
pygeoapi Documentation

Release 0.12.0

pygeoapi team

2022-02-07

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	Validating the configuration	13
4.3	Using environment variables	13
4.4	Linked Data	13
4.5	Summary	15
5	Administration	17
5.1	Creating the OpenAPI document	17
5.2	Validating the OpenAPI document	18
5.3	Setting system environment variables	18
5.4	Summary	18
6	Running	19
6.1	pygeoapi serve	19
6.1.1	Flask WSGI	19
6.1.2	Starlette ASGI	20
6.2	Running in production	20
6.2.1	Apache and mod_wsgi	21
6.2.2	Gunicorn	21
6.2.3	Gunicorn and Flask	21

6.2.4	Gunicorn and Starlette	22
6.3	Summary	22
7	Docker	23
7.1	The basics	23
7.2	Overriding the default configuration	23
7.3	Deploying on a sub-path	24
7.4	Summary	25
8	Taking a tour of pygeoapi	27
8.1	Overview	27
8.2	Landing page	27
8.3	Collections	27
8.4	Collection information	28
8.5	Vector data	28
8.5.1	Collection queryables	28
8.5.2	Collection items	28
8.5.3	Collection item	28
8.6	Raster data	28
8.6.1	Collection coverage domainset	28
8.6.2	Collection coverage rangetype	29
8.6.3	Collection coverage data	29
8.7	Tiles	29
8.7.1	URI templates	29
8.7.2	Generic metadata	29
8.8	Metadata Records	29
8.9	Processes	30
8.10	Environmental data retrieval	30
8.11	SpatioTemporal Assets	30
8.12	API Documentation	30
8.13	Conformance	31
9	OpenAPI	33
9.1	Using OpenAPI via Swagger	33
9.2	Using OpenAPI via ReDoc	38
9.3	Summary	38
10	Data publishing	39
10.1	Providers overview	39
10.1.1	Publishing vector data to OGC API - Features	39
10.1.1.1	Providers	39
10.1.1.2	Connection examples	40
10.1.1.3	Data access examples	43
10.1.2	Publishing raster data to OGC API - Coverages	44
10.1.2.1	Providers	44
10.1.2.2	Connection examples	44
10.1.2.3	Data access examples	45
10.1.3	Publishing map tiles to OGC API - Tiles	46
10.1.3.1	Providers	46
10.1.3.2	Connection examples	46
10.1.3.3	Data access examples	46
10.1.4	Publishing processes via OGC API - Processes	47
10.1.4.1	Configuration	47
10.1.4.2	Asynchronous support	47
10.1.4.3	Putting it all together	47

10.1.4.4	Processing examples	47
10.1.5	Publishing metadata to OGC API - Records	48
10.1.5.1	Providers	48
10.1.5.2	Connection examples	48
10.1.5.3	Metadata search examples	49
10.1.6	Publishing data to OGC API - Environmental Data Retrieval	50
10.1.6.1	Providers	50
10.1.6.2	Connection examples	50
10.1.6.3	Data access examples	51
10.1.7	Publishing files to a SpatioTemporal Asset Catalog	51
10.1.7.1	Connection examples	51
10.1.7.2	pygeometa metadata control files	51
10.1.7.3	Publishing ESRI Shapefiles	52
10.1.7.4	Data access examples	52
11	Customizing pygeoapi: plugins	53
11.1	Overview	53
11.2	Example: custom pygeoapi vector data provider	54
11.2.1	Python code	54
11.2.2	Connecting to pygeoapi	55
11.3	Example: custom pygeoapi raster data provider	55
11.3.1	Python code	56
11.4	Example: custom pygeoapi formatter	56
11.4.1	Python code	56
11.5	Processing plugins	57
11.6	Featured plugins	57
12	HTML Templating	59
12.1	Featured templates	60
13	CQL support	61
13.1	Limitations	61
13.2	Formats	61
13.3	Providers	61
13.3.1	Queries	61
13.3.2	Examples	61
14	Multilingual support	63
14.1	End user guide	63
14.1.1	Notes	63
14.2	Maintainer guide	64
14.2.1	Notes	64
14.2.2	Add translations for configurable text values	65
14.3	Translator guide	66
14.4	Developer guide	66
14.4.1	Notes	67
15	Development	69
15.1	Codebase	69
15.2	Testing	69
15.3	CQL extension lifecycle	69
15.3.1	Limitations	69
15.3.2	Schema	69
15.3.3	Model generation	70
15.3.4	How to merge	70

15.4	Working with Spatialite on OSX	70
15.4.1	Using pyenv	70
16	OGC Compliance	71
16.1	CITE instance	71
16.2	Setting up your own CITE testing instance	71
17	Contributing	73
18	Support	75
18.1	Community	75
19	Further Reading	77
20	License	79
20.1	Code	79
20.2	Documentation	79
21	API documentation	81
21.1	API	81
21.2	flask_app	86
21.3	Logging	89
21.4	OpenAPI	89
21.5	Plugins	90
21.6	Utils	90
21.7	Formatter package	93
21.7.1	Base class	93
21.7.2	csv	94
21.8	Process package	94
21.8.1	Base class	95
21.8.2	hello_world	95
21.9	Provider	96
21.9.1	Base class	96
21.9.2	CSV provider	98
21.9.3	Elasticsearch provider	99
21.9.4	GeoJSON	99
21.9.5	OGR	100
21.9.6	postgresql	103
21.9.7	sqlite/geopackage	104
22	Indices and tables	107
	Python Module Index	109
	Index	111



Author the pygeoapi team

Contact [pygeoapi at lists.osgeo.org](mailto:pygeoapi@lists.osgeo.org)

Release 0.12.0

Date 2022-02-07

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 - Tiles * OGC API - Processes * OGC API - Environmental Data Retrieval * 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)
- supports asynchronous processing and job management (OGC API - Processes)

1.2 Standards Support

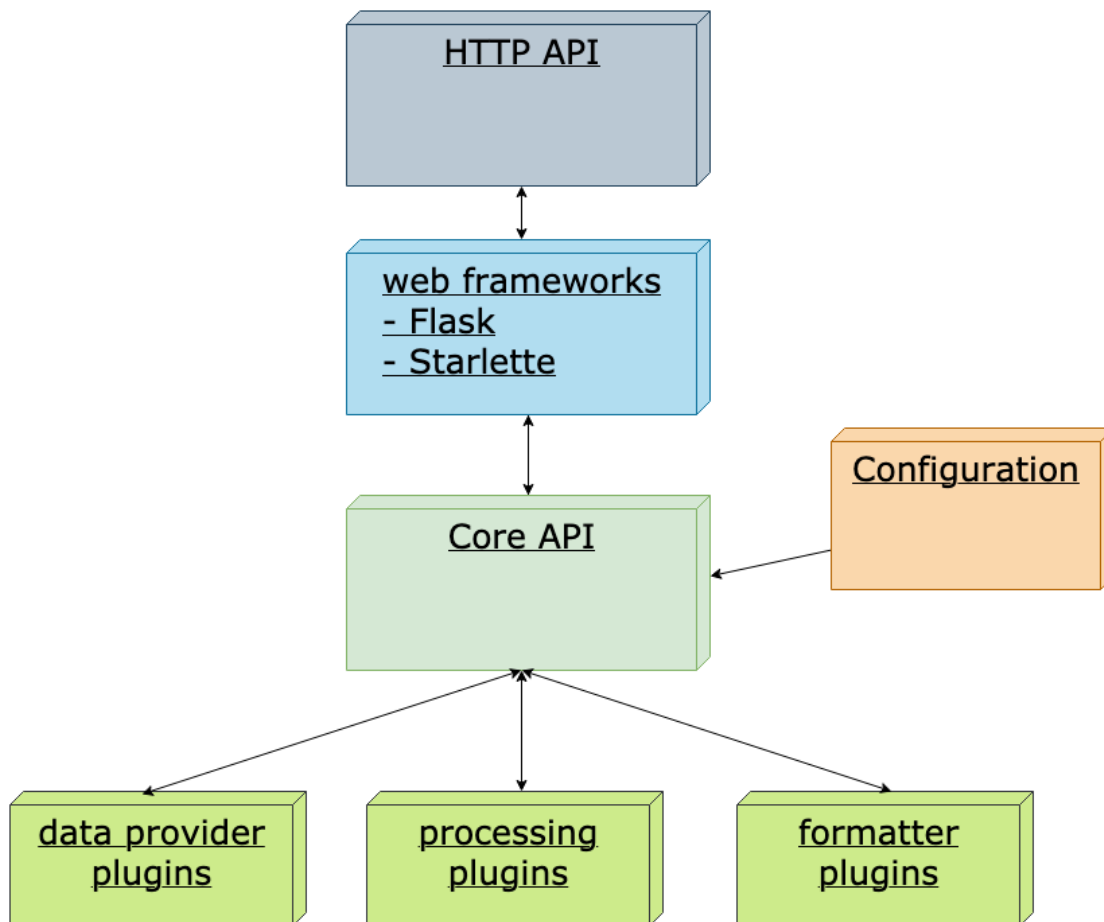
Standards are at the core of `pygeoapi`. Below is the project's standards support matrix.

- 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 - Tiles	Implementing
OGC API - Processes	Implementing
OGC API - Records	Implementing
OGC API - Environmental Data Retrieval	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 openapi generate $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.

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 `.yaml`.

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
  gzip: false # default server config to gzip/compress responses to requests with_
  ↪ gzip in the Accept-Encoding header
  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
```

(continues on next page)

(continued from previous page)

```
templates: # optional configuration to specify a different set of templates for_
↳HTML pages. Recommend using absolute paths. Omit this to use the default provided_
↳templates
    path: /path/to/jinja2/templates/folder # path to templates folder containing the_
↳jinja2 template HTML files
    static: /path/to/static/folder # path to static folder containing css, js, images_
↳and other static files referenced by the template

map: # leaflet map setup for HTML pages
    url: https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png
    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

manager: # optional OGC API - Processes asynchronous job management
    name: TinyDB # plugin name (see pygeoapi.plugin for supported process_manager
↳'s)
    connection: /tmp/pygeoapi-process-manager.db # connection info to store jobs_
↳(e.g. filepath)
    output_dir: /tmp/ # temporary file area for storing job results (files)
```

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
```

(continues on next page)

(continued from previous page)

```

    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: +xx-xxx-xxx-xxxx
    fax: +xx-xxx-xxx-xxxx
    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
- coverage
- tile

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#

```

(continues on next page)

(continued from previous page)

```

    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
    # 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, record, tile, edr)
      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
      uri_field: uri # optional field corresponding to the Uniform Resource
↪Identifier (see Linked Data section)
      time_field: datetimestamp # optional field corresponding to the
↪temporal property of the dataset
      title_field: foo # optional field of which property to display as title/
↪label on HTML pages
      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 definition

```

See also:*Linked Data* for optionally configuring linked data datasets**See also:**

Customizing pygeoapi: plugins for more information on plugins

4.2 Validating the configuration

To ensure your configuration is valid, pygeoapi provides a validation utility that can be run as follows:

```
pygeoapi config validate /path/to/my-pygeoapi-config.yml
```

