xcube geoDB

Release 1.0.6.dev0

Brockmann Consult GmbH

May 02, 2023

GETTING STARTED

xcube geoDB has been developed to store and access feature data in the form of a [GeoDataFrame](http://geopandas.org/). The feature data is stored in a custom PostgreSQL database provided by [Brockmann Consult](https://www.brockmann-consult.de)

CHAPTER

ONE

INSTALLATION

In this chapter we will describe how to install the xcube geoDB infrastructure. The infrastructure consists of four main components:

1. A Python client/API accessing the database through the PostGrest RestAPI (this version)
2. A PostGIS database (Version 14)
3. An xcube geoDB extension to be installed into the PostGIS database (this version)
4. A PostGrest RestAPI (Version 7)

1.1 1. Installing xcube geoDB Client/API

The aim of the chapter is to describe the installation of the xcube geoDB client which serves as a wrapper to accessing an existing xcube geoDB service. You can omit steps 2.-4. entirely if you have gained access to such a service (e.g. by buying xcube geoDB access through the [eurodatacube] (https://eurodatacube.com/) service).

1.1.1 Installation Using the Package Manager conda/mamba

The xcube geoDB client is preferably installed into a conda environment. Ensure that you use Python 3 (>=3.6). The xcube geoDB client has not been tested with Python 2 and will very likely not work.

You install the client using conda

```
$ conda install -c conda-forge xcube_geodb
$ conda activate xcube_geodb
```

The client comes with a jupyter lab pre-installed. To launch the lab type:

```
$ conda activate xcube_geodb
$ jupyter lab
```

We have described the installation using the standard package manager `conda`. However, we strongly encourage you to consider using `mamba` instead, particularly if you combine the xcube geoDB with xcube.

1.1.2 Installation from Sources

We discourage you to install from source. However, if you are a developer, use the following steps. Clone the repository using `git`:

```
$ git clone https://github.com/dcs4cop/xcube-geodb.git
$ cd xcube-geodb
```

You need to create the conda environment to install all dependencies:

```
$ conda env create
```

Once the environment has been created you can activate the environment and install the xcube geoDB client:

```
$ conda activate xcube-geodb
$ python setup.py develop
```

1.2 2. Installation of the Database Backend

This section describes how to set up the xcube geoDB PostGIS database. The xcube geoDB PostGIS database consists of three software components:

- A PostgreSQL database (version 10)
- A PostGIS extension (version 3+)
- The xcube geoDB extension (this version)

The easiest way is to use docker. We maintain a docker image that includes all these three components hosted on quay.io.

```
docker run -p 5432:5432 -e POSTGRES_PASSWORD=mypassword quay.io/bcdev/xcube-geodb-backend
```

For more information about the docker image refer to the PostGIS docker image.

Another option is to install the xcube geoDB extension into an existing PostGIS instance. Prerequisite is, though, that you have full access to your database. If so, you can use a Makefile in our xcube geoDB client repository. To do so clone the repository and move into the sql directory of the xcube_geodb package:

```
$ git clone https://github.com/dcs4cop/xcube-geodb.git
$ cd xcube-geodb/xcube_geodb/sql
```

You will find a `Makefile` in this directory. Before you can install the xcube geoDB extension you need to ensure that the xcube geoDB version is set properly in the filename of the SQL file.

```
export GEODB_VERSION=<version>
make release_version
```

After the execution of the above make command, you will find two new files in your directory:

- `geodb.control` and
- `geodb--<version>.sql`

It is essential that those exist and that the version in the SQL file name matches the xcube geoDB software version you attempt to install. The extension will not install otherwise.

Once the above-mentioned files exist, run `make install`. This will install all necessary files into your PostGIS directory structure.

Lastly, open a PostGreSQL console or a database GUI of your choice as super-user and enter the following SQL command:

```
CREATE EXTENSION geodb;
```

1.3 3. Installation of the Postgrest RestAPI

One of the main objectives of xcube geoDB is to offer an easy way for users to gain access to a PostGIS database via a RestAPI. xcube geoDB takes advantage of the existing PostGreSQL restAPI `Postgrest` version 7.0.1 (we aim to upgrade to version 9 as it integrates better with PostGIS in the future).

To configure a postgrest instance please refer to the postgrest configuration docs. We will give an example in the next chapter where we talk about authorization and authentication.

1.4 4. Authorization/Authentication

The xcube geoDB infrastructure was developed to run on the eurodatacube infrastructure. Hence, it was never meant to run outside a different authorization flow other than oauth2 machine to machine or password flows. Therefore, we provide an example, how to configure postgrest for proper authorization with Auth0 using the client_credentials flow. This configuration should also in principle work with other providers like Keycloak.

