

---

# pygeoapi Documentation

pygeoapi team

Jun 29, 2020



---

## Contents:

---

<b>1</b>	<b>pygeoapi project</b>	<b>3</b>
1.1	Demo server . . . . .	3
<b>2</b>	<b>Install</b>	<b>5</b>
2.1	Copy/paste install . . . . .	5
<b>3</b>	<b>OpenAPI</b>	<b>9</b>
3.1	Using OpenAPI . . . . .	9
<b>4</b>	<b>Docker</b>	<b>15</b>
4.1	Running - Basics . . . . .	15
4.2	Running - Overriding the default config . . . . .	16
4.3	Running - Running on a sub-path . . . . .	16
<b>5</b>	<b>WSGI</b>	<b>17</b>
5.1	Running gunicorn . . . . .	17
<b>6</b>	<b>ASGI</b>	<b>19</b>
6.1	Running gunicorn . . . . .	19
<b>7</b>	<b>Configuration</b>	<b>21</b>
7.1	Structured data . . . . .	21
<b>8</b>	<b>Plugins</b>	<b>23</b>
8.1	Plugin data provider plugin . . . . .	23
<b>9</b>	<b>Code documentation</b>	<b>25</b>
9.1	API . . . . .	25
9.2	flask_app . . . . .	26
9.3	Logging . . . . .	27
9.4	OpenAPI . . . . .	27
9.5	Plugins . . . . .	28
9.6	Utils . . . . .	28
9.7	Formatter package . . . . .	29
9.7.1	Base class . . . . .	29
9.7.2	csv . . . . .	30
9.8	Process package . . . . .	30
9.8.1	Base class . . . . .	30

---

9.8.2	hello_world . . . . .	31
9.9	Provider . . . . .	31
9.9.1	Base class . . . . .	31
9.9.2	CSV provider . . . . .	32
9.9.3	Elasticsearch provider . . . . .	33
9.9.4	GeoJSON . . . . .	34
9.9.5	OGR . . . . .	35
9.9.6	postgresql . . . . .	37
9.9.7	sqlite/geopackage . . . . .	38

<b>10</b>	<b>Indices and tables</b>	<b>41</b>
-----------	---------------------------	-----------

<b>Python Module Index</b>	<b>43</b>
----------------------------	-----------

<b>Index</b>	<b>45</b>
--------------	-----------





# CHAPTER 1

---

## pygeoapi project

---

pygeoapi is a Python server implementation of the emerging suite of OGC API standards. pygeoapi is development using the [12-factor app](#) approach, allowing for easy deployment in cloud systems.

pygeoapi is a OsGEO recognized project ([here](#)).

### 1.1 Demo server

There is a demo server on <https://demo.pygeoapi.io> running the latest (Docker) version from the master branch of this repo. pygeoapi runs there at <https://demo.pygeoapi.io/master>.

The demo server setup and config is maintained within a seperate GH repo: <https://github.com/geopython/demo.pygeoapi.io>.



# CHAPTER 2

---

## Install

---

pygeoapi, by default, is natively run as a Flask app (the code struct is an API and Flask is used as a wrapper). Optionally it can be run as a Starlette app.

pygeoapi uses two configuration files: **pygeoapi-config.yml** and **openapi.yml**. First configuration contains all the information and setup to run pygeoapi while the second is exclusively for the openapi.

pygeoapi requires setting PYGEOAPI\_CONFIG and PYGEOAPI\_OPENAPI env variable. PYGEOAPI\_CONFIG points to the yaml file containing the configuration, in the example below we copy the local.config.yml default configuration to pygeoapi-config.yml and use this configuration file.

PYGEOAPI\_OPENAPI variable is the path to openapi file configuration, this file **needs to be autogenerated** using the pygeoapi generate-openapi-document command and the pygeoapi config files e.g.: pygeoapi generate-openapi-document -c local.config.yml > openapi.yml. And then setting the env variable to the path: export PYGEOAPI\_OPENAPI=/path/to/openapi.yml

For production environments it is recommended to use *Docker* or specialized servers like WSGI HTTP server *WSGI* or ASGI HTTP server *ASGI*

## 2.1 Copy/paste install

It is advisable to run pygeoapi inside an isolated environment either *virtualenv* or *docker*, mainly to avoid python package conflicts.

```
# create virtualenv
virtualenv -p python pygeoapi
cd pygeoapi
bin/activate
git clone https://github.com/geopython/pygeoapi.git
cd pygeoapi

# install requirements
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

(continues on next page)

(continued from previous page)