4.3 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}
```

4.4 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 item, the default JSON-LD representation adds:

- An `@id` for the item, which is the URL for that item. If `uri_field` is specified, it is used, otherwise the URL is to its HTML representation in pygeoapi.
- Separate GeoSPARQL/WKT and *schema.org/geo* versions of the geometry. *schema.org/geo* only supports point, line, and polygon geometries. Multipart lines are merged into a single line. The rest of the multipart geometries are transformed reduced and into a polygon via unary union or convex hull transform.
- `@context` for the GeoSPARQL and schema geometries.
- The unpacked properties block into the main body of the item.

For collections, at the level of items, the default JSON-LD representation adds:

- A *schema.org* `itemList` of the `@id` and `@type` of each feature in the collection.

The optional configuration options for collections, at the level of an item of items, are:

- If `uri_field` is specified, JSON-LD will be updated such that the `@id` has the value of `uri_field` for each item in a collection

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.

An example of a data provider that includes relationships between items is the SensorThings API provider. SensorThings API, by default, has relationships between entities within its data model. Setting the `intraLink` field of the SensorThings provider to `true` sets pygeoapi to represent the relationship between configured entities as intra-pygeoapi links or URIs. This relationship can further be maintained in the JSON-LD structured data using the appropriate `@context` with the `sosa/ssn` ontology. For example:

```
Things:
  context:
  - sosa: "http://www.w3.org/ns/sosa/"
    ssn: "http://www.w3.org/ns/ssn/"
    Datastreams: sosa:ObservationCollection

Datastreams:
  context:
  - sosa: "http://www.w3.org/ns/sosa/"
```

(continues on next page)

(continued from previous page)

```
ssn: "http://www.w3.org/ns/ssn/"
Observations: sosa:hasMember
Thing: sosa:hasFeatureOfInterest
```

Observations:**context:**

- **sosa:** "http://www.w3.org/ns/sosa/"
- ssn:** "http://www.w3.org/ns/ssn/"
- Datastream:** sosa:isMemberOf

4.5 Summary

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

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
- validate OpenAPI document
- set system environment variables

5.1 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 openapi generate /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 openapi generate /path/to/my-pygeoapi-config.yml > /path/to/my-pygeoapi-  
↪openapi.yml
```

To generate the OpenAPI document as JSON, run:

```
pygeoapi openapi generate /path/to/my-pygeoapi-config.yml -f json > /path/to/my-  
↪pygeoapi-openapi.json
```

Note: Generate as YAML or JSON? If your OpenAPI YAML definition is slow to render as JSON, saving as JSON to disk will help with performance at run-time.

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

5.2 Validating the OpenAPI document

To ensure your OpenAPI document is valid, pygeoapi provides a validation utility that can be run as follows:

```
pygeoapi openapi validate /path/to/my-pygeoapi-openapi.yml
```

5.3 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
# or if OpenAPI JSON
export PYGEOAPI_OPENAPI=/path/to/my-pygeoapi-openapi.json
```

In Windows:

```
set PYGEOAPI_CONFIG=/path/to/my-pygeoapi-config.yml
set PYGEOAPI_OPENAPI=/path/to/my-pygeoapi-openapi.yml
# or if OpenAPI JSON
set PYGEOAPI_OPENAPI=/path/to/my-pygeoapi-openapi.json
```

5.4 Summary

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

RUNNING

Now we are ready to start up pygeoapi.

6.1 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).

6.1.1 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
```

To integrate `pygeoapi` as part of another Flask application, use Flask blueprints:

```
from flask import Flask
from pygeoapi.flask_app import BLUEPRINT as pygeoapi_blueprint

app = Flask(__name__)

app.register_blueprint(pygeoapi_blueprint, url_prefix='/oapi')

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

As a result, your application will be available at <http://localhost:5000/> and pygeoapi will be available at <http://localhost:5000/oapi>

6.1.2 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 Starlette ASGI server can be run as follows:

```
pygeoapi serve --starlette
```

To integrate pygeoapi as part of another Starlette application:

```
from starlette.applications import Starlette
from starlette.responses import PlainTextResponse
from starlette.routing import Route
from pygeoapi.starlette_app import app as pygeoapi_app

async def homepage(request):
    return PlainTextResponse('Hello, World!')

app = Starlette(debug=True, routes=[
    Route('/', homepage),
])

app.mount('/oapi', pygeoapi_app)
```

As a result, your application will be available at <http://localhost:5000/> and pygeoapi will be available at <http://localhost:5000/oapi>

6.2 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.

6.2.1 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:

```
WSGIDaemonProcess 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>
```

6.2.2 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](#).

6.2.3 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](#).

6.2.4 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`

6.3 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.

7.1 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
```

7.2 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
    ports:
      - "5000:80"
```

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

7.3 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

7.4 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>

8.1 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.

8.2 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.

8.3 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.

8.4 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 'Queryable' section provides a link to the dataset's properties.

8.5 Vector data

8.5.1 Collection queryables

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

The queryables endpoint provides the collection's queryable properties and associated datatypes.

8.5.2 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](#).

8.5.3 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.

8.6 Raster data

8.6.1 Collection coverage domainset

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

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

8.6.2 Collection coverage rangetype

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

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

8.6.3 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.

8.7 Tiles

A given collection or any data type can additionally be made available as tiles (map tiles, vector tiles, etc.). The following page provides an overview of a collection's tiles capabilities (tile matrix sets, URI templates, etc.)

<http://localhost:5000/collections/lakes/tiles>

8.7.1 URI templates

From the abovementioned page, we can find the URI template:

<http://localhost:5000/collections/lakes/tiles/{tileMatrixSetId}/{tileMatrix}/{tileRow}/{tileCol}/{format}?f=mvt>

8.7.2 Generic metadata

This page provides freeform tiles metadata.

<http://localhost:5000/collections/lakes/tiles/WorldCRS84Quad/metadata>

8.8 Metadata Records

<http://localhost:5000/collections/metadata-records/items?q=crops&bbox=-142,42,-52,84>

This page provides metadata catalogue search capabilities

See also:

Publishing metadata to OGC API - Records for more OGC API - Records request examples.

8.9 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.

8.10 Environmental data retrieval

<http://localhost:5000/collections/edr-test>

This page provides, in addition to a common collection description, specific link relations for EDR queries if the collection has an EDR capability, as well as supported parameter names to select.

[http://localhost:5000/collections/edr-test/position?coords=POINT\(111 13\)¶meter-name=SST&f=json](http://localhost:5000/collections/edr-test/position?coords=POINT(111 13)¶meter-name=SST&f=json)

This page executes a position query against a given parameter name, providing a response in CoverageJSON.

See also:

Publishing data to OGC API - Environmental Data Retrieval for more OGC API - EDR request examples.

8.11 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.

8.12 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

8.13 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:

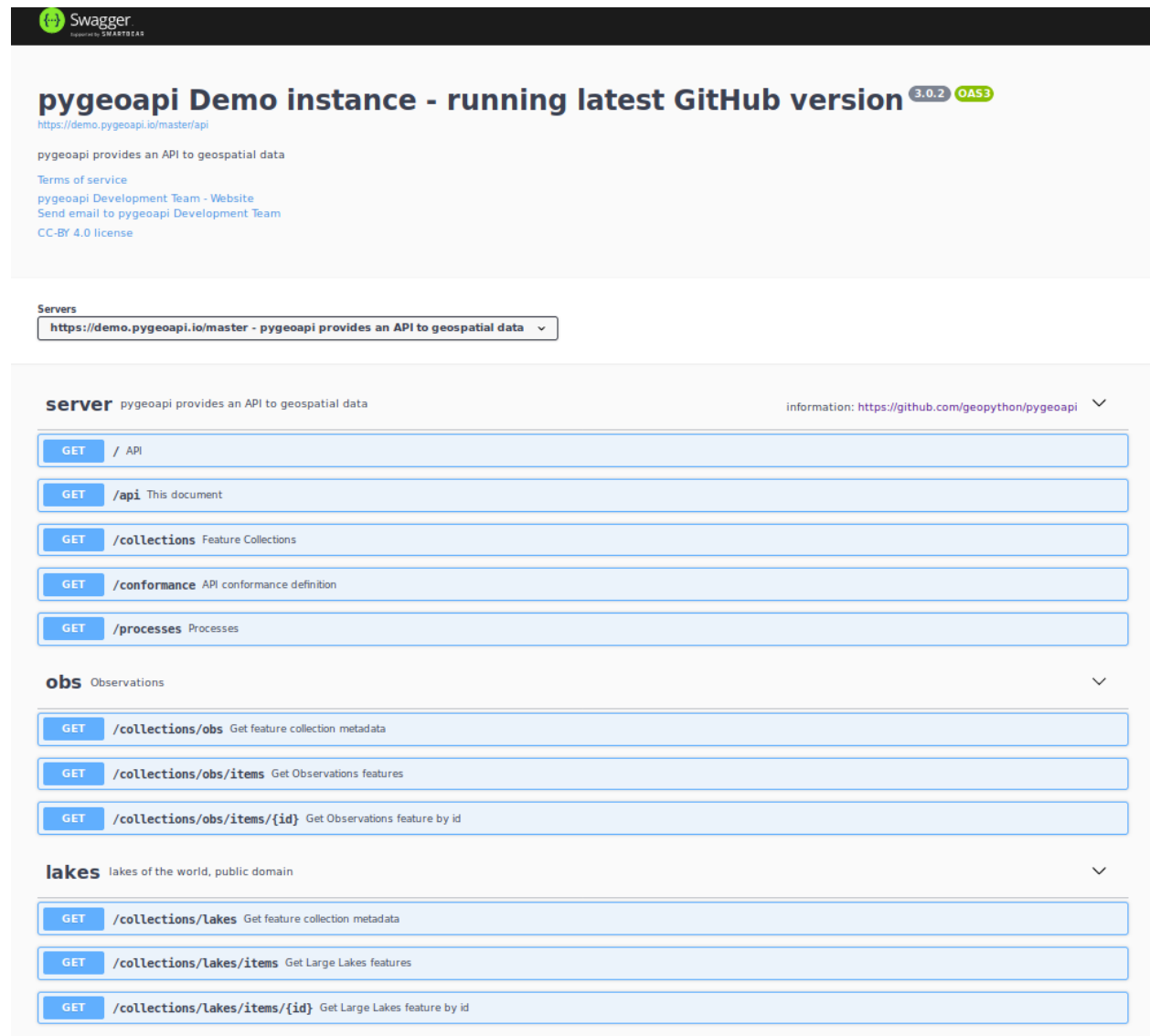
The API is accessible at the `/openapi` endpoint, providing a Swagger-based webpage of the API description..

See also:

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

9.1 Using OpenAPI via Swagger

Accessing the Swagger webpage we have the following structure:



The image shows the Swagger UI for the pygeoapi Demo instance. At the top, it says "pygeoapi Demo instance - running latest GitHub version" with version "3.0.2" and "OAS3" tags. Below this, there's a link to "https://demo.pygeoapi.io/master/api" and a description: "pygeoapi provides an API to geospatial data". There are also links for "Terms of service", "pygeoapi Development Team - Website", "Send email to pygeoapi Development Team", and "CC-BY 4.0 license".

The "Servers" section shows a single server: "https://demo.pygeoapi.io/master - pygeoapi provides an API to geospatial data".

The "server" section, titled "pygeoapi provides an API to geospatial data", lists several endpoints under the "collections" group:

- GET / API**
- GET /api** This document
- GET /collections** Feature Collections
- GET /conformance** API conformance definition
- GET /processes** Processes

There are also sections for "obs" (Observations) and "lakes" (lakes of the world, public domain):

- obs** Observations
 - GET /collections/obs** Get feature collection metadata
 - GET /collections/obs/items** Get Observations features
 - GET /collections/obs/items/{id}** Get Observations feature by id
- lakes** lakes of the world, public domain
 - GET /collections/Lakes** Get feature collection metadata
 - GET /collections/Lakes/items** Get Large Lakes features
 - GET /collections/Lakes/items/{id}** Get Large Lakes feature by id

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:

server pygeoapi provides an API to geospatial data information: <https://github.com/geopython/pygeoapi>

GET / API

GET /api This document

GET /collections Feature Collections

Feature Collections

Parameters Cancel

No parameters

Execute

Responses

Code	Description	Links
200	successful operation	No links

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

Curl

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

Request URL

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

Server response

Code **Details**

200

Response body

