

---

# **pygeoapi Documentation**

***Release 0.9.dev0***

**pygeoapi team**

**2020-12-06**



# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Features . . . . .	3
1.2	Standards Support . . . . .	3
<b>2</b>	<b>How pygeoapi works</b>	<b>5</b>
<b>3</b>	<b>Install</b>	<b>7</b>
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
<b>4</b>	<b>Configuration</b>	<b>9</b>
4.1	Reference . . . . .	9
4.1.1	server . . . . .	9
4.1.2	logging . . . . .	10
4.1.3	metadata . . . . .	10
4.1.4	resources . . . . .	11
4.2	Using environment variables . . . . .	12
4.3	Linked Data . . . . .	13
4.4	Summary . . . . .	14
<b>5</b>	<b>Administration</b>	<b>15</b>
5.1	Creating the OpenAPI document . . . . .	15
5.2	Verifying configuration files . . . . .	15
5.3	Setting system environment variables . . . . .	16
5.4	Summary . . . . .	16
<b>6</b>	<b>Running</b>	<b>17</b>
6.1	pygeoapi serve . . . . .	17
6.1.1	Flask WSGI . . . . .	17
6.1.2	Starlette ASGI . . . . .	18
6.2	Running in production . . . . .	18
6.2.1	Apache and mod_wsgi . . . . .	19
6.2.2	Gunicorn . . . . .	19
6.2.3	Gunicorn and Flask . . . . .	19
6.2.4	Gunicorn and Starlette . . . . .	20

6.3	Summary . . . . .	20
<b>7</b>	<b>Docker</b>	<b>21</b>
7.1	The basics . . . . .	21
7.2	Overriding the default configuration . . . . .	21
7.3	Deploying on a sub-path . . . . .	22
7.4	Summary . . . . .	22
<b>8</b>	<b>Taking a tour of pygeoapi</b>	<b>23</b>
8.1	Overview . . . . .	23
8.2	Landing page . . . . .	23
8.3	Collections . . . . .	23
8.4	Collection information . . . . .	24
8.5	Vector data . . . . .	24
8.5.1	Collection queryables . . . . .	24
8.5.2	Collection items . . . . .	24
8.5.3	Collection item . . . . .	24
8.6	Raster data . . . . .	24
8.6.1	Collection coverage domainset . . . . .	24
8.6.2	Collection coverage rangetype . . . . .	25
8.6.3	Collection coverage data . . . . .	25
8.7	Tiles . . . . .	25
8.7.1	URI templates . . . . .	25
8.7.2	Generic metadata . . . . .	25
8.8	SpatioTemporal Assets . . . . .	25
8.9	Processes . . . . .	26
8.10	API Documentation . . . . .	26
8.11	Conformance . . . . .	26
<b>9</b>	<b>OpenAPI</b>	<b>27</b>
9.1	Using OpenAPI . . . . .	27
9.2	Summary . . . . .	32
<b>10</b>	<b>Data publishing</b>	<b>33</b>
10.1	Providers overview . . . . .	33
10.1.1	Publishing vector data to OGC API - Features . . . . .	33
10.1.1.1	Providers . . . . .	33
10.1.1.2	Connection examples . . . . .	34
10.1.1.3	Data access examples . . . . .	36
10.1.2	Publishing raster data to OGC API - Coverages . . . . .	36
10.1.2.1	Providers . . . . .	36
10.1.2.2	Connection examples . . . . .	36
10.1.2.3	Data access examples . . . . .	38
10.1.3	Publishing map tiles to OGC API - Tiles . . . . .	38
10.1.3.1	Providers . . . . .	38
10.1.3.2	Connection examples . . . . .	38
10.1.3.3	Data access examples . . . . .	39
10.1.4	Publishing processes via OGC API - Processes . . . . .	39
10.1.4.1	Configuration . . . . .	39
10.1.4.2	Processing examples . . . . .	39
10.1.5	Publishing files to a SpatioTemporal Asset Catalog . . . . .	40
10.1.5.1	Connection examples . . . . .	40
10.1.5.2	Data access examples . . . . .	40
<b>11</b>	<b>Customizing pygeoapi: plugins</b>	<b>41</b>

11.1	Overview . . . . .	41
11.2	Example: custom pygeoapi vector data provider . . . . .	42
11.2.1	Python code . . . . .	42
11.2.2	Connecting to pygeoapi . . . . .	43
11.3	Example: custom pygeoapi raster data provider . . . . .	43
11.3.1	Python code . . . . .	43
11.4	Example: custom pygeoapi formatter . . . . .	44
11.4.1	Python code . . . . .	44
11.5	Processing plugins . . . . .	45
<b>12</b>	<b>HTML Templating</b>	<b>47</b>
<b>13</b>	<b>Development</b>	<b>49</b>
13.1	Codebase . . . . .	49
13.2	Testing . . . . .	49
13.3	Working with Spatialite on OSX . . . . .	49
13.3.1	Using pyenv . . . . .	49
<b>14</b>	<b>OGC Compliance</b>	<b>51</b>
14.1	CITE instance . . . . .	51
14.2	Setting up your own CITE testing instance . . . . .	51
<b>15</b>	<b>Contributing</b>	<b>53</b>
<b>16</b>	<b>Support</b>	<b>55</b>
16.1	Community . . . . .	55
<b>17</b>	<b>Further Reading</b>	<b>57</b>
<b>18</b>	<b>License</b>	<b>59</b>
18.1	Code . . . . .	59
18.2	Documentation . . . . .	59
<b>19</b>	<b>API documentation</b>	<b>61</b>
19.1	API . . . . .	61
19.2	flask_app . . . . .	62
19.3	Logging . . . . .	64
19.4	OpenAPI . . . . .	64
19.5	Plugins . . . . .	65
19.6	Utils . . . . .	65
19.7	Formatter package . . . . .	67
19.7.1	Base class . . . . .	68
19.7.2	csv . . . . .	68
19.8	Process package . . . . .	69
19.8.1	Base class . . . . .	69
19.8.2	hello_world . . . . .	69
19.9	Provider . . . . .	70
19.9.1	Base class . . . . .	70
19.9.2	CSV provider . . . . .	72
19.9.3	Elasticsearch provider . . . . .	73
19.9.4	GeoJSON . . . . .	74
19.9.5	OGR . . . . .	75
19.9.6	postgresql . . . . .	77
19.9.7	sqlite/geopackage . . . . .	79

<b>20 Indices and tables</b>	<b>81</b>
<b>Python Module Index</b>	<b>83</b>
<b>Index</b>	<b>85</b>