```
# optionally install requirements for starlette
pip install -r requirements-starlette.txt

# install source in current directory
pip install -e .
cp pygeoapi-config.yml local.config.yml
# edit configuration file
nano local.config.yml

export PYGEOAPI_CONFIG=/path/to/local.config.yml
# generate OpenAPI Document
pygeoapi generate-openapi-document -c local.config.yml > openapi.yml
export PYGEOAPI_OPENAPI=/path/to/openapi.yml

# run pygeoapi
pygeoapi serve

# optionally run pygeoapi with starlette
pygeoapi serve --starlette
```

If the default configuration was used then we should have pygeoapi running on

The screenshot shows the pygeoapi Demo instance - running latest GitHub version (3.0.2) OAS3 API documentation. It includes a navigation bar with links to Terms of service, pygeoapi Development Team - Website, Send email to pygeoapi Development Team, and CC-BY 4.0 license. Below this is a 'Servers' section with a dropdown menu set to <https://demo.pygeoapi.io/master/api>. The main content area is organized into sections: **server**, **obs**, and **lakes**. The **server** section lists endpoints for API, /api, /collections, /conformance, and /processes. The **obs** section lists endpoints for /collections/obs, /collections/obs/items, and /collections/obs/items/{id}. The **lakes** section lists endpoints for /collections/lakes, /collections/lakes/items, and /collections/lakes/items/{id}.

pygeoapi provides an API to geospatial data

Terms of service  
pygeoapi Development Team - Website  
Send email to pygeoapi Development Team  
CC-BY 4.0 license

Servers

<https://demo.pygeoapi.io/master/api>

**server** pygeoapi provides an API to geospatial data

information: <https://github.com/geopython/pygeoapi>

**GET** / API

**GET** /api This document

**GET** /collections Feature Collections

**GET** /conformance API conformance definition

**GET** /processes Processes

**obs** Observations

**GET** /collections/obs Get feature collection metadata

**GET** /collections/obs/items Get Observations features

**GET** /collections/obs/items/{id} Get Observations feature by id

**lakes** lakes of the world, public domain

**GET** /collections/lakes Get feature collection metadata

**GET** /collections/lakes/items Get Large Lakes features

**GET** /collections/lakes/items/{id} Get Large Lakes feature by id



# CHAPTER 3

---

## OpenAPI

---

OpenAPI spec is an open specification for REST end points, currently OGC services are being redefined using such specification. The REST structure and payload are defined using yaml file structures, the file structure is described here: <https://swagger.io/docs/specification/basic-structure/>

pygeoapi REST end points descriptions on OpenAPI standard are automatically generated based on the configuration file:

```
pygeoapi generate-openapi-document -c local.config.yml > openapi.yml
```

The api will then be accessible at `/openapi` endpoint.

For api demo please check: <https://demo.pygeoapi.io/master/openapi>

The api page has REST description but also integrated clients that can be used to send requests to the REST end points and see the response provided

### 3.1 Using OpenAPI

Accessing the openAPI webpage we have the following structure:

The screenshot shows the Swagger UI for the pygeoapi Demo instance. At the top, it displays the title "pygeoapi Demo instance - running latest GitHub version" with a version of 3.0.2 and an OAS3 badge. Below the title, there's a link to the API documentation at <https://demo.pygeoapi.io/master/api>. The page includes links to "Terms of service", "pygeoapi Development Team - Website", "Send email to pygeoapi Development Team", and "CC-BY 4.0 license". A "Servers" section shows the URL <https://demo.pygeoapi.io/master> - pygeoapi provides an API to geospatial data.

**server** pygeoapi provides an API to geospatial data

information: <https://github.com/geopython/pygeoapi>

**GET** / API

**GET** /api This document

**GET** /collections Feature Collections

**GET** /conformance API conformance definition

**GET** /processes Processes

**obs** Observations

**GET** /collections/obs Get feature collection metadata

**GET** /collections/obs/items Get Observations features

**GET** /collections/obs/items/{id} Get Observations feature by id

**lakes** lakes of the world, public domain

**GET** /collections/lakes Get feature collection metadata

**GET** /collections/lakes/items Get Large Lakes features

**GET** /collections/lakes/items/{id} Get Large Lakes feature by id

Please notice that **each dataset** will be represented as a REST end point under *collections*

In this example we will test and *GET* data concerning windmills in the Netherlands, first we will check the available datasets, by accessing the service's collections:

The screenshot shows the pygeoapi OpenAPI interface. At the top, it says "server" and "pygeoapi provides an API to geospatial data". Below that, there are several "GET" buttons for different endpoints: "/ API", "/api This document", and "/collections Feature Collections". The last one is highlighted with a red box. Underneath, there's a "Parameters" section with "No parameters" and a "Responses" section. In the "Responses" section, there's a table with one row for "200": "Code" (200), "Description" ("successful operation"), and "Links" (No links). At the bottom right of the "Responses" section, there's a "Cancel" button and a red box highlighting the "Execute" button.

The service collection metadata will contain a description of the collections provided by the server

The screenshot shows the pygeoapi API response for the "/collections" endpoint. It includes a "Curl" section with a command, a "Request URL" section with "https://demo.pygeoapi.io/master/collections", and a "Server response" section. The "Server response" section has tabs for "Code" (selected) and "Details". Under "Code", there's a "200 Response body" section containing JSON. A red box highlights the "description" field of the first dataset, which reads: "Locations of windmills within the Netherlands from Rijksdienst voor het Cultureel Erfgoed (RCE) INSPIRE WFS. Uses GeoServer WFS v2 backend via OGRProvider.". There's also a "Download" button at the bottom right of the response body.

The dataset *dutch\_windmills* will be available on the *collections* end point, in the following example we'll obtain the specific metadata of the dataset

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

**GET** /collections/dutch\_windmills Get feature collection metadata

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

**Parameters**

No parameters

**Responses**

**Curl**

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

**Request URL**

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

**Code** **Details**

200 **Response body**

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

**Download**

**Response headers**

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

features/items composing the data are aggregated on the `/items` end point, in this REST end point it is possible to obtain all dataset, or restrict it features/items to a **numerical limit, bounding box, time stamp, pagging** (start index)

The screenshot shows the pygeoapi OpenAPI interface for the `/collections/dutch_windmills/items` endpoint. The interface is a web-based form with the following fields:

- f**: A dropdown menu set to `json`.
- bbox**: An input field with a placeholder `[minx, miny, maxx, maxy]`.
- time**: An input field with a placeholder `time - The time parameter indicates an`.
- limit**: An input field set to `10`.
- sortby**: An input field with a placeholder `sortby - The optional sortby parameter`.
- startIndex**: An input field set to `0`.

At the bottom right of the interface is a red-bordered **Cancel** button.

For each feature in dataset we have a **specific identifier** (notice that the identifier is not part of the JSON properties),

The screenshot shows the response from the `/collections/dutch_windmills/items` endpoint. The response body is a JSON array of features, with one specific feature highlighted:

```

[{"id": "Molens.1", "type": "Feature", "geometry": {"type": "Point", "coordinates": [5.057482816805334, 52.17198007919141]}, "properties": {"gid": 1, "NAAM": "De Trouwe Wagter of Trouwe Wachter", "PLAATS": "Tienhoven", "CATEGORIE": "windmolens", "FUNCTIE": "poldermolens", "TYPE": "wipmolens", "STAAT": "bestaand", "RMONNUMMER": "26483", "TBGNUMMER": "00003", "INFORLINK": "https://zoeken.allermolens.nl/tenbruggencatenummer/00003", "THUMBNAIL": "https://images.memorix.nl/rce/thumb/350x350/9165dd5b-34b8-705d-0128-3196d2831677.jpg", "HDFUNCTIE": "poldermolens", "FOTOGRAAF": "Frank Terpstra", "FOTO_GROOT": "https://images.memorix.nl/rce/thumb/fullsize/9165dd5b-34b8-705d-0128-3196d2831677.jpg", "BOUWJAAR": "1832"}]

```

The `id` field of the first feature is highlighted with a red box.

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

GET /collections/dutch\_windmills/items/{id} Get Windmills within The Netherlands feature by id

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

**Parameters**

**Name** **Description**

**id** \* required  
string  
(path)

**f**  
string  
(query)

**Execute**

**cancel**

# CHAPTER 4

---

## Docker

---

Docker Images `geopython/pygeoapi:latest` and versions are available on [DockerHub](#).

Each Docker Image contains a default configuration `default.config.yml` with the project's test data and WFS3 datasets.

You can override this default config via Docker Volume mapping or by extending the Docker Image and copying in your config.

See an [example for the geoapi demo server](#) for the latter method.

<https://github.com/geopython/demo.pygeoapi.io/tree/master/services> Depending on your config you may need specific backends to be available.

## 4.1 Running - Basics

By default this Image will start a pygeoapi Docker Container using gunicorn on internal port 80.

To run with default built-in config 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
```