```
{
  "crs": [
    "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
  ],
  "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"
  ],
  "extent": [
    50.75,
    3.37,
    53.47,
    7.21
  ],
  "links": [
    {
      "type": "text/html",
      "href": "https://demo.pygeoapi.io/master/collections",
      "rel": "self"
    }
  ]
}
```

Download

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 Cancel

No parameters

Execute Clear

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": "text/html",
  "rel": "alternate",
  "title": "This document as HTML",
  "href": "https://demo.pygeoapi.io/master/collections/dutch_windmills?f=html"
},
{
  "crs": [
    "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
  ],
  "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"
  ],
  "extent": [
    50.75,
    3.37,
    53.47,
    7.21
  ]
}
```

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-spy: h2
x-powered-by: pygeoapi 0.6.0
```

Download

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 Cancel

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.
bbox array[number] (query)	The bbox parameter indicates the minimum bounding rectangle upon which to query the collection in WFS84 (minx, miny, maxx, maxy).
time string (query)	The time parameter indicates an RFC3339 formatted datetime (single, interval, open).
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.
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 <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.
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).

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

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

The screenshot displays the OpenAPI UI for the endpoint `GET /collections/dutch_windmills/items/{id}`. The interface includes a description of the endpoint, a 'Parameters' section with a table of input fields, and an 'Execute' button at the bottom.

Name	Description
id * required string (path)	The id of a feature
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.

The 'id' parameter is highlighted with a red box and contains the value 'Molens.1'. The 'f' parameter is a dropdown menu set to 'json'.

9.2 Using OpenAPI via ReDoc

pygeoapi also supports OpenAPI document rendering via [ReDoc](#).

ReDoc rendering is accessible at the same `/openapi` endpoint, adding `ui=redoc` to the request URL.

9.3 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/coverage data, processes, catalogues, and exposing filesystems of geospatial data.

10.1 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.

10.1.1 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.

10.1.1.1 Providers

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

Provider	property filters/display	resulttype	bbox	datetime	sortby	CQL
CSV	/	results/hits				
Elasticsearch	/	results/hits				
GeoJSON	/	results/hits				
MongoDB	/	results				
OGR	/	results/hits				
PostgreSQL	/	results/hits				
SQLiteGPKG	/	results/hits				
SensorThingsAPI	/	results/hits				

Below are specific connection examples based on supported providers.

10.1.1.2 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: datetimefield
```

This provider has the support for the CQL queries as indicated in the table above.

See also:

CQL support for more details on how to use the Common Query Language to filter the collection with specific queries.

OGR

GDAL/OGR supports a wide range of spatial file formats, such as shapefile, dxf, gpx, kml, but also services such as WFS. Read the full list and configuration options at <https://gdal.org/drivers/vector>. Additional formats and features are available via the [virtual format](#), use this driver for example for flat database files (CSV).

The OGR provider requires a recent (3+) version of GDAL to be installed.

```
providers:
- type: feature
  name: OGR
  data:
    source_type: ESRI Shapefile
    source: tests/data/dutch_addresses_shape_4326/inspireadressen.shp
    source_options:
      ADJUST_GEOM_TYPE: FIRST_SHAPE
    gdal_ogr_options:
      SHPT: POINT
    id_field: fid
    layer: inspireadressen
```

```
providers:
- type: feature
  name: OGR
  data:
    source_type: WFS
    source: WFS:https://geodata.nationaalgeoregister.nl/rdinfo/wfs?
    source_options:
      VERSION: 2.0.0
      OGR_WFS_PAGING_ALLOWED: YES
      OGR_WFS_LOAD_MULTIPLE_LAYER_DEFN: NO
    gdal_ogr_options:
      GDAL_CACHEMAX: 64
      GDAL_HTTP_PROXY: (optional proxy)
      GDAL_PROXY_AUTH: (optional auth for remote WFS)
      CPL_DEBUG: NO
    id_field: gml_id
    layer: rdinfo:stations
```

```
providers:
- type: feature
  name: OGR
  data:
    source_type: ESRIJSON
    source: https://map.bgs.ac.uk/arcgis/rest/services/GeoIndex_Onshore/
↳boreholes/MapServer/0/query?where=BGS_ID+%3DBGS_ID&outfields=*&orderByFields=BGS_
↳ID+ASC&f=json
    source_srs: EPSG:27700
    target_srs: EPSG:4326
    source_capabilities:
      paging: True
    open_options:
      FEATURE_SERVER_PAGING: YES
    gdal_ogr_options:
      EMPTY_AS_NULL: NO
      GDAL_CACHEMAX: 64
      # GDAL_HTTP_PROXY: (optional proxy)
```

(continues on next page)

(continued from previous page)

```
# GDAL_PROXY_AUTH: (optional auth for remote WFS)
CPL_DEBUG: NO
id_field: BGS_ID
layer: ESRIJSON
```

MongoDB

Todo: add overview and requirements

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

PostgreSQL

Must have PostGIS installed.

Todo: add overview and requirements

```
providers:
- type: feature
  name: PostgreSQL
  data:
    host: 127.0.0.1
    port: 3010 # Default 5432 if not provided
    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
```

(continues on next page)

(continued from previous page)

```
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
```

SensorThings API

The STA provider is capable of creating feature collections from OGC SensorThings API endpoints. Three of the STA entities are configurable: Things, Datastreams, and Observations. For a full description of the SensorThings entity model, see [here](#). For each entity of Things, pygeoapi will expand all entities directly related to the Thing, including its associated Location, from which the geometry for the feature collection is derived. Similarly, Datastreams are expanded to include the associated Thing, Sensor and ObservedProperty.

The default id_field is @iot.id. The STA provider adds one required field, entity, and an optional field, intralink. The entity field refers to which STA entity to use for the feature collection. The intralink field controls how the provider is acted upon by other STA providers and is by default, False. If intralink is true for an adjacent STA provider collection within a pygeoapi instance, the expanded entity is instead represented by an intra-pygeoapi link to the other entity or it's uri_field if declared.

```
providers:
- type: feature
  name: SensorThings
  data: https://sensorthings-wq.brgm-rec.fr/FROST-Server/v1.0/
  uri_field: uri
  entity: Datastreams
  time_field: phenomenonTime
  intralink: true
```

If all three entities are configured, the STA provider will represent a complete STA endpoint as OGC-API feature collections. The Things features will include links to the associated features in the Datastreams feature collection, and the Observations features will include links to the associated features in the Datastreams feature collection. Examples with three entities configured are included in the docker examples for SensorThings.

10.1.1.3 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>
- query features (temporal) and sort ascending by a property (if no +/- indicated, + is assumed) * <http://localhost:5000/collections/foo/items?datetime=2020-04-10T14:11:00Z&sortby=+datetime>
- query features (temporal) and sort descending by a property * <http://localhost:5000/collections/foo/items?datetime=2020-04-10T14:11:00Z&sortby=-datetime>
- fetch a specific feature * <http://localhost:5000/collections/foo/items/123>

Note: `.../items` queries which return an alternative representation to GeoJSON (which prompt a download) will have the response filename matching the collection name and appropriate file extension (e.g. `my-dataset.csv`)

10.1.2 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.

10.1.2.1 Providers

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

Provider	range-subset	subset	bbox	datetime
rasterio				
xarray				

Below are specific connection examples based on supported providers.

10.1.2.2 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_latlon.15x.15_2020081000_P000.grib2
  options: # optional creation options
    DATA_ENCODING: COMPLEX_PACKING
  format:
    name: GRIB
    mimetype: application/x-grib2
```

Note: The `rasterio` provider `format.name` directive **requires** a valid GDAL raster driver short name.

xarray

The `xarray` provider plugin reads and extracts `NetCDF` and `Zarr` data.

```
providers:
- type: coverage
  name: xarray
  data: tests/data/coads_sst.nc
  # optionally specify x/y/time fields, else provider will attempt
  # to derive automatically
  x_field: lat
  y_field: lon
  time_field: time
  format:
    name: netcdf
    mimetype: application/x-netcdf

providers:
- type: coverage
  name: xarray
  data: tests/data/analysed_sst.zarr
  format:
    name: zarr
    mimetype: application/zip
```

Note: `Zarr` files are directories with files and subdirectories. Therefore a zip file is returned upon request for said format.

10.1.2.3 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=GRIB`
- coverage access with comma-separated range-subset * `http://localhost:5000/collections/foo/coverage?range-subset=1,3`
- coverage access with subsetting * `http://localhost:5000/collections/foo/coverage?subset=lat(10,20)&subset=long(10,20)`

Note: `.../coverage` queries which return an alternative representation to CoverageJSON (which prompt a download) will have the response filename matching the collection name and appropriate file extension (e.g. `my-dataset.nc`)

10.1.3 Publishing map tiles to OGC API - Tiles

OGC API - Tiles provides access to geospatial data in the form of tiles (map, vector, etc.).

pygeoapi can publish tiles from local or remote data sources (including cloud object storage). To integrate tiles from a local data source, it is assumed that a directory tree of static tiles has been created on disk. Examples of tile generation software include (but are not limited to):

- [MapProxy](#)
- [tippecanoe](#)

10.1.3.1 Providers

pygeoapi core tile providers are listed below, along with supported storage types.

Provider	local	remote
MVT		

Below are specific connection examples based on supported providers.

10.1.3.2 Connection examples

MVT

The MVT provider plugin provides access to [Mapbox Vector Tiles](#).

```
providers:
- type: tile
  name: MVT
  data: tests/data/tiles/ne_110m_lakes # local directory tree
  # data: https://example.org/ne_110m_lakes/{z}/{x}/{y}.pbf
  options:
    metadata_format: raw # default | tilejson
    zoom:
      min: 0
      max: 5
    schemes:
      - WorldCRS84Quad
  format:
    name: pbf
    mimetype: application/vnd.mapbox-vector-tile
```

10.1.3.3 Data access examples

- list all collections * <http://localhost:5000/collections>
- overview of dataset * <http://localhost:5000/collections/foo>
- overview of dataset tiles * <http://localhost:5000/collections/foo/tiles>
- tile matrix metadata * <http://localhost:5000/collections/lakes/tiles/WorldCRS84Quad/metadata>
- tiles URI template * <http://localhost:5000/collections/lakes/tiles/{tileMatrixSetId}/{tileMatrix}/{tileRow}/{tileCol}?f=mvt>

10.1.4 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.

10.1.4.1 Configuration

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

10.1.4.2 Asynchronous support

By default, pygeoapi implements process execution (jobs) as synchronous mode. That is, when jobs are submitted, the process is executed and returned in real-time. Certain processes that may take time to execute, or be delegated to a scheduler/queue, are better suited to an asynchronous design pattern. This means that when a job is submitted in asynchronous mode, the server responds immediately with a reference to the job, which allows the client to periodically poll the server for the processing status of a given job.

pygeoapi provides asynchronous support by providing a ‘manager’ concept which, well, manages job execution. The manager concept is implemented as part of the pygeoapi *Customizing pygeoapi: plugins* architecture. pygeoapi provides a default manager implementation based on [TinyDB](#) for simplicity. Custom manager plugins can be developed for more advanced job management capabilities (e.g. Kubernetes, databases, etc.).

