
pygeoapi Documentation

Release 0.9.dev0

pygeoapi team

2020-11-05

TABLE OF CONTENTS

1	Introduction	3
1.1	Features	3
1.2	Standards Support	3
2	How pygeoapi works	5
3	Install	7
3.1	Requirements and dependencies	7
3.2	For developers and the truly impatient	7
3.3	pip	7
3.4	Docker	8
3.5	Conda	8
3.6	UbuntuGIS	8
3.7	FreeBSD	8
3.8	Summary	8
4	Configuration	9
4.1	Reference	9
4.1.1	server	9
4.1.2	logging	10
4.1.3	metadata	10
4.1.4	resources	11
4.2	Using environment variables	12
4.3	Linked Data	13
4.4	CQL Filter	14
4.5	Summary	14
5	CQL Filter Implementation	15
5.1	CQL Filter Predicates	15
5.2	CQL Filter Implementation for Data Providers	15
5.3	Steps to generate and execute CQL endpoints	16
5.4	Examples of CQL query filter	21
5.4.1	Getting started	21
5.4.2	Simple comparisons	23
5.4.3	String comparisons	28
5.4.4	List comparisons	29
5.4.5	Combination filters	31
5.4.6	Spatial filters	33
5.4.7	Temporal filters	36
6	Administration	39

6.1	Creating the OpenAPI document	39
6.2	Verifying configuration files	39
6.3	Setting system environment variables	40
6.4	Summary	40
7	Running	41
7.1	pygeoapi serve	41
7.1.1	Flask WSGI	41
7.1.2	Starlette ASGI	41
7.2	Running in production	42
7.2.1	Apache and mod_wsgi	42
7.2.2	Gunicorn	42
7.2.3	Gunicorn and Flask	43
7.2.4	Gunicorn and Starlette	43
7.3	Summary	43
8	Docker	45
8.1	The basics	45
8.2	Overriding the default configuration	45
8.3	Deploying on a sub-path	46
8.4	Summary	46
9	Taking a tour of pygeoapi	47
9.1	Overview	47
9.2	Landing page	47
9.3	Collections	47
9.4	Collection information	48
9.5	Vector data	48
9.5.1	Collection queryables	48
9.5.2	Collection items	48
9.5.3	Collection item	48
9.6	Raster data	48
9.6.1	Collection coverage domainset	48
9.6.2	Collection coverage rangetype	49
9.6.3	Collection coverage data	49
9.7	SpatioTemporal Assets	49
9.8	Processes	49
9.9	API Documentation	49
9.10	Conformance	50
10	OpenAPI	51
10.1	Using OpenAPI	51
10.2	Summary	56
11	Data publishing	57
11.1	Providers overview	57
11.1.1	Publishing vector data to OGC API - Features	57
11.1.1.1	Providers	57
11.1.1.2	Connection examples	58
11.1.1.3	Data access examples	60
11.1.2	Publishing raster data to OGC API - Coverages	60
11.1.2.1	Providers	60
11.1.2.2	Connection examples	60
11.1.2.3	Data access examples	61
11.1.3	Publishing processes via OGC API - Processes	61

11.1.3.1 Configuration	61
11.1.3.2 Processing examples	61
11.1.4 Publishing files to a SpatioTemporal Asset Catalog	62
11.1.4.1 Connection examples	62
11.1.4.2 Data access examples	62
12 Customizing pygeoapi: plugins	63
12.1 Overview	63
12.2 Example: custom pygeoapi vector data provider	64
12.2.1 Python code	64
12.2.2 Connecting to pygeoapi	65
12.3 Example: custom pygeoapi raster data provider	65
12.3.1 Python code	65
12.4 Example: custom pygeoapi formatter	66
12.4.1 Python code	66
12.5 Processing plugins	67
13 Development	69
13.1 Codebase	69
13.2 Testing	69
13.3 Working with Spatialite on OSX	69
13.3.1 Using pyenv	69
14 OGC Compliance	71
14.1 CITE instance	71
14.2 Setting up your own CITE testing instance	71
15 Contributing	73
16 Support	75
16.1 Community	75
17 Further Reading	77
18 License	79
18.1 Code	79
18.2 Documentation	79
19 API documentation	81
19.1 API	81
19.2 flask_app	82
19.3 Logging	83
19.4 OpenAPI	84
19.5 Plugins	84
19.6 Utils	85
19.7 Formatter package	87
19.7.1 Base class	87
19.7.2 csv	87
19.8 Process package	88
19.8.1 Base class	88
19.8.2 hello_world	88
19.9 Provider	89
19.9.1 Base class	89
19.9.2 CSV provider	91
19.9.3 Elasticsearch provider	92

19.9.4	GeoJSON	93
19.9.5	OGR	94
19.9.6	postgresql	96
19.9.7	sqlite/geopackage	98
20	Indices and tables	101
	Python Module Index	103
	Index	105



Author the pygeoapi team

Contact pygeoapi at lists.osgeo.org

Release 0.9.dev0

Date 2020-11-05

INTRODUCTION

`pygeoapi` is a Python server implementation of the OGC API suite of standards. The project emerged as part of the next generation [OGC API](#) efforts in 2018 and provides the capability for organizations to deploy a RESTful OGC API endpoint using OpenAPI, GeoJSON, and HTML. `pygeoapi` is [open source](#) and released under an MIT [License](#).

1.1 Features

- out of the box modern OGC API server
- certified OGC Compliant and Reference Implementation for OGC API - Features
- additionally implements OGC API - Coverages, OGC API - Processes and SpatioTemporal Asset Library
- out of the box data provider plugins for rasterio, GDAL/OGR, Elasticsearch, PostgreSQL/PostGIS
- easy to use OpenAPI / Swagger documentation for developers
- supports JSON, GeoJSON, HTML and CSV output
- supports data filtering by spatial, temporal or attribute queries
- easy to install: install a full implementation via `pip` or `git`
- simple YAML configuration
- easy to deploy: via UbuntuGIS or the official Docker image
- flexible: built on a robust plugin framework to build custom data connections, formats and processes
- supports any Python web framework (included are Flask [default], Starlette)

1.2 Standards Support

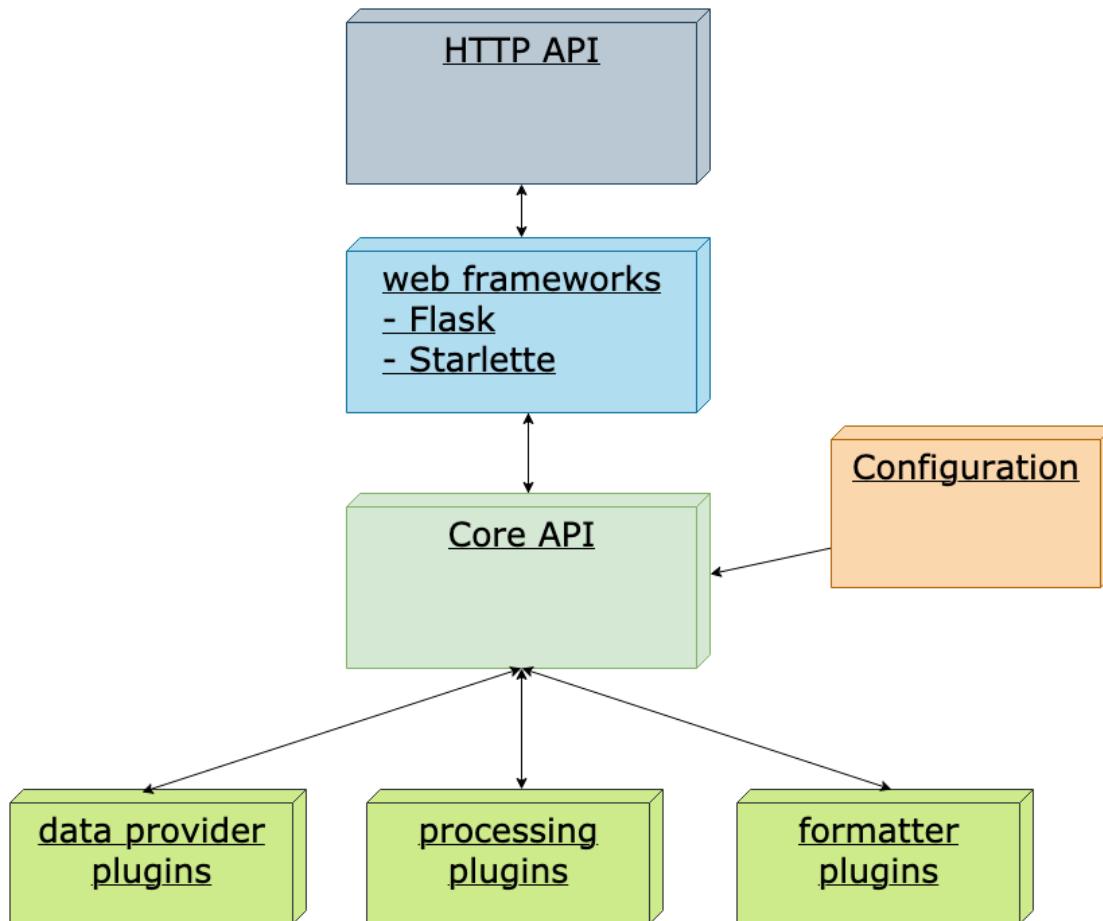
Standards are at the core of `pygeoapi`. Below is the project's standards support matrix.

- Implementing: implements standard (good)
- Compliant: conforms to OGC compliance requirements (great)
- Reference Implementation: provides a reference for the standard (awesome!)

Standard	Support
OGC API - Features	Reference Implementation
OGC API - Coverages	Implementing
OGC API - Processes	Implementing
SpatioTemporal Asset Catalog	Implementing

HOW PYGEOAPI WORKS

pygeoapi is a Python-based HTTP server implementation of the OGC API standards. As a server implementation, pygeoapi listens to HTTP requests from web browsers, mobile or desktop applications and provides responses accordingly.



At its core, pygeoapi provides a core Python API that is driven by two required YAML configuration files, specified with the following environment variables:

- PYGEOAPI_CONFIG: runtime configuration settings
- PYGEOAPI_OPENAPI: the OpenAPI document autogenerated from the runtime configuration

See also:

Configuration for more details on pygeoapi settings

The core Python API provides the functionality to list, describe, query, and access geospatial data. From here, standard Python web frameworks like [Flask](#), [Django](#) and [Starlette](#) provide the web API/wrapper atop the core Python API.

Note: pygeoapi ships with Flask and Starlette as web framework options.

INSTALL

pygeoapi is easy to install on numerous environments. Whether you are a user, administrator or developer, below are multiple approaches to getting pygeoapi up and running depending on your requirements.

3.1 Requirements and dependencies

pygeoapi runs on Python 3.

Core dependencies are included as part of a given pygeoapi installation procedure. More specific requirements details are described below depending on the platform.

3.2 For developers and the truly impatient

```
python -m venv pygeoapi
cd pygeoapi
. bin/activate
git clone https://github.com/geopython/pygeoapi.git
cd pygeoapi
pip install -r requirements.txt
python setup.py install
cp pygeoapi-config.yml example-config.yml
vi example-config.yml
export PYGEOAPI_CONFIG=example-config.yml
export PYGEOAPI_OPENAPI=example-openapi.yml
pygeoapi generate-openapi-document -c $PYGEOAPI_CONFIG > $PYGEOAPI_OPENAPI
pygeoapi serve
curl http://localhost:5000
```

3.3 pip

PyPI package info

```
pip install pygeoapi
```

3.4 Docker

Docker image

```
docker pull geopython/pygeoapi:latest
```

3.5 Conda

Conda package info

```
conda install -c conda-forge pygeoapi
```

3.6 UbuntuGIS

UbuntuGIS package (stable)

UbuntuGIS package (unstable)

```
apt-get install python3-pygeoapi
```

3.7 FreeBSD

FreeBSD port

```
pkg install py-pygeoapi
```

3.8 Summary

Congratulations! Whichever of the abovementioned methods you chose, you have successfully installed pygeoapi onto your system.

CHAPTER FOUR

CONFIGURATION

Once you have installed pygeoapi, it's time to setup a configuration. pygeoapi's runtime configuration is defined in the **YAML** format which is then referenced via the `PYGEOAPI_CONFIG` environment variable. You can name the file whatever you wish; typical filenames end with `.yml`.

Note: A sample configuration can always be found in the pygeoapi [GitHub](#) repository.

pygeoapi configuration contains the following core sections:

- `server`: server-wide settings
- `logging`: logging configuration
- `metadata`: server-wide metadata (contact, licensing, etc.)
- `resources`: dataset collections, processes and stac-collections offered by the server

Note: Standard **YAML** mechanisms can be used (anchors, references, etc.) for reuse and compactness.

Configuration directives and reference are described below via annotated examples.

4.1 Reference

4.1.1 server

The `server` section provides directives on binding and high level tuning.

```
server:  
  bind:  
    host: 0.0.0.0 # listening address for incoming connections  
    port: 5000 # listening port for incoming connections  
    url: http://localhost:5000/ # url of server  
    mimetype: application/json; charset=UTF-8 # default MIME type  
    encoding: utf-8 # default server encoding  
    language: en-US # default server language  
    cors: true # boolean on whether server should support CORS  
    pretty_print: true # whether JSON responses should be pretty-printed  
    limit: 10 # server limit on number of items to return  
    map: # leaflet map setup for HTML pages  
      url: https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png
```

(continues on next page)

(continued from previous page)

```
attribution: '<a href="https://wikimediafoundation.org/wiki/Maps_Terms_of_Use">
    ↵Wikimedia maps</a> | Map data &copy; <a href="https://openstreetmap.org/copyright">
    ↵OpenStreetMap contributors</a>'
ogc_schemas_location: /opt/schemas.opengis.net # local copy of http://schemas.
    ↵opengis.net
```

4.1.2 logging

The logging section provides directives for logging messages which are useful for debugging.

```
logging:
    level: ERROR # the logging level (see https://docs.python.org/3/library/logging.
        ↵html#logging-levels)
    logfile: /path/to/pygeoapi.log # the full file path to the logfile
```

Note: If `level` is defined and `logfile` is undefined, logging messages are output to the server's `stdout`.

4.1.3 metadata

The metadata section provides settings for overall service metadata and description.

```
metadata:
    identification:
        title: pygeoapi default instance # the title of the service
        description: pygeoapi provides an API to geospatial data # some descriptive_
            ↵text about the service
        keywords: # list of keywords about the service
            - geospatial
            - data
            - api
        keywords_type: theme # keyword type as per the ISO 19115 MD_KeywordTypeCode_
            ↵codelist). Accepted values are discipline, temporal, place, theme, stratum
        terms_of_service: https://creativecommons.org/licenses/by/4.0/ # terms of_
            ↵service
        url: http://example.org # informative URL about the service
    license: # licensing details
        name: CC-BY 4.0 license
        url: https://creativecommons.org/licenses/by/4.0/
    provider: # service provider details
        name: Organization Name
        url: https://pygeoapi.io
    contact: # service contact details
        name: Lastname, Firstname
        position: Position Title
        address: Mailing Address
        city: City
        stateorprovince: Administrative Area
        postalcode: Zip or Postal Code
        country: Country
        phone: +xxx-xxx-xxx-xxxx
        fax: +xxx-xxx-xxx-xxxx
```

(continues on next page)

(continued from previous page)

```

email: you@example.org
url: Contact URL
hours: Mo-Fr 08:00-17:00
instructions: During hours of service. Off on weekends.
role: pointOfContact

```

4.1.4 resources

The resources section lists 1 or more dataset collections to be published by the server.

The resource.type property is required. Allowed types are:

- collection
- process
- stac-collection

The providers block is a list of 1..n providers with which to operate the data on. Each provider requires a type property. Allowed types are:

- feature

A collection's default provider can be qualified with default: true in the provider configuration. If default is not included, the first provider is assumed to be the default.

```

resources:
  obs:
    type: collection # REQUIRED (collection, process, or stac-collection)
    title: Observations # title of dataset
    description: My cool observations # abstract of dataset
    keywords: # list of related keywords
      - observations
      - monitoring
    context: # linked data configuration (see Linked Data section)
      - datetime: https://schema.org/DateTime
      - vocab: https://example.com/vocab#
        stn_id: "vocab:stn_id"
        value: "vocab:value"
    links: # list of 1..n related links
      - type: text/csv # MIME type
        rel: canonical # link relations per https://www.iana.org/assignments/
        ↵link-relations/link-relations.xhtml
        title: data # title
        href: https://github.com/mapserver/mapserver/blob/branch-7-0/msautotest/
        ↵wxs/data/obs.csv # URL
        hreflang: en-US # language
    extents: # spatial and temporal extents
      spatial: # required
        bbox: [-180,-90,180,90] # list of minx, miny, maxx, maxy
        crs: http://www.opengis.net/def/crs/OGC/1.3/CRS84 # CRS
      temporal: # optional
        begin: 2000-10-30T18:24:39Z # start datetime in RFC3339
        end: 2007-10-30T08:57:29Z # end datetime in RFC3339
    providers: # list of 1..n required connections information
      # provider name
      # see pygeoapi.plugin for supported providers

```

(continues on next page)

(continued from previous page)

```
# for custom built plugins, use the import path (e.g. mypackage.provider.  
↳MyProvider)  
    # see Plugins section for more information  
    - type: feature # underlying data geospatial type: (allowed values are:  
↳feature, coverage)  
        default: true # optional: if not specified, the first provider  
↳definition is considered the default  
        name: CSV  
        data: tests/data/obs.csv # required: the data filesystem path or URL,  
↳depending on plugin setup  
        id_field: id # required for vector data, the field corresponding to  
↳the ID  
        time_field: datetimestamp # optional field corresponding to the  
↳temporal property of the dataset  
        format: # optional default format  
        name: GeoJSON # required: format name  
        mimetype: application/json # required: format mimetype  
    options: # optional options to pass to provider (i.e. GDAL creation)  
        option_name: option_value  
    properties: # optional: only return the following properties, in order  
        - stn_id  
        - value  
  
hello-world: # name of process  
    type: collection # REQUIRED (collection, process, or stac-collection)  
    processor:  
        name: HelloWorld # Python path of process defition
```

See also:

[Linked Data](#) for optionally configuring linked data datasets

See also:

[Customizing pygeoapi: plugins](#) for more information on plugins

4.2 Using environment variables

pygeoapi configuration supports using system environment variables, which can be helpful for deploying into 12 factor environments for example.

Below is an example of how to integrate system environment variables in pygeoapi.

```
server:  
    bind:  
        host: ${MY_HOST}  
        port: ${MY_PORT}
```

4.3 Linked Data



pygeoapi supports structured metadata about a deployed instance, and is also capable of presenting data as structured data. [JSON-LD](#) equivalents are available for each HTML page, and are embedded as data blocks within the corresponding page for search engine optimisation (SEO). Tools such as the [Google Structured Data Testing Tool](#) can be used to check the structured representations.

The metadata for an instance is determined by the content of the `metadata` section of the configuration. This metadata is included automatically, and is sufficient for inclusion in major indices of datasets, including the [Google Dataset Search](#).

For collections, at the level of an item or items, by default the JSON-LD representation adds:

- The GeoJSON JSON-LD vocabulary and context to the `@context`.
- An `@id` for each item in a collection, that is the URL for that item (resolving to its HTML representation in pygeoapi)

Note: While this is enough to provide valid RDF (as GeoJSON-LD), it does not allow the *properties* of your items to be unambiguously interpretable.

pygeoapi currently allows for the extension of the `@context` to allow properties to be aliased to terms from vocabularies. This is done by adding a `context` section to the configuration of a dataset.