## 4.2 Running - Overriding the default config

Normally you would override the `default.config.yml` with your own `pygeoapi` config. This can be effected best 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
```

But better/cleaner is to use `docker-compose`. Something like:

```
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 config:

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

See how the demo server is setup ([here](#))

## 4.3 Running - Running 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 correct you need to set `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
```

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

Or within a `docker-compose.yml` full example:

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

See `pygeoapi` demo service for an full example.

# CHAPTER 5

---

## WSGI

---

Web Server Gateway Interface (WSGI) is standard for forwarding request to web applications written on Python language. pygeoapi it self doesn't implement WSGI since it is an API, therefore it is required a webframework to access HTTP requests and pass the information to pygeoapi

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

the pygeoapi package integrates [Flask](#) as webframework for defining the API routes/end points and WSGI support.

The flask WSGI server can be easily run as a pygeoapi command with the option *-flask*:

```
pygeoapi serve --flask
```

Running a native Flask server is not advisable, the prefered option is as follows:

```
HTTP request --> WSGI server (gunicorn) --> Flask (flask_app.py) --> pygeoapi API
```

By having a specific WSGI server, the HTTPS are efficiently processed into threads/processes. The current docker pygeoapi implement such strategy (see section: [Docker](#)), it is prefered to implement pygeopai using docker solutions than running host native WSGI servers.

## 5.1 Running gunicorn

Gunicorn is one of several WSGI supporting server on python (list of server supporting WSGI: [here](#)). This server is simple to run from the command, e.g:

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

For extra configuration parameters like port binding, workers please consult the gunicorn [settings](#)



# CHAPTER 6

---

## ASGI

---

Asynchronous Server Gateway Interface (ASGI) is standard interface between async-capable web servers, frameworks, and applications written on Python language. pygeoapi itself doesn't implement ASGI since it is an API, therefore it is required a webframework to access HTTP requests and pass the information to pygeoapi

```
HTTP request --> Starlette (starlette_app.py) --> pygeoapi API
```

the pygeoapi package integrates `starlette_app` as webframework for defining the API routes/end points and WSGI support.

The starlette ASGI server can be easily run as a pygeoapi command with the option `-starlette`:

```
pygeoapi serve --starlette
```

Running a Uvicorn server is not advisable, the preferred option is as follows:

```
HTTP request --> ASGI server (gunicorn) --> Starlette (starlette_app.py) --> pygeoapi API
```

By having a specific ASGI server, the HTTPS are efficiently processed into threads/processes. The current docker pygeoapi implement such strategy (see section: [Docker](#)), it is prefered to implement pygeopai using docker solutions than running host native ASGI servers.

## 6.1 Running 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. This server is simple to run from the command, e.g:

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

For extra configuration parameters like port binding, workers please consult the `gunicorn` [settings](#)



# CHAPTER 7

---

## Configuration

---

pygeoapi uses a yaml file as configuration source and the file location is read from the PYGEOAPI\_CONFIG env variable

---

**Note:** pygeoapi is under high development, and new configuration parameters are constantly being added. For the latest parameters please consult the [pygeoapi-config.yml](#) file provided on github

---

Using `pygeoapi-config.yml` as reference we will have the following sections:

- *server* for server related configurations
- *logging* for logging configuration
- *metadata* server and content metadata (information used to populate multiple content)
- *datasets* data content offered by server (collections in WFS3.0)

### 7.1 Structured data



pygeoapi supports structured metadata about a deployed instance, and is also capable of presenting feature 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 YAML. 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 feature in a collection, that is the URL for that feature (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 features 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 YAML 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 features can be expressed but is dependent on such relationships being expressed by the dataset provider, not pygeoapi.

# CHAPTER 8

## Plugins

In this section we will explain how pygeoapi uses a plugin approach for data providers, formatters and processes.

### 8.1 Plugin data provider plugin

Plugins are in general modules containing derived classed classes that ensure minimal requirements for the plugin to work. Lets consider the steps for a data provider plugin (source code is located here: [Provider](#))

1. create a new module file on the *provider folder* (e.g myprovider.py)
2. copy code from *base.py*
3. import base provider class

```
from pygeoapi.provider.base import BaseProvider
```

4. create a child class from the *BaseProvider* class with a specific name

```
class BaseProvider(object):
    """generic Provider ABC"""

    def __init__(self, provider_def):
        """
        Initialize object
        """


```

to become:

```
class MyDataProvider(object):
    """My data provider"""