```
server:
  manager:
    name: TinyDB
    connection: /tmp/pygeoapi-process-manager.db
    output_dir: /tmp/
```

10.1.4.3 Putting it all together

To summarize how pygeoapi processes and managers work together:

```
* process plugins implement the core processing / workflow functionality
* manager plugins control and manage how processes are executed
```

10.1.4.4 Processing examples

- list all processes * <http://localhost:5000/processes>
- describe the `hello-world` process * <http://localhost:5000/processes/hello-world>
- show all jobs * <http://localhost:5000/jobs>
- execute a job for the `hello-world` process * `curl -X POST "http://localhost:5000/processes/hello-world/execution" -H "Content-Type: application/json" -d '{"inputs":{"name": "hi there2"}}'`

- execute a job for the hello-world process with a raw response (default) *

```
curl -X POST "http://localhost:5000/processes/hello-world/execution" -H "Content-Type: application/json" -d '{"inputs":{"name": "hi there2"}}'
```
- execute a job for the hello-world process with a response document *

```
curl -X POST "http://localhost:5000/processes/hello-world/execution" -H "Content-Type: application/json" -d '{"inputs":{"name": "hi there2"},"response":{"document": "hi there2"}}'
```
- execute a job for the hello-world process in asynchronous mode *

```
curl -X POST "http://localhost:5000/processes/hello-world/execution" -H "Content-Type: application/json" -d '{"mode": "async", "inputs":{"name": "hi there2"}}'
```

Todo: add more examples once OAProc implementation is complete

10.1.5 Publishing metadata to OGC API - Records

OGC API - Records 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.

10.1.5.1 Providers

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

Provider	properties (filters)	result-type	q	bbox	date-time	sortby	properties (display)
ElasticsearchCatalogue		results/hits					
TinyDBCatalogue		results/hits					

Below are specific connection examples based on supported providers.

10.1.5.2 Connection examples

ElasticsearchCatalogue

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 [OGC API - Records GeoJSON Features](#)
- index mappings must define the GeoJSON geometry as a `geo_shape`

```
providers:
- type: record
  name: ElasticsearchCatalogue
  data: http://localhost:9200/some_metadata_index
  id_field: identifier
  time_field: datetimefield
```

TinyDBCatalogue

Note: Elasticsearch 7 or greater is supported.

To publish a TinyDB index, the following are required in your index:

- indexes must be documents of valid [OGC API - Records GeoJSON Features](#)

```
providers:
- type: record
  name: TinyDBCatalogue
  data: /path/to/file.db
  id_field: identifier
  time_field: datetimefield
```

10.1.5.3 Metadata search examples

- overview of record collection * <http://localhost:5000/collections/metadata-records>
- queryables * <http://localhost:5000/collections/foo/queryables>
- browse records * <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 records (spatial) * <http://localhost:5000/collections/foo/items?bbox=-180,-90,180,90>
- query records (attribute) * <http://localhost:5000/collections/foo/items?propertyname=foo>
- query records (temporal) * <http://localhost:5000/collections/my-metadata/items?datetime=2020-04-10T14:11:00Z>
- query features (temporal) and sort ascending by a property (if no +/- indicated, + is assumed) * <http://localhost:5000/collections/my-metadata/items?datetime=2020-04-10T14:11:00Z&sortby=datetime>
- query features (temporal) and sort descending by a property * <http://localhost:5000/collections/my-metadata/items?datetime=2020-04-10T14:11:00Z&sortby=-datetime>
- fetch a specific record * <http://localhost:5000/collections/my-metadata/items/123>

10.1.6 Publishing data to OGC API - Environmental Data Retrieval

The [OGC Environmental Data Retrieval \(EDR\) \(API\)](#) provides a family of lightweight query interfaces to access spatio-temporal data resources.

To add spatio-temporal data to pygeoapi for EDR query interfaces, you can use the dataset example in [Configuration](#) as a baseline and modify accordingly.

10.1.6.1 Providers

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

Provider	coords	parameter-name	datetime
xarray-edr			

Below are specific connection examples based on supported providers.

10.1.6.2 Connection examples

xarray-edr

The *xarray-edr* provider plugin reads and extracts [NetCDF](#) and [Zarr](#) data via [xarray](#).

```
providers:
- type: edr
  name: xarray-edr
  data: tests/data/coads_sst.nc
  # optionally specify x/y/time fields, else provider will attempt
  # to derive automagically
  x_field: lat
  y_field: lon
  time_field: time
  format:
    name: netcdf
    mimetype: application/x-netcdf

providers:
- type: edr
  name: xarray-edr
  data: tests/data/analysed_sst.zarr
  format:
    name: zarr
    mimetype: application/zip
```

Note: [Zarr](#) files are directories with files and subdirectories. Therefore a zip file is returned upon request for said format.

10.1.6.3 Data access examples

- list all collections * `http://localhost:5000/collections`
- overview of dataset * `http://localhost:5000/collections/foo`
- dataset position query * `http://localhost:5000/collections/foo/position?coords=POINT(-75%2045)`
- dataset position query for a specific parameter * `http://localhost:5000/collections/foo/position?coords=POINT(-75%2045)¶meter-name=SST`
- dataset position query for a specific parameter and time step * `http://localhost:5000/collections/foo/position?coords=POINT(-75%2045)¶meter-name=SST&datetime=2000-01-16`

10.1.7 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:

10.1.7.1 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.

10.1.7.2 pygeometa metadata control files

pygeoapi's STAC filesystem functionality supports [pygeometa](#) MCF files residing in the same directory as data files. If an MCF file is found, it will be used as part of generating the STAC item metadata (e.g. a file named `birds.csv` having an associated `birds.yml` file). If no MCF file is found, then pygeometa will generate the STAC item metadata from configuration and by reading the data's properties.

10.1.7.3 Publishing ESRI Shapefiles

ESRI Shapefile publishing requires to specify all required component file extensions (.shp, .shx, .dbf) with the provider `file_types` option.

10.1.7.4 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.

11.1 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

11.2 Example: custom pygeoapi vector data provider

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

11.2.1 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"""

        super().__init__(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=[],
              select_properties=[], skip_geometry=False, **kwargs):

        # optionally specify the output filename pygeoapi can use as part
        # of the response (HTTP Content-Disposition header)
        self.filename = "my-cool-filename.dat"

        # 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.

Note: You can add language support to your plugin using [these guides](#).

11.2.2 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
```

Note: The United States Geological Survey has created a [Cookiecutter](#) project for creating pygeoapi plugins. See the [pygeoapi-plugin-cookiecutter](#) project to get started.

11.3 Example: custom pygeoapi raster data provider

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

11.3.1 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"""

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

    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', **kwargs):
        # process bands and subsets parameters
        # query/extract coverage data

        # optionally specify the output filename pygeoapi can use as part
        # of the response (HTTP Content-Disposition header)
        self.filename = "my-cool-filename.dat"

        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.

11.4 Example: custom pygeoapi formatter

11.4.1 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"""
```

(continues on next page)

(continued from previous page)

```

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

    super().__init__({'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

```

11.5 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.

11.6 Featured plugins

The following plugins provide useful examples of pygeoapi plugins implemented by downstream applications.

Plugin(s)	Organization/Project	Description
msc-pygeoapi	Meteorological Service of Canada	processes for weather/climate/water data work-flows
pygeoapi-kubernetes-papermill	Euro Data Cube	processes for executing Jupyter notebooks via Kubernetes
local-outlier-factor-plugin	Manaaki Whenua – Landcare Research	processes for local outlier detection
ogc-edc	Euro Data Cube	coverage provider atop the EDC API
nldi_xstool	United States Geological Survey	Water data processing

HTML TEMPLATING

pygeoapi uses [Jinja](#) as its templating engine to render HTML and [Flask](#) to provide route paths of the API that returns HTTP responses. For complete details on how to use these modules, refer to the [Jinja documentation](#) and the [Flask documentation](#).

The default pygeoapi configuration has `server.templates` commented out and defaults to the pygeoapi `pygeoapi/templates` and `pygeoapi/static` folder. To point to a different set of template configuration, you can edit your configuration:

```
server:
  templates:
    path: /path/to/jinja2/templates/folder # jinja2 template HTML files
    static: /path/to/static/folder # css, js, images and other static files_
    ↪referenced by the template
```

Note: the URL path to your static folder will always be `/static` in your deployed web instance of pygeoapi.

Your templates folder should mimic the same file names and structure of the default pygeoapi templates. Otherwise, you will need to modify `api.py` accordingly.

Note that you need only copy and edit the templates you are interested in updating. For example, if you are only interested in updating the `landing_page.html` template, then create your own version of the only that same file. When pygeoapi detects that a custom HTML template is being used, it will look for the custom template in `server.templates.path`. If it does not exists, pygeoapi will render the default HTML template for the given endpoint/request.

Linking to a static file in your HTML templates can be done using Jinja syntax and the exposed `config['server']['url']`:

```
<!-- CSS example -->
<link rel="stylesheet" href="{ config['server']['url'] }/static/css/default.css">
<!-- JS example -->
<script src="{ config['server']['url'] }/static/js/main.js"></script>
<!-- Image example with metadata -->

```

12.1 Featured templates

The following themes provide useful examples of pygeoapi templates implemented by downstream applications.

Plugin(s)	Organization/Project	Description
pygeoapi-skin-dashboard	GeoCat bv	skin for pygeoapi based on a typical dashboard interface

CQL SUPPORT

13.1 Limitations

The support to CQL is limited to [Simple CQL filter](#) and thus it allows to query with the following predicates:

- comparison predicates
- spatial predicates
- temporal predicates

13.2 Formats

At the moment pygeoapi supports only the CQL dialect with the JSON encoding [CQL-JSON](#).

13.3 Providers

As of now the available providers supported for CQL filtering are limited to only [Elasticsearch](#).

13.3.1 Queries

The following type of queries are supported right now:

- between predicate query
- Logical and query with between and eq expression
- Spatial query with bbox

13.3.2 Examples

A between example for a specific property through an HTTP POST request:

```
curl --location --request POST 'http://localhost:5000/collections/nhsl_hazard_threat_
↪all_indicators_s_bc/items?f=json&limit=50&filter-lang=cql-json' \
--header 'Content-Type: application/query-cql-json' \
--data-raw '{
  "between": {
    "value": { "property": "properties.MHn_Intensity" },
```

(continues on next page)

(continued from previous page)