The default pygeoapi configuration includes an example for the `obs` sample dataset:

```
context:
  - datetime: https://schema.org/DateTime
  - vocab: https://example.com/vocab#
    stn_id: "vocab:stn_id"
    value: "vocab:value"
```

This is a non-existent vocabulary included only to illustrate the expected data structure within the configuration. In particular, the links for the `stn_id` and `value` properties do not resolve. We can extend this example to one with terms defined by schema.org:

```
context:
  - schema: https://schema.org/
    stn_id: schema:identifier
    datetime:
      "@id": schema:observationDate
      "@type": schema:DateTime
    value:
      "@id": schema:value
      "@type": schema:Number
```

Now this has been elaborated, the benefit of a structured data representation becomes clearer. What was once an unexplained property called `datetime` in the source CSV, it can now be expanded to <https://schema.org/observationDate>, thereby eliminating ambiguity and enhancing interoperability. Its type is also expressed as <https://schema.org/DateTime>.

This example demonstrates how to use this feature with a CSV data provider, using included sample data. The implementation of JSON-LD structured data is available for any data provider but is currently limited to defining a @context. Relationships between items can be expressed but is dependent on such relationships being expressed by the dataset provider, not pygeoapi.

4.4 CQL Filter

A fundamental operation performed by pygeoapi on a collection of features is that of querying in order to obtain a subset of the data which contains feature instances that satisfy some filtering criteria. The filtering criteria can be a simpler expression or an arbitrarily complex expression. To implement these enhanced filtering criteria in a request to a server, CQL is used. CQL extension on pygeoapi specifies how resource instances in a source collection should be filtered to identify a result set.

CQL helps in query operations to identify the subset of resources that should be included in a response document. Each resource instance in the source collection is evaluated using a CQL filtering expression. The overall filter expression always evaluates to true or false. If the expression evaluates to true, the resource instance satisfies the expression and is marked as being in the result set. If the overall filter expression evaluates to false, the data instance is not in the result set.

This section is implemented at provider level and based on [OGC API - Features - Part 3: Common Query Language](#) document that defines the schema for a JSON document and exposes the set of properties or keys that are used to construct CQL expressions for pygeoapi.

CQL filter extension can be enabled for a resource by adding `filters` section to the configuration of a resource in pygeoapi config file.

The default pygeoapi configuration for CQL extension includes an example for the `obs` sample dataset:

resources:

obs:

providers:

extensions:

- type: CQL

filters:

- cql-text
- cql-json

4.5 Summary

At this point, you have the configuration ready to administer the server.

CQL FILTER IMPLEMENTATION

pygeoapi is a Python server implementation of the OGC API suite of standards. OGC API standards define modular API building blocks to spatially enable Web API in a consistent way. This standard specifies the fundamental API building blocks for interacting with features. pygeoapi provides the capability for organizations to deploy a RESTful OGC API endpoint using OpenAPI, GeoJSON, and HTML. Project/code is structured to provide functionality via plugins where data can be fetched from any backend services like remote services or local files.

Querying is one of the fundamental operations performed on a collection of features. It is in order to obtain a subset of the data which contains feature instances that satisfy some filtering criteria. This project implements these enhanced filtering criteria in a request to a server. CQL is used to specify how resource instances in a source collection should be filtered to identify a result set. Typically, CQL is used here in query operations because it can be written in human readable format. So its the best query language that can be used to identify the subset of resources that should be included in a response document. Each resource instance in the source collection is evaluated using a filtering expression. The overall filter expression always evaluates to true or false. If the expression evaluates to true, the resource instance satisfies the expression and is marked as being in the result set. If the overall filter expression evaluates to false, the data instance is not in the result set.

This project is based on [OGC API - Features - Part 3: Common Query Language](#) document that defines the schema for a JSON document that exposes the set of properties or keys that may be used to construct CQL expressions for pygeoapi.

5.1 CQL Filter Predicates

The following CQL predicates are implemented in pygeoapi to support filtering functionality on features:

Simple Condition Predicate, Combination Predicate, Not Condition Predicate, Between Predicate, Like Predicate, In Predicate, Null Predicate, BBox Predicate, Spatial Predicate and Temporal Predicate

5.2 CQL Filter Implementation for Data Providers

CQL implementation are provider for following data providers:

- **CQL for CSV and GeoJSON data providers:** Evaluation of the Abstract Syntax Tree to filter the feature collections supported by CSV and GeoJSON data providers. pycql library has implementation connection to databases using ORM, but in pygeoapi the data providers don't work with ORM. So the evaluation for all the CQL query operations are developed from scratch and by using efficient methodology. The evaluated output is the response from the API.
- **CQL for SQLite data provider:** Evaluation of the Abstract Syntax Tree to filter the feature collections supported by SQLite data provider. The AST of the CQL filter request is translated into SQL queries and then used as a request to the database. The evaluated output from the SQLite database is the response from the API.

- **CQL for PostGreSQL data provider:** Evaluation of the Abstract Syntax Tree to filter the feature collections supported by PostGreSQL data provider. Like SQLite queries, the AST of the CQL filter request is translated into PostGreSQL queries by following the syntax of psycopg2 database adapter. The query is then used as a request to the database. The evaluated output from the PostGreSQL database is the response from the API.

5.3 Steps to generate and execute CQL endpoints

1. Install and run pygeoapi on localhost following the steps specified here
2. Go to OpenAPI documentation

The screenshot shows the OpenAPI documentation for the pygeoapi default instance. The left sidebar contains sections for Collections, SpatioTemporal Assets, Processes, API Definition (with 'Documentation' highlighted), and Conformance. The main content area displays the API definition, divided into Provider and Contact point sections. The Provider section includes fields for Organization Name (https://pygeoapi.io) and Contact URL. The Contact point section includes fields for Address (Mailing Address, City, Administrative Area, Zip or Postal Code), Country, Email (you@example.org), Telephone (+xx-xxx-xxx-xxxx), Fax (+xx-xxx-xxx-xxxx), and Hours (Mo-Fr 08:00-17:00). The Contact instructions field notes 'During hours of service. Off on weekends.'

Provider	
Organization Name	https://pygeoapi.io

Contact point	
Mailing Address	
City, Administrative Area	
Zip or Postal Code	
Country	
Email	you@example.org
Telephone	+xx-xxx-xxx-xxxx
Fax	+xx-xxx-xxx-xxxx
Contact URL	
Contact URL	
Hours	Mo-Fr 08:00-17:00
Contact instructions	During hours of service. Off on weekends.

pygeoapi currently supports two collections obs and lakes from CSV and GeoJSON data providers in OpenAPI Documentation

3. Providing CQL query filter along with other query parameters. For the following parameters, the default value of limit is 10, startindex 0, CQL query language is in text, resulttype is results and output format is GeoJSON

lakes lakes of the world, public domain

GET /collections/lakes Get collection metadata

GET /collections/lakes/items Get Large Lakes items

lakes of the world, public domain

Parameters

Name **Description**

f string (query) The optional f parameter indicates the output format which the server shall provide as part of the response document. The default format is GeoJSON.
json

bbox array(number) (query) Only features that have a geometry that intersects the bounding box are selected. The bounding box is provided as four or six numbers, depending on whether the coordinate reference system includes a vertical axis (height or depth):

- Lower left corner, coordinate axis 1
- Lower left corner, coordinate axis 2
- Minimum value, coordinate axis 3 (optional)
- Upper right corner, coordinate axis 1
- Upper right corner, coordinate axis 2
- Maximum value, coordinate axis 3 (optional)

The coordinate reference system of the values is WGS 84 longitude/latitude (<http://www.opengis.net/def/crs/OGC/1.3/CRS84>) unless a different coordinate reference system is specified in the parameter `bbox-crs`.

For WGS 84 longitude/latitude the values are in most cases the sequence of minimum longitude, minimum latitude, maximum longitude and maximum latitude. However, in cases where the box spans the antimeridian the first value (west-most box edge) is larger than the third value (east-most box edge).

If the vertical axis is included, the third and the sixth number are the bottom and the top of the 3-dimensional bounding box.

If a feature has multiple spatial geometry properties, it is the decision of the server whether only a single spatial geometry property is used to determine the extent or all relevant geometries.

Add item

limit integer (query) The optional limit parameter limits the number of items that are presented in the response document.
Only items are counted that are on the first level of the collection in the response document. Nested objects contained within the explicitly requested items shall not be counted.
Minimum = 1. Maximum = 10000. Default = 10.
Default value : 10
10

Sortby string (query) The optional sortby parameter indicates the sort property and order on which the server shall present results in the response document using the convention `sortby=PROPERTY:X`, where `PROPERTY` is the sort property and `X` is the sort order (`A` is ascending, `D` is descending). Sorting by multiple properties is supported by providing a comma-separated list.
sortby - The optional sortby parameter indica

startindex integer (query) The optional startindex parameter indicates the index within the result set from which the server shall begin presenting results in the response document. The first element has an index of 0 (default).
Default value : 0
startindex - The optional startindex paramete

filter string (query) The optional filter parameter to provide filters on the collection items
filter - The optional filter parameter to provide

filter-lang string (query) The optional parameter to provide filter lang
Available values : cql-text, cql-json
Default value : cql-text
cql-text

admin string (query) admin

featureclass string (query) featureclass

name string (query) name

<code>name_alt</code> string (query)	<input type="text" value="name_alt"/>
<code>scalerank</code> string (query)	<input type="text" value="scalerank"/>

The parameter values of any collection item can be changed to generate different API endpoint

4. Click on Try it out to give the parameters value

GET /collections/lakes/items Get Large Lakes items

lakes of the world, public domain

Parameters

Try it out

Name	Description
<code>f</code> string (query)	The optional f parameter indicates the output format which the server shall provide as part of the response document. The default format is GeoJSON. Available values : json, html, jsonld, csv Default value : json <input type="text" value="json"/>
<code>bbox</code> array[number] (query)	Only features that have a geometry that intersects the bounding box are selected. The bounding box is provided as four or six numbers, depending on whether the coordinate reference system includes a vertical axis (height or depth): <ul style="list-style-type: none"> Lower left corner, coordinate axis 1 Lower left corner, coordinate axis 2 Minimum value, coordinate axis 3 (optional) Upper right corner, coordinate axis 1 Upper right corner, coordinate axis 2 Maximum value, coordinate axis 3 (optional) The coordinate reference system of the values is WGS 84 longitude/latitude (http://www.opengis.net/def/crs/OGC/1.3/CRS84) unless a different coordinate reference system is specified in the parameter <code>bbox-crs</code> . For WGS 84 longitude/latitude the values are in most cases the sequence of minimum longitude, minimum latitude, maximum longitude and maximum latitude. However, in cases where the box spans the antimeridian the first value (west-most box edge) is larger than the third value (east-most box edge). If the vertical axis is included, the third and the sixth number are the bottom and the top of the 3-dimensional bounding box. If a feature has multiple spatial geometry properties, it is the decision of the server whether only a single spatial geometry property is used to determine the extent or all relevant geometries.

5. Provide the CQL query parameter in text to filter the collection features Here assigning CQL filter as **WITHIN(geometry, POLYGON((-80.0 -80.0,-80.0 50,80.0 50,-80.0 -80.0))) AND id<>371** and keeping the default values of all the other parameters.

<code>limit</code> integer (query)	The optional limit parameter limits the number of items that are presented in the response document. Only items are counted that are on the first level of the collection in the response document. Nested objects contained within the explicitly requested items shall not be counted. Minimum = 1. Maximum = 10000. Default = 10. <input type="text" value="10"/>
<code>sortby</code> string (query)	The optional sortby parameter indicates the sort property and order on which the server shall present results in the response document using the convention <code>sortby=PROPERTY:X</code> , where <code>PROPERTY</code> is the sort property and <code>X</code> is the sort order (<code>A</code> is ascending, <code>D</code> is descending). Sorting by multiple properties is supported by providing a comma-separated list. <input type="text" value="sortby - The optional sortby parameter indica"/>
<code>startindex</code> integer (query)	The optional startindex parameter indicates the index within the result set from which the server shall begin presenting results in the response document. The first element has an index of 0 (default). <input type="text" value="startIndex - The optional startindex paramete"/>
<code>filter</code> string (query)	The optional filter parameter to provide filters on the collection items <input type="text" value="WITHIN(geometry, POLYGON((-80.0 -80.0,-80.0 50,80.0 50,-80.0 -80.0)) AND id<>371)"/>

6. After filling the values of parameters (including CQL filter expression), click on execute. If the CQL expression is valid then an endpoint will be generated with Success code 200 and response body.

Responses

Execute 

Curl

```
curl -X GET "http://localhost:5000/collections/lakes/items?f=json&limit=10&filter=WITHIN%28geometry%2C%2B-80.0%2B-80.0%2C-80.0%2B50%2C80.0%2B50%2C-80.0%2B-80.0%2B50%2BAND%20id%3CK3E371&filter-lang=cql-text" -H "accept: application/json"
```

Request URL

```
http://localhost:5000/collections/lakes/items?f=json&limit=10&filter=WITHIN%28geometry%2C%2B-80.0%2B-80.0%2C-80.0%2B50%2C80.0%2B50%2C-80.0%2B-80.0%2B50%2BAND%20id%3CK3E371&filter-lang=cql-text
```

Server response

Code Details

200 Response body

```
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          [
            [
              [
                [
                  -79.05630591502026,
                  43.2540431576152
                ],
                [
                  -79.36168779164988,
                  43.20237620703736
                ],
                [
                  -79.76947481964547,
                  43.29720246029295
                ],
                [
                  -79.05630591502026,
                  43.2540431576152
                ]
              ]
            ]
          ]
        ]
      }
    }
  ]
}
```

7. Furthermore the response body can be investigated by hitting the generated URL:

```
http://localhost:5000/collections/lakes/items?f=json&filter-lang=cql-text&  
filter=WITHIN(geometry, POLYGON((-80.0 -80.0,-80.0 50,80.0 50,-80.0 -80.0)))  
AND id<>371
```

8. Since the output format was specified as GeoJSON the response from API is the following:

```
{ "features": [ { "geometry": { "coordinates": [ [ [ -79.05630591502026, 43.25410431576152 ], [ -79.36168779164908, 43.20237620703736 ], [ -79.76047481964547, 43.29720246029295 ], [ -79.46116492381094, 43.639197089200565 ], [ -79.1561706204243, 43.75743276628437 ], [ -78.45052893747877, 43.9031861435636 ], [ -77.60530688345149, 44.03932774436545 ], [ -77.16148617217414, 43.85014028515994 ], [ -76.88269181995948, 44.0694550646427 ], [ -76.56555355948425, 44.20802514765336 ], [ -76.3530422718391, 44.134670722015045 ], [ -76.23926856149336, 43.9791504990326565 ], [ -76.17999563635458, 43.5900011256857 ], [ -76.9300015937227, 43.25999542904281 ], [ -77.74915056019732, 43.342832750006664 ], [ -78.53499406605984, 43.379988104824534 ], [ -79.05630591502026, 43.25410431576152 ] ], "type": "Polygon" }, "id": "3", "properties": { "admin": "admin-0", "featureclass": "Lake", "name": "L_Ontario", "name_alt": "https://en.wikipedia.org/wiki/Lake_Ontario", "scalerank": 0, "type": "Feature" }, "geometry": { "coordinates": [ [ [ 60.05285159692946, 44.264636946229114 ], [ 59.77002648299605, 44.15999217383806 ], [ 59.062886138315405, 44.363928172462015 ], [ 59.34571129744745, 44.9972053538971 ], [ 59.35930219914022, 45.1900068222796851 ], [ 58.96139367049281, 45.375008484984859 ], [ 58.92144778833119, 45.1018468287879746 ], [ 58.73556427670495, 45.4871326082782614, 44.212340397479735 ], [ 58.285008048224456, 44.473952338227335 ], [ 58.285008048224456, 44.89255727800766 ], [ 58.68998048095896, 45.50001373959863 ], [ 58.78000939314833, 45.88673432065487 ], [ 59.20427290226462, 45.93905708350391 ], [ 59.553002068932585, 45.70501414650949 ], [ 59.55784305200561, 46.30539260453519 ], [ 59.77002648299603, 46.25299103452352 ], [ 60.12355963273702, 46.060023871436955 ], [ 60.12359663273702, 45.88673432065487 ], [ 59.94675988143425, 45.808211981787366 ], [ 59.84071984237133, 45.502477606786216 ], [ 60.12359663273702, 45.572774156265595 ], [ 60.23997195828534, 46.0025825159692946, 44.264636949229114 ], "type": "Polygon" }, "id": "11", "properties": { "admin": null, "featureclass": "Lake", "name": "Aral Sea", "name_alt": "https://en.wikipedia.org/wiki/Aral_Sea", "scalerank": 0, "type": "Feature" }, "geometry": { "coordinates": [ [ [ -69.40673987494259, -16.12619882575063 ], [ -69.7290974595793, -15.928794854397104 ], [ -69.8365556504908, -15.73717864345884 ], [ -69.87870212470148, -15.66984252182529 ], [ -69.86877797183637, -15.546079196843493 ], [ -69.88559534775224, -15.35425628017606 ], [ -69.59675411647981, -15.410480238509614 ], [ -68.98697221543571, -15.889503415594846 ], [ -68.95986708751856, -15.91329192470954 ], [ -68.74623755560401, -16.356003920154023 ], [ -68.9052453776611, -16.566640720284835 ], [ -69.00115739690983, -16.536406343258492 ], [ -69.09084184432438, -16.461992827874657 ], [ -69.18205074733753, -16.40116912197712 ], [ -69.25098710801488, -16.227536309476427 ], [ -69.40673987494259, -16.126198825752063 ], [ -69.40673987494259, -16.126198825752063 ] ], "type": "Polygon" }, "id": "16", "properties": { "admin": null, "featureclass": "Lake", "name": "Lago Titicaca", "name_alt": "https://en.wikipedia.org/wiki/Lake_Titicaca", "scalerank": 1, "type": "Feature" }, "geometry": { "coordinates": [ [ [ 60.23997195828534, 46.0025825159692946, 44.264636949229114 ], "type": "FeatureCollection", "numberMatched": 3, "numberReturned": 3, "links": [ { "type": "application/geo+json", "rel": "self", "title": "This document as GeoJSON", "href": "http://localhost:5000/collections/lakes/items?f=json&limit=10&filter=WITHIN%28geometry,%20POLYGON%28%28-80.0%20-80.0%20-80.0%29%28%20AND%20id%3C%3E371&filter=lang=cql-text" }, { "type": "application/ld+json", "title": "This document as RDF (JSON-LD)", "href": "http://localhost:5000/collections/lakes/items?f=json&limit=10&filter=WITHIN%28geometry,%20POLYGON%28%28-80.0%20-80.0,-80.0%2050,80.0%2050,80.0%20-80.0%29%28%29%20AND%20id%3C%3E371&filter=lang=cql-text" }, { "type": "text/html", "rel": "alternate", "title": "This document as HTML", "href": "http://localhost:5000/collections/lakes/items?f=html&limit=10&filter=WITHIN%28geometry,%20POLYGON%28%28-80.0%20-80.0,-80.0%2050,80.0%2050,-80.0%20-80.0%29%28%29%20AND%20id%3C%3E371&filter=lang=cql-text" }, { "type": "application/json", "title": "Large Lakes", "rel": "collection", "href": "http://localhost:5000/collections/lakes" } ], "timestamp": "2020-08-31T01:02:24.100013Z" } ] }
```

9. For the same CQL filter expression if the resulttype is changed to hits. The API response will have only the total count of features that satisfied the given filter expression.