Step 1: Create an API in Auth0

Ensure that you name the API (this will be the audience in the client configuration) and make sure that `Add Permissions in the Access Token` is enabled.

Step 2: Create an Application in Auth0

You need to create a 'Machine to Machine' application in Auth0 in order for this example to work. Other providers call this 'client'. Select the API created above and note down the client_id and client_secret Auth0 will provide after the creation of the application.

Step 3: Configure the client for the `client_credentials` auth flow

The Auth0 application is used by xcube geoDB client when connecting to the Auth0 token end points. In our example the xcube geoDB client uses aclient_id/client_secret pair and sends it to the authentication provider. The provider returns the `bearer token`. The token contains information about the client as given in the example below (this example might be incomplete):

```
{
  "iss": "an issuer",
  "aud": [
    "an audience"
  ],
  "scope": "scoped that are authorized",
  "gty": "client-credentials",
  "email": "email of the user",
  "permissions": [
    "read:collections"
  ],
  "https://geodb.brockmann-consult.de/dbrole": "the posgresql role",
```

(continues on next page)

(continued from previous page)

```
  "exp": "expiry date"
}
```

The xcube geoDB client will use that token every time it connects to the postgrest service. The postgrest service will test, whether the token (and hence the user/client) is authorized to access the PostGIS instance. The token also contains client/user information like the PostGreSQL role the client/user is assigned to.

The client can be configured using dotenv for your convenience. Add a .env file in your working directory. Add the following entries of you use client credentials:

```
GEODB_AUTH_CLIENT_ID = "the auth0 client id"
GEODB_AUTH_CLIENT_SECRET = "the auth0 client secret"
GEODB_AUTH_MODE = "client-credentials"
GEODB_AUTH_AUD = "the auth0 audience (The name of your API)"
GEODB_AUTH_DOMAIN="The auth0 domain (Look in yout auth0 application under 'Endpoints/
↪OAuth Token URL')"
GEODB_API_SERVER_URL = "The postgrest API server URL"
GEODB_API_SERVER_PORT = "The postgrest API server port"
```

Step 3 (alternative): Configure the client for the `password` auth flow

The configuration for the password flow is very similar to the client_credentials flow. You need to create a single machine to machine application in Auth0 instead of an application per user. Use the id and secret in your `.env` file. The main difference in the configuration is that you need users with a username and password. You can add those in Auth0 as many as you need. You need to use a `username-password` connection. The username and password can also be configured as environment variable. This is meant to be used in JupyterLabs to provide the user's credentials automatically in the user's notebook.

```
GEODB_AUTH_CLIENT_ID = "the auth0 password flow client id"
GEODB_AUTH_CLIENT_SECRET = "the auth0 password flow client secret"
GEODB_AUTH_MODE = "password"
GEODB_AUTH_AUD = "the auth0 audience (The name of your API)"
GEODB_AUTH_DOMAIN="The auth0 domain (Look in yout auth0 application under 'Endpoints/
↪OAuth Token URL')"
GEODB_AUTH_USERNAME = "the auth0 username"
GEODB_AUTH_PASSWORD = "the auth0 password"
GEODB_API_SERVER_URL = "The postgrest API server URL"
GEODB_API_SERVER_PORT = "The postgrest API server port"
```

Please be aware that the username/password flow is discouraged for security reasons. However, Auth0 has a strict limit on the number of applications (100). Hence, it might be necessary to use the username/password flow in Auth0 if you have a large number of users. Please refer to the Auth0 docs how to set up that flow.

Step 4: Configure the postgrest Service

The postgrest service needs a key to check the signature of the token. This is done using the `jwt-secret` in the postgrest configuration file using asymmetric encryption (tested with Auth0) (see postgrest docs chapter 'JWT from Auth0').

```
db-uri = "postgres://user:mapassword@localhost:5432/geodb"
db-schema = "public, geodb_user_info"
db-anon-role = "anonymous"
jwt-secret = ""{\"alg\":\"RS256\",\"e\":\"AQAB\",\"key_ops\":[\"verify\"],\"kty\":\"RSA\
↪",\"n\":\"aav7svBqEXAw-5D29LO...\"}""
```

The entry in section "n" is provided by Auth0 as a so-called 'public key' of the application you have configured in Auth0.