```
"lower": 0.59,  
"upper": 0.60  
}  
'
```

MULTILINGUAL SUPPORT

pygeoapi is language-aware and can handle multiple languages if these have been defined in pygeoapi's configuration (see [maintainer guide](#)). Providers can also handle multiple languages if configured. These may even be different from the languages that pygeoapi supports. Out-of-the-box, pygeoapi “speaks” English. System messages and exceptions are always English only.

The following sections provide more information how to use and set up languages in pygeoapi.

14.1 End user guide

There are 2 ways to affect the language of the results returned by pygeoapi, both for the HTML and JSON(-LD) formats:

1. After the requested pygeoapi URL, append a `lang=<code>` query parameter, where `<code>` should be replaced by a well-known language code. This can be an ISO 639-1 code (e.g. *de* for German), optionally accompanied by an ISO 3166-1 alpha-2 country code (e.g. *de-CH* for Swiss-German). Please refer to this [W3C article](#) for more information or this [list of language codes](#) for more examples. Another option is to send a complex definition with quality weights (e.g. *de-CH, de;q=0.9, en;q=0.8, fr;q=0.7, *;q=0.5*). pygeoapi will then figure out the best match for the requested language.

For example, to view the pygeoapi landing page in Canadian-French, you could use this URL:

<https://demo.pygeoapi.io/master?lang=fr-CA>

2. Alternatively, you can set an `Accept-Language` HTTP header for the requested pygeoapi URL. Language tags that are valid for the `lang` query parameter are also valid for this header value. Please note that if your client application (e.g. browser) is configured for a certain language, it will likely set this header by default, so the returned response should be translated to the language of your client app. If you don't want this, you can either change the language of your client app or append the `lang` parameter to the URL, which will override any language defined in the `Accept-Language` header.

14.1.1 Notes

- If pygeoapi cannot find a good match to the requested language, the response is returned in the default language (US English mostly). The default language is the *first* language defined in pygeoapi's server configuration YAML (see [maintainer guide](#)).
- Even if pygeoapi *itself* supports the requested language, provider plugins may not support that particular language or perhaps don't even support any language at all. In that case the provider will reply in its own “unknown” language, which may not be the same language as the default pygeoapi server language set in the `Content-Language` HTTP response header.

- It is up to the creator of the provider to properly define at least 1 supported language in the provider configuration, as described in the [developer guide](#). This will ensure that the `Content-Language` HTTP response header is always set properly.
- If pygeoapi found a match to the requested language, the response will include a `Content-Language` HTTP header, set to the best-matching server language code. This is the default behavior for most pygeoapi requests. However, note that some responses (e.g. exceptions) always have a `Content-Language: en-US` header, regardless of the requested language.
- For results returned by a **provider**, the `Content-Language` HTTP header will be set to the best-matching provider language or the best-matching pygeoapi server language if the provider is not language-aware.
- If the provider supports a requested language, but pygeoapi does *not* support that same language, the `Content-Language` header will contain both the provider language *and* the best-matching pygeoapi server language.
- Please note that the `Content-Language` HTTP response header only *indicates the language of the intended audience*. It does not necessarily mean that the content is actually written in that particular language.

14.2 Maintainer guide

Every pygeoapi instance should support at least 1 language. In the server configuration, there must be a `language` or a `languages` (note the *s*) property. The property can be set to a single language tag or a list of tags respectively.

If you wish to set up a multilingual pygeoapi instance, you will have to add more than 1 language to the server configuration YAML file (i.e. `pygeoapi-config.yml`). First, you will have to add the supported language tags/codes as a list. For example, if you wish to support American English and Canadian French, you could do:

```
server:
  bind: ...
  url: ...
  mimetype: ...
  encoding: ...
  languages:
    - en-US
    - fr-CA
```

Next, you will have to provide translations for the configured languages. This involves 3 steps:

1. *Add translations for configurable text values* in the pygeoapi configuration;
2. Verify if there are any Jinja2 HTML template translations for the configured language(s);
3. Make sure that the provider plugins you need can handle this language as well, if you have the ability to do so. See the [developer guide](#) for more details.

14.2.1 Notes

- The **first** language you define in the configuration determines the default language, i.e. the language that pygeoapi will use if no other language was requested or no best match for the requested language could be found.
- It is not possible to **disable** language support in pygeoapi. The functionality is always on and a `Content-Language` HTTP response header is always set. If results should be available in a single language, you'd have to set that language only in the pygeoapi configuration.

- Results returned from a provider may be in a different language than pygeoapi’s own server language. The “raw” requested language is always passed on to the provider, even if pygeoapi itself does not support it. For more information, see the *end user guide* and the *developer guide*.

14.2.2 Add translations for configurable text values

For most of the text values in pygeoapi’s server configuration where it makes sense, you can add translations. Consider the `metadata` section for example. The English-only version looks similar to this:

```
metadata:
  identification:
    title: pygeoapi default instance
    description: pygeoapi provides an API to geospatial data
    keywords:
      - geospatial
      - data
      - api
```

If you wish to make these text values available in English and French, you could use the following language struct:

```
metadata:
  identification:
    title:
      en: pygeoapi default instance
      fr: instance par défaut de pygeoapi
    description:
      en: pygeoapi provides an API to geospatial data
      fr: pygeoapi fournit une API aux données géospatiales
    keywords:
      en:
        - geospatial
        - data
        - api
      fr:
        - géospatiale
        - données
        - api
```

In other words: each plain text value should be replaced by a dictionary, where the language code is the key and the translated text represents the matching value. For lists, this can be applied as well (see `keywords` example above), as long as you nest the entire list under a language key instead of each list item.

Note that the example above uses generic language tags, but you can also supply more localized tags (with a country code) if required. pygeoapi should always be able find the best match to the requested language, i.e. if the user wants Swiss-French (*fr-CH*) but pygeoapi can only find *fr* tags, those values will be returned. However, if a *fr-CH* tag can also be found, that value will be returned and not the *fr* value.

Warning: A language struct is only translated if all language tags (keys) in the struct are valid locales.

Todo: Add docs on HTML templating.

14.3 Translator guide

Hardcoded strings in pygeoapi templates are translated using the Babel translation system. Translation files are stored on the `/locale` folder. Translators can follow these steps to prepare their environment for translations.

1. Extract from latest code the keys to be translated. These keys are captured in a `.pot` file. Note that the `.pot` file is not to be stored in version control, but as an intermediary file used to update `/locale/*/LC_MESSAGES/messages.po` files:

```
pybabel extract -F babel-mapping.ini -o locale/messages.pot ./
```

2. Update the existing `.po` language file:

```
pybabel update -d locale -l fr -i locale/messages.pot
```

3. Open the relevant `.po` file and contribute your translations. Then compile a `.mo` file to be used by the application:

```
pybabel compile -d locale -l fr
```

Within jinja templates keys are prepared to be translated by wrapping them in:

```
{% trans %}Key{% endtrans %}
```

14.4 Developer guide

If you are a developer who wishes to create a pygeoapi provider plugin that “speaks” a certain language, you will have to fully implement this yourself. Needless to say, if your provider depends on some backend, it will only make sense to implement language support if the backend can be queried in another language as well.

You are free to set up the language support anyway you like, but there are a couple of steps you’ll have to walk through:

1. You will have to define the supported languages in the provider configuration YAML. This can be done in a similar fashion as the `languages` configuration for pygeoapi itself, as described in the [maintainer guide](#) section above. For example, a TinyDB records provider that supports English and French could be set up like:

```
my-records:
  type: collection
  ..
  providers:
    - type: record
      name: TinyDBCatalogue
      data: ..
      languages:
        - en
        - fr
```

2. If your provider implements any of the `query`, `get` or `get_metadata` methods of the base class and you wish to make them language-aware, either add an implicit `**kwargs` parameter or an explicit `language=None` parameter to the method signature.

An example Python code block for a custom provider with a language-aware `query` method could look like this:

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

(continues on next page)

(continued from previous page)

```
def __init__(self, provider_def):
    super().__init__(provider_def)

def query(self, startindex=0, limit=10, resulttype='results', bbox=[],
          datetime_=None, properties=[], sortby=[], select_properties=[],
          skip_geometry=False, q=None, language=None):
    LOGGER.debug(f'Provider queried in {language.english_name} language')
    # Implement your logic here, returning JSON in the requested language
```

Alternatively, you could also use `**kwargs` in the `query` method and get the `language` value:

```
def query(self, **kwargs):
    LOGGER.debug(f"Provider locale set to: {kwargs.get('language')}")
    # Implement your logic here, returning JSON in the requested language
```

This is all that is required. The `pygeoapi` API class will make sure that the correct HTTP `Content-Language` headers are set on the response object.

14.4.1 Notes

- If your provider implements any of the aforementioned `query`, `get` and `get_metadata` methods, it **must** add a `**kwargs` or `language=None` parameter, even if it does not need to use the `language` parameter.
- Contrary to the `pygeoapi` server configuration, adding a `language` or `languages` (both are supported) property to the provider definition is **not** required and may be omitted. In that case, the passed-in `language` parameter language-aware provider methods (`query`, `get`, etc.) will be set to `None`. This results in the following behavior:
 - HTML responses returned from the providers will have the `Content-Language` header set to the best-matching `pygeoapi` server language.
 - JSON(-LD) responses returned from providers will **not** have a `Content-Language` header if `language` is `None`.
- If the provider supports a requested language, the passed-in `language` will be set to the best matching [Babel Locale instance](#). Note that this may be the provider default language if no proper match was found. No matter the output format, API responses returned from providers will always contain a best-matching `Content-Language` header if one or more supported provider languages were defined.
- For general information about building plugins, please visit the [Customizing pygeoapi: plugins](#) page.

DEVELOPMENT

15.1 Codebase

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

15.2 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 [GitHub Actions 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.

15.3 CQL extension lifecycle

15.3.1 Limitations

This workflow is valid only for the *CQL-JSON* format.

15.3.2 Schema

The Common Query Language (CQL) is the part 3 of the standard OGC API - Features. This extension has its specification available at [OGC API - Features - Part 3: Filtering and the Common Query Language \(CQL\)](#) and the schema exists in development at [cql.json](#).

15.3.3 Model generation

pygeoapi uses a class-based python model interface to translate the schema into python objects defined by `pydantic` models. The model is generated with the pre-processing of the schema through the utility `datamodel-codegen`:

```
# Generate from local downloaded json schema file
datamodel-codegen --input ~/Download/cql-schema.json --input-file-type jsonschema --
↳output ./pygeoapi/models/cql_update.py --class-name CQLModel
```

15.3.4 How to merge

Once the new `pydantic` models have been generated then the content of the python file `cql_update.py` can be used to replace the old classes within the `cql.py` file. Update everything above the function `get_next_node` and then verify if the tests for the CQL are still passing, for example `test_post_cql_json_between_query` in `tests/test_elasticsearch_provider.py`.

15.4 Working with Spatialite on OSX

15.4.1 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:

```
LD_FLAGS="-L/usr/local/opt/sqlite/lib -L/usr/local/opt/zlib/lib" CPP_FLAGS="-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
```

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.

16.1 CITE instance

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

16.2 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
SEVENTEEN

CONTRIBUTING

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

18.1 Community

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

FURTHER READING

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

- [Default pygeoapi presentation](#)
- [OGC API](#)

20.1 Code

The MIT License (MIT)

Copyright © 2018-2022 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.

20.2 Documentation

The documentation is released under the [Creative Commons Attribution 4.0 International \(CC BY 4.0\)](https://creativecommons.org/licenses/by/4.0/) license.

API DOCUMENTATION

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

21.1 API

Root level code of pygeoapi, parsing content provided by web framework. 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)

    delete_job (job_id) → Tuple[dict, int, str]
        Delete a process job

        Parameters job_id – job identifier
        Returns tuple of headers, status code, content

    get_exception (status, headers, format_, code, description) → Tuple[dict, int, str]
        Exception handler

        Parameters
        • status – HTTP status code
        • headers – dict of HTTP response headers
        • format – format string
        • code – OGC API exception code
        • description – OGC API exception code

        Returns tuple of headers, status, and message

    get_format_exception (request) → Tuple[dict, int, str]
        Returns a format exception.

        Parameters request – An APIRequest instance.
```

Returns tuple of (headers, status, message)

class `pygeoapi.api.APIRequest` (*request, supported_locales*)

Transforms an incoming server-specific Request into an object with some generic helper methods and properties.

Note: Typically, this instance is created automatically by the `pre_process()` decorator. **Every** API method that has been routed to a REST endpoint should be decorated by the `pre_process()` function. Therefore, **all** routed API methods should at least have 1 argument that holds the (transformed) request.