Requested API:

```
http://localhost:5000/collections/lakes/items?f=json&filter-lang=cql-text&resulttype=hits&filter=WITHIN(geometry, POLYGON((-80.0 -80.0,-80.0 50,80.0 50,-80.0 -80.0)))  
AND id<>371
```

Response:

```
← → ⌂ ⓘ http://localhost:5000/collections/lakes/items?f=json&resulttype=hits&limit=10&filter=WITHIN%28geometry%2C%20POLYGON%28%28-80.0%20-80.0,-80.0%2050,80.0%2050,-80.0%20-80.0%29%29%20AND%20id%3C%3E371&filter-lang=cql-text", {"rel": "alternate", "type": "application/ld+json", "title": "This document as RDF (JSON-LD)", "href": "http://localhost:5000/collections/lakes/items?f=json&resulttype=hits&limit=10&filter=WITHIN%28geometry,%20POLYGON%28%28-80.0%20-80.0,-80.0%2050,80.0%2050,-80.0%20-80.0%29%29%20AND%20id%3C%3E371&filter-lang=cql-text"}, {"type": "text/html", "rel": "alternate", "title": "This document as HTML", "href": "http://localhost:5000/collections/lakes/items?f=html&resulttype=hits&limit=10&filter=WITHIN%28geometry,%20POLYGON%28%28-80.0%20-80.0,-80.0%2050,80.0%2050,-80.0%20-80.0%29%29%20AND%20id%3C%3E371&filter-lang=cql-text"}, {"type": "application/json", "title": "Large Lakes", "rel": "collection", "href": "http://localhost:5000/collections/lakes"}], "timeStamp": "2020-08-31T10:08:08.460312Z"}
```

- To overlay the response from API on a map, we can change the output format of the endpoint from JSON to HTML

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter-lang=cql-text&filter=WITHIN(geometry, POLYGON((-80.0 -80.0,-80.0 50,80.0 50,-80.0 -80.0))) AND id<>371
```

Response:

ⓘ http://localhost:5000/collections/lakes/items?f=html&filter=WITHIN(geometry,%20POLYGON((-80.0%20-80.0,-80.0%2050,80.0%2050,-80.0%20-80.0%29%29%20AND%20id%3C%3E371&filter-lang=cql-text))

 Contact

Home / Collections / Large Lakes / Items JSON JSON-LD

Large Lakes

Items in this collection.



Warning: Higher limits not recommended!

Limit: 10 (default) ▾

id	admin	featureclass	name	name_alt
3	admin-0	Lake	L. Ontario	https://en.wikipedia.org/wiki/Lake_Ontario
11	None	Lake	Aral Sea	https://en.wikipedia.org/wiki/Aral_Sea
16	None	Lake	Lago Titicaca	https://en.wikipedia.org/wiki/Lake_Titicaca

- If any invalid CQL filter expression is provided then the API raises an exception and the response is as follows:

Requested API:

```
http://localhost:5000/collections/obs/items?f=json&filter-lang=cql-text&filter=INTERSECTION(geometry,POINT (-75 45))
```

Response:

```
← → ⌂ ⓘ http://localhost:5000/collections/obs/items?f=json&filter=INTERSECTION%28geometry%2CPOINT%20%28-75%2045%29%29&filter-lang=cql-text
```

```
{"code": "ParameterValue", "description": "Invalid CQL filter expression"}
```

Requested API:

```
http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=id IN ['A', 'B']
```

Response:


HTTP 400 Bad Request

http://localhost:5000/collections/obs/items?f=html&filter=id%20IN%20["A","B"]

```
{"code": "InvalidParameterValue", "description": "Invalid CQL filter expression"}
```

Requested API:

`http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=name@obs`

Response:

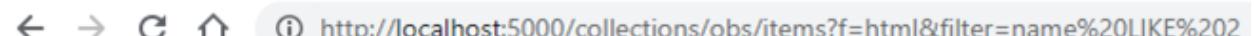

HTTP 400 Bad Request

http://localhost:5000/collections/obs/items?f=html&filter=name@obs

```
{"code": "InvalidParameterValue", "description": "Invalid CQL filter expression"}
```

Requested API:

`http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=name LIKE 2`

Response:


HTTP 400 Bad Request

http://localhost:5000/collections/obs/items?f=html&filter=name%20LIKE%202

```
{"code": "InvalidParameterValue", "description": "Invalid CQL filter expression"}
```

5.4 Examples of CQL query filter

Following are few examples of CQL query filter implemented on pygeoapi data providers-

5.4.1 Getting started

The collections used for the project demonstration here are observation and lake features from CSV and GeoJSON data providers respectively. The attribute table for observation and lake features are as follows:

obs.csv

id	stn_id	datetime	value
371	35	2001-10-30T14:24:55Z	89.9
377	35	2002-10-30T18:31:38Z	93.9
238	2147	2007-10-30T08:57:29Z	103.5
297	2147	2003-10-30T07:37:29Z	93.5
964	604	2000-10-30T18:24:39Z	99.9

lakes.geojson

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wikipedia.org/w/index.php?title=Lake_Baikal&oldid=9100000
1	None	Lake	Lake Winnipeg	https://en.wikipedia.org/w/index.php?title=Lake_Winnipeg&oldid=9100000
2	None	Lake	Great Slave Lake	https://en.wikipedia.org/w/index.php?title=Great_Slave_Lake&oldid=9100000
3	admin-0	Lake	L. Ontario	https://en.wikipedia.org/w/index.php?title=Lake_Ontario&oldid=9100000
4	admin-0	Lake	L. Erie	https://en.wikipedia.org/w/index.php?title=Lake_Erie&oldid=9100000
5	admin-0	Lake	Lake Superior	https://en.wikipedia.org/w/index.php?title=Lake_Superior&oldid=9100000
6	admin-0	Lake	Lake Victoria	https://en.wikipedia.org/w/index.php?title=Lake_Victoria&oldid=9100000
7	None	Lake	Lake Ladoga	https://en.wikipedia.org/w/index.php?title=Lake_Ladoga&oldid=9100000
8	None	Lake	Balqash Köli	Lake Balkhash
9	admin-0	Lake	Lake Tanganyika	https://en.wikipedia.org/w/index.php?title=Lake_Tanganyika&oldid=9100000
10	admin-0	Lake	Lake Malawi	Lake Nyasa
11	None	Lake	Aral Sea	https://en.wikipedia.org/w/index.php?title=Aral_Sea&oldid=9100000
12	None	Lake	Vänern	None
13	None	Lake	Lake Okeechobee	https://en.wikipedia.org/w/index.php?title=Lake_Okeechobee&oldid=9100000

14	None	Lake	Lago de Nicaragua	https://en.wikipedia.org/wiki/Lake_Nicaragua
15	None	Lake	Lake Tana	https://en.wikipedia.org/wiki/Lake_Tana
16	None	Lake	Lago Titicaca	https://en.wikipedia.org/wiki/Lake_Titicaca
17	None	Lake	Lake Winnipegosis	https://en.wikipedia.org/wiki/Lake_Winnipegosis
18	None	Lake	Lake Onega	https://en.wikipedia.org/wiki/Lake_Onega
19	None	Lake	Great Salt Lake	https://en.wikipedia.org/wiki/Great_Salt_Lake
20	None	Lake	Great Bear Lake	https://en.wikipedia.org/wiki/Great_Bear_Lake
21	None	Lake	Lake Athabasca	https://en.wikipedia.org/wiki/Lake_Athabasca
22	None	Lake	Reindeer Lake	https://en.wikipedia.org/wiki/Reindeer_Lake
23	admin-0	Lake	Lake Huron	https://en.wikipedia.org/wiki/Lake_Huron
24	admin-0	Lake	Lake Michigan	https://en.wikipedia.org/wiki/Lake_Michigan

For the following API requests the default value of limit is 10, startIndex is 0 and CQL query language is text

5.4.2 Simple comparisons

Let's get started with the simple examples. In CQL comparisons are expressed using plain text.

- The filter `stn_id >= 35` will filter the observations that have `stn_id` value greater than or equals to 35:

Requested API:

`http://localhost:5000/collections/obs/items?f=html&filter=stn_id>=35&filter-lang=cql-text`

Response:

The screenshot shows a map of North America with observation points marked in blue. A legend indicates that blue dots represent observations. Below the map is a table of observations:

id	stn_id	datetime	value
371	35	2001-10-30T14:24:55Z	89.9
377	35	2002-10-30T18:31:38Z	93.9
238	2147	2007-10-30T08:57:29Z	103.5
297	2147	2003-10-30T07:37:29Z	93.5
964	604	2000-10-30T18:24:39Z	99.9

Below the table, there is a note: "Warning: Higher limits not recommended!" and a dropdown menu set to "10 (default)".

- The filter **stn_id <= 604** will select observations that have **stn_id** less than or equals than 604:

Requested API:

```
http://localhost:5000/collections/obs/items?f=html&filter=stn_id<=604&filter-lang=cql-text
```

Response:

The screenshot shows a map of North America with observation points marked in blue. A legend indicates that blue dots represent observations. Below the map is a table of observations:

id	stn_id	datetime	value
371	35	2001-10-30T14:24:55Z	89.9
377	35	2002-10-30T18:31:38Z	93.9
964	604	2000-10-30T18:24:39Z	99.9

Below the table, there is a note: "Warning: Higher limits not recommended!" and a dropdown menu set to "10 (default)".

- If we want to look for Lake Baikal on the map, then the filter **name='Lake Baikal'** will fetch its details and display its location on the world's map.

The requested API to GeoJSON Data provider for filtering Lake Baikal should be:

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter-lang=cql-text&filter=name='Lake Baikal'
```

Response:

<http://localhost:5000/collections/lakes/items?f=json&filter=name=%27Lake%20Baikal%27>



id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wikipedia.org/wiki/Lake_Baikal

Items in this collection.

Warning: Higher limits not recommended!

Limit: 10 (default)

- To filter lakes whose **id** is not equals to 0, than the filter `id<>0` will response with all the lake features except the one with **id=0**.

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=100&filter=lang=cql-text&filter=id<>0
```

Response:

<http://localhost:5000/collections/lakes/items?limit=100&filter=id<>0>



id	admin	featureclass	name	name_alt
1	None	Lake	Lake Winnipeg	https://en.wikipedia.org/wiki/Lake_Winnipeg
2	None	Lake	Great Slave Lake	https://en.wikipedia.org/wiki/Great_Slave_Lake
3	admin-0	Lake	L. Ontario	https://en.wikipedia.org/wiki/Lake_Ontario
4	admin-0	Lake	L. Erie	https://en.wikipedia.org/wiki/Lake_Erie
5	admin-0	Lake	Lake Superior	https://en.wikipedia.org/wiki/Lake_Superior
6	admin-0	Lake	Lake Victoria	https://en.wikipedia.org/wiki/Lake_Victoria
7	None	Lake	Lake Ladoga	https://en.wikipedia.org/wiki/Lake_Ladoga
8	None	Lake	Balqash Koli	https://en.wikipedia.org/wiki/Balqash_Koli

Items in this collection.

Warning: Higher limits not recommended!

Limit: 100

- If there is a requirement to fetch only 5 lakes starting from index 10 and having filter as `id>10`.

pygeoapi supports limit and startindex request parameters, so an API call is possible with CQL query filter along with other query parameters.

Requested API:

<http://localhost:5000/collections/lakes/items?limit=5&startIndex=10&filter-lang=cql-text&filter=id>10>

Response:

The screenshot shows the pygeoapi interface. At the top, there's a browser header with the URL 'localhost:5000/collections/lakes/items?limit=5&startIndex=10&filter-lang=cql-text&filter=id>10'. Below the header is the pygeoapi logo and navigation links for 'Home', 'Collections', 'Large Lakes', and 'Items'. On the right, there are links for 'Contact', 'JSON', and 'JSON-LD'. The main content area has a title 'Large Lakes' and a subtitle 'Items in this collection.' To the left is a map of North America focusing on the Great Lakes and surrounding areas, with labels for 'Calgary', 'Minneapolis', 'Chicago', 'Toronto', and 'Montreal'. A legend indicates 'Canada' and 'United States of'. Below the map is a table with the following data:

id	admin	featureclass	name	name_alt
21	None	Lake	Lake Athabasca	https://en.wikipedia.org/w/index.php?title=Lake_Athabasca&oldid=91311110
22	None	Lake	Reindeer Lake	https://en.wikipedia.org/w/index.php?title=Reindeer_Lake&oldid=91311110
23	admin-0	Lake	Lake Huron	https://en.wikipedia.org/w/index.php?title=Lake_Huron&oldid=91311110
24	admin-0	Lake	Lake Michigan	https://en.wikipedia.org/w/index.php?title=Lake_Michigan&oldid=91311110

Due to the implementation of CQL extension on pygeoapi, all the simple comparison operations are now supported on any number of feature collections.

The common comparison operators are: <, >, <=, >=, =, <>

- To select a range of values the BETWEEN operator can be used like **id BETWEEN 20 AND 25**

Requested API:

<http://localhost:5000/collections/lakes/items?limit=100&filter-lang=cql-text&filter=id&filter=BETWEEN%20%20AND%2025>

Response:

localhost:5000/collections/lakes/items?limit=100&filter=id%20BETWEEN%202020%20AND%202025

 Contact

Home / Collections / Large Lakes / Items JSON JSON-LD

Large Lakes

Items in this collection.



Warning: Higher limits not recommended!

Limit:

id	admin	featureclass	name	name_alt
20	None	Lake	Great Bear Lake	https://en.wikipedia.org/w/index.php?title=Great_Bear_Lake&oldid=9700000
21	None	Lake	Lake Athabasca	https://en.wikipedia.org/w/index.php?title=Lake_Athabasca&oldid=9700000
22	None	Lake	Reindeer Lake	https://en.wikipedia.org/w/index.php?title=Reindeer_Lake&oldid=9700000
23	admin-0	Lake	Lake Huron	https://en.wikipedia.org/w/index.php?title=Lake_Huron&oldid=9700000
24	admin-0	Lake	Lake Michigan	https://en.wikipedia.org/w/index.php?title=Lake_Michigan&oldid=9700000

- If needed to filter out lake features with no admin then **admin IS NULL** will response with required lakes.

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=1000&filter=lang=cql-text&filter=admin IS NULL
```

Response:

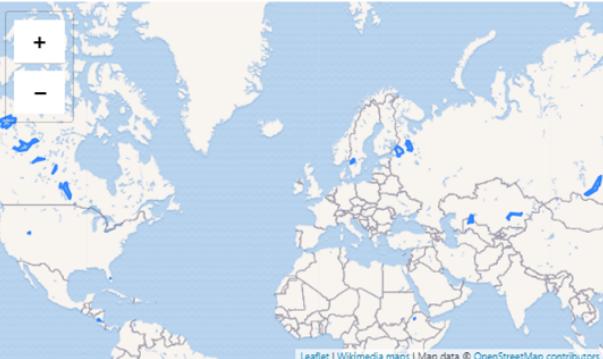
localhost:5000/collections/lakes/items?limit=1000&filter=admin%20IS%20NULL

 Contact

Home / Collections / Large Lakes / Items JSON JSON-LD

Large Lakes

Items in this collection.



Warning: Higher limits not recommended!

Limit:

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wikipedia.org/w/index.php?title=Lake_Baikal&oldid=9700000
1	None	Lake	Lake Winnipeg	https://en.wikipedia.org/w/index.php?title=Lake_Winnipeg&oldid=9700000
2	None	Lake	Great Slave Lake	https://en.wikipedia.org/w/index.php?title=Great_Slave_Lake&oldid=9700000
7	None	Lake	Lake Ladoga	https://en.wikipedia.org/w/index.php?title=Lake_Ladoga&oldid=9700000
8	None	Lake	Balqash Koli	Lake Balkhash
11	None	Lake	Aral Sea	https://en.wikipedia.org/w/index.php?title=Aral_Sea&oldid=9700000
12	None	Lake	Vänern	None
13	None	Lake	Lake	https://en.wikipedia.org/w/index.php?title=Lake&oldid=9700000

5.4.3 String comparisons

- In one of the above example we have already seen that comparison operators also support text values. For instance, to select only Lake Baikal, the filter was name='Lake Baikal'. But more general text/string comparisons can be made using the LIKE operator. name **NOT LIKE '%Lake%'** will extract all lakes that does not have 'Lake' anywhere in their name.

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&&filter-lang=cql-textfilter=name NOT LIKE '%Lake%'
```

Response:

The screenshot shows the pygeoapi web interface. At the top, there is a search bar and a navigation menu with links for Home, Collections, Large Lakes, and Items. Below the menu, the title 'Large Lakes' is displayed. A map of the world highlights major lakes in blue. A warning message 'Warning: Higher limits not recommended!' is shown below the map. To the right of the map is a table of lake data. The table has columns for id, admin, featureclass, name, and name_alt. The data includes entries for Lake Ontario, Lake Erie, Balqash Köli, Aral Sea, Vänern, Lago de Nicaragua, and Lago Titicaca.

id	admin	featureclass	name	name_alt
3	admin-0	Lake	L. Ontario	https://en.wi...
4	admin-0	Lake	L. Erie	https://en.wi...
8	None	Lake	Balqash Köli	Lake Balkhash
11	None	Lake	Aral Sea	https://en.wi...
12	None	Lake	Vänern	None
14	None	Lake	Lago de Nicaragua	https://en.wi...
16	None	Lake	Lago Titicaca	https://en.wi...

- Suppose we want to find all lakes whose name contains an 'great', regardless of letter case. We cannot use LIKE operator here as it is case sensitive. ILIKE operator can be used to ignore letter casing: **name ILIKE '%great%'**

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&&filter-lang=cql-text&filter=name ILIKE "%great%"
```

Response:

The screenshot shows the pygeoapi interface at localhost:5000/collections/lakes/items?f=html&filter=name%20ILIKE%20%25great%25. The page title is "Large Lakes". The left side features a map of North America with highlighted large lakes. Below the map is a warning about higher limits and a dropdown for setting the limit to 10 (default). To the right is a table listing three lakes:

id	admin	featureclass	name	name_alt
2	None	Lake	Great Slave Lake	https://en.wikipedia.org/wiki/Great_Slave_Lake
19	None	Lake	Great Salt Lake	https://en.wikipedia.org/wiki/Great_Salt_Lake
20	None	Lake	Great Bear Lake	https://en.wikipedia.org/wiki/Great_Bear_Lake

The comparison on strings can be performed with either of the following: LIKE, NOT LIKE, ILIKE , NOT LIKE

The CQL extension on pygeoapi supports all the above specified formats for comparing strings.

5.4.4 List comparisons

- If we want to extract only specific lakes whose **name** is in a given list, then we can use the IN operator specifying an attribute name as in **name IN ('Lake Baikal','Lake Huron','Lake Onega','Lake Victoria')**

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=1000&filter-lang=cql-text&filter=name IN ('Lake Baikal','Lake Huron','Lake Onega','Lake Victoria')
```

Response:

The screenshot shows the pygeoapi interface at [> localhost:5000/collections/lakes/items?limit=1000&filter=name%20IN%20\(%27Lake%20Baikal%27,%27Lake%20Huron%27,%27Lake%20Onega%27,%27Lake%20Victoria%27\)](http://localhost:5000/collections/lakes/items?limit=1000&filter=name%20IN%20(%27Lake%20Baikal%27,%27Lake%20Huron%27,%27Lake%20Onega%27,%27Lake%20Victoria%27)). The page title is "Large Lakes". A map of the world highlights major lakes. To the right is a table of lake items:

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wikipedia.org/wiki/Lake_Baikal
6	admin-0	Lake	Lake Victoria	https://en.wikipedia.org/wiki/Lake_Victoria
18	None	Lake	Lake Onega	https://en.wikipedia.org/wiki/Lake_Onega
23	admin-0	Lake	Lake Huron	https://en.wikipedia.org/wiki/Lake_Huron

Below the map is a warning: "Warning: Higher limits not recommended!". A dropdown menu shows "Limit: 1,000".

- If the requirement is to get all the lakes from the collection except the ones specified in the list then **name NOT IN ('Lake Baikal','Lake Huron','Lake Onega','Lake Victoria')** will serve our purpose.

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=1000&filter=lang=cql-text&filter=name NOT IN ('Lake Baikal','Lake Huron','Lake Onega','Lake Victoria')
```

Response:

The screenshot shows the pygeoapi interface at [> localhost:5000/collections/lakes/items?limit=1000&filter=name%20NOT%20IN%20\(%27Lake%20Baikal%27,%27Lake%20Huron%27,%27Lake%20Onega%27,%27Lake%20Victoria%27\)](http://localhost:5000/collections/lakes/items?limit=1000&filter=name%20NOT%20IN%20(%27Lake%20Baikal%27,%27Lake%20Huron%27,%27Lake%20Onega%27,%27Lake%20Victoria%27)). The page title is "Large Lakes". A map of the world highlights major lakes. To the right is a table of lake items:

id	admin	featureclass	name	name_alt
1	None	Lake	Lake Winnipeg	https://en.wikipedia.org/wiki/Lake_Winnipeg
2	None	Lake	Great Slave Lake	https://en.wikipedia.org/wiki/Great_Slave_Lake
3	admin-0	Lake	L. Ontario	https://en.wikipedia.org/wiki/Lake_Ontario
4	admin-0	Lake	L. Erie	https://en.wikipedia.org/wiki/Lake_Erie
5	admin-0	Lake	Lake Superior	https://en.wikipedia.org/wiki/Lake_Superior
7	None	Lake	Lake Ladoga	https://en.wikipedia.org/wiki/Lake_Ladoga
8	None	Lake	Balqash Koli	Lake Balkhash

Below the map is a warning: "Warning: Higher limits not recommended!". A dropdown menu shows "Limit: 1,000".