**Author** the pygeoapi team

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

**Release** 0.9.dev0

**Date** 2020-12-06





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

### 1.2 Standards Support

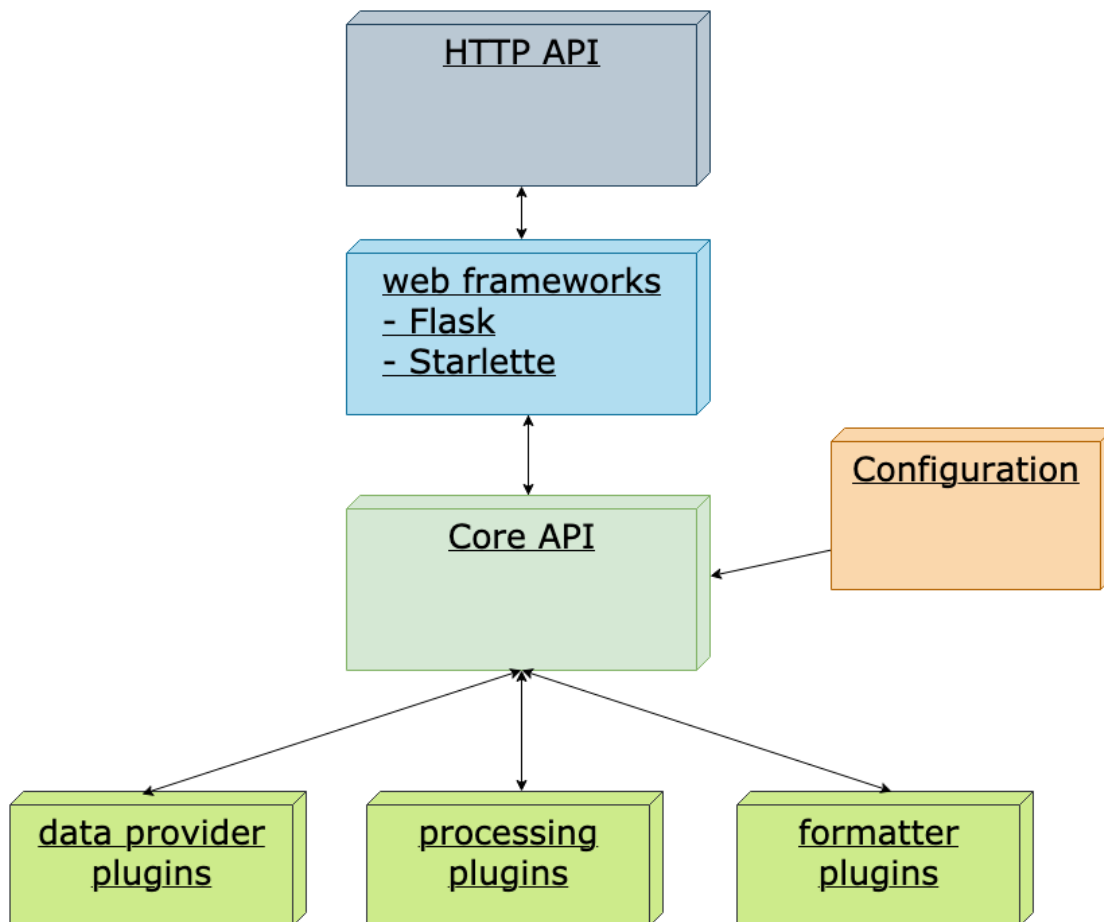
Standards are at the core of `pygeoapi`. Below is the project's standards support matrix.

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

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

## HOW PYGEOAPI WORKS

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



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

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

**See also:**

[Configuration](#) for more details on pygeoapi settings

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

---

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

---

## INSTALL

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

### 3.1 Requirements and dependencies

pygeoapi runs on Python 3.

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

### 3.2 For developers and the truly impatient

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

### 3.3 pip

PyPI package info

```
pip install pygeoapi
```

## 3.4 Docker

Docker image

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

## 3.5 Conda

Conda package info

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

## 3.6 UbuntuGIS

UbuntuGIS package (stable)

UbuntuGIS package (unstable)

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

## 3.7 FreeBSD

FreeBSD port

```
pkg install py-pygeoapi
```

## 3.8 Summary

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

## 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
  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
  template: # optional configuration to specify a different set of templates for HTML
    ↪ pages. Recommend using absolute paths. Omit this to use the default provided
    ↪ templates
```

(continues on next page)

(continued from previous page)

```

    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

```

## 4.1.2 logging

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

```

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

```

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

## 4.1.3 metadata

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

```

metadata:
    identification:
        title: pygeoapi default instance # the title of the service
        description: pygeoapi provides an API to geospatial data # some descriptive
    ↪ text about the service
        keywords: # list of keywords about the service
            - geospatial
            - data
            - api
        keywords_type: theme # keyword type as per the ISO 19115 MD_KeywordTypeCode
    ↪ codelist. Accepted values are discipline, temporal, place, theme, stratum
        terms_of_service: https://creativecommons.org/licenses/by/4.0/ # terms of
    ↪ service
        url: http://example.org # informative URL about the service
        license: # licensing details
            name: CC-BY 4.0 license
            url: https://creativecommons.org/licenses/by/4.0/
        provider: # service provider details
            name: Organization Name
            url: https://pygeoapi.io
        contact: # service contact details
            name: Lastname, Firstname
            position: Position Title
            address: Mailing Address

```

(continues on next page)



(continued from previous page)

```

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#
        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

```

(continues on next page)

(continued from previous page)

```

    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)
        default: true # optional: if not specified, the first provider
        ↪definition is considered the default
        name: CSV
        data: tests/data/obs.csv # required: the data filesystem path or URL,
        ↪depending on plugin setup
        id_field: id # required for vector data, the field corresponding to
        ↪the ID
        time_field: datetimestamp # optional field corresponding to the
        ↪temporal property of the dataset
        format: # optional default format
            name: GeoJSON # required: format name
            mimetype: application/json # required: format mimetype
        options: # optional options to pass to provider (i.e. GDAL creation)
            option_name: option_value
        properties: # optional: only return the following properties, in order
            - stn_id
            - value

    hello-world: # name of process
        type: collection # REQUIRED (collection, process, or stac-collection)
        processor:
            name: HelloWorld # Python path of process definition

```

**See also:**

*Linked Data* for optionally configuring linked data datasets

**See also:**

*Customizing pygeoapi: plugins* for more information on plugins

## 4.2 Using environment variables

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

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

```

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

```

## 4.3 Linked Data



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

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

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

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

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

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

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

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

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

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

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

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

## 4.4 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
- 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 generate-openapi-document -c /path/to/my-pygeoapi-config.yml
```

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

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

---

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

---

**See also:**

[OpenAPI](#) for more information on pygeoapi's OpenAPI support

### 5.2 Verifying configuration files

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

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

In Windows:

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

## 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
```

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](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](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 a list of queryable properties and their 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 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.

## 8.9 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 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*

## 8.11 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:

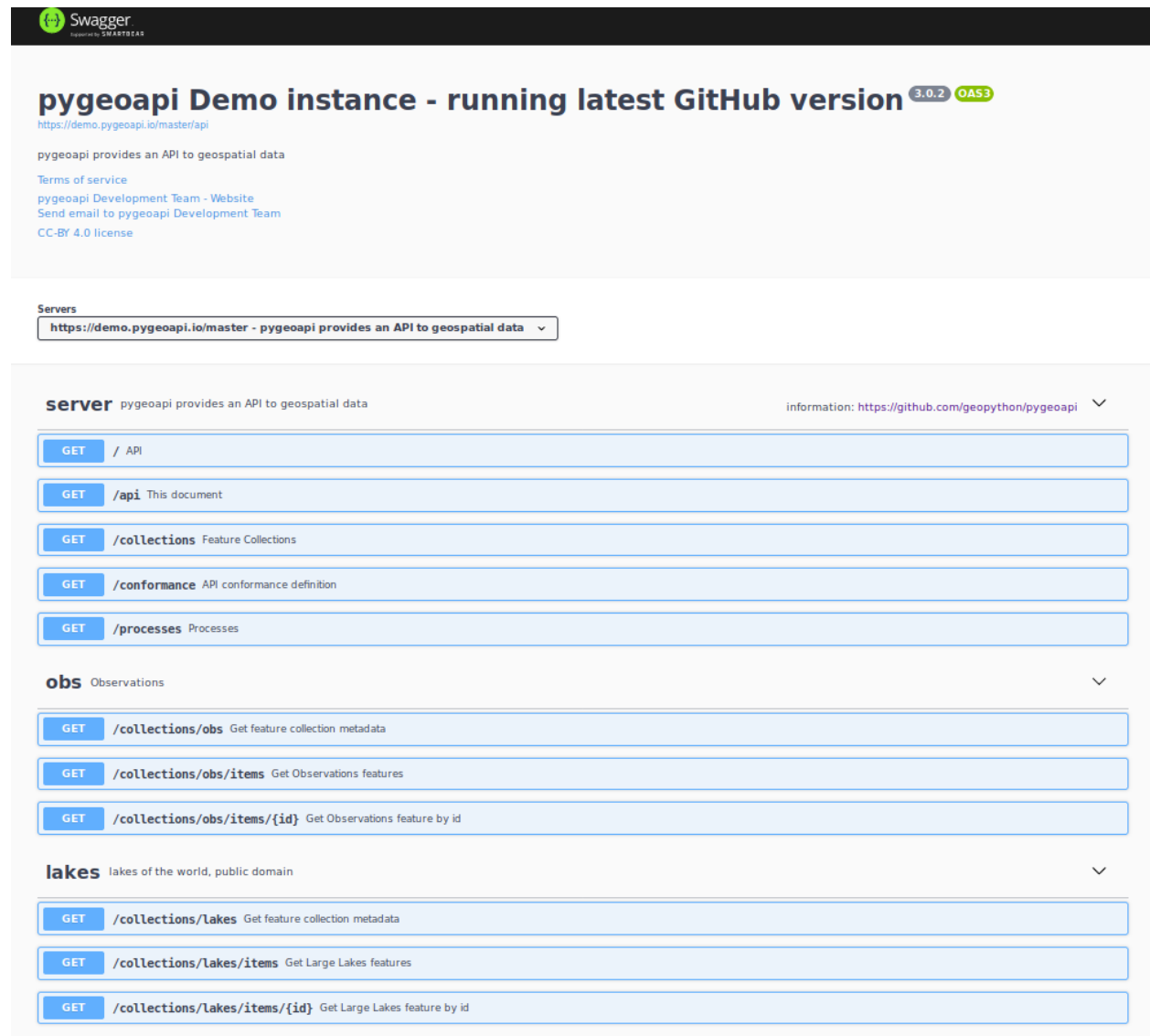
The API is accessible at the `/openapi` endpoint, providing a Swagger-based webpage of the API description..

**See also:**

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

### **9.1 Using OpenAPI**

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:

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

The "obs" section, titled "Observations", lists three endpoints:

- GET /collections/obs** Get feature collection metadata
- GET /collections/obs/items** Get Observations features
- GET /collections/obs/items/{id}** Get Observations feature by id

The "lakes" section, titled "lakes of the world, public domain", lists three endpoints:

- 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

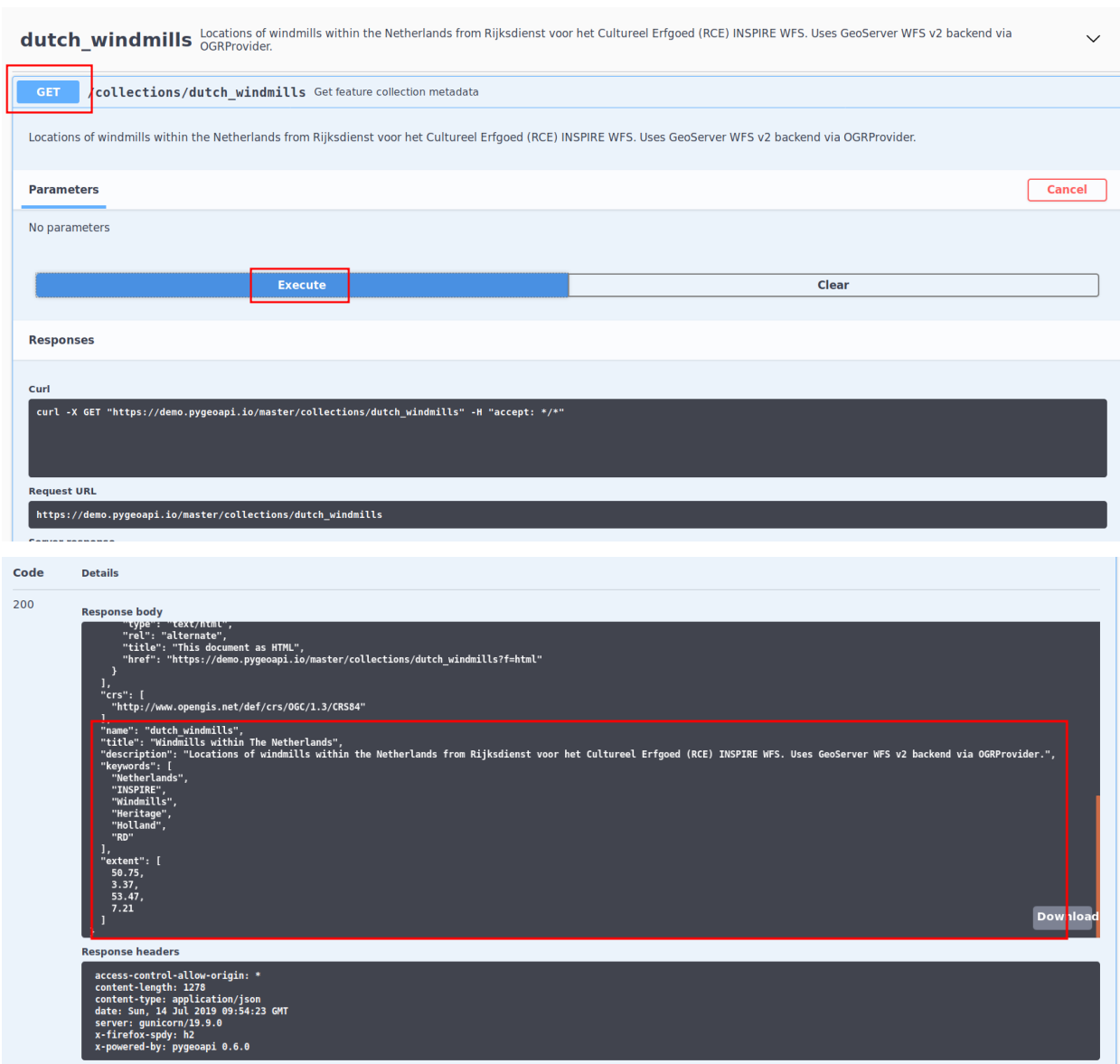
200

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	<p><b>Response body</b></p> <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",       "RHONUMMER": "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",       "FOTOGRAAF": "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",       "RHONUMMER": "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",       "FOTOGRAAF": "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 shows the OpenAPI UI for the endpoint `GET /collections/dutch_windmills/items/{id}`. The description states: "Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider." The parameters section includes a required path parameter `id` (string) with the value `Molens.1` entered, and an optional query parameter `f` (string) with the value `json` selected from a dropdown menu. A red box highlights the endpoint URL and the `id` parameter input field. A blue `Execute` button is at the bottom.

Name	Description
<code>id</code> * required string (path)	The id of a feature
<code>f</code> string (query)	The optional f parameter indicates the output format which the server shall provide as part of the response document. The default format is GeoJSON.

## 9.2 Summary

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

## DATA PUBLISHING

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

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

Below are specific connection examples based on supported providers.

### 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
```