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

5. implement class methods.

```
def query(self):  
  
    def get(self, identifier):  
  
        def create(self, new_feature):  
  
            def update(self, identifier, new_feature):  
  
                def delete(self, identifier):
```

The above class methods are related to the specific URLs defined on the OGC openapi specification:

# CHAPTER 9

---

## Code documentation

---

Top level code documentation. Follow link in section for module/class member information

### 9.1 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 feature 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 0.7'}
```

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*

## 9.2 flask\_app

Flask module providing the route paths to the api

```
pygeoapi.flask_app.conformance()
```

OGC open api conformance access point

**Returns** HTTP response

```
pygeoapi.flask_app.dataset(feature_collection, feature=None)
```

OGC open api collections/{dataset}/items/{feature} access point

**Returns** HTTP response

```
pygeoapi.flask_app.describe_collections(name=None)
```

OGC open api collections access point

**Parameters** **name** – identifier of collection name

**Returns** HTTP response

```
pygeoapi.flask_app.describe_processes(name=None)
```

OGC open api processes access point (experimental)

**Parameters** **name** – identifier of process to describe

**Returns** HTTP response

```
pygeoapi.flask_app.execute_process(name=None)
    OGC open api jobs from processes access point (experimental)
```

**Parameters** `name` – identifier of process to execute

**Returns** HTTP response

```
pygeoapi.flask_app.openapi()
    OpenAPI access point
```

**Returns** HTTP response

```
pygeoapi.flask_app.root()
    HTTP root content of pygeoapi. Intro page access point
```

**Returns** HTTP response

## 9.3 Logging

Logging system

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

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

**Returns** void (creates logging instance)

## 9.4 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

## 9.5 Plugins

---

**Note:** Please consult section [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

## 9.6 Utils

Generic util functions used in the code

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

Attempts to obtain a date value from a collection.

**Parameters**

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

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

`pygeoapi.util.get_typed_value(value)`

Derive true type from data value

**Parameters** `value` – value

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

`pygeoapi.util.is_url(urlstring)`

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

`pygeoapi.util.json_serial(obj)`

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

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

render Jinja2 template

#### Parameters

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

**Returns** string of rendered template

`pygeoapi.util.str2bool(value)`

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

#### Parameters value – value to be evaluated

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

`pygeoapi.util.to_json(dict_)`

Serialize dict to json

#### Parameters dict – dict of JSON representation

**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

## 9.7 Formatter package

Output formatter package

### 9.7.1 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
```

## 9.7.2 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
```

## 9.8 Process package

OGC process package, each process is an independent module

### 9.8.1 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)

```

## 9.8.2 hello\_world

Hello world example process

```

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

    Hello World Processor example

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

    __repr__()
        Return repr(self).

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

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

```

## 9.9 Provider

Provider module containing the plugins wrapping data sources

### 9.9.1 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)**  
Updates an existing feature id with new\_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 ()**  
query the provider

**Returns** 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

**exception** `pygeoapi.provider.base.ProviderConnectionError`  
Bases: Exception  
query / backend error

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

**exception** `pygeoapi.provider.base.ProviderQueryError`  
Bases: Exception  
query / backend error

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

**exception** `pygeoapi.provider.base.ProviderVersionError`  
Bases: Exception  
Incorrect provider version

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

## 9.9.2 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

```
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

### 9.9.3 Elasticsearch provider

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

Elasticsearch Provider

```
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

```
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

## 9.9.4 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)
```

Updates an existing feature id with new\_feature

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

```
get(identifier)
```

query the provider by id

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

**Returns** dict of single GeoJSON feature

---

```
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

## 9.9.5 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://www.gdal.org/ogr\\_sql.html](https://www.gdal.org/ogr_sql.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.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)

```
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-gdal-ogr-cookbook/> [https://www.gdal.org/ogr\\_formats.html](https://www.gdal.org/ogr_formats.html) (per-driver specifics).

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

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

```
_load_source_helper(source_type)
```

Loads Source Helper by name.

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

**Returns** Source Helper object

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

Assembles output from Layer query as GeoJSON FeatureCollection structure.