9	admin-0	Lake	Lake Tanganyika	https://en.wi...
10	admin-0	Lake	Lake Malawi	Lake Nyasa
11	None	Lake	Aral Sea	https://en.wi...
12	None	Lake	Vänern	None
13	None	Lake	Lake Okeechobee	https://en.wi...
14	None	Lake	Lago de Nicaragua	https://en.wi...
15	None	Lake	Lake Tana	https://en.wi...
16	None	Lake	Lago Titicaca	https://en.wi...
17	None	Lake	Lake Winnipegosis	https://en.wi...
19	None	Lake	Great Salt Lake	https://en.wi...
Okeechobee				
14	None	Lake	Lago de Nicaragua	https://en.wi...
15	None	Lake	Lake Tana	https://en.wi...
16	None	Lake	Lago Titicaca	https://en.wi...
17	None	Lake	Lake Winnipegosis	https://en.wi...
19	None	Lake	Great Salt Lake	https://en.wi...
20	None	Lake	Great Bear Lake	https://en.wi...
21	None	Lake	Lake Athabasca	https://en.wi...
22	None	Lake	Reindeer Lake	https://en.wi...
24	admin-0	Lake	Lake Michigan	https://en.wi...

5.4.5 Combination filters

The CQL extension on pygeoapi is eligible to support filters that are a combination of more than one simple query filters.

The logical operators are: AND, OR

- To extract all the lakes whose id is less than 5 and name starts with ‘Lake’ then the combination of two filters can be formed as **id<5 AND name LIKE “Lake%”**

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=100&filter-lang=cql-text&filter=id<5 AND name LIKE "Lake%"
```

Response:

The screenshot shows the pygeoapi interface for a 'Large Lakes' collection. On the left is a world map with several large blue areas representing lakes. On the right is a table with two rows of data.

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wi...
1	None	Lake	Lake Winnipeg	https://en.wi...

Items in this collection.

Warning: Higher limits not recommended!

Limit:

- Furthermore, if a lake has an admin and its id is greater than 5 or its name contains ‘lake’ string irrespective of letter case, then the complex CQL filter query will be like: **admin IS NOT NULL AND id>5 OR name ILIKE “%lake%**

Requested API:

```
http://localhost:5000/collections/lakes/items?limit=100&filter=lang=cql-text&
filter=admin IS NOT NULL AND id>5 OR name ILIKE "%lake%"
```

Response:

The screenshot shows the pygeoapi interface for a 'Large Lakes' collection. On the left is a world map with several large blue areas representing lakes. On the right is a table with six rows of data.

id	admin	featureclass	name	name_alt
5	admin-0	Lake	Lake Superior	https://en.wi...
6	admin-0	Lake	Lake Victoria	https://en.wi...
9	admin-0	Lake	Lake Tanganyika	https://en.wi...
10	admin-0	Lake	Lake Malawi	Lake Nyasa
23	admin-0	Lake	Lake Huron	https://en.wi...
24	admin-0	Lake	Lake Michigan	https://en.wi...

Items in this collection.

Warning: Higher limits not recommended!

Limit:

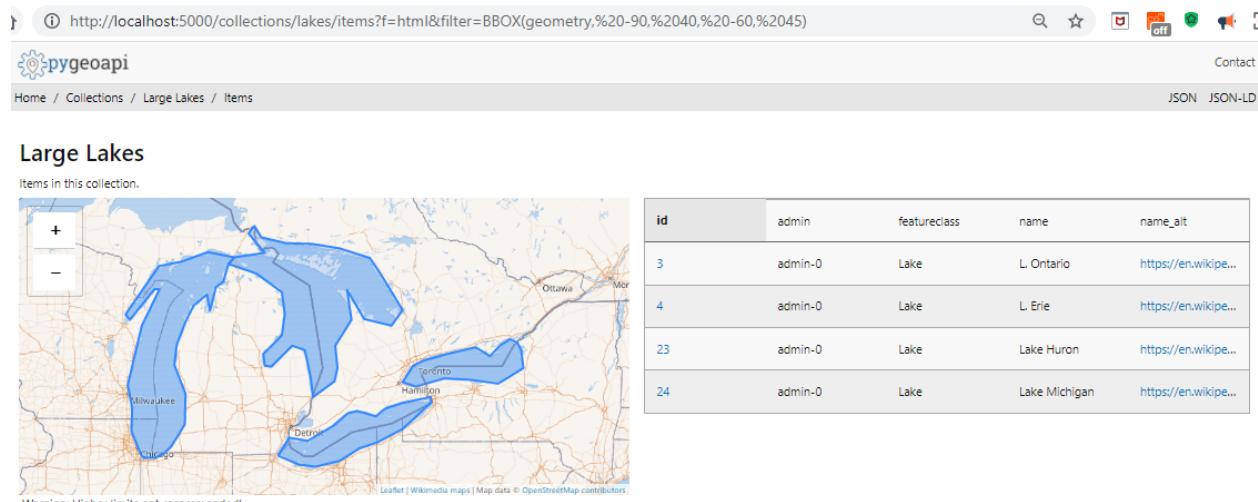
5.4.6 Spatial filters

- CQL provides a full set of geometric filter capabilities. Say, for example, if we want to display only the lakes that intersect the (-90,40,-60,45) bounding box. The filter will be **BBOX(geometry, -90, 40, -60, 45)**

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter-lang=cql-text&filter=BBOX(geometry, -90, 40, -60, 45)
```

Response:



The screenshot shows the pygeoapi interface for the 'Large Lakes' collection. At the top, there's a navigation bar with links for Home, Collections, Large Lakes, and Items. On the right, there are links for Contact, JSON, and JSON-LD. Below the navigation, the title 'Large Lakes' is displayed, followed by a sub-header 'Items in this collection.' To the left is a map of the Great Lakes area, specifically the Great Lakes and parts of the St. Lawrence River, with a blue polygon highlighting the bounding box filter. The map includes labels for cities like Milwaukee, Chicago, Detroit, and Toronto, and towns like Hamilton. A legend on the left side of the map shows zoom levels (+ and -). Below the map, a warning message reads 'Warning: Higher limits not recommended!' and a dropdown menu shows 'Limit: 10 (default)'. To the right of the map is a table listing four items:

id	admin	featureclass	name	name_alt
3	admin-0	Lake	L. Ontario	https://en.wikipedia.org/wiki/Lake_Ontario
4	admin-0	Lake	L. Erie	https://en.wikipedia.org/wiki/Lake_Erie
23	admin-0	Lake	Lake Huron	https://en.wikipedia.org/wiki/Lake_Huron
24	admin-0	Lake	Lake Michigan	https://en.wikipedia.org/wiki/Lake_Michigan

- Conversely, we can select the states that do not intersect the bounding box with the filter: **DISJOINT(the_geom, POLYGON((-90 40, -90 45, -60 45, -60 40, -90 40)))**

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter-lang=cql-text&filter=DISJOINT(the_geom, POLYGON((-90 40, -90 45, -60 45, -60 40, -90 40)))
```

Response:

The screenshot shows a web browser displaying the pygeoapi interface. The URL is [http://localhost:5000/collections/lakes/items?f=html&limit=5&filter=DISJOINT\(geometry,%20POLYGON\(\(-90%2040,%20-90%2045,%20-...](http://localhost:5000/collections/lakes/items?f=html&limit=5&filter=DISJOINT(geometry,%20POLYGON((-90%2040,%20-90%2045,%20-...))). The page title is "Large Lakes". The left side features a map of the world with blue shaded areas representing large lakes. Below the map is a legend with zoom controls (+, -, fit). The right side displays a table of lake data:

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wi...
1	None	Lake	Lake Winnipeg	https://en.wi...
2	None	Lake	Great Slave Lake	https://en.wi...
5	admin-0	Lake	Lake Superior	https://en.wi...
6	admin-0	Lake	Lake Victoria	https://en.wi...

- If needed to extract the information of a lake that contains a particular geometry. Then **CONTAINS(geometry, POLYGON((108.58 54.19, 108.37 54.04, 108.48 53.94, 108.77 54.01, 108.77 54.11, 108.58 54.19)))** will return the feature that contains a polygon of specified coordinates.

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter=lang=cql-text&
filter=CONTAINS(geometry, POLYGON((108.58 54.19, 108.37 54.04, 108.48 53.94,
108.77 54.01, 108.77 54.11, 108.58 54.19)))
```

Response:

The screenshot shows the same pygeoapi interface as before, but with a different filter applied. The URL is now [http://localhost:5000/collections/lakes/items?f=html&filter=CONTAINS\(geometry,POLYGON\(\(108.58%2054.19,%20108.37%2054.04,%20108.48%2053.94,%20108.77%2054.01,%20108.77%2054.11,%20108.58%2054.19\)\)\)](http://localhost:5000/collections/lakes/items?f=html&filter=CONTAINS(geometry,POLYGON((108.58%2054.19,%20108.37%2054.04,%20108.48%2053.94,%20108.77%2054.01,%20108.77%2054.11,%20108.58%2054.19)))). The map now highlights a specific region around Lake Baikal in Russia. The table data remains the same as in the previous screenshot:

id	admin	featureclass	name	name_alt
0	None	Lake	Lake Baikal	https://en.wi...

- But if needed to extract the information of lakes that are within a particular geometry. Then **WITHIN(geometry,POLYGON((-112.32 49.83, -94.21 49.83, -94.21 59.97, -112.32 59.97, -112.32 49.83)))** will return the features that are within a polygon of specified coordinates.

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&filter-lang=cql-text&
filter=WITHIN(geometry,POLYGON((-112.32 49.83, -94.21 49.83, -94.21 59.97,
-112.32 59.97, -112.32 49.83)))
```

Response:

The screenshot shows a web browser displaying the pygeoapi interface. The URL is [http://localhost:5000/collections/lakes/items?f=html&filter=WITHIN\(geometry,POLYGON\(\(-112.32%2049.83,%20-94.21%2049.83,%20-94.21%2059.97,%20-112.32%2059.97,%20-112.32%2049.83\)\)\)](http://localhost:5000/collections/lakes/items?f=html&filter=WITHIN(geometry,POLYGON((-112.32%2049.83,%20-94.21%2049.83,%20-94.21%2059.97,%20-112.32%2059.97,%20-112.32%2049.83)))).

Map View: A Leaflet map of Canada highlights several large lakes, including Lake Winnipeg, Lake Winnipegosis, Lake Athabasca, and Reindeer Lake. Labels for Edmonton and Calgary are visible. A legend on the left shows zoom controls (+, -, x). Below the map is a warning: "Warning: Higher limits not recommended!" and a dropdown limit selector set to "10 (default)".

Table View: A table lists the filtered lake items.

id	admin	featureclass	name	name_alt
1	None	Lake	Lake Winnipeg	https://en.wikipedia.org/wiki/Lake_Winnipeg
17	None	Lake	Lake Winnipegosis	https://en.wikipedia.org/wiki/Lake_Winnipegosis
21	None	Lake	Lake Athabasca	https://en.wikipedia.org/wiki/Lake_Athabasca
22	None	Lake	Reindeer Lake	https://en.wikipedia.org/wiki/Reindeer_Lake

- To filter all the lakes that lies beyond 10000 meters from a location (-85 75) but its id should be between 15 and 25. Then the query filter can be **BEYOND(geometry,POINT(-85 75),10000,meters)** AND **id BETWEEN 15 AND 25**

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&limit=5&filter-lang=cql-text&
filter=BEYOND(geometry,POINT(-85 75),10000,meters) AND id BETWEEN 15 AND 25
```

Response:

The screenshot shows a web browser displaying the pygeoapi interface. The URL is [http://localhost:5000/collections/lakes/items?f=html&limit=5&filter=BEYOND\(geometry,POINT\(-85%2075\),10000,meters\)%20AND%20id%20BETWEEN%2015%20AND%2025](http://localhost:5000/collections/lakes/items?f=html&limit=5&filter=BEYOND(geometry,POINT(-85%2075),10000,meters)%20AND%20id%20BETWEEN%2015%20AND%2025).

Map View: A Leaflet map of the world highlights several large lakes, including Lake Tana, Lago Titicaca, Lake Winnipegos, Lake Onega, and Great Salt Lake.

Table View: A table lists the filtered lake items.

id	admin	featureclass	name	name_alt
15	None	Lake	Lake Tana	https://en.wikipedia.org/wiki/Lake_Tana
16	None	Lake	Lago Titicaca	https://en.wikipedia.org/wiki/Lago_Titicaca
17	None	Lake	Lake Winnipegos	https://en.wikipedia.org/wiki/Lake_Winnipegosis
18	None	Lake	Lake Onega	https://en.wikipedia.org/wiki/Lake_Onega
19	None	Lake	Great Salt Lake	https://en.wikipedia.org/wiki/Great_Salt_Lake

- But if to filter all the lakes that lies within 10000 meters from a location (-85 75) but its id should be between 15

and 25. Then the query filter can be **DWITHIN(geometry,POINT(-85 75),10000,meters) AND id BETWEEN 15 AND 25**

Requested API:

```
http://localhost:5000/collections/lakes/items?f=html&limit=5&filter-lang=cql-text&filter=DWITHIN(geometry,POINT(-85 75),10000,meters) AND id BETWEEN 15 AND 25
```

Response:

The screenshot shows the pygeoapi interface with a search bar containing the URL. Below it, a navigation bar includes 'Home', 'Collections', 'Large Lakes', and 'Items'. The main content area is titled 'Large Lakes' and displays a message: 'Items in this collection.' followed by 'No items'. A note at the bottom states '**No such lakes found**'. A list of geometric predicates is also present.

Large Lakes

Items in this collection.

No items

***No such lakes found*

The full list of geometric predicates are: EQUALS, DISJOINT, INTERSECTS, TOUCHES, CROSSES, WITHIN, CONTAINS, OVERLAPS, RELATE, DWITHIN, BEYOND

The CQL extension on pygeoapi supports all the above geometric predicates to perform spatial filters on any feature collection.

5.4.7 Temporal filters

- Get all the features whose time value is before a point in time such as **datetime BEFORE 2001-10-30T14:24:54Z**

Requested API:

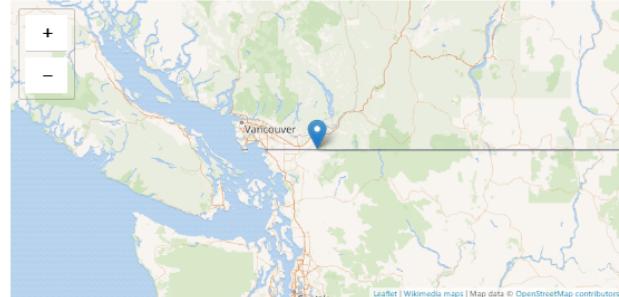
```
http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=datetime BEFORE 2001-10-30T14:24:54Z
```

Response:

The screenshot shows the pygeoapi interface with a search bar containing the URL. Below it, a navigation bar includes 'Home', 'Collections', 'Observations', and 'Items'. The main content area is titled 'Observations' and displays a message: 'Items in this collection.' A map of Vancouver is shown with a blue marker indicating a location. To the right is a table with one row of data.

Observations

Items in this collection.



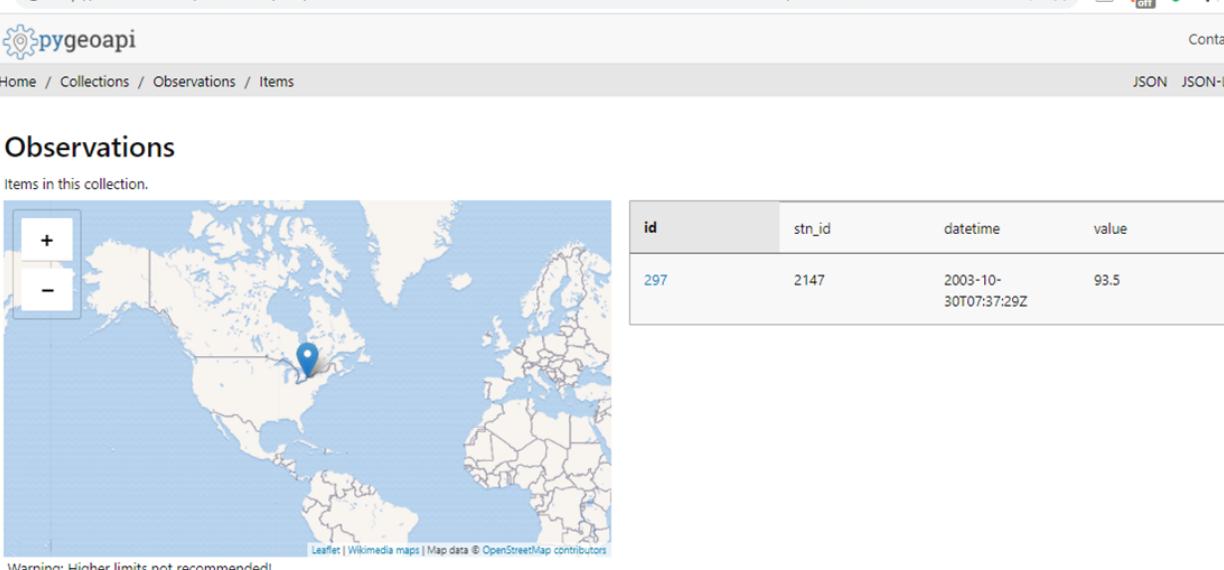
ID	stn_id	datetime	value
964	604	2000-10-30T14:24:54Z	99.9

- Get all the features whose time value is during a time period such as **datetime DURING 2003-01-01T00:00:00Z/2005-01-01T00:00:00Z**

Requested API:

```
http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=datetime DURING 2003-01-01T00:00:00Z/2005-01-01T00:00:00Z
```

Response:



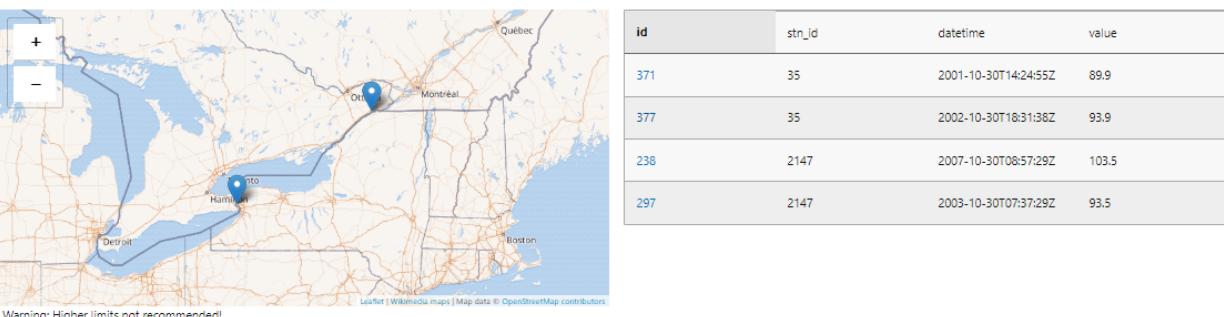
id	str_id	datetime	value
297	2147	2003-10-30T07:37:29Z	93.5

- Get all the features whose time value is after a point in time such as **datetime AFTER 2001-10-30T14:24:54Z**

Requested API:

```
http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=datetime AFTER 2001-10-30T14:24:54Z
```

Response:



id	str_id	datetime	value
371	35	2001-10-30T14:24:55Z	89.9
377	35	2002-10-30T18:31:38Z	93.9
238	2147	2007-10-30T08:57:29Z	103.5
297	2147	2003-10-30T07:37:29Z	93.5

- Get all the features whose time value is during or after a time period such as **datetime DURING OR AFTER 2003-01-01T00:00:00Z/2005-01-01T00:00:00Z**

Requested API:

```
http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=datetime DURING OR AFTER 2003-01-01T00:00:00Z/2005-01-01T00:00:00Z
```

Response:

The screenshot shows the pygeoapi interface. At the top, there is a header bar with the URL `http://localhost:5000/collections/obs/items?f=html&filter-lang=cql-text&filter=datetime DURING OR AFTER 2003-01-01T00:00:00Z/2005-01-01T00:00:00Z`, a search icon, a star icon, a copy icon, a refresh icon, and a contact link. Below the header is a navigation bar with links for Home, Collections, Observations, and Items. On the right of the navigation bar are JSON and JSON-LD links. The main content area has a title "Observations" and a subtitle "Items in this collection." Below this is a map of Western New York, specifically the area around Niagara Falls, Chippawa, Tonawanda, and Fort Erie. A blue location pin is placed near the Niagara River. The map includes street names like Main Street, Water Street, and Gandy's Road, and various landmarks. To the right of the map is a table with the following data:

id	stn_id	datetime	value
238	2147	2007-10-30T08:57:29Z	103.5
297	2147	2003-10-30T07:37:29Z	93.5

At the bottom left of the map area, there is a warning: "Warning: Higher limits not recommended!" and a dropdown menu set to "10 (default)".