## OGR

---

**Todo:** add overview and requirements

---

## MongoDB

---

**Todo:** add overview and requirements

---

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

## PostgreSQL

---

**Todo:** add overview and requirements

---

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

## SQLiteGPKG

---

**Todo:** add overview and requirements

---

SQLite file:

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

GeoPackage file:

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

### 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>
- fetch a specific feature - <http://localhost:5000/collections/foo/items/123>

## 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	rangeSubset	subset
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 automagically
  x_field: lat
  x_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 rangeSubset - <http://localhost:5000/collections/foo/coverage?rangeSubset=1,3>
- coverage access with subsetting - [http://localhost:5000/collections/foo/coverage?subset=lat\(10,20\)&subset=long\(10,20\)](http://localhost:5000/collections/foo/coverage?subset=lat(10,20)&subset=long(10,20))

### 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:
```

(continues on next page)

(continued from previous page)

```

    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}/{tileCol}/{tileRow}?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 Processing examples

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

---

**Todo:** add more examples once OAProc implementation is complete

---

### 10.1.5 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:

#### 10.1.5.1 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.

---

#### 10.1.5.2 Data access examples

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

From here, browse the filesystem accordingly.

## CUSTOMIZING PYGEOAPI: PLUGINS

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

### 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"""

        BaseProvider.__init__(self, provider_def)

    def get_fields(self):

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

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

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

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

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



## 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
```

BEGIN

## 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"""

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

(continues on next page)

(continued from previous page)

```

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

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

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

```

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

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

END

## 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"""

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

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

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

        out_data = {'rows': []}

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

        return out_data

```

## 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.



## 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.template` 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:
  template:
    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.