The following example API method will:

- transform the incoming Flask/Starlette *Request* into an *APIRequest* using the `pre_process()` decorator;
- call `is_valid()` to check if the incoming request was valid, i.e. that the user requested a valid output format or no format at all (which means the default format);
- call `API.get_format_exception()` if the requested format was invalid;
- create a *dict* with the appropriate *Content-Type* header for the requested format and a *Content-Language* header if any specific language was requested.

```
@pre_process
def example_method(self, request: Union[APIRequest, Any], custom_arg):
    if not request.is_valid():
        return self.get_format_exception(request)

    headers = request.get_response_headers()

    # generate response_body here

    return headers, 200, response_body
```

The following example API method is similar as the one above, but will also allow the user to request a non-standard format (e.g. `f=xml`). If *xml* was requested, we set the *Content-Type* ourselves. For the standard formats, the *APIRequest* object sets the *Content-Type*.

```
@pre_process
def example_method(self, request: Union[APIRequest, Any], custom_arg):
    if not request.is_valid(['xml']):
        return self.get_format_exception(request)

    content_type = 'application/xml' if request.format == 'xml' else None
    headers = request.get_response_headers(content_type)

    # generate response_body here

    return headers, 200, response_body
```

Note that you don't *have* to call `is_valid()`, but that you can also perform a custom check on the requested output format by looking at the `format` property. Other query parameters are available through the `params` property as a *dict*. The request body is available through the `data` property.

Note: If the request data (body) is important, **always** create a new *APIRequest* instance using the `with_data()` factory method. The `pre_process()` decorator will use this automatically.

Parameters

- **request** – The web platform specific Request instance.
- **supported_locales** – List or set of supported Locale instances.

`__init__` (*request, supported_locales*)

Initialize self. See help(type(self)) for accurate signature.

`__weakref__`

list of weak references to the object (if defined)

`_get_format` (*headers*) → Optional[*str*]

Get *Request* format type from query parameters or headers.

Parameters *headers* – Dict of Request headers

Returns format value or None if not found/specified

`_get_locale` (*headers, supported_locales*)

Detects locale from “lang=<language>” param or *Accept-Language* header. Returns a tuple of (raw, locale) if found in params or headers. Returns a tuple of (raw default, default locale) if not found.

Parameters

- **headers** – A dict with Request headers
- **supported_locales** – List or set of supported Locale instances

Returns A tuple of (str, Locale)

static `_get_params` (*request*)

Extracts the query parameters from the *Request* object.

Parameters *request* – A Flask or Starlette Request instance

Returns *ImmutableMultiDict* or empty *dict*

property data

Returns the additional data send with the Request (bytes)

property format

Returns the content type format from the request query parameters or headers.

Returns Format name or None

`get_linkrel` (*format_*: *str*) → *str*

Returns the hyperlink relationship (rel) attribute value for the given API format string.

The string is compared against the request format and if it matches, the value ‘self’ is returned. Otherwise, ‘alternate’ is returned. However, if *format_* is ‘json’ and *no* request format was found, the relationship ‘self’ is returned as well (JSON is the default).

Parameters *format* – The format to compare the request format against.

Returns A string ‘self’ or ‘alternate’.

`get_request_headers` (*headers*) → *dict*

Obtains and returns a dictionary with Request object headers.

This method adds the headers of the original request and makes them available to the API object.

Returns A header dict

get_response_headers (*force_lang*: `babel.core.Locale = None`, *force_type*: `str = None`,
force_encoding: `str = None`) → `dict`

Prepares and returns a dictionary with Response object headers.

This method always adds a ‘Content-Language’ header, where the value is determined by the ‘lang’ query parameter or ‘Accept-Language’ header from the request. If no language was requested, the default pygeoapi language is used, unless a *force_lang* override was specified (see notes below).

A ‘Content-Type’ header is also always added to the response. If the user does not specify *force_type*, the header is based on the *format* APIRequest property. If that is invalid, the default MIME type *application/json* is used.

..note:: If a *force_lang* override is applied, that language is always set as the ‘Content-Language’, regardless of a ‘lang’ query parameter or ‘Accept-Language’ header. If an API response always needs to be in the same language, ‘force_lang’ should be set to that language.

Parameters

- **force_lang** – An optional Content-Language header override.
- **force_type** – An optional Content-Type header override.
- **force_encoding** – An optional Content-Encoding header override.

Returns A header dict

property headers

Returns the dictionary of the headers from the request.

Returns Request headers dictionary

is_valid (*additional_formats*=None) → `bool`

Returns True if:

- the format is not set (None)
- the requested format is supported
- the requested format exists in a list if additional formats

Note: Format names are matched in a case-insensitive manner.

Parameters **additional_formats** – Optional additional supported formats list

Returns `bool`

property locale

Returns the user-defined locale from the request object. If no locale has been defined or if it is invalid, the default server locale is returned.

Note: The locale here determines the language in which pygeoapi should return its responses. This may not be the language that the user requested. It may also not be the language that is supported by a collection provider, for example. For this reason, you should pass the *raw_locale* property to the `l10n.get_plugin_locale()` function, so that the best match for the provider can be determined.

Returns `babel.core.Locale`

property params

Returns the Request query parameters dict

property path_info

Returns the web server request path info part

property raw_locale

Returns the raw locale string from the *Request* object. If no “lang” query parameter or *Accept-Language* header was found, *None* is returned. Pass this value to the `l10n.get_plugin_locale()` function to let the provider determine a best match for the locale, which may be different from the locale used by pygeoapi’s UI.

Returns a locale string or None

classmethod with_data (*request, supported_locales*) → *pygeoapi.api.APIRequest*

Factory class method to create an *APIRequest* instance with data.

If the request body is required, an *APIRequest* should always be instantiated using this class method. The reason for this is, that the Starlette request body needs to be awaited (async), which cannot be achieved in the `__init__()` method of the *APIRequest*. However, *APIRequest* can still be initialized using `__init__()`, but then the *data* property value will always be empty.

Parameters

- **request** – The web platform specific Request instance.
- **supported_locales** – List or set of supported Locale instances.

Returns An *APIRequest* instance with data.

```
pygeoapi.api.FORMAT_TYPES = {'html': 'text/html', 'json': 'application/json', 'jsonld':
```

Formats allowed for ?f= requests (order matters for complex MIME types)

```
pygeoapi.api.HEADERS = {'Content-Type': 'application/json', 'X-Powered-By': 'pygeoapi 0.12.0'}
```

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

```
pygeoapi.api.SYSTEM_LOCALE = Locale('en', territory='US')
```

Locale used for system responses (e.g. exceptions)

```
pygeoapi.api.gzip(func)
```

Decorator that compresses the content of an outgoing API result instance if the Content-Encoding response header was set to gzip.

Parameters *func* – decorated function

Returns *func*

```
pygeoapi.api.pre_process(func)
```

Decorator that transforms an incoming Request instance specific to the web framework (i.e. Flask or Starlette) into a generic *APIRequest* instance.

Parameters *func* – decorated function

Returns *func*

```
pygeoapi.api.validate_bbox(value=None) → list
```

Helper function to validate bbox parameter

Parameters *value* – list of minx, miny, maxx, maxy

Returns bbox as list of float values

```
pygeoapi.api.validate_datetime(resource_def, datetime_=None) → str
```

Helper function to validate temporal parameter

Parameters

- **resource_def** – *dict* of configuration resource definition
- **datetime** – *str* of datetime parameter

Returns *str* of datetime input, if valid

`pygeoapi.api.validate_subset` (*value: str*) → *dict*
Helper function to validate subset parameter

Parameters **value** – *subset* parameter

Returns *dict* of axis/values

21.2 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.execute_process_jobs(process_id)`
OGC API - Processes execution endpoint

Parameters `process_id` – process identifier

Returns HTTP response

`pygeoapi.flask_app.get_collection_edr_query(collection_id, instance_id=None)`
OGC EDR API endpoints

Parameters

- `collection_id` – collection identifier
- `instance_id` – instance identifier

Returns HTTP response

`pygeoapi.flask_app.get_collection_tiles(collection_id=None)`
OGC open api collections tiles access point

Parameters `collection_id` – collection identifier

Returns HTTP response

`pygeoapi.flask_app.get_collection_tiles_data(collection_id=None, tileMa-
trixSetId=None, tileMatrix=None,
tileRow=None, tileCol=None)`
OGC open api collection tiles service data

Parameters

- `collection_id` – collection identifier
- `tileMatrixSetId` – identifier of tile matrix set
- `tileMatrix` – identifier of {z} matrix index
- `tileRow` – identifier of {y} matrix index
- `tileCol` – identifier of {x} matrix index

Returns HTTP response

`pygeoapi.flask_app.get_collection_tiles_metadata(collection_id=None, tileMa-
trixSetId=None)`
OGC open api collection tiles service metadata

Parameters

- `collection_id` – collection identifier
- `tileMatrixSetId` – identifier of tile matrix set

Returns HTTP response

`pygeoapi.flask_app.get_job_result(job_id=None)`
OGC API - Processes job result endpoint

Parameters `job_id` – job identifier

Returns HTTP response

`pygeoapi.flask_app.get_job_result_resource(job_id, resource)`
OGC API - Processes job result resource endpoint

Parameters

- **job_id** – job identifier
- **resource** – job resource

Returns HTTP response

`pygeoapi.flask_app.get_jobs(job_id=None)`
OGC API - Processes jobs endpoint

Parameters **job_id** – job identifier

Returns HTTP response

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

Parameters **process_id** – process identifier

Returns HTTP response

`pygeoapi.flask_app.get_response(result: tuple)`
Creates a Flask Response object and updates matching headers.

Parameters **result** – The result of the API call. This should be a tuple of (headers, status, content).

Returns A Response instance.

`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.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

21.3 Logging

Logging system

`pygeoapi.log.setup_logger(logging_config)`

Setup configuration

Parameters `logging_config` – logging specific configuration

Returns void (creates logging instance)

21.4 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

`pygeoapi.openapi.validate_openapi_document(instance_dict)`

Validate an OpenAPI document against the OpenAPI schema

Parameters `instance_dict` – dict of OpenAPI instance

Returns *bool* of validation

21.5 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 = {'formatter': {'CSV': 'pygeoapi.formatter.csv.CSVFormatter'}}`,

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

21.6 Utils

Generic util functions used in the code

class `pygeoapi.util.JobStatus`

Bases: `enum.Enum`

Enum for the job status options specified in the WPS 2.0 specification

`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 (allowing for an open interval using null)

`pygeoapi.util.file_modified_iso8601(filepath)`

Provide a file's ctime in ISO8601

Parameters **filepath** – path to file

Returns string of ISO8601

`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.filter_providers_by_type` (*providers*, *type*)
 helper function to filter a list of providers by type

Parameters

- **providers** – list
- **type** – str

Returns filtered dict provider

`pygeoapi.util.format_datetime` (*value*, *format_*='%Y-%m-%dT%H:%M:%S.%fZ')
 Parse datetime as ISO 8601 string; re-present it in particular format for display in HTML

Parameters

- **value** – str of ISO datetime
- **format** – str of datetime format for strftime

Returns string

`pygeoapi.util.format_duration` (*start*, *end*=None)
 Parse a start and (optional) end datetime as ISO 8601 strings, calculate the difference, and return that duration as a string.

Parameters

- **start** – str of ISO datetime
- **end** – str of ISO datetime, defaults to *start* for a 0 duration

Returns string

`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_envelope` (*coords_list*: List[List[float]])
 helper function to get the envelope for a given coordinates list through the Shapely API.