ADMINISTRATION

Now that you have pygeoapi installed and a basic configuration setup, it's time to complete the administrative steps required before starting up the server. The remaining steps are:

- create OpenAPI document
- set system environment variables

6.1 Creating the OpenAPI document

The OpenAPI document is a YAML configuration which is generated from the pygeoapi configuration, and describes the server information, endpoints, and parameters.

To generate the OpenAPI document, run the following:

```
pygeoapi generate-openapi-document -c /path/to/my-pygeoapi-config.yml
```

This will dump the OpenAPI document as YAML to your system's stdout. To save to a file on disk, run:

```
pygeoapi generate-openapi-document -c /path/to/my-pygeoapi-config.yml > /path/to/my-  
↳pygeoapi-openapi.yml
```

Note: The OpenAPI document provides detailed information on query parameters, and dataset property names and their data types. Whenever you make changes to your pygeoapi configuration, always refresh the accompanying OpenAPI document.

See also:

OpenAPI for more information on pygeoapi's OpenAPI support

6.2 Verifying configuration files

To ensure your YAML configurations are correctly formatted, you can use any YAML validator, or try the Python one-liner per below:

```
python -c 'import yaml, sys; yaml.safe_load(sys.stdin)' < /path/to/my-pygeoapi-config.  
↳yml  
python -c 'import yaml, sys; yaml.safe_load(sys.stdin)' < /path/to/my-pygeoapi-  
↳openapi.yml
```

6.3 Setting system environment variables

Now, let's set our system environment variables.

In UNIX:

```
export PYGEOAPI_CONFIG=/path/to/my-pygeoapi-config.yml  
export PYGEOAPI_OPENAPI=/path/to/my-pygeoapi-openapi.yml
```

In Windows:

```
set PYGEOAPI_CONFIG=/path/to/my-pygeoapi-config.yml  
set PYGEOAPI_OPENAPI=/path/to/my-pygeoapi-openapi.yml
```

6.4 Summary

At this point you are ready to run the server. Let's go!

RUNNING

Now we are ready to start up pygeoapi.

7.1 pygeoapi serve

The `pygeoapi serve` command starts up an instance using Flask as the default server. pygeoapi can be served via Flask [WSGI](#) or Starlette [ASGI](#).

Since pygeoapi is a Python API at its core, it can be served via numerous web server scenarios.

Note: Changes to either of the pygeoapi or OpenAPI configurations requires a server restart (configurations are loaded once at server startup for performance).

7.1.1 Flask WSGI

Web Server Gateway Interface (WSGI) is a standard for how web servers communicate with Python applications. By having a WSGI server, HTTP requests are processed into threads/processes for better performance. Flask is a WSGI implementation which pygeoapi utilizes to communicate with the core API.

```
HTTP request <--> Flask (pygeoapi/flask_app.py) <--> pygeoapi API (pygeoapi/api.py)
```

The Flask WSGI server can be run as follows:

```
pygeoapi serve --flask
pygeoapi serve # uses Flask by default
```

7.1.2 Starlette ASGI

Asynchronous Server Gateway Interface (ASGI) is standard interface between async-capable web servers, frameworks, and applications written in Python. ASGI provides the benefits of WSGI as well as asynchronous capabilities. Starlette is an ASGI implementation which pygeoapi utilizes to communicate with the core API in asynchronous mode.

```
HTTP request <--> Starlette (pygeoapi/starlette_app.py) <--> pygeoapi API (pygeoapi/
˓→api.py)
```

The Flask WSGI server can be run as follows:

```
pygeoapi serve --starlette
```

7.2 Running in production

Running `pygeoapi serve` in production is not recommended or advisable. Preferred options are described below.

See also:

[Docker](#) for container-based production installations.

7.2.1 Apache and mod_wsgi

Deploying pygeoapi via `mod_wsgi` provides a simple approach to enabling within Apache.

To deploy with `mod_wsgi`, your Apache instance must have `mod_wsgi` enabled within Apache. At this point, set up the following Python WSGI script:

```
import os

os.environ['PYGEOAPI_CONFIG'] = '/path/to/my-pygeoapi-config.yml'
os.environ['PYGEOAPI_OPENAPI'] = '/path/to/my-pygeoapi-openapi.yml'

from pygeoapi.flask_app import APP as application
```

Now configure in Apache:

```
WSGIProcessGroup pygeoapi processes=1 threads=1
WSGIScriptAlias /pygeoapi /path/to/pygeoapi.wsgi process-group=pygeoapi application-
group=%{GLOBAL}

<Location /pygeoapi>
    Header set Access-Control-Allow-Origin "*"
</Location>
```

7.2.2 Gunicorn

Gunicorn (for UNIX) is one of several Python WSGI HTTP servers that can be used for production environments.

```
HTTP request --> WSGI or ASGI server (gunicorn) <--> Flask or Starlette (pygeoapi/
flask_app.py or pygeoapi/starlette_app.py) <--> pygeoapi API
```

Note: Gunicorn is as easy to install as `pip install gunicorn`

Note: For a complete list of WSGI server implementations, see the [WSGI server list](#).

7.2.3 Gunicorn and Flask

Gunicorn and Flask is simple to run:

```
gunicorn pygeoapi.flask_app:APP
```

Note: For extra configuration parameters like port binding, workers, and logging please consult the [Gunicorn settings](#).

7.2.4 Gunicorn and Starlette

Running Gunicorn with Starlette requires the [Uvicorn](#) which provides async capabilities along with Gunicorn. Uvicorn includes a Gunicorn worker class allowing you to run ASGI applications, with all of Uvicorn's performance benefits, while also giving you Gunicorn's fully-featured process management.

is simple to run from the command, e.g:

```
gunicorn pygeoapi.starlette_app:app -w 4 -k uvicorn.workers.UvicornWorker
```

Note: Uvicorn is as easy to install as `pip install guvicorn`

7.3 Summary

pygeoapi has many approaches for deploying depending on your requirements. Choose one that works for you and modify accordingly.

Note: Additional approaches are welcome and encouraged; see [Contributing](#) for more information on how to contribute to and improve the documentation

DOCKER

pygeoapi provides an official Docker image which is made available on the [geopython Docker Hub](#). Additional Docker examples can be found in the [pygeoapi GitHub repository](#), each with sample configurations, test data, deployment scenarios and provider backends.

The [pygeoapi demo server](#) runs various services from Docker images which also serve as useful examples.

Note: Both Docker and Docker Compose are required on your system to run pygeoapi images.

8.1 The basics

The official pygeoapi Docker image will start a pygeoapi Docker container using Gunicorn on internal port 80.

To run with the default built-in configuration and data:

```
docker run -p 5000:80 -it geopython/pygeoapi run  
# or simply  
docker run -p 5000:80 -it geopython/pygeoapi
```

...then browse to <http://localhost:5000>

You can also run all unit tests to verify:

```
docker run -it geopython/pygeoapi test
```

8.2 Overriding the default configuration

Normally you would override the `default.config.yml` with your own pygeoapi configuration. This can be done via Docker Volume Mapping.

For example, if your config is in `my.config.yml`:

```
docker run -p 5000:80 -v $(pwd)/my.config.yml:/pygeoapi/local.config.yml -it  
geopython/pygeoapi
```

For a cleaner approach, You can use `docker-compose` as per below:

```
version: "3"
services:
  pygeoapi:
    image: geopython/pygeoapi:latest
    volumes:
      - ./my.config.yml:/pygeoapi/local.config.yml
```

Or you can create a Dockerfile extending the base image and **copy** in your configuration:

```
FROM geopython/pygeoapi:latest
COPY ./my.config.yml /pygeoapi/local.config.yml
```

A corresponding example can be found in https://github.com/geopython/demo.pygeoapi.io/tree/master/services/pygeoapi_master

8.3 Deploying on a sub-path

By default the pygeoapi Docker image will run from the root path (/). If you need to run from a sub-path and have all internal URLs properly configured, you can set the `SCRIPT_NAME` environment variable.

For example to run with `my.config.yml` on `http://localhost:5000/mypygeoapi`:

```
docker run -p 5000:80 -e SCRIPT_NAME='/mypygeoapi' -v $(pwd)/my.config.yml:/pygeoapi/
→local.config.yml -it geopython/pygeoapi
```

...then browse to **http://localhost:5000/mypygeoapi**

Below is a corresponding docker-compose approach:

```
version: "3"
services:
  pygeoapi:
    image: geopython/pygeoapi:latest
    volumes:
      - ./my.config.yml:/pygeoapi/local.config.yml
    ports:
      - "5000:80"
    environment:
      - SCRIPT_NAME=/pygeoapi
```

A corresponding example can be found in https://github.com/geopython/demo.pygeoapi.io/tree/master/services/pygeoapi_master

8.4 Summary

Docker is an easy and reproducible approach to deploying systems.

Note: Additional approaches are welcome and encouraged; see [Contributing](#) for more information on how to contribute to and improve the documentation

TAKING A TOUR OF PYGEOAPI

At this point, you've installed pygeoapi, set configurations and started the server.

pygeoapi's default configuration comes setup with two simple vector datasets, a STAC collection and a sample process. Note that these resources are straightforward examples of pygeoapi's baseline functionality, designed to get the user up and running with as little barriers as possible.

Let's check things out. In your web browser, go to <http://localhost:5000>

9.1 Overview

All pygeoapi URLs have HTML and JSON representations. If you are working through a web browser, HTML is always returned as the default, whereas if you are working programmatically, JSON is always returned.

To explicitly ask for HTML or JSON, simply add `f=html` or `f=json` to any URL accordingly.

Each web page provides breadcrumbs for navigating up/down the server's data. In addition, the upper right of the UI always has JSON and JSON-LD links to provide you with the current page in JSON if desired.

9.2 Landing page

<http://localhost:5000>

The landing page provides a high level overview of the pygeoapi server (contact information, licensing), as well as specific sections to browse data, processes and geospatial files.

9.3 Collections

<http://localhost:5000/collections>

The collections page displays all the datasets available on the pygeoapi server with their title and abstract. Let's drill deeper into a given dataset.

9.4 Collection information

<http://localhost:5000/collections/obs>

Let's drill deeper into a given dataset. Here we can see the `obs` dataset is described along with related links (other related HTML pages, dataset download, etc.).

The ‘View’ section provides the default to start browsing the data.

The ‘Queryables’ section provides a link to the dataset’s properties.

9.5 Vector data

9.5.1 Collection queryables

<http://localhost:5000/collections/obs/queryables>

The queryables endpoint provides a list of queryable properties and their associated datatypes.

9.5.2 Collection items

<http://localhost:5000/collections/obs/items>

This page displays a map and tabular view of the data. Features are clickable on the interactive map, allowing the user to drill into more information about the feature. The table also allows for drilling into a feature by clicking the link in a given table row.

Let's inspect the feature close to [Toronto, Ontario, Canada](#).

9.5.3 Collection item

<http://localhost:5000/collections/obs/items/297>

This page provides an overview of the feature and its full set of properties, along with an interactive map.

See also:

[Publishing vector data to OGC API - Features](#) for more OGC API - Features request examples.

9.6 Raster data

9.6.1 Collection coverage domainset

This page provides information on a collection coverage spatial properties and axis information.

<http://localhost:5000/collections/gdps-temperature/coverage/domainset>

9.6.2 Collection coverage rangetype

This page provides information on a collection coverage rangetype (bands) information.

<http://localhost:5000/collections/gdps-temperature/coverage/rangetype>

9.6.3 Collection coverage data

This page provides a coverage in CoverageJSON format.

<http://localhost:5000/collections/gdps-temperature/coverage>

See also:

Publishing raster data to OGC API - Coverages for more OGC API - Coverages request examples.

9.7 SpatioTemporal Assets

<http://localhost:5000/stac>

This page provides a Web Accessible Folder view of raw geospatial data files. Users can navigate and click to browse directory contents or inspect files. Clicking on a file will attempt to display the file's properties/metadata, as well as an interactive map with a footprint of the spatial extent of the file.

See also:

Publishing files to a SpatioTemporal Asset Catalog for more STAC request examples.

9.8 Processes

The processes page provides a list of process integrated onto the server, along with a name and description.

Todo: Expand with more info once OAProc HTML is better flushed out.

See also:

Publishing processes via OGC API - Processes for more OGC API - Processes request examples.

9.9 API Documentation

<http://localhost:5000/openapi>

<http://localhost:5000/openapi?f=json>

The API documentation links provide a [Swagger](#) page of the API as a tool for developers to provide example request/response/query capabilities. A JSON representation is also provided.

See also:

[OpenAPI](#)

9.10 Conformance

<http://localhost:5000/conformance>

The conformance page provides a list of URLs corresponding to the OGC API conformance classes supported by the pygeoapi server. This information is typically useful for developers and client applications to discover what is supported by the server.

OPENAPI

The [OpenAPI specification](#) is an open specification for RESTful endpoints. OGC API specifications leverage OpenAPI to describe the API in great detail with developer focus.

The RESTful structure and payload are defined using JSON or YAML file structures (pygeoapi uses YAML). The basic structure is described here: <https://swagger.io/docs/specification/basic-structure/>

The official OpenAPI specification can be found on [GitHub](#). pygeoapi supports OpenAPI version 3.0.2.

As described in [Administration](#), the pygeoapi OpenAPI document is automatically generated based on the configuration file:

The API is accessible at the `/openapi` endpoint, providing a Swagger-based webpage of the API description..

See also:

the pygeoapi demo OpenAPI/Swagger endpoint at <https://demo.pygeoapi.io/master/openapi>

10.1 Using OpenAPI

Accessing the Swagger webpage we have the following structure:

The screenshot shows the Swagger UI for the pygeoapi Demo instance. At the top, it displays the title "pygeoapi Demo instance - running latest GitHub version" with a version of 3.0.2 and an OAS3 badge. Below the title, there's a brief description: "pygeoapi provides an API to geospatial data". Underneath, there are links for "Terms of service", "pygeoapi Development Team - Website", "Send email to pygeoapi Development Team", and "CC-BY 4.0 license". A "Servers" section shows the URL "https://demo.pygeoapi.io/master - pygeoapi provides an API to geospatial data". The main content area is organized into sections: "server", "obs", and "lakes". The "server" section lists various REST endpoints with their methods and descriptions. The "obs" section and "lakes" section each have three listed endpoints. On the right side of the interface, there's a link to "information: https://github.com/geopython/pygeoapi" with a dropdown arrow.

Notice that each dataset is represented as a RESTful endpoint under `collections`.

In this example we will test GET capability of data concerning windmills in the Netherlands. Let's start by accessing the service's dataset collections:

The screenshot shows the pygeoapi OpenAPI documentation interface. At the top left, it says "server" and "pygeoapi provides an API to geospatial data". At the top right, there's a link to "information: https://github.com/geopython/pygeoapi". Below this, there are several blue "GET" buttons with URLs: "/ API", "/api This document", and "/collections Feature Collections". The third button, "/collections", is highlighted with a red box. The "Feature Collections" section is expanded, showing a "Parameters" table with "No parameters" and a "Responses" table with a single entry for "Code 200" which has "Description successful operation" and "Links No links". At the bottom of the "Parameters" section is a blue "Execute" button, also highlighted with a red box.

The service collection metadata will contain a description of each collection:

The screenshot shows the API response for the "/collections" endpoint. It includes a "Curl" section with the command "curl -X GET "https://demo.pygeoapi.io/master/collections" -H "accept: */*" and a "Request URL" section with "https://demo.pygeoapi.io/master/collections". The "Server response" section is expanded, showing a "Code 200" entry under "Details". The "Response body" is a JSON object with various fields like "crs", "name", "title", "description", "keywords", "extents", and "links". A specific part of the "keywords" array is highlighted with a red box, showing values such as "Netherlands", "INSPIRE", "Windmills", "Heritage", "Holland", and "RD". At the bottom right of the "Response body" section is a "Download" button.

Here, we see that the `dutch_windmills` dataset is available. Next, let's obtain the specific metadata of the dataset:

dutch_windmills Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.

GET /collections/dutch_windmills Get feature collection metadata

Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.

Parameters

No parameters

Responses

Curl

```
curl -X GET "https://demo.pygeoapi.io/master/collections/dutch_windmills" -H "accept: */*"
```

Request URL

```
https://demo.pygeoapi.io/master/collections/dutch_windmills
```

Code **Details**

200 **Response body**

```
{
  "type": "FeatureCollection",
  "version": "1.0.0",
  "name": "dutch_windmills",
  "title": "Windmills within The Netherlands",
  "description": "Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.",
  "keywords": [
    "Netherlands",
    "INSPIRE",
    "Windmills",
    "Heritage",
    "Holland",
    "RD"
  ],
  "extents": [
    {
      "bbox": [
        50.75,
        3.37,
        53.47,
        7.21
      ]
    }
  ],
  "crs": [
    "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
  ],
  "features": []
}
```

Response headers

```
access-control-allow-origin: *
content-length: 1278
content-type: application/json
date: Sun, 14 Jul 2019 09:54:23 GMT
server: gunicorn/19.9.0
x-firefox-spdy: h2
x-powered-by: pygeoapi 0.6.0
```

We also see that the dataset has an `items` endpoint which provides all data, along with specific parameters for filtering, paging and sorting:

GET /collections/dutch_windmills/items Get Windmills within The Netherlands features

Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.