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 -->

```



## DEVELOPMENT

### 13.1 Codebase

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

### 13.2 Testing

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

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

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

### 13.3 Working with Spatialite on OSX

#### 13.3.1 Using pyenv

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

Make Homebrew and pyenv play nicely together:

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

Install python with the option to enable SQLite extensions:

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

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

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

Install Spatialite from Homebrew:

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

Set the variable for the Spatialite library under OSX:

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



## OGC COMPLIANCE

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

### 14.1 CITE instance

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

### 14.2 Setting up your own CITE testing instance

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



## CONTRIBUTING

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



## 16.1 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](#)





## 18.1 Code

# The MIT License (MIT)

Copyright © 2018-2020 Tom Kralidis

• - \*

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

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

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

## 18.2 Documentation

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



## API DOCUMENTATION

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

### 19.1 API

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

**class** `pygeoapi.api.API` (*config*)

API object

`__init__` (*config*)

constructor

**Parameters** *config* – configuration dict

**Returns** *pygeoapi.API* instance

`__weakref__`

list of weak references to the object (if defined)

**execute\_process** (*headers, args, data, process*)

Execute process

**Parameters**

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

**Returns** tuple of headers, status code, content

**get\_collection\_items** (*headers, args, dataset, pathinfo=None*)

Queries collection

**Parameters**

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

**Returns** tuple of headers, status code, content

`pygeoapi.api.FORMATS = ['json', 'html', 'jsonld']`  
Formats allowed for ?f= requests

`pygeoapi.api.HEADERS = {'Content-Type': 'application/json', 'X-Powered-By': 'pygeoapi 0.9'}`  
Return headers for requests (e.g:X-Powered-By)

`pygeoapi.api.check_format(args, headers)`  
check format requested from arguments or headers

**Parameters**

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

**Returns** format value

`pygeoapi.api.pre_process(func)`  
Decorator performing header copy and format checking before sending arguments to methods

**Parameters** **func** – decorated function

**Returns** *func*

## 19.2 flask\_app

Flask module providing the route paths to the api

`pygeoapi.flask_app.collection_coverage(collection_id)`  
OGC API - Coverages coverage endpoint

**Parameters** **collection\_id** – collection identifier

**Returns** HTTP response

`pygeoapi.flask_app.collection_coverage_domainset(collection_id)`  
OGC API - Coverages coverage domainset endpoint

**Parameters** **collection\_id** – collection identifier

**Returns** HTTP response

`pygeoapi.flask_app.collection_coverage_rangetype(collection_id)`  
OGC API - Coverages coverage rangetype endpoint

**Parameters** **collection\_id** – collection identifier

**Returns** HTTP response

`pygeoapi.flask_app.collection_items(collection_id, item_id=None)`  
OGC API collections items endpoint

**Parameters**

- **collection\_id** – collection identifier
- **item\_id** – item identifier

**Returns** HTTP response

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

**Parameters** **collection\_id** – collection identifier

**Returns** HTTP response

`pygeoapi.flask_app.collections` (*collection\_id=None*)  
OGC API collections endpoint

**Parameters** `collection_id` – collection identifier

**Returns** HTTP response

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

**Returns** HTTP response

`pygeoapi.flask_app.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*, *tileMatrixSetId=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*, *tileMatrixSetId=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.landing_page` ()  
OGC API landing page endpoint

**Returns** HTTP response

`pygeoapi.flask_app.openapi` ()  
OpenAPI endpoint

**Returns** HTTP response

`pygeoapi.flask_app.process_jobs` (*process\_id=None*)  
OGC API - Processes jobs endpoint

**Parameters** `process_id` – process identifier

**Returns** HTTP response

`pygeoapi.flask_app.processes` (*process\_id=None*)  
OGC API - Processes description endpoint

**Parameters** `process_id` – process identifier

**Returns** HTTP response

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

**Parameters** `path` – path

**Returns** HTTP response

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

**Returns** HTTP response

## 19.3 Logging

Logging system

`pygeoapi.log.setup_logger` (*logging\_config*)  
Setup configuration

**Parameters** `logging_config` – logging specific configuration

**Returns** void (creates logging instance)

## 19.4 OpenAPI

`pygeoapi.openapi.gen_media_type_object` (*media\_type, api\_type, path*)  
Generates an OpenAPI Media Type Object

**Parameters**

- **media\_type** – MIME type
- **api\_type** – OGC API type
- **path** – local path of OGC API parameter or schema definition

**Returns** *dict* of media type object

`pygeoapi.openapi.gen_response_object` (*description, media\_type, api\_type, path*)  
Generates an OpenAPI Response Object

**Parameters**

- **description** – text description of response
- **media\_type** – MIME type
- **api\_type** – OGC API type

**Returns** *dict* of response object

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

**Parameters**

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

**Returns** OpenAPI definition YAML dict

`pygeoapi.openapi.get_oas_30(cfg)`

Generates an OpenAPI 3.0 Document

**Parameters** **cfg** – configuration object

**Returns** OpenAPI definition YAML dict

## 19.5 Plugins

**See also:**

*Customizing pygeoapi: plugins*

Plugin loader

**exception** `pygeoapi.plugin.InvalidPluginError`

Bases: `Exception`

Invalid plugin

`__weakref__`

list of weak references to the object (if defined)

`pygeoapi.plugin.PLUGINS = {'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

## 19.6 Utils

Generic util functions used in the code

`pygeoapi.util.dategetter(date_property, collection)`

Attempts to obtains a date value from a collection.

**Parameters**

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

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

`pygeoapi.util.filter_dict_by_key_value(dict_, key, value)`

helper function to filter a dict by a dict key

**Parameters**

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

**Returns** filtered dict

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

**Parameters** `filename` – filename (with extension)

**Returns** MIME type of given filename

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

**Parameters** `urlpath` – URL path

**Returns** string of basename of URL path

`pygeoapi.util.get_provider_by_type(providers, provider_type)`  
helper function to load a provider by a provider type

**Parameters**

- **providers** – list of providers
- **provider\_type** – type of provider (feature)

**Returns** provider based on type

`pygeoapi.util.get_provider_default(providers)`  
helper function to get a resource's default provider

**Parameters** `providers` – list of providers

**Returns** filtered dict

`pygeoapi.util.get_typed_value(value)`  
Derive true type from data value

**Parameters** `value` – value

**Returns** value as a native Python data type

`pygeoapi.util.is_url(urlstring)`  
Validation function that determines whether a candidate URL should be considered a URI. No remote resource



is obtained; this does not check the existence of any remote resource. :param urlstring: *str* to be evaluated as candidate URL. :returns: *bool* of whether the URL looks like a URL.

`pygeoapi.util.json_serial(obj)`

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

`pygeoapi.util.read_data(path)`

helper function to read data (file or network)

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

render Jinja2 template

#### Parameters

- **config** – dict of configuration
- **template** – template (relative path)
- **data** – dict of data

**Returns** string of rendered template

`pygeoapi.util.str2bool(value)`

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

**Parameters** **value** – value to be evaluated

**Returns** *bool* of whether the value is boolean-ish

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

Serialize dict to json

#### Parameters

- **dict** – *dict* of JSON representation
- **pretty** – *bool* of whether to prettify JSON (default is *False*)

**Returns** JSON string representation

`pygeoapi.util.yaml_load(fh)`

serializes a YAML files into a pyyaml object

**Parameters** **fh** – file handle

**Returns** *dict* representation of YAML

## 19.7 Formatter package

Output formatter package

### 19.7.1 Base class

**class** `pygeoapi.formatter.base.BaseFormatter` (*formatter\_def*)  
Bases: `object`  
generic Formatter ABC

**\_\_init\_\_** (*formatter\_def*)  
Initialize object

**Parameters** **formatter\_def** – formatter definition

**Returns** `pygeoapi.providers.base.BaseFormatter`

**\_\_repr\_\_** ()  
Return repr(self).

**\_\_weakref\_\_**  
list of weak references to the object (if defined)

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

**Parameters**

- **options** – CSV formatting options
- **data** – dict representation of GeoJSON object

**Returns** string representation of format

### 19.7.2 csv

**class** `pygeoapi.formatter.csv_.CSVFormatter` (*formatter\_def*)  
Bases: `pygeoapi.formatter.base.BaseFormatter`  
CSV formatter

**\_\_init\_\_** (*formatter\_def*)  
Initialize object

**Parameters** **formatter\_def** – formatter definition

**Returns** `pygeoapi.formatter.csv_.CSVFormatter`

**\_\_repr\_\_** ()  
Return repr(self).

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

**Parameters**

- **options** – CSV formatting options
- **data** – dict of GeoJSON data

**Returns** string representation of format

## 19.8 Process package

OGC process package, each process is an independent module

### 19.8.1 Base class

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

    generic Processor ABC. Processes are inherited from this class

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

    __repr__ ()
        Return repr(self).

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

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

exception pygeoapi.process.base.ProcessorExecuteError
    Bases: Exception

    query / backend error

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

### 19.8.2 hello\_world

Hello world example process

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

    Hello World Processor example

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

    __repr__ ()
        Return repr(self).

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

pygeoapi.process.hello_world.PROCESS_METADATA = {'description': 'Hello World process', 'ex
```