Parameters *coords_list* – list of coordinates

Returns list of the envelope's coordinates

`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.human_size` (*nbytes*)
Provides human readable file size

source: <https://stackoverflow.com/a/14996816>

Parameters

- **nbytes** – int of file size (bytes)
- **units** – list of unit abbreviations

Returns string of human readable filesize

`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.read_data` (*path*)
helper function to read data (file or network)

`pygeoapi.util.render_j2_template` (*config, template, data, locale_=None*)
render Jinja2 template

Parameters

- **config** – dict of configuration
- **template** – template (relative path)
- **data** – dict of data
- **locale** – the requested output Locale

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.url_join(*parts)`
helper function to join a URL from a number of parts/fragments. Implemented because `urllib.parse.urljoin` strips subpaths from host urls if they are specified

Per <https://github.com/geopython/pygeoapi/issues/695>

Parameters **parts** – list of parts to join

Returns str of resulting URL

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

Parameters **fh** – file handle

Returns *dict* representation of YAML

21.7 Formatter package

Output formatter package

21.7.1 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.formatter.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

exception `pygeoapi.formatter.base.FormatterGenericError`

Bases: `Exception`

formatter generic error

__weakref__

list of weak references to the object (if defined)

exception `pygeoapi.formatter.base.FormatterSerializationError`

Bases: `pygeoapi.formatter.base.FormatterGenericError`

formatter serialization error

21.7.2 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

21.8 Process package

OGC process package, each process is an independent module

21.8.1 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

Parameters

- **processor_def** – processor definition
- **process_metadata** – process metadata *dict*

Returns pygeoapi.processor.base.BaseProvider

__repr__ ()
 Return repr(self).

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

execute ()
 execute the process

Returns tuple of MIME type and process response

exception pygeoapi.process.base.**ProcessorExecuteError**
 Bases: `pygeoapi.process.base.ProcessorGenericError`

query / backend error

exception pygeoapi.process.base.**ProcessorGenericError**
 Bases: `Exception`

processor generic error

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

21.8.2 hello_world

Hello world example process

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

Hello World Processor example

__init__ (*processor_def*)
 Initialize object

Parameters **processor_def** – provider definition

Returns pygeoapi.process.hello_world>HelloWorldProcessor

__repr__ ()
 Return repr(self).

execute (*data*)
 execute the process

Returns tuple of MIME type and process response

```
pygeoapi.process.hello_world.PROCESS_METADATA = {'description': {'en': 'An example process'}}
Process metadata and description
```

21.9 Provider

Provider module containing the plugins wrapping data sources

21.9.1 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.provider.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.ProviderNoDataError

Bases: *pygeoapi.provider.base.ProviderGenericError*

provider no data 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

21.9.2 CSV provider

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

load (*startindex=0, limit=10, resulttype='results', identifier=None, bbox=[], datetime=None, properties=[], select_properties=[], skip_geometry=False, q=None*)
Load CSV data

Parameters

- **startindex** – starting record to return (default 0)
- **limit** – number of records to return (default 10)
- **datetime** – temporal (datestamp or extent)
- **resulttype** – return results or hit limit (default results)
- **properties** – list of tuples (name, value)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

Returns dict of GeoJSON FeatureCollection

get (*identifier, **kwargs*)
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=[], select_properties=[], skip_geometry=False, q=None, **kwargs*)
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)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

Returns dict of GeoJSON FeatureCollection

21.9.3 Elasticsearch provider

21.9.4 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 (*skip_geometry=None, properties=[], select_properties=[]*)

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, **kwargs*)

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=[], select_properties=[], skip_geometry=False, q=None, **kwargs*)
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)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

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

21.9.5 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 Shape-
    file, 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/hotosm\_bdi\_waterways/items?resulttype=hits

        Returns GeoJSON FeaturesCollection

    get(identifier, **kwargs)
        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=[],
        sortby=[], select_properties=[], skip_geometry=False, q=None, **kwargs)
        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)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

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

21.9.6 postgresql

class pygeoapi.provider.postgresql.DatabaseConnection(*conn_dic, table, properties=[], context='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=[]*)

Generates 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

__make_orderby (*sortby*)

Private function: Make STA filter from query properties

Parameters *sortby* – list of dicts (property, order)

Returns STA \$orderby string

get (*identifier, **kwargs*)

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=[], select_properties=[], skip_geometry=False, q=None, **kwargs*)
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)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

Returns GeoJSON FeaturesCollection

21.9.7 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=[]*)

Generates 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 spatialite, 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*, ***kwargs*)

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=[]*, *orderby=[]*, *select_properties=[]*, *skip_geometry=False*, *q=None*, ***kwargs*)

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)
- **orderby** – list of dicts (property, order)
- **select_properties** – list of property names
- **skip_geometry** – bool of whether to skip geometry (default False)
- **q** – full-text search term(s)

Returns GeoJSON FeaturesCollection

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

- `pygeoapi.api`, 81
- `pygeoapi.flask_app`, 86
- `pygeoapi.formatter`, 93
- `pygeoapi.formatter.base`, 93
- `pygeoapi.formatter.csv_`, 94
- `pygeoapi.log`, 89
- `pygeoapi.openapi`, 89
- `pygeoapi.plugin`, 90
- `pygeoapi.process`, 94
- `pygeoapi.process.base`, 95
- `pygeoapi.process.hello_world`, 95
- `pygeoapi.provider`, 96
- `pygeoapi.provider.base`, 96
- `pygeoapi.provider.csv_`, 98
- `pygeoapi.provider.geojson`, 99
- `pygeoapi.provider.ogr`, 100
- `pygeoapi.provider.postgresql`, 103
- `pygeoapi.provider.sqlite`, 104
- `pygeoapi.util`, 90

Symbols

<code>__repr__()</code>	(<i>pygeoapi.provider.base.BaseProvider</i> method), 96
<code>_PostgreSQLProvider__get_where_clauses()</code>	(<i>pygeoapi.provider.postgresql.PostgreSQLProvider</i> method), 103
<code>_PostgreSQLProvider__response_feature()</code>	(<i>pygeoapi.provider.postgresql.PostgreSQLProvider</i> method), 103
<code>_PostgreSQLProvider__response_feature_hits()</code>	(<i>pygeoapi.provider.postgresql.PostgreSQLProvider</i> method), 103
<code>_SQLiteGPKGProvider__get_where_clauses()</code>	(<i>pygeoapi.provider.sqlite.SQLiteGPKGProvider</i> method), 104
<code>_SQLiteGPKGProvider__load()</code>	(<i>pygeoapi.provider.sqlite.SQLiteGPKGProvider</i> method), 104
<code>_SQLiteGPKGProvider__response_feature()</code>	(<i>pygeoapi.provider.sqlite.SQLiteGPKGProvider</i> method), 104
<code>_SQLiteGPKGProvider__response_feature_hits()</code>	(<i>pygeoapi.provider.sqlite.SQLiteGPKGProvider</i> method), 104
<code>__init__()</code>	(<i>pygeoapi.api.API</i> method), 81
<code>__init__()</code>	(<i>pygeoapi.api.APIRequest</i> method), 83
<code>__init__()</code>	(<i>pygeoapi.formatter.base.BaseFormatter</i> method), 93
<code>__init__()</code>	(<i>pygeoapi.formatter.csv.CSVFormatter</i> method), 94
<code>__init__()</code>	(<i>pygeoapi.process.base.BaseProcessor</i> method), 95
<code>__init__()</code>	(<i>pygeoapi.process.hello_world.HelloWorldProcessor</i> method), 95
<code>__init__()</code>	(<i>pygeoapi.provider.base.BaseProvider</i> method), 96
<code>__repr__()</code>	(<i>pygeoapi.formatter.base.BaseFormatter</i> method), 93
<code>__repr__()</code>	(<i>pygeoapi.formatter.csv.CSVFormatter</i> method), 94
<code>__repr__()</code>	(<i>pygeoapi.process.base.BaseProcessor</i> method), 95
<code>__repr__()</code>	(<i>pygeoapi.process.hello_world.HelloWorldProcessor</i> method), 95
<code>__weakref__</code>	(<i>pygeoapi.api.API</i> attribute), 81
<code>__weakref__</code>	(<i>pygeoapi.api.APIRequest</i> attribute), 83
<code>__weakref__</code>	(<i>pygeoapi.formatter.base.BaseFormatter</i> attribute), 93
<code>__weakref__</code>	(<i>pygeoapi.formatter.base.FormatterGenericError</i> attribute), 94
<code>__weakref__</code>	(<i>pygeoapi.plugin.InvalidPluginError</i> attribute), 90
<code>__weakref__</code>	(<i>pygeoapi.process.base.BaseProcessor</i> attribute), 95
<code>__weakref__</code>	(<i>pygeoapi.process.base.ProcessorGenericError</i> attribute), 95
<code>__weakref__</code>	(<i>pygeoapi.provider.base.BaseProvider</i> attribute), 96
<code>__weakref__</code>	(<i>pygeoapi.provider.base.ProviderGenericError</i> attribute), 97
<code>_get_format()</code>	(<i>pygeoapi.api.APIRequest</i> method), 83
<code>_get_locale()</code>	(<i>pygeoapi.api.APIRequest</i> method), 83
<code>_get_params()</code>	(<i>pygeoapi.api.APIRequest</i> static method), 83
<code>_ignore_gdal_error()</code>	(in module <i>pygeoapi.provider.ogr</i>), 102
<code>_load()</code>	(<i>pygeoapi.provider.csv.CSVProvider</i> method), 98
<code>_load()</code>	(<i>pygeoapi.provider.geojson.GeoJSONProvider</i> method), 99
<code>_load_source_helper()</code>	(<i>pygeoapi.provider.ogr.OGRProvider</i> method), 101
<code>_make_orderby()</code>	(<i>pygeoapi.provider.postgresql.PostgreSQLProvider</i> method), 103
<code>_response_feature_collection()</code>	(<i>pygeoapi.provider.ogr.OGRProvider</i> method), 101
<code>_response_feature_hits()</code>	(<i>pygeoapi.provider.ogr.OGRProvider</i> method), 101