**Returns** GeoJSON FeatureCollection

```
_response_feature_hits(layer)
```

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

**Returns** GeoJSON FeaturesCollection

```
get(identifier)
```

Get Feature by id

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

**Returns** feature collection

```
get_fields()
```

Get provider field information (names, types)

**Returns** dict of fields

```
query(startindex=0, limit=10, resulttype='results', bbox=[], datetime=None, properties=[], 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 (datestamp or extent)
- **properties** – list of tuples (name, value)
- **sortby** – list of dicts (property, order)

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

---

```
class pygeoapi.provider.ogr.SourceHelper(provider)
Bases: object

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

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

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

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

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

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

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

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

## 9.9.6 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__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/   ho-
    tosm_bdi_waterways/items?resulttype=hits

        Returns GeoJSON FeaturesCollection

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

        Parameters identifier – feature id

        Returns GeoJSON FeaturesCollection

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

**Returns** dict of fields

**get\_next (cursor, identifier)**  
Query next ID given current ID

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

**Returns** feature id

**get\_previous (cursor, identifier)**  
Query previous ID given current ID

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

**Returns** feature id

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

## 9.9.7 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\_\_load()**  
Private method for loading spatiallite, get the table structure and dump geometry

**Returns** sqlite3.Cursor

**\_SQLiteGPKGProvider\_\_response\_feature (row\_data)**  
Assembles GeoJSON output from DB query

**Parameters** **row\_data** – DB row result

**Returns** dict of GeoJSON Feature

**\_SQLiteGPKGProvider\_\_response\_feature\_hits (hits)**  
Assembles GeoJSON/Feature number

**Returns** GeoJSON FeaturesCollection

**get (identifier)**  
Query the provider for a specific feature id e.g: /collections/countries/items/1

**Parameters** `identifier` – feature id  
**Returns** GeoJSON FeaturesCollection

**get\_fields()**  
Get fields from sqlite table (columns are field)  
**Returns** dict of fields

**query** (`startindex=0, limit=10, resulttype='results', bbox=[], datetimemode=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]
- `datetimemode` – temporal (datestamp or extent)
- `properties` – list of tuples (name, value)
- `sortby` – list of dicts (property, order)

**Returns** GeoJSON FeaturesCollection



# CHAPTER 10

---

## Indices and tables

---

- genindex
- modindex
- search



---

## Python Module Index

---

### p

pygeoapi.api, 25  
pygeoapi.flask\_app, 26  
pygeoapi.formatter, 29  
pygeoapi.formatter.base, 29  
pygeoapi.formatter.csv\_, 30  
pygeoapi.log, 27  
pygeoapi.openapi, 27  
pygeoapi.plugin, 28  
pygeoapi.process, 30  
pygeoapi.process.base, 30  
pygeoapi.process.hello\_world, 31  
pygeoapi.provider, 31  
pygeoapi.provider.base, 31  
pygeoapi.provider.csv\_, 32  
pygeoapi.provider.elasticsearch\_, 33  
pygeoapi.provider.geojson, 34  
pygeoapi.provider.ogr, 35  
pygeoapi.provider.postgresql, 37  
pygeoapi.provider.sqlite, 38  
pygeoapi.util, 28



### Symbols

\_PostgreSQLProvider\_\_response\_feature()      (`pygeoapi.provider.postgresql.PostgreSQLProvider` method), 37  
  \_\_\_\_weakref\_\_(`pygeoapi.plugin.InvalidPluginError` attribute), 28  
\_PostgreSQLProvider\_\_response\_feature\_hits()      (`pygeoapi.provider.postgresql.PostgreSQLProvider` method), 37  
  \_\_\_\_weakref\_\_(`pygeoapi.process.base.BaseProcessor` attribute), 31  
\_SQLiteGPKGProvider\_\_load()      (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` method), 38  
  \_\_\_\_weakref\_\_(`pygeoapi.provider.base.BaseProvider` attribute), 32  
  \_\_\_\_weakref\_\_(`pygeoapi.provider.base.ProviderConnectionError` attribute), 32  
\_SQLiteGPKGProvider\_\_response\_feature()      (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` method), 38  
  \_\_\_\_weakref\_\_(`pygeoapi.provider.base.ProviderQueryError` attribute), 32  
\_SQLiteGPKGProvider\_\_response\_feature\_hits()      (`pygeoapi.provider.sqlite.SQLiteGPKGProvider` method), 38  
  \_\_\_\_weakref\_\_(`pygeoapi.provider.base.ProviderVersionError` attribute), 32  
  \_\_\_\_load()      (`pygeoapi.provider.csv.CSVProvider` method), 33  
\_\_init\_\_() (`pygeoapi.api.API` method), 25  
\_\_init\_\_() (`pygeoapi.formatter.base.BaseFormatter` method), 29  
\_\_init\_\_() (`pygeoapi.formatter.csv.CSVFormatter` method), 30  
\_\_init\_\_() (`pygeoapi.process.base.BaseProcessor` method), 30  
\_\_init\_\_() (`pygeoapi.process.hello_world.HelloWorldProcessor` method), 31  
\_\_init\_\_() (`pygeoapi.provider.base.BaseProvider` method), 31  
\_\_repr\_\_() (`pygeoapi.formatter.base.BaseFormatter` method), 29  
\_\_repr\_\_() (`pygeoapi.formatter.csv.CSVFormatter` method), 30  
\_\_repr\_\_() (`pygeoapi.process.base.BaseProcessor` method), 30  
\_\_repr\_\_() (`pygeoapi.process.hello_world.HelloWorldProcessor` method), 31  
\_\_repr\_\_() (`pygeoapi.provider.base.BaseProvider` method), 31  
\_\_\_\_weakref\_\_(`pygeoapi.api.API` attribute), 25  
\_\_\_\_weakref\_\_(`pygeoapi.formatter.base.BaseFormatter` attribute), 30  
\_\_\_\_weakref\_\_(`pygeoapi.plugin.InvalidPluginError` attribute), 28  
\_\_\_\_weakref\_\_(`pygeoapi.process.base.BaseProcessor` attribute), 31  
\_\_\_\_weakref\_\_(`pygeoapi.provider.base.BaseProvider` attribute), 31  
\_\_\_\_load\_source\_helper()      (`pygeoapi.provider.ogr.OGRProvider` method), 36  
  \_\_\_\_response\_feature\_collection()      (`pygeoapi.provider.ogr.OGRProvider` method), 36  
  \_\_\_\_response\_feature\_hits()      (`pygeoapi.provider.ogr.OGRProvider` method), 36