Process metadata and description

## 19.9 Provider

Provider module containing the plugins wrapping data sources

### 19.9.1 Base class

```
class pygeoapi.provider.base.BaseProvider(provider_def)
    Bases: object
    generic Provider ABC

    __init__(provider_def)
        Initialize object

        Parameters provider_def – provider definition

        Returns pygeoapi.providers.base.BaseProvider

    __repr__()
        Return repr(self).

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

    create(new_feature)
        Create a new feature

    delete(identifier)
        Deletes an existing feature

        Parameters identifier – feature id

    get(identifier)
        query the provider by id

        Parameters identifier – feature id

        Returns dict of single GeoJSON feature

    get_coverage_domainset()
        Provide coverage domainset

        Returns CIS JSON object of domainset metadata

    get_coverage_rangetype()
        Provide coverage rangetype

        Returns CIS JSON object of rangetype metadata

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

        Parameters

        • baseurl – base URL of endpoint

        • urlpath – base path of URL

        • dirpath – directory basepath (equivalent of URL)

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

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

**Returns** dict of fields

**get\_metadata()**

Provide data/file metadata

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

**query()**

query the provider

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

**update(identifier, new\_feature)**

Updates an existing feature id with new\_feature

**Parameters**

- **identifier** – feature id
- **new\_feature** – new GeoJSON feature dictionary

**exception** pygeoapi.provider.base.ProviderConnectionError

Bases: [pygeoapi.provider.base.ProviderGenericError](#)

provider connection error

**exception** pygeoapi.provider.base.ProviderGenericError

Bases: [Exception](#)

provider generic error

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**exception** pygeoapi.provider.base.ProviderInvalidQueryError

Bases: [pygeoapi.provider.base.ProviderGenericError](#)

provider invalid query error

**exception** pygeoapi.provider.base.ProviderItemNotFoundError

Bases: [pygeoapi.provider.base.ProviderGenericError](#)

provider item not found query error

**exception** pygeoapi.provider.base.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

## 19.9.2 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=[]*)  
Load CSV data

### Parameters

- **startindex** – starting record to return (default 0)
- **limit** – number of records to return (default 10)
- **resulttype** – return results or hit limit (default results)
- **properties** – list of tuples (name, value)

**Returns** dict of GeoJSON FeatureCollection

**get** (*identifier*)  
query CSV id

**Parameters** **identifier** – feature id

**Returns** dict of single GeoJSON feature

**get\_fields** ()

Get provider field information (names, types)

**Returns** dict of fields

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

**Returns** dict of GeoJSON FeatureCollection

### 19.9.3 Elasticsearch provider

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

    Elasticsearch Provider

    esdoc2geojson (doc)
        generate GeoJSON dict from ES document

        Parameters doc – dict of ES document

        Returns GeoJSON dict

    get (identifier)
        Get ES document by id

        Parameters identifier – feature id

        Returns dict of single GeoJSON feature

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

        Returns dict of fields

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

        Parameters property_name – property name

        Returns masked property name

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

        Parameters

        • startindex – starting record to return (default 0)

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

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

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

        • datetime – temporal (datestamp or extent)

        • properties – list of tuples (name, value)

        • sortby – list of dicts (property, order)

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

## 19.9.4 GeoJSON

**class** pygeoapi.provider.geojson.**GeoJSONProvider** (*provider\_def*)

Bases: *pygeoapi.provider.base.BaseProvider*

Provider class backed by local GeoJSON files

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

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

Not thread safe, a single server process is assumed

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

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

**\_load()**

Load and validate the source GeoJSON file at self.data

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

**create** (*new\_feature*)

Create a new feature

**Parameters** **new\_feature** – new GeoJSON feature dictionary

**delete** (*identifier*)

Deletes an existing feature

**Parameters** **identifier** – feature id

**get** (*identifier*)

query the provider by id

**Parameters** **identifier** – feature id

**Returns** dict of single GeoJSON feature

**get\_fields()**

Get provider field information (names, types)

**Returns** dict of fields

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

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)

**Returns** FeatureCollection dict of 0..n GeoJSON features

**update** (*identifier, new\_feature*)

Updates an existing feature id with new\_feature

**Parameters**

- **identifier** – feature id
- **new\_feature** – new GeoJSON feature dictionary

## 19.9.5 OGR

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

Bases: [pygeoapi.provider.ogr.SourceHelper](#)

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

**close** ()

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

**disable\_paging** ()

Disable paged access to dataset (OGR Driver-specific)

**enable\_paging** (*startindex=- 1, limit=- 1*)

Enable paged access to dataset (OGR Driver-specific) using OGR SQL [https://gdal.org/user/ogr\\_sql\\_dialect.html](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](https://gdal.org/ogr_formats.html) (per-driver specifics).

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

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

**`_load_source_helper`** (*source\_type*)

Loads Source Helper by name.

**Parameters** **type** (*Source*) – Source type name

**Returns** Source Helper object

**`_response_feature_collection`** (*layer*, *limit*)

Assembles output from Layer query as GeoJSON FeatureCollection structure.

**Returns** GeoJSON FeatureCollection

**`_response_feature_hits`** (*layer*)

Assembles GeoJSON hits from OGR Feature count e.g: [http://localhost:5000/collections/hotosm\\_bdi\\_waterways/items?resulttype=hits](http://localhost:5000/collections/hotosm_bdi_waterways/items?resulttype=hits)

**Returns** GeoJSON FeaturesCollection

**`get`** (*identifier*)

Get Feature by id

**Parameters** **identifier** – feature id

**Returns** feature collection

**`get_fields`** ()

Get provider field information (names, types)

**Returns** dict of fields

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

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 (timestamp or extent)
- **properties** – list of tuples (name, value)
- **sortby** – list of dicts (property, order)

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

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

Bases: `object`

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

**`close`** ()

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