1.5 5. Installation of the geoserver

The xcube geoDB Python client provides a wrapper around publishing xcube geoDB collections as an e.g. WMS service to a Geoserver instance. In order toaccess such a server, xcube geoDB client needs access to a Geoserver instance using the credentials of a generic Geoserver user. The current xcube geoDB setup uses the docker image of the Geoserver version 2.19.1 (image: terrestris/geoserver:2.19.1).

When installing this docker image we ran into CORS issues and a wrong redirect to a http not https URL after login. The redirect and CORS issues have been resolved by the following settings in the Kubernetes setup:

```
geoserver:
  geoserverCsrfWhitelist: xcube-geodb.brockmann-consult.de
  proxyBaseUrl: https://xcube-geodb.brockmann-consult.de/geoserver
```

These values are imputed as environment variables into the Geoserver container and should also be configurable in the `web.xml` in `/usr/local/tomcat/webapps/geoserver/WEB-INF`.

In addition, a vectortile plugin has been added to the geoserver image by building a custom docker image hosted on quay.io (current version: `quay. io/bcdev/xcube-geoserv:1.0.3`) build using this Dockerfile.

For any more detailed information about installation, please refer to this Dockerfile or the original Installation instructions.

Please be aware that the admin credentials should be changed after installation. Otherwise, any user with even the most mediocre intelligence will be able to log on as admin.

CHAPTER

TWO

MANAGE GEODB COLLECTIONS

The geoDB is a service provided by the Euro Data Cube project (EDC) as a paid service. It comes with a Python client that provides hugh level access to your data, and a certain amount of space in a PostGreSQL database. For managing your data you will need a management (read/write) account to your database which you can purchase at the EDC market place.

You can access the service in two ways:

- By using the Jupyter Python notebook provided by EOX (configuration free, `geodb = GeoDBClient()`)
- By using you own Jupyter notebook or Python script by providing a client id and secret to the GeoDBClient (`geodb = GeoDBClient(client_id="myid", client_secret="mysecet")`)

The client ID and secret is also provided by EOX in the latter case. You will find them in your EOX hub account section. You can also provide the credentials via system environment variables (GEODB_AUTH_CLIENT_ID and GEODB_AUTH_CLIENT_SECRET). These variables can be supplied via a .env file.

There are two different types of geoDB accounts: a read only, and a management (read/write) access. The system will determine your access right through your authentication credentials.

2.1 Manage Collections in your GeoDB

[3]:
```
from xcube_geodb.core.geodb import GeoDBClient
```

2.2 Login from any maschine

Install xcube geoDB with command:

```
conda install xcube_geodb -c conda-forge
```

[1]:
```
### uncomment if not on EDC

#client_id=YourID
#client_secret=YourSecret
#geodb = GeoDBClient(client_id=client_id, client_secret=client_secret, auth_mode="client-
↪credentials")
```

2.3 Login in EDC environment

[5]:
```
### comment if not on EDC

geodb = GeoDBClient()
```

2.4 Get your user name

[5]:
```
geodb.whoami
```

[5]:
```
'geodb_ci_test_user'
```

[6]:
```
# Lets get already existing collections
ds = geodb.get_my_collections()
ds
```

[6]:
```
                           owner                          database table_name
0  geodb_9bfgsdfg-453f-445b-a459  geodb_9bfgsdfg-453f-445b-a459   land_use
1                             tt                              tt      tt300
```

2.5 Creating collections

Once the connection has been established you will be able to create a collection. The collection will contain standard properties (fields) plus custom properties which you can add at your discretion. Please use PostGreSQL type definitions. We recommend stying simple with your data types as we have not tested every single type.

[7]:
```
# Have a look at fiona feature schema
collections = {
        "land_use":
        {
            "crs": 3794,
            "properties":
            {
                "RABA_PID": "float",
                "RABA_ID": "float",
                "D_OD": "date"
            }
        }
    }


geodb.create_collections(collections, clear=True)
```

[7]:
```
{'collections': {'geodb_ci_test_user_land_use': {'crs': 3794,
                                                 'properties': {'D_OD': 'date',
                                                                'RABA_ID': 'float',
                                                                'RABA_PID': 'float'}}}}
```

2.6 Loading data into a dataset

Once the table has been created, you can load data into the dataset. The example below loads a shapefile. The attributes of the shapefile correspond to the dataset's properties.