### A

API (class in `pygeoapi.api`), 25

### B

BaseFormatter (class in `pygeoapi.formatter.base`), 29  
BaseProcessor (class in `pygeoapi.process.base`), 30  
BaseProvider (class in `pygeoapi.provider.base`), 31

### C

check\_format() (in module `pygeoapi.api`), 26

close() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 35  
close() (*pygeoapi.provider.ogr.SourceHelper method*), 37  
CommonSourceHelper (class in *pygeoapi.provider.ogr*), 35  
conformance() (*in module pygeoapi.flask\_app*), 26  
create() (*pygeoapi.provider.base.BaseProvider method*), 32  
create() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 34  
CSVFormatter (*class in pygeoapi.formatter.csv\_*), 30  
CSVProvider (*class in pygeoapi.provider.csv\_*), 32

**D**

DatabaseConnection (*class in pygeoapi.provider.postgresql*), 37  
dataset() (*in module pygeoapi.flask\_app*), 26  
dategetter() (*in module pygeoapi.util*), 28  
delete() (*pygeoapi.provider.base.BaseProvider method*), 32  
delete() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 34  
describe\_collections() (*in module pygeoapi.flask\_app*), 26  
describe\_processes() (*in module pygeoapi.flask\_app*), 26  
disable\_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 35  
disable\_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 35  
disable\_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 37  
disable\_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 37

**E**

ElasticsearchProvider (*class in pygeoapi.provider.elasticsearch\_*), 33  
enable\_paging() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 35  
enable\_paging() (*pygeoapi.provider.ogr.ESRIJSONHelper method*), 35  
enable\_paging() (*pygeoapi.provider.ogr.SourceHelper method*), 37  
enable\_paging() (*pygeoapi.provider.ogr.WFSHelper method*), 37

37  
ESRIJSONHelper (*class in pygeoapi.provider.ogr*), 35  
execute() (*pygeoapi.process.base.BaseProcessor method*), 31  
execute() (*pygeoapi.process.hello\_world.HelloWorldProcessor method*), 31  
execute\_process() (*in module pygeoapi.flask\_app*), 26  
execute\_process() (*pygeoapi.api.API method*), 25