**disable\_paging()**  
Disable paged access to dataset (OGR Driver-specific)

**enable\_paging**(*startindex=- 1, limit=- 1*)  
Enable paged access to dataset (OGR Driver-specific)

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

**class** pygeoapi.provider.ogr.**WFSHelper**(*provider*)  
Bases: [pygeoapi.provider.ogr.SourceHelper](#)

**disable\_paging()**  
Disable paged access to dataset (OGR Driver-specific)

**enable\_paging**(*startindex=- 1, limit=- 1*)  
Enable paged access to dataset (OGR Driver-specific)

pygeoapi.provider.ogr.**\_\_ignore\_gdal\_error**(*inst, fn, \*args, \*\*kwargs*) → Any  
Evaluate the function with the object instance.

#### Parameters

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

**Returns** Any function evaluation result

pygeoapi.provider.ogr.**\_\_silent\_gdal\_error**(*f*)  
Decorator function for gdal

## 19.9.6 postgresql

**class** pygeoapi.provider.postgresql.**DatabaseConnection**(*conn\_dic, table, con-  
text='query'*)  
Bases: [object](#)

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

**class** pygeoapi.provider.postgresql.**PostgreSQLProvider**(*provider\_def*)  
Bases: [pygeoapi.provider.base.BaseProvider](#)

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

**\_\_PostgreSQLProvider\_\_get\_where\_clauses**(*properties=[], bbox=[]*)  
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](http://localhost:5000/collections/hotosm_bdi_waterways/items?resulttype=hits)

**Returns** GeoJSON FeaturesCollection

**`get`** (*identifier*)  
Query the provider for a specific feature id e.g: [/collections/hotosm\\_bdi\\_waterways/items/13990765](http://localhost:5000/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=[]*)  
Query Postgis for all the content. e.g: [http://localhost:5000/collections/hotosm\\_bdi\\_waterways/items?limit=1&resulttype=results](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)

**Returns** GeoJSON FeaturesCollection

## 19.9.7 sqlite/geopackage

**class** pygeoapi.provider.sqlite.SQLiteGPKGProvider(provider\_def)

Bases: *pygeoapi.provider.base.BaseProvider*

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

**\_\_SQLiteGPKGProvider\_\_get\_where\_clauses**(properties=[], bbox=[])

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)

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

**Parameters** identifier – feature id

**Returns** GeoJSON FeaturesCollection

**get\_fields**()

Get fields from sqlite table (columns are field)

**Returns** dict of fields

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

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

**Parameters**

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

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

**Returns** GeoJSON FeaturesCollection

## INDICES AND TABLES

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





## PYTHON MODULE INDEX

### p

- `pygeoapi.api`, 61
- `pygeoapi.flask_app`, 62
- `pygeoapi.formatter`, 67
  - `pygeoapi.formatter.base`, 68
  - `pygeoapi.formatter.csv_`, 68
- `pygeoapi.log`, 64
- `pygeoapi.openapi`, 64
- `pygeoapi.plugin`, 65
- `pygeoapi.process`, 69
  - `pygeoapi.process.base`, 69
  - `pygeoapi.process.hello_world`, 69
- `pygeoapi.provider`, 70
  - `pygeoapi.provider.base`, 70
  - `pygeoapi.provider.csv_`, 72
  - `pygeoapi.provider.elasticsearch_`, 73
  - `pygeoapi.provider.geojson`, 74
  - `pygeoapi.provider.ogr`, 75
  - `pygeoapi.provider.postgresql`, 77
  - `pygeoapi.provider.sqlite`, 79
- `pygeoapi.util`, 65



## Symbols

`_PostgreSQLProvider__get_where_clauses()` `__weakref__` (`pygeoapi.api.API` attribute), 61  
 (`pygeoapi.provider.postgresql.PostgreSQLProvider` `__weakref__` (`pygeoapi.formatter.base.BaseFormatter` attribute), 68  
 method), 77  
`_PostgreSQLProvider__response_feature()` `__weakref__` (`pygeoapi.plugin.InvalidPluginError` attribute), 65  
 (`pygeoapi.provider.postgresql.PostgreSQLProvider` `__weakref__` (`pygeoapi.process.base.BaseProcessor` attribute), 69  
 method), 77  
`_PostgreSQLProvider__response_feature_hits()` `__weakref__` (`pygeoapi.process.base.ProcessorExecuteError` attribute), 69  
 (`pygeoapi.provider.postgresql.PostgreSQLProvider` `__weakref__` (`pygeoapi.provider.base.BaseProvider` attribute), 70  
 method), 79  
`_SQLiteGPKGProvider__get_where_clauses()` `__weakref__` (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` `__weakref__` (`pygeoapi.provider.base.ProviderGenericError` attribute), 71  
 method), 79  
`_SQLiteGPKGProvider__load()` (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` `__ignore_gdal_error()` (in module `pygeoapi.provider.ogr`), 77  
 method), 79  
`_SQLiteGPKGProvider__response_feature()` `__load()` (`pygeoapi.provider.csv.CSVProvider` method), 72  
 (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` `__load()` (`pygeoapi.provider.geojson.GeoJSONProvider` method), 74  
 method), 79  
`_SQLiteGPKGProvider__response_feature_hits()` `__load_source_helper()` (`pygeoapi.provider.ogr.OGRProvider` method), 76  
 (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` `__response_feature_collection()` (`pygeoapi.provider.ogr.OGRProvider` method), 76  
 method), 79  
`__init__()` (`pygeoapi.api.API` method), 61  
`__init__()` (`pygeoapi.formatter.base.BaseFormatter` method), 68  
`__init__()` (`pygeoapi.formatter.csv.CSVFormatter` method), 68  
`__init__()` (`pygeoapi.process.base.BaseProcessor` method), 69  
`__init__()` (`pygeoapi.process.hello_world.HelloWorldProcessor` `__silent_gdal_error()` (in module `pygeoapi.provider.ogr`), 77  
 method), 69  
`__init__()` (`pygeoapi.provider.base.BaseProvider` method), 70  
`__repr__()` (`pygeoapi.formatter.base.BaseFormatter` method), 68  
`__repr__()` (`pygeoapi.formatter.csv.CSVFormatter` method), 68  
`__repr__()` (`pygeoapi.process.base.BaseProcessor` method), 69  
`__repr__()` (`pygeoapi.process.hello_world.HelloWorldProcessor` method), 69  
`__repr__()` (`pygeoapi.provider.base.BaseProvider` method), 70

## A

API (class in `pygeoapi.api`), 61

## B

`BaseFormatter` (class in `pygeoapi.formatter.base`), 68

`BaseProcessor` (class in `pygeoapi.process.base`), 69

`BaseProvider` (class in `pygeoapi.provider.base`), 70

## C

`check_format()` (in module `pygeoapi.api`), 62

close() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 75

close() (*pygeoapi.provider.ogr.SourceHelper method*), 76

collection\_coverage() (*in module pygeoapi.flask\_app*), 62

collection\_coverage\_domainset() (*in module pygeoapi.flask\_app*), 62

collection\_coverage\_rangetype() (*in module pygeoapi.flask\_app*), 62

collection\_items() (*in module pygeoapi.flask\_app*), 62

collection\_queryables() (*in module pygeoapi.flask\_app*), 62

collections() (*in module pygeoapi.flask\_app*), 63

CommonSourceHelper (*class in pygeoapi.provider.ogr*), 75

conformance() (*in module pygeoapi.flask\_app*), 63

create() (*pygeoapi.provider.base.BaseProvider method*), 70

create() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 74

CSVFormatter (*class in pygeoapi.formatter.csv\_*), 68

CSVProvider (*class in pygeoapi.provider.csv\_*), 72

## D

DatabaseConnection (*class in pygeoapi.provider.postgresql*), 77

dategetter() (*in module pygeoapi.util*), 65

delete() (*pygeoapi.provider.base.BaseProvider method*), 70

delete() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 74

disable\_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 75

disable\_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 75

disable\_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 76

disable\_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 77

## E

ElasticsearchProvider (*class in pygeoapi.provider.elasticsearch\_*), 73

enable\_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 75

