Paging

Im folgenden Beispiel werden die Positionen der E-Ladestandorte über die STA der UDP_HH abgefragt:

- https://iot.hamburg.de/v1.1/Things?$filter=Datastreams/properties/serviceName+eq+'HH_STA_E-Ladestationen'&$resultMetadata=none&$count=true

Per Default werden maximal 100 Entitäten zurückgegeben. Ist die Ergebnismenge größer, greift das Paging. Der Link zur jeweils nächsten Ergebnisseite wird dann im Antwortdokument unter dem Schlüssel `@iot.nextLink` ausgegeben.

⚠ Achtung: Wird mit Paging gearbeitet, sollte immer der Parameter `$orderby=ID` angefügt werden! **Andernfalls können fehlende Entitäten oder Duplikate auftreten!**

Und so ließe sich der Download aller E-Ladesäulen (Things) mit Paging in Python umsetzen:

Beispielcode

```python
import requests
import json

# define query
next_data_url = "https://iot.hamburg.de/v1.1/Things?$orderby=id&$resultMetadata=none"

# list to be filled with downloaded things while looping through pages
all_things = []

# loop whie next_data_url is assigned a truthy value:
while next_data_url:
    response = requests.get(next_data_url)
    try:
      json_data = response_frost.json()
    except json.JSONDecodeError as e:
      print("Not a json: ", e) # in case an error occurs on the server side or the URL is wrong

    # extract things from dictionary
    things = json_data.get("values")

    # append the things to list of all things
    all_things.append(things)

    # get next_data_url - assigns None if no @iot.nextLink exists
    next_data_url = json_data.get("@iot.nextLink")
```

5 Einbindung via MQTT

Wie eingangs erwähnt unterstützt unsere STA auch das MQTT-Protokoll. Es dient der zeitnahen Bereitstellung von Sensordaten. Damit könnt ihr Abonnements auf sogenannte “Topics” abschließen und bekommt so immer die neuesten Informationen, sobald sie verfügbar sind.

Im nachfolgenden Beispiel abonnieren wir das Primärsignal einer exemplarischen Ampel für eine einzelne KFZ-Fahrspur. Weil die Schaltzeit ein bis zwei Minuten dauern kann, ist etwas Geduld gefragt. Vorab könnt ihr prüfen, ob die ausgewählte Ampel gerade überhaupt Signale liefert:

- https://tld.iot.hamburg.de/v1.1/Datastreams(56859)/Observations?$orderby=id+desc

Wollt ihr gleich mehrere hundert Ampeln abonnieren, habt ihr es schnell mit Datenraten von mehreren hundert Observations pro Sekunde zu tun. Da beim MQTT-Protokoll der Client die Geschwindigkeit steuert, mit der er aktuelle Daten erhält, will sichergestellt sein, dass alle Daten so schnell wie möglich verarbeitet werden. Mit einem asynchronen MQTT-Client seid ihr grundsätzlich besser gerüstet. Im exemplarischen Beispielcode nutzen wir zu diesem Zweck aiomqtt. Wer mit asynchroner Programmierung in Python noch keine Übung hat, findet hier einen guten Einstieg.

Beispielcode

```python
import asyncio
import aiomqtt

# create an async function
async def main():
  # use an async context manager to create a Client object
  async with aiomqtt.Client("tld.iot.hamburg.de") as client:
      # subscribe to a topic
      await client.subscribe("v1.1/Datastreams(56859)/Observations")
      # print the received data
      async for message in client.messages:
          print(message.payload)

# the sync function
asyncio.run(main())

```

Zum Ausprobieren kann auch ein freier MQTT-Client mit **Benutzeroberfläche** verwendet werden, wie z.B. MQTTX: Your All-in-one MQTT Client Toolbox.