Parameters

Name	Description
f string (query)	The optional f parameter indicates the output format which the server shall provide as part of the response document. The default format is GeoJSON. <input type="button" value="json"/>
bbox array[number] (query)	The bbox parameter indicates the minimum bounding rectangle upon which to query the collection in WFS84 (minx, miny, maxx, maxy). <input type="button" value="Add item"/>
time string (query)	The time parameter indicates an RFC3339 formatted datetime (single, interval, open). <input type="text" value="time - The time parameter indicates an"/>
limit integer (query)	The optional limit parameter limits the number of items that are presented in the response document. Only items are counted that are on the first level of the collection in the response document. Nested objects contained within the explicitly requested items shall not be counted. Minimum = 1. Maximum = 10000. Default = 10. <input type="text" value="10"/>
sortby string (query)	The optional sortby parameter indicates the sort property and order on which the server shall present results in the response document using the convention <code>sortby=PROPERTY:X</code> , where PROPERTY is the sort property and X is the sort order (A is ascending, D is descending). Sorting by multiple properties is supported by providing a comma-separated list. <input type="text" value="sortby - The optional sortby parameter"/>
startindex integer (query)	The optional startindex parameter indicates the index within the result set from which the server shall begin presenting results in the response document. The first element has an index of 0 (default). <input type="text" value="0"/>

Execute

Responses

For each item in our dataset we have a specific identifier. Notice that the identifier is not part of the GeoJSON properties, but is provided as a GeoJSON root property of `id`.

Request URL

```
https://demo.pygeoapi.io/master/collections/dutch_windmills/items?f=json&limit=10&startindex=0
```

Server response

Code	Details
200	Response body <pre> "id": "Molens.1" }, "properties": { "gid": 1, "NAAM": "De Trouwe Waghter of Trouwe Wachter", "PLAATS": "Tienhoven", "CATEGORIE": "windmolen", "FUNCTIE": "poldermolen", "TYPE": "wipmolen", "STAAT": "bestaand", "RMONNUMMER": "26483", "TBGNUMMER": "00003", "INFOLINK": "https://zoeken.allermolens.nl/tenbruggencatenummer/00003", "THUMBNAIL": "https://images.memorix.nl/rce/thumb/350x350/9165dd5b-34b8-705d-0128-3196d2831677.jpg", "HDFUNCTIE": "poldermolen", "FOTOGRAAF": "Frank Terpstra", "FOTO_GROOT": "https://images.memorix.nl/rce/thumb/fullsize/9165dd5b-34b8-705d-0128-3196d2831677.jpg", "BOUWJAAR": "1832" }, "type": "Feature", "geometry": { "type": "Point", "coordinates": [5.057482816805334, 52.17198007919141] } } </pre>

This identifier can be used to obtain a specific item from the dataset using the `items{id}` endpoint as follows:

GET /collections/dutch_windmills/items/{id} Get Windmills within The Netherlands feature by id

Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.

Parameters

cancel

Name	Description
id * required string (path)	The id of a feature <input type="text" value="Molens.1"/>
f string (query)	The optional f parameter indicates the output format which the server shall provide as part of the response document. The default format is GeoJSON. <input type="text" value="json"/>

Execute

10.2 Summary

Using pygeoapi's OpenAPI and Swagger endpoints provides a useful user interface to query data, as well as for developers to easily understand pygeoapi when building downstream applications.

DATA PUBLISHING

Let's start working on integrating your data into pygeoapi. pygeoapi provides the capability to publish vector data, processes, and exposing filesystems of geospatial data.

11.1 Providers overview

A key component to data publishing is the pygeoapi provider framework. Providers allow for configuring data files, databases, search indexes, other APIs, cloud storage, to be able to return back data to the pygeoapi API framework in a plug and play fashion.

11.1.1 Publishing vector data to OGC API - Features

[OGC API - Features](#) provides geospatial data access functionality to vector data.

To add vector data to pygeoapi, you can use the dataset example in [Configuration](#) as a baseline and modify accordingly.

11.1.1.1 Providers

pygeoapi core feature providers are listed below, along with a matrix of supported query parameters.

Provider	properties	resulttype	bbox	datetime	sortby
CSV	✓	results/hits			
Elasticsearch	✓	results/hits	✓	✓	✓
GeoJSON	✓	results/hits			
MongoDB	✓	results	✓	✓	✓
OGR	✓	results/hits	✓		
PostgreSQL	✓	results/hits	✓		
SQLiteGPKG	✓	results/hits	✓		

Below are specific connection examples based on supported providers.

11.1.1.2 Connection examples

CSV

To publish a CSV file, the file must have columns for x and y geometry which need to be specified in `geometry` section of the provider definition.

```
providers:  
  - type: feature  
    name: CSV  
    data: tests/data/obs.csv  
    id_field: id  
    geometry:  
      x_field: long  
      y_field: lat
```

GeoJSON

To publish a GeoJSON file, the file must be a valid GeoJSON FeatureCollection.

```
providers:  
  - type: feature  
    name: GeoJSON  
    data: tests/data/file.json  
    id_field: id
```

Elasticsearch

Note: Elasticsearch 7 or greater is supported.

To publish an Elasticsearch index, the following are required in your index:

- indexes must be documents of valid GeoJSON Features
- index mappings must define the GeoJSON geometry as a `geo_shape`

```
providers:  
  - type: feature  
    name: Elasticsearch  
    data: http://localhost:9200/ne_110m_populated_places_simple  
    id_field: geonameid  
    time_field: datetimfield
```

OGR

Todo: add overview and requirements

MongoDB

Todo: add overview and requirements

```
providers:
  - type: feature
    name: MongoDB
    data: mongodb://localhost:27017/testdb
    collection: testplaces
```

PostgreSQL

Todo: add overview and requirements

```
providers:
  - type: feature
    name: PostgreSQL
    data:
      host: 127.0.0.1
      dbname: test
      user: postgres
      password: postgres
      search_path: [osm, public]
    id_field: osm_id
    table: hotosm_bdi_waterways
    geom_field: foo_geom
```

SQLiteGPKG

Todo: add overview and requirements

SQLite file:

```
providers:
  - type: feature
    name: SQLiteGPKG
    data: ./tests/data/ne_110m_admin_0_countries.sqlite
    id_field: ogc_fid
    table: ne_110m_admin_0_countries
```

GeoPackage file:

```
providers:
  - type: feature
    name: SQLiteGPKG
    data: ./tests/data/poi_portugal.gpkg
    id_field: osm_id
    table: poi_portugal
```

11.1.1.3 Data access examples

- list all collections - <http://localhost:5000/collections>
- overview of dataset - <http://localhost:5000/collections/foo>
- queryables - <http://localhost:5000/collections/foo/queryables>
- browse features - <http://localhost:5000/collections/foo/items>
- paging - <http://localhost:5000/collections/foo/items?startIndex=10&limit=10>
- CSV outputs - <http://localhost:5000/collections/foo/items?f=csv>
- query features (spatial) - <http://localhost:5000/collections/foo/items?bbox=-180,-90,180,90>
- query features (attribute) - <http://localhost:5000/collections/foo/items?propertyName=foo>
- query features (temporal) - <http://localhost:5000/collections/foo/items?datetime=2020-04-10T14:11:00Z>
- fetch a specific feature - <http://localhost:5000/collections/foo/items/123>

11.1.2 Publishing raster data to OGC API - Coverages

OGC API - Coverages provides geospatial data access functionality to raster data.

To add raster data to pygeoapi, you can use the dataset example in [Configuration](#) as a baseline and modify accordingly.

11.1.2.1 Providers

pygeoapi core feature providers are listed below, along with a matrix of supported query parameters.

Provider	rangeSubset	subset
rasterio	✓	✓

Below are specific connection examples based on supported providers.

11.1.2.2 Connection examples

rasterio

The `rasterio` provider plugin reads and extracts any data that rasterio is capable of handling.

```
providers:
  - type: coverage
    name: rasterio
    data: tests/data/CMC_glb_TMP_TGL_2_llatlon.15x.15_2020081000_P000.grib2
```

(continues on next page)

(continued from previous page)

```
options: # optional creation options
  DATA_ENCODING: COMPLEX_PACKING
format:
  name: GRIB2
  mimetype: application/x-grib2
```

11.1.2.3 Data access examples

- list all collections - <http://localhost:5000/collections>
- overview of dataset - <http://localhost:5000/collections/foo>
- coverage rangetype - <http://localhost:5000/collections/foo/coverage/rangetype>
- coverage domainset - <http://localhost:5000/collections/foo/coverage/domainset>
- coverage access via CoverageJSON (default) - <http://localhost:5000/collections/foo/coverage?f=json>
- coverage access via native format (as defined in provider.format.name) - <http://localhost:5000/collections/foo/coverage?f=GRIB2>
- coverage access with comma-separated rangeSubset - <http://localhost:5000/collections/foo/coverage?rangeSubset=1,3>
- coverage access with subsetting - [http://localhost:5000/collections/foo/coverage?subset=lat\(10,20\)&subset=long\(10,20\)](http://localhost:5000/collections/foo/coverage?subset=lat(10,20)&subset=long(10,20))

11.1.3 Publishing processes via OGC API - Processes

OGC API - Processes provides geospatial data processing functionality in a standards-based fashion (inputs, outputs). pygeoapi implements OGC API - Processes functionality by providing a plugin architecture, thereby allowing developers to implement custom processing workflows in Python.

A sample hello-world process is provided with the pygeoapi default configuration.

11.1.3.1 Configuration

```
processes:
  hello-world:
    processor:
      name: HelloWorld
```

11.1.3.2 Processing examples

- list all processes - <http://localhost:5000/processes>
- describe the hello-world process - <http://localhost:5000/processes/hello-world>
- show all jobs for the hello-world process - <http://localhost:5000/processes/hello-world/jobs>
- execute a job for the hello-world process - curl -X POST "http://localhost:5000/processes/hello-world/jobs" -H "Content-Type: application/json" -d "{\"inputs\":[{\"id\":\"name\", \"type\":\"text/plain\", \"value\":\"hi there2\"}]}"

- execute a job for the hello-world process with a raw response - curl -X POST "http://localhost:5000/processes/hello-world/jobs?response=raw" -H "Content-Type: application/json" -d "{\"inputs\": [{\"id\": \"name\", \"type\": \"text/plain\", \"value\": \"hi there2\"}]}"

Todo: add more examples once OAProc implementation is complete

11.1.4 Publishing files to a SpatioTemporal Asset Catalog

The [SpatioTemporal Asset Catalog \(STAC\)](#) specification provides an easy approach for describing geospatial assets. STAC is typically implemented for imagery and other raster data.

pygeoapi implements STAC as an geospatial file browser through the FileSystem provider, supporting any level of file/directory nesting/hierarchy.

Configuring STAC in pygeoapi is done by simply pointing the `data` provider property to the given directory and specifying allowed file types:

11.1.4.1 Connection examples

```
my-stac-resource:  
    type: stac-collection  
    ...  
    providers:  
        - type: stac  
          name: FileSystem  
          data: /Users/tomkralidis/Dev/data/gdps  
          file_types:  
              - .grib2
```

Note: `rasterio` and `fiona` are required for describing geospatial files.

11.1.4.2 Data access examples

- STAC root page - <http://localhost:5000/stac>

From here, browse the filesystem accordingly.

CUSTOMIZING PYGEOAPI: PLUGINS

In this section we will explain how pygeoapi provides plugin architecture for data providers, formatters and processes. Plugin development requires knowledge of how to program in Python as well as Python's package/module system.

12.1 Overview

pygeoapi provides a robust plugin architecture that enables developers to extend functionality. Infact, pygeoapi itself implements numerous formats, data providers and the process functionality as plugins.

The pygeoapi architecture supports the following subsystems:

- data providers
- output formats
- processes

The core pygeoapi plugin registry can be found in `pygeoapi.plugin.PLUGINS`.

Each plugin type implements its relevant base class as the API contract:

- data providers: `pygeoapi.provider.base`
- output formats: `pygeoapi.formatter.base`
- processes: `pygeoapi.process.base`

Todo: link PLUGINS to API doc

Plugins can be developed outside of the pygeoapi codebase and be dynamically loaded by way of the pygeoapi configuration. This allows your custom plugins to live outside pygeoapi for easier maintenance of software updates.

Note: It is recommended to store pygeoapi plugins outside of pygeoapi for easier software updates and package management

12.2 Example: custom pygeoapi vector data provider

Lets consider the steps for a vector data provider plugin (source code is located here: [Provider](#)).

12.2.1 Python code

The below template provides a minimal example (let's call the file `mycoolvectordata.py`):

```
from pygeoapi.provider.base import BaseProvider

class MyCoolVectorDataProvider(BaseProvider):
    """My cool vector data provider"""

    def __init__(self, provider_def):
        """Inherit from parent class"""

        BaseProvider.__init__(self, provider_def)

    def get_fields(self):

        # open dat file and return fields and their datatypes
        return {
            'field1': 'string',
            'field2': 'string'
        }

    def query(self, startindex=0, limit=10, resulttype='results',
              bbox=[], datetime=None, properties=[], sortby=[]):

        # open data file (self.data) and process, return
        return {
            'type': 'FeatureCollection',
            'features': [
                {
                    'type': 'Feature',
                    'id': '371',
                    'geometry': {
                        'type': 'Point',
                        'coordinates': [ -75, 45 ]
                    },
                    'properties': {
                        'stn_id': '35',
                        'datetime': '2001-10-30T14:24:55Z',
                        'value': '89.9'
                    }
                }
            ]
        }
```

For brevity, the above code will always return the single feature of the dataset. In reality, the plugin developer would connect to a data source with capabilities to run queries and return a relevant result set, as well as implement the `get` method accordingly. As long as the plugin implements the API contract of its base provider, all other functionality is left to the provider implementation.

Each base class documents the functions, arguments and return types required for implementation.

12.2.2 Connecting to pygeoapi

The following methods are options to connect the plugin to pygeoapi:

Option 1: Update in core pygeoapi:

- copy `mycoolvectordata.py` into `pygeoapi/provider`
- update the plugin registry in `pygeoapi/plugin.py`:`PLUGINS['provider']` with the plugin's shortname (say `MyCoolVectorData`) and dotted path to the class (i.e. `pygeoapi.provider.mycoolvectordata.MyCoolVectorDataProvider`)
- specify in your dataset provider configuration as follows:

```
providers:
  - type: feature
    name: MyCoolVectorData
    data: /path/to/file
    id_field: stn_id
```

Option 2: implement outside of pygeoapi and add to configuration (recommended)

- create a Python package of the `mycoolvectordata.py` module (see [Cookiecutter](#) as an example)
- install your Python package onto your system (`python setup.py install`). At this point your new package should be in the `PYTHONPATH` of your pygeoapi installation
- specify in your dataset provider configuration as follows:

```
providers:
  - type: feature
    name: mycooldatapackage.mycoolvectordata.MyCoolVectorDataProvider
    data: /path/to/file
    id_field: stn_id
```

BEGIN

12.3 Example: custom pygeoapi raster data provider

Lets consider the steps for a raster data provider plugin (source code is located here: [Provider](#)).

12.3.1 Python code

The below template provides a minimal example (let's call the file `mycoolrasterdata.py`):

```
from pygeoapi.provider.base import BaseProvider

class MyCoolRasterDataProvider(BaseProvider):
    """My cool raster data provider"""

    def __init__(self, provider_def):
        """Inherit from parent class"""

        BaseProvider.__init__(self, provider_def)
        self.num_bands = 4
        self.axes = ['Lat', 'Long']
```

(continues on next page)

(continued from previous page)

```

def get_coverage_domainset(self):
    # return a CIS JSON DomainSet

def get_coverage_rangetype(self):
    # return a CIS JSON RangeType

def query(self, bands=[], subsets={}, format_='json'):
    # process bands and subsets parameters
    # query/extract coverage data
    if format_ == 'json':
        # return a CoverageJSON representation
        return {'type': 'Coverage', ...} # trimmed for brevity
    else:
        # return default (likely binary) representation
        return bytes(112)

```

For brevity, the above code will always JSON for metadata and binary or CoverageJSON for the data. In reality, the plugin developer would connect to a data source with capabilities to run queries and return a relevant result set. As long as the plugin implements the API contract of its base provider, all other functionality is left to the provider implementation.

Each base class documents the functions, arguments and return types required for implementation.

END

12.4 Example: custom pygeoapi formatter

12.4.1 Python code

The below template provides a minimal example (let's call the file `mycooljsonformat.py`):

```

import json
from pygeoapi.formatter.base import BaseFormatter

class MyCoolJSONFormatter(BaseFormatter):
    """My cool JSON formatter"""

    def __init__(self, formatter_def):
        """Inherit from parent class"""

        BaseFormatter.__init__(self, {'name': 'cooljson', 'geom': None})
        self.mimetype = 'text/json; subtype:mycooljson'

    def write(self, options={}, data=None):
        """custom writer"""

        out_data = {'rows': []}

        for feature in data['features']:
            out_data.append(feature['properties'])

        return out_data

```

12.5 Processing plugins

Processing plugins are following the OGC API - Processes development. Given that the specification is under development, the implementation in `pygeoapi/process/hello_world.py` provides a suitable example for the time being.

DEVELOPMENT

13.1 Codebase

The pygeoapi codebase exists at <https://github.com/geopython/pygeoapi>.

13.2 Testing

pygeoapi uses `pytest` for managing its automated tests. Tests exist in `/tests` and are developed for providers, formatters, processes, as well as the overall API.

Tests can be run locally as part of development workflow. They are also run on pygeoapi's `Travis` setup against all commits and pull requests to the code repository.

To run all tests, simply run `pytest` in the repository. To run a specific test file, run `pytest tests/test_api.py`, for example.

13.3 Working with Spatialite on OSX

13.3.1 Using pyenv

It is common among OSX developers to use the package manager homebrew for the installation of pyenv to being able to manage multiple versions of Python. They can encounter errors about the load of some SQLite extensions that pygeoapi uses for handling spatial data formats. In order to run properly the server you are required to follow these steps below carefully.

Make Homebrew and pyenv play nicely together:

```
# see https://github.com/pyenv/pyenv/issues/106
alias brew='env PATH=${PATH//$(pyenv root)\/shims:/} brew'
```

Install python with the option to enable SQLite extensions:

```
LDFLAGS="-L/usr/local/opt/sqlite/lib -L/usr/local/opt/zlib/lib" CPPFLAGS="-I/usr/
˓→local/opt/sqlite/include -I/usr/local/opt/zlib/include" PYTHON_CONFIGURE_OPTS="-
˓→enable-loadable-sqlite-extensions" pyenv install 3.7.6
```

Configure SQLite from Homebrew over that one shipped with the OS:

```
export PATH="/usr/local/opt/sqlite/bin:$PATH"
```

Install Spatialite from Homebrew:

```
brew update  
brew install spatialite-tools  
brew libspatialite
```

Set the variable for the Spatialite library under OSX:

```
SPATIALITE_LIBRARY_PATH=/usr/local/lib/mod_spatialite.dylib
```

CHAPTER
FOURTEEN

OGC COMPLIANCE

As mentioned in the *Introduction*, pygeoapi strives to implement the OGC API standards to be compliant as well as achieving reference implementation status. pygeoapi works closely with the OGC CITE team to achieve compliance through extensive testing as well as providing feedback in order to improve the tests.

14.1 CITE instance

The pygeoapi CITE instance is at <https://demo.pygeoapi.io/cite>

14.2 Setting up your own CITE testing instance

Please see the pygeoapi OGC Compliance for up to date information as well as technical details on setting up your own CITE instance.

CHAPTER
FIFTEEN

CONTRIBUTING

Please see the [Contributing page](#) for information on contributing to the project.

CHAPTER
SIXTEEN

SUPPORT

16.1 Community

Please see the pygeoapi [Community](#) page for information on the community, getting support, and how to get involved.

CHAPTER
SEVENTEEN

FURTHER READING

The following list provides information on pygeoapi and OGC API efforts.