**F**

FORMATS (*in module pygeoapi.api*), 26

**G**

gen\_media\_type\_object() (*in module pygeoapi.openapi*), 27  
gen\_response\_object() (*in module pygeoapi.openapi*), 27  
GeoJSONProvider (class in *pygeoapi.provider.geojson*), 34  
get() (*pygeoapi.provider.base.BaseProvider method*), 32  
get() (*pygeoapi.provider.csv\_.CSVProvider method*), 33  
get() (*pygeoapi.provider.elasticsearch\_ElasticsearchProvider method*), 33  
get() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 34  
get() (*pygeoapi.provider.ogr.OGRProvider method*), 36  
get() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 37  
get() (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 38  
get\_collection\_items() (*pygeoapi.api.API method*), 25  
get\_fields() (*pygeoapi.provider.base.BaseProvider method*), 32  
get\_fields() (*pygeoapi.provider.elasticsearch\_ElasticsearchProvider method*), 33  
get\_fields() (*pygeoapi.provider.ogr.OGRProvider method*), 36  
get\_fields() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 37  
get\_fields() (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 39  
get\_layer() (*pygeoapi.provider.ogr.CommonSourceHelper method*), 35  
get\_layer() (*pygeoapi.provider.ogr.SourceHelper method*), 37  
get\_next() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 38  
get\_oas() (*in module pygeoapi.openapi*), 27  
get\_oas\_30() (*in module pygeoapi.openapi*), 28

get\_previous() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 38  
 get\_typed\_value() (*in module pygeoapi.util*), 28

**H**

HEADERS (*in module pygeoapi.api*), 26  
 HelloWorldProcessor (*class in pygeoapi.process.hello\_world*), 31

**I**

InvalidHelperError, 35  
 InvalidPluginError, 28  
 is\_url() (*in module pygeoapi.util*), 29

**J**

json\_serial() (*in module pygeoapi.util*), 29

**L**

load\_plugin() (*in module pygeoapi.plugin*), 28

**O**

ogrProvider (*class in pygeoapi.provider.ogr*), 36  
 openapi() (*in module pygeoapi.flask\_app*), 27

**P**

PLUGINS (*in module pygeoapi.plugin*), 28  
 PostgreSQLProvider (*class in pygeoapi.provider.postgresql*), 37  
 pre\_process() (*in module pygeoapi.api*), 26  
 PROCESS\_METADATA (*in module pygeoapi.process.hello\_world*), 31  
 ProcessorExecuteError, 31  
 ProviderConnectionError, 32  
 ProviderQueryError, 32  
 ProviderVersionError, 32  
 pygeoapi.api (*module*), 25  
 pygeoapi.flask\_app (*module*), 26  
 pygeoapi.formatter (*module*), 29  
 pygeoapi.formatter.base (*module*), 29  
 pygeoapi.formatter.csv\_ (*module*), 30  
 pygeoapi.log (*module*), 27  
 pygeoapi.openapi (*module*), 27  
 pygeoapi.plugin (*module*), 28  
 pygeoapi.process (*module*), 30  
 pygeoapi.process.base (*module*), 30  
 pygeoapi.process.hello\_world (*module*), 31  
 pygeoapi.provider (*module*), 31  
 pygeoapi.provider.base (*module*), 31  
 pygeoapi.provider.csv\_ (*module*), 32  
 pygeoapi.provider.elasticsearch\_ (*module*), 33  
 pygeoapi.provider.geojson (*module*), 34

pygeoapi.provider.ogr (*module*), 35  
 pygeoapi.provider.postgresql (*module*), 37  
 pygeoapi.provider.sqlite (*module*), 38  
 pygeoapi.util (*module*), 28

**Q**

query() (*pygeoapi.provider.base.BaseProvider method*), 32  
 query() (*pygeoapi.provider.csv\_.CSVProvider method*), 33  
 query() (*pygeoapi.provider.elasticsearch\_.ElasticsearchProvider method*), 33  
 query() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 34  
 query() (*pygeoapi.provider.ogr.OGRProvider method*), 36  
 query() (*pygeoapi.provider.postgresql.PostgreSQLProvider method*), 38  
 query() (*pygeoapi.provider.sqlite.SQLiteGPKGProvider method*), 39

**R**

render\_j2\_template() (*in module pygeoapi.util*), 29  
 root() (*in module pygeoapi.flask\_app*), 27

**S**

setup\_logger() (*in module pygeoapi.log*), 27  
 SourceHelper (*class in pygeoapi.provider.ogr*), 36  
 SQLiteGPKGProvider (*class in pygeoapi.provider.sqlite*), 38  
 str2bool() (*in module pygeoapi.util*), 29

**T**

to\_json() (*in module pygeoapi.util*), 29

**U**

update() (*pygeoapi.provider.base.BaseProvider method*), 32  
 update() (*pygeoapi.provider.geojson.GeoJSONProvider method*), 35

**W**

WFSHelper (*class in pygeoapi.provider.ogr*), 37  
 write() (*pygeoapi.formatter.base.BaseFormatter method*), 30  
 write() (*pygeoapi.formatter.csv\_.CSVFormatter method*), 30

**Y**

yaml\_load() (*in module pygeoapi.util*), 29