enable\_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*),

75

enable\_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 77

enable\_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 77

esdoc2geojson() (*pygeoapi.provider.elasticsearch\_.ElasticsearchProvider method*), 73

ESRIJSONHelper (*class in pygeoapi.provider.ogr*), 75

execute() (*pygeoapi.process.base.BaseProcessor method*), 69

execute() (*pygeoapi.process.hello\_world.HelloWorldProcessor method*), 69

execute\_process() (*pygeoapi.api.API method*), 61

## F

filter\_dict\_by\_key\_value() (*in module pygeoapi.util*), 65

filter\_providers\_by\_type() (*in module pygeoapi.util*), 66

FORMATS (*in module pygeoapi.api*), 61

## G

gen\_media\_type\_object() (*in module pygeoapi.openapi*), 64

gen\_response\_object() (*in module pygeoapi.openapi*), 64

GeoJSONProvider (*class in pygeoapi.provider.geojson*), 74

get() (*pygeoapi.provider.base.BaseProvider method*), 70

get() (*pygeoapi.provider.csv\_.CSVProvider method*), 72

get() (*pygeoapi.provider.elasticsearch\_.ElasticsearchProvider method*), 73

get() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 74

get() (*pygeoapi.provider.ogr.OGRProvider method*), 76

get() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 78

get() (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 79

get\_breadcrumbs() (*in module pygeoapi.util*), 66

get\_collection\_items() (*pygeoapi.api.API method*), 61

get\_collection\_tiles() (*in module pygeoapi.flask\_app*), 63

get\_collection\_tiles\_data() (*in module pygeoapi.flask\_app*), 63

get\_collection\_tiles\_metadata() (*in module pygeoapi.flask\_app*), 63

- [get\\_coverage\\_domainset\(\)](#) (*pygeoapi.provider.base.BaseProvider* method), 70  
[get\\_coverage\\_rangetype\(\)](#) (*pygeoapi.provider.base.BaseProvider* method), 70  
[get\\_data\\_path\(\)](#) (*pygeoapi.provider.base.BaseProvider* method), 70  
[get\\_fields\(\)](#) (*pygeoapi.provider.base.BaseProvider* method), 70  
[get\\_fields\(\)](#) (*pygeoapi.provider.csv.CSVProvider* method), 72  
[get\\_fields\(\)](#) (*pygeoapi.provider.elasticsearch.ElasticsearchProvider* method), 73  
[get\\_fields\(\)](#) (*pygeoapi.provider.geojson.GeoJSONProvider* method), 74  
[get\\_fields\(\)](#) (*pygeoapi.provider.ogr.ogr.Provider* method), 76  
[get\\_fields\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider* method), 78  
[get\\_fields\(\)](#) (*pygeoapi.provider.sqlite.SQLiteGPKGProvider* method), 79  
[get\\_layer\(\)](#) (*pygeoapi.provider.ogr.CommonSourceHelper* method), 75  
[get\\_layer\(\)](#) (*pygeoapi.provider.ogr.ESRIJSONHelper* method), 75  
[get\\_layer\(\)](#) (*pygeoapi.provider.ogr.SourceHelper* method), 77  
[get\\_metadata\(\)](#) (*pygeoapi.provider.base.BaseProvider* method), 71  
[get\\_mimetype\(\)](#) (*in module pygeoapi.util*), 66  
[get\\_next\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider* method), 78  
[get\\_oas\(\)](#) (*in module pygeoapi.openapi*), 64  
[get\\_oas\\_30\(\)](#) (*in module pygeoapi.openapi*), 65  
[get\\_path\\_basename\(\)](#) (*in module pygeoapi.util*), 66  
[get\\_previous\(\)](#) (*pygeoapi.provider.postgresql.PostgreSQLProvider* method), 78  
[get\\_provider\\_by\\_type\(\)](#) (*in module pygeoapi.util*), 66  
[get\\_provider\\_default\(\)](#) (*in module pygeoapi.util*), 66  
[get\\_typed\\_value\(\)](#) (*in module pygeoapi.util*), 66
- ## H
- [HEADERS](#) (*in module pygeoapi.api*), 62  
[HelloWorldProcessor](#) (class *in pygeoapi.process.hello\_world*), 69
- ## I
- [InvalidHelperError](#), 75  
[InvalidPluginError](#), 65  
[is\\_url\(\)](#) (*in module pygeoapi.util*), 66
- ## J
- [json\\_serial\(\)](#) (*in module pygeoapi.util*), 67
- ## L
- [landing\\_page\(\)](#) (*in module pygeoapi.flask\_app*), 63  
[load\\_plugin\(\)](#) (*in module pygeoapi.plugin*), 65
- ## M
- [mask\\_prop\(\)](#) (*pygeoapi.provider.elasticsearch.ElasticsearchProvider* method), 73  
[module](#)  
[pygeoapi.api](#), 61  
[pygeoapi.flask\\_app](#), 62  
[pygeoapi.formatter](#), 67  
[pygeoapi.formatter.base](#), 68  
[pygeoapi.formatter.csv](#), 68  
[pygeoapi.log](#), 64  
[pygeoapi.openapi](#), 64  
[pygeoapi.plugin](#), 65  
[pygeoapi.process](#), 69  
[pygeoapi.process.base](#), 69  
[pygeoapi.process.hello\\_world](#), 69  
[pygeoapi.provider](#), 70  
[pygeoapi.provider.base](#), 70  
[pygeoapi.provider.csv](#), 72  
[pygeoapi.provider.elasticsearch](#), 73  
[pygeoapi.provider.geojson](#), 74  
[pygeoapi.provider.ogr](#), 75  
[pygeoapi.provider.postgresql](#), 77  
[pygeoapi.provider.sqlite](#), 79  
[pygeoapi.util](#), 65
- ## O
- [OGRProvider](#) (class *in pygeoapi.provider.ogr*), 75  
[openapi\(\)](#) (*in module pygeoapi.flask\_app*), 63
- ## P
- [PLUGINS](#) (*in module pygeoapi.plugin*), 65  
[PostgreSQLProvider](#) (class *in pygeoapi.provider.postgresql*), 77  
[pre\\_process\(\)](#) (*in module pygeoapi.api*), 62  
[process\\_jobs\(\)](#) (*in module pygeoapi.flask\_app*), 63  
[PROCESS\\_METADATA](#) (*in module pygeoapi.process.hello\_world*), 69  
[processes\(\)](#) (*in module pygeoapi.flask\_app*), 63  
[ProcessorExecuteError](#), 69  
[ProviderConnectionError](#), 71  
[ProviderGenericError](#), 71

`ProviderInvalidQueryError`, 71  
`ProviderItemNotFoundError`, 71  
`ProviderNoDataError`, 71  
`ProviderNotFoundError`, 71  
`ProviderQueryError`, 71  
`ProviderTypeError`, 71  
`ProviderVersionError`, 71  
`pygeoapi.api`  
    module, 61  
`pygeoapi.flask_app`  
    module, 62  
`pygeoapi.formatter`  
    module, 67  
`pygeoapi.formatter.base`  
    module, 68  
`pygeoapi.formatter.csv_`  
    module, 68  
`pygeoapi.log`  
    module, 64  
`pygeoapi.openapi`  
    module, 64  
`pygeoapi.plugin`  
    module, 65  
`pygeoapi.process`  
    module, 69  
`pygeoapi.process.base`  
    module, 69  
`pygeoapi.process.hello_world`  
    module, 69  
`pygeoapi.provider`  
    module, 70  
`pygeoapi.provider.base`  
    module, 70  
`pygeoapi.provider.csv_`  
    module, 72  
`pygeoapi.provider.elasticsearch_`  
    module, 73  
`pygeoapi.provider.geojson`  
    module, 74  
`pygeoapi.provider.ogr`  
    module, 75  
`pygeoapi.provider.postgresql`  
    module, 77  
`pygeoapi.provider.sqlite`  
    module, 79  
`pygeoapi.util`  
    module, 65

## Q

`query()` (*pygeoapi.provider.base.BaseProvider*  
    *method*), 71  
`query()` (*pygeoapi.provider.csv\_.CSVProvider*  
    *method*), 72

`query()` (*pygeoapi.provider.elasticsearch\_.ElasticsearchProvider*  
    *method*), 73  
`query()` (*pygeoapi.provider.geojson.GeoJSONProvider*  
    *method*), 74  
`query()` (*pygeoapi.provider.ogr.ogr.Provider* *method*),  
    76  
`query()` (*pygeoapi.provider.postgresql.PostgreSQLProvider*  
    *method*), 78  
`query()` (*pygeoapi.provider.sqlite.SQLiteGPKGProvider*  
    *method*), 79

## R

`read_data()` (*in module pygeoapi.util*), 67  
`render_j2_template()` (*in module pygeoapi.util*),  
    67

## S

`setup_logger()` (*in module pygeoapi.log*), 64  
`SourceHelper` (*class in pygeoapi.provider.ogr*), 76  
`SQLiteGPKGProvider` (*class in py-*  
    *geoapi.provider.sqlite*), 79  
`stac_catalog_path()` (*in module py-*  
    *geoapi.flask\_app*), 64  
`stac_catalog_root()` (*in module py-*  
    *geoapi.flask\_app*), 64  
`str2bool()` (*in module pygeoapi.util*), 67

## T

`to_json()` (*in module pygeoapi.util*), 67

## U

`update()` (*pygeoapi.provider.base.BaseProvider*  
    *method*), 71  
`update()` (*pygeoapi.provider.geojson.GeoJSONProvider*  
    *method*), 75

## W

`WFSHelper` (*class in pygeoapi.provider.ogr*), 77  
`write()` (*pygeoapi.formatter.base.BaseFormatter*  
    *method*), 68  
`write()` (*pygeoapi.formatter.csv\_.CSVFormatter*  
    *method*), 68

## Y

`yaml_load()` (*in module pygeoapi.util*), 67