- Default pygeoapi presentation
- OGC API

CHAPTER
EIGHTEEN

LICENSE

18.1 Code

The MIT License (MIT)

Copyright © 2018-2020 Tom Kralidis

• — *

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

18.2 Documentation

The documentation is released under the [Creative Commons Attribution 4.0 International \(CC BY 4.0\)](#) license.

API DOCUMENTATION

Top level code documentation. Follow the links in each section for module/class member information.

19.1 API

Root level code of pygeoapi, parsing content provided by webframework. Returns content from plugins and sets responses

class `pygeoapi.api.API(config)`

API object

__init__(config)

constructor

Parameters `config` – configuration dict

Returns `pygeoapi.API` instance

__weakref__

list of weak references to the object (if defined)

execute_process(headers, args, data, process)

Execute process

Parameters

- `headers` – dict of HTTP headers
- `args` – dict of HTTP request parameters
- `data` – process data
- `process` – name of process

Returns tuple of headers, status code, content

get_collection_items(headers, args, dataset, pathinfo=None)

Queries collection

Parameters

- `headers` – dict of HTTP headers
- `args` – dict of HTTP request parameters
- `dataset` – dataset name
- `pathinfo` – path location

Returns tuple of headers, status code, content

`pygeoapi.api.FORMATS = ['json', 'html', 'jsonld']`

Formats allowed for ?f= requests

`pygeoapi.api.HEADERS = {'Content-Type': 'application/json', 'X-Powered-By': 'pygeoapi 0.9'}`

Return headers for requests (e.g:X-Powered-By)

`pygeoapi.api.check_format(args, headers)`

check format requested from arguments or headers

Parameters

- **args** – dict of request keyword value pairs
- **headers** – dict of request headers

Returns format value

`pygeoapi.api.pre_process(func)`

Decorator performing header copy and format checking before sending arguments to methods

Parameters **func** – decorated function

Returns *func*

19.2 flask_app

Flask module providing the route paths to the api

`pygeoapi.flask_app.collection_coverage(collection_id)`

OGC API - Coverages coverage endpoint

Parameters **collection_id** – collection identifier

Returns HTTP response

`pygeoapi.flask_app.collection_coverage_domainset(collection_id)`

OGC API - Coverages coverage domainset endpoint

Parameters **collection_id** – collection identifier

Returns HTTP response

`pygeoapi.flask_app.collection_coverage_rangetype(collection_id)`

OGC API - Coverages coverage rangetype endpoint

Parameters **collection_id** – collection identifier

Returns HTTP response

`pygeoapi.flask_app.collection_items(collection_id, item_id=None)`

OGC API collections items endpoint

Parameters

- **collection_id** – collection identifier
- **item_id** – item identifier

Returns HTTP response

`pygeoapi.flask_app.collection_queryables(collection_id=None)`

OGC API collections queryables endpoint

Parameters **collection_id** – collection identifier

Returns HTTP response

`pygeoapi.flask_app.collections(collection_id=None)`
OGC API collections endpoint

Parameters `collection_id` – collection identifier

Returns HTTP response

`pygeoapi.flask_app.conformance()`
OGC API conformance endpoint

Returns HTTP response

`pygeoapi.flask_app.landing_page()`
OGC API landing page endpoint

Returns HTTP response

`pygeoapi.flask_app.openapi()`
OpenAPI endpoint

Returns HTTP response

`pygeoapi.flask_app.process_jobs(process_id=None)`
OGC API - Processes jobs endpoint

Parameters `process_id` – process identifier

Returns HTTP response

`pygeoapi.flask_app.processes(process_id=None)`
OGC API - Processes description endpoint

Parameters `process_id` – process identifier

Returns HTTP response

`pygeoapi.flask_app.stac_catalog_path(path)`
STAC path endpoint

Parameters `path` – path

Returns HTTP response

`pygeoapi.flask_app.stac_catalog_root()`
STAC root endpoint

Returns HTTP response

19.3 Logging

Logging system

`pygeoapi.log.setup_logger(logging_config)`
Setup configuration

Parameters `logging_config` – logging specific configuration

Returns void (creates logging instance)

19.4 OpenAPI

`pygeoapi.openapi.gen_media_type_object(media_type, api_type, path)`
Generates an OpenAPI Media Type Object

Parameters

- `media_type` – MIME type
- `api_type` – OGC API type
- `path` – local path of OGC API parameter or schema definition

Returns `dict` of media type object

`pygeoapi.openapi.gen_response_object(description, media_type, api_type, path)`
Generates an OpenAPI Response Object

Parameters

- `description` – text description of response
- `media_type` – MIME type
- `api_type` – OGC API type

Returns `dict` of response object

`pygeoapi.openapi.get_oas(cfg, version='3.0')`
Stub to generate OpenAPI Document

Parameters

- `cfg` – configuration object
- `version` – version of OpenAPI (default 3.0)

Returns OpenAPI definition YAML dict

`pygeoapi.openapi.get_oas_30(cfg)`
Generates an OpenAPI 3.0 Document

Parameters `cfg` – configuration object

Returns OpenAPI definition YAML dict

19.5 Plugins

See also:

Customizing pygeoapi: plugins

Plugin loader

`exception pygeoapi.plugin.InvalidPluginError`

Bases: `Exception`

Invalid plugin

`__weakref__`

list of weak references to the object (if defined)

`pygeoapi.plugin.PLUGINS = {'extensions': {'CQL': 'pygeoapi.cql.CQLHandler'}, 'formatter': ...}`

Loads provider plugins to be used by pygeoapi,formatters and processes availablecql classes available

```
pygeoapi.plugin.load_plugin(plugin_type, plugin_def)
    loads plugin by name
```

Parameters

- **plugin_type** – type of plugin (provider, formatter)
- **plugin_def** – plugin definition

Returns plugin object

19.6 Utils

Generic util functions used in the code

```
pygeoapi.util.dategetter(date_property, collection)
    Attempts to obtain a date value from a collection.
```

Parameters

- **date_property** – property representing the date
- **collection** – dictionary to check within

Returns str (ISO8601) representing the date. (‘..’ if null or “now”, allowing for an open interval).

```
pygeoapi.util.filter_dict_by_key_value(dict_, key, value)
    helper function to filter a dict by a dict key
```

Parameters

- **dict** – dict
- **key** – dict key
- **value** – dict key value

Returns filtered dict

```
pygeoapi.util.get_breadcrumbs(urlpath)
    helper function to make breadcrumbs from a URL path
```

Parameters **urlpath** – URL path**Returns** list of dict objects of labels and links

```
pygeoapi.util.get_extension_by_type(providers, extension_type)
    helper function to check the provider's extension support by an extension type
```

Parameters

- **providers** – list of providers
- **extension_type** – type of provider (feature)

Returns extension based on type

```
pygeoapi.util.get_mimetype(filename)
    helper function to return MIME type of a given file
```

Parameters **filename** – filename (with extension)**Returns** MIME type of given filename

```
pygeoapi.util.get_path_basename(urlpath)
    Helper function to derive file basename
```

Parameters `urlpath` – URL path

Returns string of basename of URL path

`pygeoapi.util.get_provider_by_type(providers, provider_type)`

helper function to load a provider by a provider type

Parameters

- `providers` – list of providers

- `provider_type` – type of provider (feature)

Returns provider based on type

`pygeoapi.util.get_provider_default(providers)`

helper function to get a resource's default provider

Parameters `providers` – list of providers

Returns filtered dict

`pygeoapi.util.get_typed_value(value)`

Derive true type from data value

Parameters `value` – value

Returns value as a native Python data type

`pygeoapi.util.is_url(urlstring)`

Validation function that determines whether a candidate URL should be considered a URI. No remote resource is obtained; this does not check the existence of any remote resource. :param urlstring: str to be evaluated as candidate URL. :returns: bool of whether the URL looks like a URL.

`pygeoapi.util.json_serial(obj)`

helper function to convert to JSON non-default types (source: <https://stackoverflow.com/a/22238613>) :param obj: object to be evaluated :returns: JSON non-default type to str

`pygeoapi.util.render_j2_template(config, template, data)`

render Jinja2 template

Parameters

- `config` – dict of configuration

- `template` – template (relative path)

- `data` – dict of data

Returns string of rendered template

`pygeoapi.util.str2bool(value)`

helper function to return Python boolean type (source: <https://stackoverflow.com/a/715468>)

Parameters `value` – value to be evaluated

Returns bool of whether the value is boolean-ish

`pygeoapi.util.to_json(dict_, pretty=False)`

Serialize dict to json

Parameters

- `dict` – dict of JSON representation

- `pretty` – bool of whether to prettify JSON (default is `False`)

Returns JSON string representation

```
pygeoapi.util.yaml_load(fh)
    serializes a YAML files into a pyyaml object

Parameters fh – file handle

Returns dict representation of YAML
```

19.7 Formatter package

Output formatter package

19.7.1 Base class

```
class pygeoapi.formatter.base.BaseFormatter(formatter_def)
    Bases: object

    generic Formatter ABC

    __init__(formatter_def)
        Initialize object

            Parameters formatter_def – formatter definition

            Returns pygeoapi.providers.base.BaseFormatter

    __repr__()
        Return repr(self).

    __weakref__
        list of weak references to the object (if defined)

    write(options={}, data=None)
        Generate data in specified format

            Parameters

                • options – CSV formatting options

                • data – dict representation of GeoJSON object

            Returns string representation of format
```

19.7.2 csv

```
class pygeoapi.formatter.csv_.CSVFormatter(formatter_def)
    Bases: pygeoapi.formatter.base.BaseFormatter

    CSV formatter

    __init__(formatter_def)
        Initialize object

            Parameters formatter_def – formatter definition

            Returns pygeoapi.formatter.csv_.CSVFormatter

    __repr__()
        Return repr(self).
```

```
write(options={}, data=None)
    Generate data in CSV format

Parameters
    • options – CSV formatting options
    • data – dict of GeoJSON data

Returns string representation of format
```

19.8 Process package

OGC process package, each process is an independent module

19.8.1 Base class

```
class pygeoapi.process.base.BaseProcessor(processor_def, process_metadata)
Bases: object

generic Processor ABC. Processes are inherited from this class

__init__(processor_def, process_metadata)
    Initialize object :param processor_def: processor definition :returns: py-
        geoapi.processors.base.BaseProvider

__repr__()
    Return repr(self).

__weakref__
    list of weak references to the object (if defined)

execute()
    execute the process :returns: dict of process response

exception pygeoapi.process.base.ProcessorExecuteError
Bases: Exception

query / backend error

__weakref__
    list of weak references to the object (if defined)
```

19.8.2 hello_world

Hello world example process

```
class pygeoapi.process.hello_world.HelloWorldProcessor(provider_def)
Bases: pygeoapi.process.base.BaseProcessor

Hello World Processor example

__init__(provider_def)
    Initialize object :param provider_def: provider definition :returns: py-
        geoapi.process.hello_world.HelloWorldProcessor

__repr__()
    Return repr(self).
```

```

execute (data)
    execute the process :returns: dict of process response

pygeoapi.process.hello_world.PROCESS_METADATA = {'description': 'Hello World process', 'e'
    Process metadata and description

```

19.9 Provider

Provider module containing the plugins wrapping data sources

19.9.1 Base class

```

class pygeoapi.provider.base.BaseProvider (provider_def)
    Bases: object

    generic Provider ABC

    __init__ (provider_def)
        Initialize object

        Parameters provider_def – provider definition

        Returns pygeoapi.providers.base.BaseProvider

    __repr__ ()
        Return repr(self).

    __weakref__
        list of weak references to the object (if defined)

    create (new_feature)
        Create a new feature

    delete (identifier)
        Deletes an existing feature

        Parameters identifier – feature id

    get (identifier)
        query the provider by id

        Parameters identifier – feature id

        Returns dict of single GeoJSON feature

    get_coverage_domainset ()
        Provide coverage domainset

        Returns CIS JSON object of domainset metadata

    get_coverage_rangetype ()
        Provide coverage rangetype

        Returns CIS JSON object of rangetype metadata

    get_data_path (baseurl, urlpath, dirpath)
        Gets directory listing or file description or raw file dump

        Parameters
            • baseurl – base URL of endpoint

```

- **urlpath** – base path of URL
- **dirpath** – directory basepath (equivalent of URL)

Returns *dict* of file listing or *dict* of GeoJSON item or raw file

get_fields()

Get provider field information (names, types)

Returns dict of fields

get_metadata()

Provide data/file metadata

Returns *dict* of metadata construct (format determined by provider/standard)

query()

query the provider

Returns dict of 0..n GeoJSON features or coverage data

update(identifier, new_feature)

Updates an existing feature id with new_feature

Parameters

- **identifier** – feature id
- **new_feature** – new GeoJSON feature dictionary

exception pygeoapi.provider.base.ProviderConnectionError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider connection error

exception pygeoapi.provider.base.ProviderGenericError

Bases: *Exception*

provider generic error

__weakref__

list of weak references to the object (if defined)

exception pygeoapi.provider.base.ProviderInvalidQueryError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider invalid query error

exception pygeoapi.provider.base.ProviderItemNotFoundError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider item not found query error

exception pygeoapi.provider.base.ProviderNotFoundError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider not found error

exception pygeoapi.provider.base.ProviderQueryError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider query error

exception pygeoapi.provider.base.ProviderTypeError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider type error

```
exception pygeoapi.provider.base.ProviderVersionError
    Bases: pygeoapi.provider.base.ProviderGenericError
        provider incorrect version error
```

19.9.2 CSV provider

```
class pygeoapi.provider.csv_.CSVProvider(provider_def)
    Bases: pygeoapi.provider.base.BaseProvider

    CSV provider

    _load(identifier=None)
        Load CSV data

        Parameters identifier – feature id

    get(identifier)
        query CSV id

        Parameters identifier – feature id

        Returns dict of single GeoJSON feature

    get_fields()
        Get provider field information (names, types)

        Returns dict of fields

    query(startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], sortby=[], cql_expression=None, identifier=None)
        CSV query

        Parameters

            • startindex – starting record to return (default 0)

            • limit – number of records to return (default 10)

            • resulttype – return results or hit limit (default results)

            • bbox – bounding box [minx,miny,maxx,maxy]

            • datetime – temporal (datestamp or extent)

            • properties – list of tuples (name, value)

            • sortby – list of dicts (property, order)

            • cql_expression – string of cql filter expression

            • identifier – feature id

        Returns dict of GeoJSON FeatureCollection
```

19.9.3 Elasticsearch provider

```
class pygeoapi.provider.elasticsearch_.ElasticsearchProvider(provider_def)
    Bases: pygeoapi.provider.base.BaseProvider

    Elasticsearch Provider

    esdoc2geojson(doc)
        generate GeoJSON dict from ES document

            Parameters doc – dict of ES document

            Returns GeoJSON dict

    get(identifier)
        Get ES document by id

            Parameters identifier – feature id

            Returns dict of single GeoJSON feature

    get_fields()
        Get provider field information (names, types)

            Returns dict of fields

    mask_prop(property_name)
        generate property name based on ES backend setup

            Parameters property_name – property name

            Returns masked property name

    query(startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], sortby=[])
        query Elasticsearch index

            Parameters

                • startindex – starting record to return (default 0)
                • limit – number of records to return (default 10)
                • resulttype – return results or hit limit (default results)
                • bbox – bounding box [minx,miny,maxx,maxy]
                • datetime – temporal (datestamp or extent)
                • properties – list of tuples (name, value)
                • sortby – list of dicts (property, order)

            Returns dict of 0..n GeoJSON features
```

19.9.4 GeoJSON

class pygeoapi.provider.geojson.**GeoJSONProvider** (*provider_def*)
 Bases: *pygeoapi.provider.base.BaseProvider*

Provider class backed by local GeoJSON files

This is meant to be simple (no external services, no dependencies, no schema)

at the expense of performance (no indexing, full serialization roundtrip on each request)

Not thread safe, a single server process is assumed

This implementation uses the feature ‘id’ heavily and will override any ‘id’ provided in the original data. The feature ‘properties’ will be preserved.

TODO: * query method should take bbox * instead of methods returning FeatureCollections, we should be yielding Features and aggregating in the view * there are strict id semantics; all features in the input GeoJSON file must be present and be unique strings. Otherwise it will break. * How to raise errors in the provider implementation such that * appropriate HTTP responses will be raised

_load()

Load and validate the source GeoJSON file at self.data

Yes loading from disk, deserializing and validation happens on every request. This is not efficient.

create (*new_feature*)

Create a new feature

Parameters **new_feature** – new GeoJSON feature dictionary

delete (*identifier*)

Deletes an existing feature

Parameters **identifier** – feature id

get (*identifier*)

query the provider by id

Parameters **identifier** – feature id

Returns dict of single GeoJSON feature

get_fields()

Get provider field information (names, types)

Returns dict of fields

query (*startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], sortby=[]*, *cql_expression=None*)
 query the provider

Parameters

- **startindex** – starting record to return (default 0)
- **limit** – number of records to return (default 10)
- **resulttype** – return results or hit limit (default results)
- **bbox** – bounding box [minx,miny,maxx,maxy]
- **datetime** – temporal (datestamp or extent)
- **properties** – list of tuples (name, value)

- **sortby** – list of dicts (property, order)
- **cql_expression** – string of filter expression

Returns FeatureCollection dict of 0..n GeoJSON features

update (*identifier*, *new_feature*)

Updates an existing feature id with *new_feature*

Parameters

- **identifier** – feature id
- **new_feature** – new GeoJSON feature dictionary

19.9.5 OGR

class pygeoapi.provider.ogr.**CommonSourceHelper** (*provider*)

Bases: *pygeoapi.provider.ogr.SourceHelper*

SourceHelper for most common OGR Source types: Shapefile, GeoPackage, SQLite, GeoJSON etc.

close ()

OGR Driver-specific handling of closing dataset. If ExecuteSQL has been (successfully) called must close ResultSet explicitly. <https://gis.stackexchange.com/questions/114112/explicitly-close-a-ogr-result-object-from-a-call-to-executesql> # noqa

disable_paging ()

Disable paged access to dataset (OGR Driver-specific)

enable_paging (*startindex*=- 1, *limit*=- 1)

Enable paged access to dataset (OGR Driver-specific) using OGR SQL https://gdal.org/user/ogr_sql_dialect.html e.g. SELECT * FROM poly LIMIT 10 OFFSET 30

get_layer ()

Gets OGR Layer from opened OGR dataset. When startindex defined 1 or greater will invoke OGR SQL SELECT with LIMIT and OFFSET and return as Layer as ResultSet from ExecuteSQL on dataset. :return: OGR layer object

class pygeoapi.provider.ogr.**ESRIJSONHelper** (*provider*)

Bases: *pygeoapi.provider.ogr.CommonSourceHelper*

disable_paging ()

Disable paged access to dataset (OGR Driver-specific)

enable_paging (*startindex*=- 1, *limit*=- 1)

Enable paged access to dataset (OGR Driver-specific)

get_layer ()

Gets OGR Layer from opened OGR dataset. When startindex defined 1 or greater will invoke OGR SQL SELECT with LIMIT and OFFSET and return as Layer as ResultSet from ExecuteSQL on dataset. :return: OGR layer object

exception pygeoapi.provider.ogr.**InvalidHelperError**

Bases: *Exception*

Invalid helper

class pygeoapi.provider.ogr.**OGRProvider** (*provider_def*)

Bases: *pygeoapi.provider.base.BaseProvider*

OGR Provider. Uses GDAL/OGR Python-bindings to access OGR Vector sources. References: <https://pcjericks.github.io/py-gdalogr-cookbook/> https://gdal.org/ogr_formats.html (per-driver specifics).

In theory any OGR source type (Driver) could be used, although some Source Types are Driver-specific handling. This is handled in Source Helper classes, instantiated per Source-Type.

The following Source Types have been tested to work: GeoPackage (GPKG), SQLite, GeoJSON, ESRI Shapefile, WFS v2.

_load_source_helper (source_type)

Loads Source Helper by name.