`_silent_gdal_error()` (in module `pygeoapi.provider.ogr`), 102

A

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

`APIRequest` (class in `pygeoapi.api`), 82

B

`BaseFormatter` (class in `pygeoapi.formatter.base`), 93

`BaseProcessor` (class in `pygeoapi.process.base`), 95

`BaseProvider` (class in `pygeoapi.provider.base`), 96

C

`close()` (`pygeoapi.provider.ogr.CommonSourceHelper` method), 100

`close()` (`pygeoapi.provider.ogr.SourceHelper` method), 102

`collection_coverage()` (in module `pygeoapi.flask_app`), 86

`collection_coverage_domainset()` (in module `pygeoapi.flask_app`), 86

`collection_coverage_rangetype()` (in module `pygeoapi.flask_app`), 86

`collection_items()` (in module `pygeoapi.flask_app`), 86

`collection_queryables()` (in module `pygeoapi.flask_app`), 86

`collections()` (in module `pygeoapi.flask_app`), 86

`CommonSourceHelper` (class in `pygeoapi.provider.ogr`), 100

`conformance()` (in module `pygeoapi.flask_app`), 86

`create()` (`pygeoapi.provider.base.BaseProvider` method), 96

`create()` (`pygeoapi.provider.geojson.GeoJSONProvider` method), 99

`CSVFormatter` (class in `pygeoapi.formatter.csv_`), 94

`CSVProvider` (class in `pygeoapi.provider.csv_`), 98

D

`data()` (`pygeoapi.api.APIRequest` property), 83

`DatabaseConnection` (class in `pygeoapi.provider.postgresql`), 103

`dategetter()` (in module `pygeoapi.util`), 90

`delete()` (`pygeoapi.provider.base.BaseProvider` method), 96

`delete()` (`pygeoapi.provider.geojson.GeoJSONProvider` method), 99

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

`disable_paging()` (`pygeoapi.provider.ogr.CommonSourceHelper` method), 100

`disable_paging()` (`pygeoapi.provider.ogr.ESRIJSONHelper` method), 101

`disable_paging()` (`pygeoapi.provider.ogr.SourceHelper` method), 102

`disable_paging()` (`pygeoapi.provider.ogr.WFSHelper` method), 102

E

`enable_paging()` (`pygeoapi.provider.ogr.CommonSourceHelper` method), 100

`enable_paging()` (`pygeoapi.provider.ogr.ESRIJSONHelper` method), 101

`enable_paging()` (`pygeoapi.provider.ogr.SourceHelper` method), 102

`enable_paging()` (`pygeoapi.provider.ogr.WFSHelper` method), 102

`ESRIJSONHelper` (class in `pygeoapi.provider.ogr`), 100

`execute()` (`pygeoapi.process.base.BaseProcessor` method), 95

`execute()` (`pygeoapi.process.hello_world.HelloWorldProcessor` method), 95

`execute_process_jobs()` (in module `pygeoapi.flask_app`), 87

F

`file_modified_iso8601()` (in module `pygeoapi.util`), 90

`filter_dict_by_key_value()` (in module `pygeoapi.util`), 90

`filter_providers_by_type()` (in module `pygeoapi.util`), 91

`format()` (`pygeoapi.api.APIRequest` property), 83

`format_datetime()` (in module `pygeoapi.util`), 91

`format_duration()` (in module `pygeoapi.util`), 91

`FORMAT_TYPES` (in module `pygeoapi.api`), 85

`FormatterGenericError`, 94

`FormatterSerializationError`, 94

G

`gen_media_type_object()` (in module `pygeoapi.openapi`), 89

`gen_response_object()` (in module `pygeoapi.openapi`), 89

`GeoJSONProvider` (class in `pygeoapi.provider.geojson`), 99

`get()` (`pygeoapi.provider.base.BaseProvider` method), 96

`get()` (`pygeoapi.provider.csv_.CSVProvider` method), 98

[get\(\)](#) (*pygeoapi.provider.geojson.GeoJSONProvider method*), 99
[get\(\)](#) (*pygeoapi.provider.ogr.OGRProvider method*), 101
[get\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 103
[get\(\)](#) (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 105
[get_breadcrumbs\(\)](#) (*in module pygeoapi.util*), 91
[get_collection_edr_query\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_collection_tiles\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_collection_tiles_data\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_collection_tiles_metadata\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_coverage_domainset\(\)](#) (*pygeoapi.provider.base.BaseProvider method*), 96
[get_coverage_rangetype\(\)](#) (*pygeoapi.provider.base.BaseProvider method*), 96
[get_data_path\(\)](#) (*pygeoapi.provider.base.BaseProvider method*), 96
[get_envelope\(\)](#) (*in module pygeoapi.util*), 91
[get_exception\(\)](#) (*pygeoapi.api.API method*), 81
[get_fields\(\)](#) (*pygeoapi.provider.base.BaseProvider method*), 97
[get_fields\(\)](#) (*pygeoapi.provider.csv.CSVProvider method*), 98
[get_fields\(\)](#) (*pygeoapi.provider.geojson.GeoJSONProvider method*), 99
[get_fields\(\)](#) (*pygeoapi.provider.ogr.OGRProvider method*), 101
[get_fields\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 103
[get_fields\(\)](#) (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 105
[get_format_exception\(\)](#) (*pygeoapi.api.API method*), 81
[get_job_result\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_job_result_resource\(\)](#) (*in module pygeoapi.flask_app*), 87
[get_jobs\(\)](#) (*in module pygeoapi.flask_app*), 88
[get_layer\(\)](#) (*pygeoapi.provider.ogr.CommonSourceHelper method*), 100
[get_layer\(\)](#) (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 101
[get_layer\(\)](#) (*pygeoapi.provider.ogr.SourceHelper method*), 102
[get_linkrel\(\)](#) (*pygeoapi.api.APIRequest method*), 83
[get_metadata\(\)](#) (*pygeoapi.provider.base.BaseProvider method*), 97
[get_mimetype\(\)](#) (*in module pygeoapi.util*), 91
[get_next\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 103
[get_oas\(\)](#) (*in module pygeoapi.openapi*), 89
[get_oas_30\(\)](#) (*in module pygeoapi.openapi*), 89
[get_path_basename\(\)](#) (*in module pygeoapi.util*), 91
[get_previous\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 103
[get_processes\(\)](#) (*in module pygeoapi.flask_app*), 88
[get_provider_by_type\(\)](#) (*in module pygeoapi.util*), 91
[get_provider_default\(\)](#) (*in module pygeoapi.util*), 92
[get_request_headers\(\)](#) (*pygeoapi.api.APIRequest method*), 83
[get_response\(\)](#) (*in module pygeoapi.flask_app*), 88
[get_response_headers\(\)](#) (*pygeoapi.api.APIRequest method*), 83
[get_typed_value\(\)](#) (*in module pygeoapi.util*), 92
[gzip\(\)](#) (*in module pygeoapi.api*), 85

H

[HEADERS](#) (*in module pygeoapi.api*), 85
[headers\(\)](#) (*pygeoapi.api.APIRequest property*), 84
[HelloWorldProcessor](#) (*class in pygeoapi.process.hello_world*), 95
[human_size\(\)](#) (*in module pygeoapi.util*), 92

I

[InvalidHelperError](#), 101
[InvalidPluginError](#), 90
[is_url\(\)](#) (*in module pygeoapi.util*), 92
[is_valid\(\)](#) (*pygeoapi.api.APIRequest method*), 84

J

[JobStatus](#) (*class in pygeoapi.util*), 90
[json_serial\(\)](#) (*in module pygeoapi.util*), 92

L

[landing_page\(\)](#) (*in module pygeoapi.flask_app*), 88
[load_plugin\(\)](#) (*in module pygeoapi.plugin*), 90
[locale\(\)](#) (*pygeoapi.api.APIRequest property*), 84

M

[module](#)
[pygeoapi.api](#), 81

- `pygeoapi.flask_app`, 86
- `pygeoapi.formatter`, 93
- `pygeoapi.formatter.base`, 93
- `pygeoapi.formatter.csv_`, 94
- `pygeoapi.log`, 89
- `pygeoapi.openapi`, 89
- `pygeoapi.plugin`, 90
- `pygeoapi.process`, 94
- `pygeoapi.process.base`, 95
- `pygeoapi.process.hello_world`, 95
- `pygeoapi.provider`, 96
- `pygeoapi.provider.base`, 96
- `pygeoapi.provider.csv_`, 98
- `pygeoapi.provider.geojson`, 99
- `pygeoapi.provider.ogr`, 100
- `pygeoapi.provider.postgresql`, 103
- `pygeoapi.provider.sqlite`, 104
- `pygeoapi.util`, 90

O

`OGRProvider` (*class in `pygeoapi.provider.ogr`*), 101

`openapi()` (*in module `pygeoapi.flask_app`*), 88

P

`params()` (*`pygeoapi.api.APIRequest` property*), 84

`path_info()` (*`pygeoapi.api.APIRequest` property*), 85

PLUGINS (*in module `pygeoapi.plugin`*), 90

`PostgreSQLProvider` (*class in `pygeoapi.provider.postgresql`*), 103

`pre_process()` (*in module `pygeoapi.api`*), 85

`PROCESS_METADATA` (*in module `pygeoapi.process.hello_world`*), 96

`ProcessorExecuteError`, 95

`ProcessorGenericError`, 95

`ProviderConnectionError`, 97

`ProviderGenericError`, 97

`ProviderInvalidQueryError`, 97

`ProviderItemNotFoundError`, 97

`ProviderNoDataError`, 97

`ProviderNotFoundError`, 97

`ProviderQueryError`, 97

`ProviderTypeError`, 97

`ProviderVersionError`, 98

`pygeoapi.api`

- `module`, 81

`pygeoapi.flask_app`

- `module`, 86

`pygeoapi.formatter`

- `module`, 93

`pygeoapi.formatter.base`

- `module`, 93

`pygeoapi.formatter.csv_`

- `module`, 94

`pygeoapi.log`

- `module`, 89

`pygeoapi.openapi`

- `module`, 89

`pygeoapi.plugin`

- `module`, 90

`pygeoapi.process`

- `module`, 94

`pygeoapi.process.base`

- `module`, 95

`pygeoapi.process.hello_world`

- `module`, 95

`pygeoapi.provider`

- `module`, 96

`pygeoapi.provider.base`

- `module`, 96

`pygeoapi.provider.csv_`

- `module`, 98

`pygeoapi.provider.geojson`

- `module`, 99

`pygeoapi.provider.ogr`

- `module`, 100

`pygeoapi.provider.postgresql`

- `module`, 103

`pygeoapi.provider.sqlite`

- `module`, 104

`pygeoapi.util`

- `module`, 90

Q

`query()` (*`pygeoapi.provider.base.BaseProvider` method*), 97

`query()` (*`pygeoapi.provider.csv_.CSVProvider` method*), 98

`query()` (*`pygeoapi.provider.geojson.GeoJSONProvider` method*), 100

`query()` (*`pygeoapi.provider.ogr.OGRProvider` method*), 101

`query()` (*`pygeoapi.provider.postgresql.PostgreSQLProvider` method*), 103

`query()` (*`pygeoapi.provider.sqlite.SQLiteGPKGProvider` method*), 105

R

`raw_locale()` (*`pygeoapi.api.APIRequest` property*), 85

`read_data()` (*in module `pygeoapi.util`*), 92

`render_j2_template()` (*in module `pygeoapi.util`*), 92

S

`setup_logger()` (*in module `pygeoapi.log`*), 89

`SourceHelper` (*class in `pygeoapi.provider.ogr`*), 102

`SQLiteGPKGProvider` (*class in `pygeoapi.provider.sqlite`*), 104

`stac_catalog_path()` (in module `pygeoapi.flask_app`), 88
`stac_catalog_root()` (in module `pygeoapi.flask_app`), 88
`str2bool()` (in module `pygeoapi.util`), 92
`SYSTEM_LOCALE` (in module `pygeoapi.api`), 85

T

`to_json()` (in module `pygeoapi.util`), 93

U

`update()` (`pygeoapi.provider.base.BaseProvider` method), 97
`update()` (`pygeoapi.provider.geojson.GeoJSONProvider` method), 100
`url_join()` (in module `pygeoapi.util`), 93

V

`validate_bbox()` (in module `pygeoapi.api`), 85
`validate_datetime()` (in module `pygeoapi.api`), 85
`validate_openapi_document()` (in module `pygeoapi.openapi`), 89
`validate_subset()` (in module `pygeoapi.api`), 86

W

`WFShelper` (class in `pygeoapi.provider.ogr`), 102
`with_data()` (`pygeoapi.api.APIRequest` class method), 85
`write()` (`pygeoapi.formatter.base.BaseFormatter` method), 93
`write()` (`pygeoapi.formatter.csv.CSVFormatter` method), 94

Y

`yaml_load()` (in module `pygeoapi.util`), 93