6 Einbindung in Geoinformationssysteme

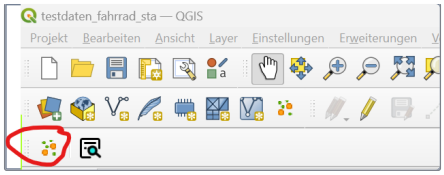
Die Echtzeitdaten der UDP_HH können via STA auch direkt in Geoinformationssystemen (GIS) eingebunden werden. Für QGIS braucht es hierzu eine Erweiterung. Hierzu haben wir eine Kurzanleitung vorbereitet:

Schritt 1: Erweiterung installieren

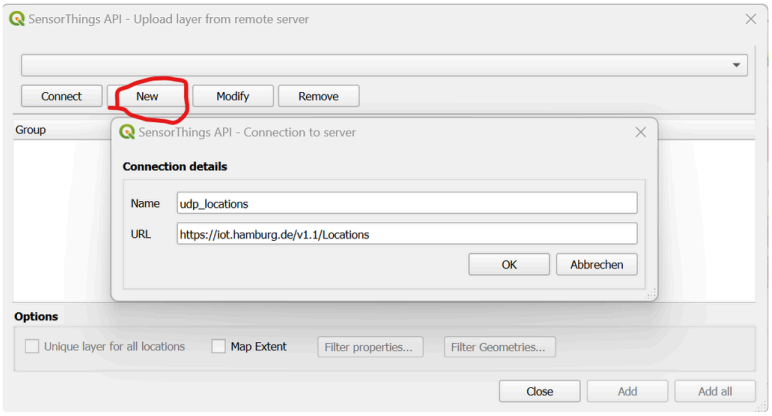
Zunächst muss die STA-Erweiterung in QGIS installiert werden. Dies funktioniert momentan nicht über den Erweiterungs-Katalog in QGIS sondern muss hier heruntergeladen werden. Zur Installation in QGIS auf “Erweiterungen” → “Erweiterungen verwalten und installieren” gehen und dann im Erweiterungs-Menü auf “aus ZIP installieren” klicken.

Schritt 2: Verbindung mit der STA herstellen

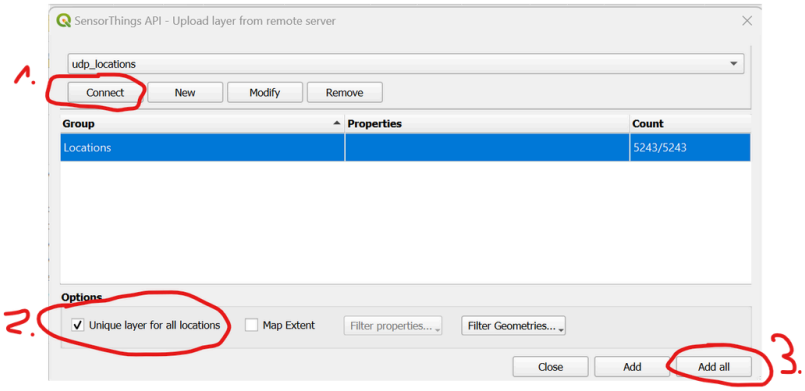
Nach der Installation auf “Upload Layer from remote Sever” anklicken.



In dem Popup-Fenster auf “New” klicken. In der Maske den Gewünschten Namen und die **URL zu den Locations** eintragen (https://iot.hamburg.de/v1.1/Locations). Mit “Ok” bestätigen.



Nun auf “Connect” klicken (1), woraufhin sich eine Liste mit den allen Locations öffnet. Mit “Unique layer for all locations” (2) werden alle Locations in einem Layer zusammengefasst. Mit “Add all” (3) wird der Layer mit den Locations dem QGIS-Projekt hinzugefügt.



⚠ Es ist derzeit leider **nicht möglich**, **vorab gefilterte Locations einzubinden**. Die Erweiterung ist auf direkte Abfragen der Entitäten ausgelegt (also nur /Locations). Die Locations **können in QGIS aber über die Attributtabelle nach den Namen gefiltert** werden. Alternativ könnt ihr den benötigten Datenausschnitt z.B. mit Python herunterladen und einfach als GeoJSON in QGIS bearbeiten.

7 Weiterführende Ressourcen

Weitere Tutorials/Handbücher zur Vertiefung.

SensorThings API — OGC e-Learning 2.0.0 documentation

OGC SensorThings API Documentation | SensorUp OGC SensorThings API Developer Centre

https://wiki.gdi-de.org/pages/viewpage.action?pageId=786464774

Dokumentation zum Frost-Server (STA-Implementierung) mit Abfragebeispielen

Impressum