Parameters `type` (`Source`) – Source type name

Returns Source Helper object

_response_feature_collection (layer, limit)

Assembles output from Layer query as GeoJSON FeatureCollection structure.

Returns GeoJSON FeatureCollection

_response_feature_hits (layer)

Assembles GeoJSON hits from OGR Feature count e.g.: http://localhost:5000/collections/tosm_bdi_waterways/items?resulttype=hits

Returns GeoJSON FeaturesCollection

get (identifier)

Get Feature by id

Parameters `identifier` – feature id

Returns feature collection

get_fields ()

Get provider field information (names, types)

Returns dict of fields

query (startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[])

Query OGR source

Parameters

- **startindex** – starting record to return (default 0)
- **limit** – number of records to return (default 10)
- **resulttype** – return results or hit limit (default results)
- **bbox** – bounding box [minx,miny,maxx,maxy]
- **datetime** – temporal (datestamp or extent)
- **properties** – list of tuples (name, value)
- **sortby** – list of dicts (property, order)

Returns dict of 0..n GeoJSON features

class pygeoapi.provider.ogr.**SourceHelper** (`provider`)

Bases: `object`

Helper classes for OGR-specific Source Types (Drivers). For some actions Driver-specific settings or processing is required. This is delegated to the OGR SourceHelper classes.

close ()

OGR Driver-specific handling of closing dataset. Default is no specific handling.

```
    disable_paging()
        Disable paged access to dataset (OGR Driver-specific)

    enable_paging(startindex=-1, limit=-1)
        Enable paged access to dataset (OGR Driver-specific)

    get_layer()
        Default action to get a Layer object from opened OGR Driver. :return:

class pygeoapi.provider.ogr.WFSHelper(provider)
    Bases: pygeoapi.provider.ogr.SourceHelper

    disable_paging()
        Disable paged access to dataset (OGR Driver-specific)

    enable_paging(startindex=-1, limit=-1)
        Enable paged access to dataset (OGR Driver-specific)

pygeoapi.provider.ogr._ignore_gdal_error(inst,fn, *args, **kwargs) → Any
    Evaluate the function with the object instance.
```

Parameters

- **inst** – Object instance
- **fn** – String function name
- **args** – List of positional arguments
- **kwargs** – Keyword arguments

Returns Any function evaluation result

```
pygeoapi.provider.ogr._silent_gdal_error(f)
    Decorator function for gdal
```

19.9.6 postgresql

```
class pygeoapi.provider.postgresql.DatabaseConnection(conn_dic,      table,      con-
                                                               text='query')
    Bases: object
```

Database connection class to be used as ‘with’ statement. The class returns a connection object.

```
class pygeoapi.provider.postgresql.PostgreSQLProvider(provider_def)
    Bases: pygeoapi.provider.base.BaseProvider
```

Generic provider for Postgresql based on psycopg2 using sync approach and server side cursor (using support class DatabaseCursor)

```
_PostgreSQLProvider__get_where_clauses(properties=[], bbox=[])
    Generarates WHERE conditions to be implemented in query. Private method mainly associated
```

with query method :param properties: list of tuples (name, value) :param bbox: bounding box [minx,miny,maxx,maxy]

Returns psycopg2.sql.Composed or psycopg2.sql.SQL

```
_PostgreSQLProvider__response_feature(row_data)
    Assembles GeoJSON output from DB query
```

Parameters **row_data** – DB row result

Returns *dict* of GeoJSON Feature

`_PostgreSQLProvider__response_feature_hits(hits)`
Assembles GeoJSON/Feature number e.g: http://localhost:5000/collections/hotosm_bdi_waterways/items?resulttype=hits

Returns GeoJSON FeaturesCollection

`get(identifier)`
Query the provider for a specific feature id e.g: /collections/hotosm_bdi_waterways/items/13990765

Parameters `identifier` – feature id

Returns GeoJSON FeaturesCollection

`get_fields()`
Get fields from PostgreSQL table (columns are field)

Returns dict of fields

`get_next(cursor, identifier)`
Query next ID given current ID

Parameters `identifier` – feature id

Returns feature id

`get_previous(cursor, identifier)`
Query previous ID given current ID

Parameters `identifier` – feature id

Returns feature id

`query(startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], sortby=[], cql_expression=None)`
Query Postgis for all the content. e.g: http://localhost:5000/collections/hotosm_bdi_waterways/items?limit=1&resulttype=results

Parameters

- **`startindex`** – starting record to return (default 0)
- **`limit`** – number of records to return (default 10)
- **`resulttype`** – return results or hit limit (default results)
- **`bbox`** – bounding box [minx,miny,maxx,maxy]
- **`datetime`** – temporal (datestamp or extent)
- **`properties`** – list of tuples (name, value)
- **`sortby`** – list of dicts (property, order)
- **`cql_expression`** – cql query filter expression

Returns GeoJSON FeaturesCollection

19.9.7 sqlite/geopackage

```
class pygeoapi.provider.sqlite.SQLiteGPKGProvider(provider_def)
    Bases: pygeoapi.provider.base.BaseProvider
```

Generic provider for SQLITE and GPKG using sqlite3 module. This module requires install of libsqlite3-mod-spatialite TODO: DELETE, UPDATE, CREATE

```
_SQLiteGPKGProvider__get_where_clauses(properties=[], bbox=[])
    Generarates WHERE conditions to be implemented in query. Private method mainly associated with query
```

method.

Method returns part of the SQL query, plus tuple to be used in the sqlite query method

Parameters

- **properties** – list of tuples (name, value)
- **bbox** – bounding box [minx,miny,maxx,maxy]

Returns str, tuple

```
_SQLiteGPKGProvider__load()
```

Private method for loading spatiallite, get the table structure and dump geometry

Returns sqlite3.Cursor

```
_SQLiteGPKGProvider__response_feature(row_data)
```

Assembles GeoJSON output from DB query

Parameters row_data – DB row result

Returns dict of GeoJSON Feature

```
_SQLiteGPKGProvider__response_feature_hits(hits)
```

Assembles GeoJSON/Feature number

Returns GeoJSON FeaturesCollection

```
get(identifier)
```

Query the provider for a specific feature id e.g: /collections/countries/items/1

Parameters identifier – feature id

Returns GeoJSON FeaturesCollection

```
get_fields()
```

Get fields from sqlite table (columns are field)

Returns dict of fields

```
query(startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], sortby=[],
       cql_expression=None)
```

Query SQLite/GPKG for all the content. e,g: <http://localhost:5000/collections/countries/items?limit=5&startindex=2&resulttype=results&continent=Europe&admin=Albania&bbox=29.3373,-3.4099,29.3761,-3.3924> <http://localhost:5000/collections/countries/items?continent=Africa&bbox=29.3373,-3.4099,29.3761,-3.3924>

Parameters

- **startindex** – starting record to return (default 0)
- **limit** – number of records to return (default 10)
- **resulttype** – return results or hit limit (default results)

- **bbox** – bounding box [minx,miny,maxx,maxy]
- **datetime** – temporal (datestamp or extent)
- **properties** – list of tuples (name, value)
- **sortby** – list of dicts (property, order)

Returns GeoJSON FeaturesCollection

CHAPTER
TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pygeoapi.api, 81
pygeoapi.flask_app, 82
pygeoapi.formatter, 87
pygeoapi.formatter.base, 87
pygeoapi.formatter.csv_, 87
pygeoapi.log, 83
pygeoapi.openapi, 84
pygeoapi.plugin, 84
pygeoapi.process, 88
pygeoapi.process.base, 88
pygeoapi.process.hello_world, 88
pygeoapi.provider, 89
pygeoapi.provider.base, 89
pygeoapi.provider.csv_, 91
pygeoapi.provider.elasticsearch_, 92
pygeoapi.provider.geojson, 93
pygeoapi.provider.ogr, 94
pygeoapi.provider.postgresql, 96
pygeoapi.provider.sqlite, 98
pygeoapi.util, 85

INDEX

Symbols

_PostgreSQLProvider__get_where_clauses() —weakref__ (`pygeoapi.api.API` attribute), 81
(`pygeoapi.provider.postgresql.PostgreSQLProvider` —weakref__ (`pygeoapi.formatter.base.BaseFormatter` attribute), 87
method), 96

_PostgreSQLProvider__response_feature() —weakref__ (`pygeoapi.plugin.InvalidPluginError` attribute), 84
(`pygeoapi.provider.postgresql.PostgreSQLProvider` —weakref__ (`pygeoapi.process.base.BaseProcessor` attribute), 88
method), 96

_PostgreSQLProvider__response_feature_hits() —weakref__ (`pygeoapi.process.base.ProcessorExecuteError` attribute), 88
(`pygeoapi.provider.postgresql.PostgreSQLProvider` —weakref__ (`pygeoapi.provider.base.ProviderGenericError` attribute), 90
method), 96

_SQLiteGPKGProvider__get_where_clauses() —weakref__ (`pygeoapi.provider.base.BaseProvider` attribute), 89
(`pygeoapi.provider.sqlite.SQLiteGPKGProvider` —weakref__ (`pygeoapi.provider.base.ProviderGenericError` attribute), 90
method), 98

_SQLiteGPKGProvider__load() —weakref__ (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` —weakref__ (`pygeoapi.provider.ogr.OGRProvider` attribute), 91
method), 98

_SQLiteGPKGProvider__response_feature() —weakref__ (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` —weakref__ (`pygeoapi.provider.ogr.OGRProvider` attribute), 93
method), 98

_SQLiteGPKGProvider__response_feature_hits() —weakref__ (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` —weakref__ (`pygeoapi.provider.ogr.OGRProvider` attribute), 95
method), 98

__init__() (`pygeoapi.api.API` method), 81

__init__() (`pygeoapi.formatter.base.BaseFormatter` method), 87

__init__() (`pygeoapi.formatter.csv_.CSVFormatter` method), 87

__init__() (`pygeoapi.process.base.BaseProcessor` method), 88

__init__() (`pygeoapi.provider.base.BaseProvider` method), 89

__repr__() (`pygeoapi.formatter.base.BaseFormatter` method), 87

__repr__() (`pygeoapi.formatter.csv_.CSVFormatter` method), 87

__repr__() (`pygeoapi.process.base.BaseProcessor` method), 88

__repr__() (`pygeoapi.process.hello_world.HelloWorldProcessor` method), 88

silent_gdal_error() —weakref__ (`pygeoapi.provider.ogr.OGRProvider` attribute), 96

A

API (class in `pygeoapi.api`), 81

B

BaseFormatter (class in `pygeoapi.formatter.base`), 87

BaseProcessor (class in `pygeoapi.process.base`), 88

BaseProvider (class in `pygeoapi.provider.base`), 89

C

check_format() (in module `pygeoapi.api`), 82

A

B

BaseFormatter (*class in pygeoapi.formatter.base*), 87
BaseProcessor (*class in pygeoapi.process.base*), 88
BaseProvider (*class in pygeoapi.provider.base*), 89

proc
G

`check_format()` (*in module* `pygeoapi.api`), 82

close() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 94
close() (*pygeoapi.provider.ogr.SourceHelper method*), 95
collection_coverage() (*in module pygeoapi.flask_app*), 82
collection_coverage_domainset() (*in module pygeoapi.flask_app*), 82
collection_coverage_rangetype() (*in module pygeoapi.flask_app*), 82
collection_items() (*in module pygeoapi.flask_app*), 82
collection_queryables() (*in module pygeoapi.flask_app*), 82
collections() (*in module pygeoapi.flask_app*), 83
CommonSourceHelper (*class in pygeoapi.provider.ogr*), 94
conformance() (*in module pygeoapi.flask_app*), 83
create() (*pygeoapi.provider.base.BaseProvider method*), 89
create() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 93
CSVFormatter (*class in pygeoapi.formatter.csv_*), 87
CSVProvider (*class in pygeoapi.provider.csv_*), 91

D

DatabaseConnection (*class in pygeoapi.provider.postgresql*), 96
dategetter() (*in module pygeoapi.util*), 85
delete() (*pygeoapi.provider.base.BaseProvider method*), 89
delete() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 93
disable_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 94
disable_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 94
disable_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 95
disable_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 96

E

ElasticsearchProvider (*class in pygeoapi.provider.elasticsearch_*), 92
enable_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 94
enable_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 94

94
enable_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 96
enable_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 96
esdoc2geojson() (*pygeoapi.provider.elasticsearch_.ElasticsearchProvider method*), 92
ESRIJSONHelper (*class in pygeoapi.provider.ogr*), 94
execute() (*pygeoapi.process.base.BaseProcessor method*), 88
execute() (*pygeoapi.process.hello_world.HelloWorldProcessor method*), 88
execute_process() (*pygeoapi.api.API method*), 81

F

filter_dict_by_key_value() (*in module pygeoapi.util*), 85
FORMATS (*in module pygeoapi.api*), 81

G

gen_media_type_object() (*in module pygeoapi.openapi*), 84
gen_response_object() (*in module pygeoapi.openapi*), 84
GeoJSONProvider (*class in pygeoapi.provider.geojson*), 93
get() (*pygeoapi.providerbase.BaseProvider method*), 89
get() (*pygeoapi.provider.csv_.CSVProvider method*), 91
get() (*pygeoapi.provider.elasticsearch_.ElasticsearchProvider method*), 92
get() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 93
get() (*pygeoapi.provider.ogr.OGRProvider method*), 95
get() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 97
get() (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 98
get_breadcrumbs() (*in module pygeoapi.util*), 85
get_collection_items() (*pygeoapi.api.API method*), 81
get_coverage_domainset() (*pygeoapi.providerbase.BaseProvider method*), 89
get_coverage_rangetype() (*pygeoapi.providerbase.BaseProvider method*), 89
get_data_path() (*pygeoapi.providerbase.BaseProvider method*), 89

get_extension_by_type() (in module `pygeoapi.util`), 85

get_fields() (`pygeoapi.provider.base.BaseProvider` method), 90

get_fields() (`pygeoapi.provider.csv_.CSVProvider` method), 91

get_fields() (`pygeoapi.provider.elasticsearch_.ElasticsearchProvider` method), 92

get_fields() (`pygeoapi.provider.geojson.GeoJSONProvider` method), 93

get_fields() (`pygeoapi.provider.ogr.OGRProvider` method), 95

get_fields() (`pygeoapi.provider.postgresql.PostgreSQLProvider` method), 97

get_fields() (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` method), 98

get_layer() (`pygeoapi.provider.ogr.CommonSourceHelper` method), 94

get_layer() (`pygeoapi.provider.ogr.ESRIJSONHelper` method), 94

get_layer() (`pygeoapi.provider.ogr.SourceHelper` method), 96

get_metadata() (`pygeoapi.provider.base.BaseProvider` method), 90

get_mimetype() (in module `pygeoapi.util`), 85

get_next() (`pygeoapi.provider.postgresql.PostgreSQLProvider` method), 97

get_oas() (in module `pygeoapi.openapi`), 84

get_oas_30() (in module `pygeoapi.openapi`), 84

get_path_basename() (in module `pygeoapi.util`), 85

get_previous() (`pygeoapi.provider.postgresql.PostgreSQLProvider` method), 97

get_provider_by_type() (in module `pygeoapi.util`), 86

get_provider_default() (in module `pygeoapi.util`), 86

get_typed_value() (in module `pygeoapi.util`), 86

H

HEADERS (in module `pygeoapi.api`), 82

HelloWorldProcessor (class in `pygeoapi.process.hello_world`), 88

I

InvalidHelperError, 94

InvalidPluginError, 84

is_url() (in module `pygeoapi.util`), 86

J

json_serial() (in module `pygeoapi.util`), 86

L

landing_page() (in module `pygeoapi.flask_app`), 83

load_plugin() (in module `pygeoapi.plugin`), 84

M

mask_prop() (`pygeoapi.provider.elasticsearch_.ElasticsearchProvider` method), 92

module

- `pygeoapi.api`, 81
- `pygeoapi.flask_app`, 82
- `pygeoapi.formatter`, 87
- `pygeoapi.formatter.base`, 87
- `pygeoapi.formatter.csv_`, 87
- `pygeoapi.log`, 83
- `pygeoapi.openapi`, 84
- `pygeoapi.plugin`, 84
- `pygeoapi.process`, 88
- `pygeoapi.process.base`, 88
- `pygeoapi.process.hello_world`, 88
- `pygeoapi.provider`, 89
- `pygeoapi.provider.base`, 89
- `pygeoapi.provider.csv_`, 91
- `pygeoapi.provider.elasticsearch_`, 92
- `pygeoapi.provider.geojson`, 93
- `pygeoapi.provider.ogr`, 94
- `pygeoapi.provider.postgresql`, 96
- `pygeoapi.provider.sqlite`, 98
- `pygeoapi.util`, 85

O

OGRProvider (class in `pygeoapi.provider.ogr`), 94

openapi() (in module `pygeoapi.flask_app`), 83

P

PLUGINS (in module `pygeoapi.plugin`), 84

PostgreSQLProvider (class in `pygeoapi.provider.postgresql`), 96

pre_process() (in module `pygeoapi.api`), 82

process_jobs() (in module `pygeoapi.flask_app`), 83

PROCESS_METADATA (in module `pygeoapi.process.hello_world`), 89

processes() (in module `pygeoapi.flask_app`), 83

ProcessorExecuteError, 88

ProviderConnectionError, 90

ProviderGenericError, 90

ProviderInvalidQueryError, 90

ProviderItemNotFoundError, 90

ProviderNotFoundError, 90

ProviderQueryError, 90

ProviderTypeError, 90

ProviderVersionError, 90

pygeoapi.api module, 81

pygeoapi.flask_app
 module, 82
pygeoapi.formatter
 module, 87
pygeoapi.formatter.base
 module, 87
pygeoapi.formatter.csv_
 module, 87
pygeoapi.log
 module, 83
pygeoapi.openapi
 module, 84
pygeoapi.plugin
 module, 84
pygeoapi.process
 module, 88
pygeoapi.process.base
 module, 88
pygeoapi.process.hello_world
 module, 88
pygeoapi.provider
 module, 89
pygeoapi.provider.base
 module, 89
pygeoapi.provider.csv_
 module, 91
pygeoapi.provider.elasticsearch_
 module, 92
pygeoapi.provider.geojson
 module, 93
pygeoapi.provider.ogr
 module, 94
pygeoapi.provider.postgresql
 module, 96
pygeoapi.provider.sqlite
 module, 98
pygeoapi.util
 module, 85

Q

query() (pygeoapi.provider.base.BaseProvider
 method), 90
query() (pygeoapi.provider.csv_.CSVProvider
 method), 91
query() (pygeoapi.provider.elasticsearch_.ElasticsearchProvider
 method), 92
query() (pygeoapi.provider.geojson.GeoJSONProvider
 method), 93
query() (pygeoapi.provider.ogr.OGRProvider method),
 95
query() (pygeoapi.provider.postgresql.PostgreSQLProvider
 method), 97
query() (pygeoapi.provider.sqlite.SQLiteGPKGProvider
 method), 98

R

render_j2_template() (in module pygeoapi.util),
 86

S

setup_logger() (in module pygeoapi.log), 83
SourceHelper (class in pygeoapi.provider.ogr), 95
SQLiteGPKGProvider (class in py-
 geoapi.provider.sqlite), 98
stac_catalog_path() (in module py-
 geoapi.flask_app), 83
stac_catalog_root() (in module py-
 geoapi.flask_app), 83
str2bool() (in module pygeoapi.util), 86

T

to_json() (in module pygeoapi.util), 86

U

update() (pygeoapi.provider.base.BaseProvider
 method), 90
update() (pygeoapi.provider.geojson.GeoJSONProvider
 method), 94

W

WFSHelper (class in pygeoapi.provider.ogr), 96
write() (pygeoapi.formatter.base.BaseFormatter
 method), 87
write() (pygeoapi.formatter.csv_.CSVFormatter
 method), 87

Y

yaml_load() (in module pygeoapi.util), 86