[7]:
```
import geopandas
gdf = geopandas.read_file('data/sample/land_use.shp')
gdf
```

[7]:
```
       RABA_PID  RABA_ID        D_OD  \
0     4770326.0     1410  2019-03-26
1     4770325.0     1300  2019-03-26
2     2305689.0     7000  2019-02-25
3     2305596.0     1100  2019-02-25
4     2310160.0     1100  2019-03-11
...         ...      ...         ...
9822  6253989.0     1600  2019-03-08
9823  6252044.0     1600  2019-03-26
9824  6245985.0     2000  2019-04-08
9825  6245986.0     2000  2019-02-20
9826  6245987.0     2000  2019-03-11

                                              geometry
0     POLYGON ((453952.629 91124.177, 453952.696 911...
1     POLYGON ((453810.376 91150.199, 453812.552 911...
2     POLYGON ((456099.635 97696.070, 456112.810 976...
3     POLYGON ((455929.405 97963.785, 455933.284 979...
4     POLYGON ((461561.512 96119.256, 461632.114 960...
...                                                 ...
9822  POLYGON ((460637.334 96865.891, 460647.927 969...
9823  POLYGON ((459467.868 96839.686, 459467.770 968...
9824  POLYGON ((459488.998 94066.248, 459498.145 940...
9825  POLYGON ((459676.680 94000.000, 459672.469 939...
9826  POLYGON ((459690.580 94042.607, 459686.872 940...

[9827 rows x 4 columns]
```

[8]:
```
geodb.insert_into_collection('land_use', gdf.iloc[:100,:]) # minimizing rows to 100, if␣
↪you are in EDC, you dont need to make the subset.
```

[8]:
```
Data inserted into land_use
```

[10]:
```
geodb.get_collection('land_use', query="raba_id=eq.7000")
```

[10]:
```
   id                        created_at modified_at  \
0   3  2021-01-22T10:13:54.418035+00:00        None
1  26  2021-01-22T10:13:54.418035+00:00        None
2  95  2021-01-22T10:13:54.418035+00:00        None

                                          geometry  raba_pid  raba_id  \
0  POLYGON ((456099.635 97696.070, 456112.810 976...   2305689     7000
1  POLYGON ((459898.930 100306.841, 459906.288 10...   2301992     7000
2  POLYGON ((459591.248 92619.056, 459592.745 926...   2333229     7000
```

(continues on next page)

(continued from previous page)

```
         d_od
0  2019-02-25
1  2019-04-06
2  2019-02-20
```

2.6.1 Delete from a Collection

[11]:
```
geodb.delete_from_collection('land_use', query="raba_id=eq.7000")
```

[11]:
```
Data from land_use deleted
```

[12]:
```
geodb.get_collection('land_use', query="raba_id=eq.7000")
```

[12]:
```
Empty DataFrame
Columns: [Empty Result]
Index: []
```

2.7 Updating a Collection

[13]:
```
geodb.get_collection('land_use', query="raba_id=eq.1300")
```

[13]:
```
   id                        created_at modified_at  \
0   2  2021-01-22T10:13:54.418035+00:00        None
1  10  2021-01-22T10:13:54.418035+00:00        None
2  63  2021-01-22T10:13:54.418035+00:00        None
3  86  2021-01-22T10:13:54.418035+00:00        None
4  87  2021-01-22T10:13:54.418035+00:00        None
5  92  2021-01-22T10:13:54.418035+00:00        None

                                        geometry  raba_pid  raba_id  \
0  POLYGON ((453810.376 91150.199, 453812.552 911...   4770325     1300
1  POLYGON ((456547.427 91543.640, 456544.255 915...   2318555     1300
2  POLYGON ((456201.531 98685.274, 456199.109 986...   2304287     1300
3  POLYGON ((454709.766 97354.278, 454704.878 973...   2331038     1300
4  POLYGON ((453820.737 98574.017, 453816.740 985...   2357574     1300
5  POLYGON ((461723.552 99635.913, 461729.649 996...   2332405     1300

         d_od
0  2019-03-26
1  2019-03-14
2  2019-02-25
3  2019-01-05
4  2019-01-16
5  2019-03-27
```

[14]:
```
geodb.update_collection('land_use', query="raba_id=eq.1300", values={'d_od': '2000-01-01
↪'})
```

[14]:
```
land_use updated
```

[15]:
```
geodb.get_collection('land_use', query="raba_id=eq.1300")
```

[15]:
```
   id                        created_at                       modified_at  \
0  10  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00
1  86  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00
2   2  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00
3  63  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00
4  87  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00
5  92  2021-01-22T10:13:54.418035+00:00  2021-01-22T10:15:11.329375+00:00

                                            geometry  raba_pid  raba_id  \
0  POLYGON ((456547.427 91543.640, 456544.255 915...   2318555     1300
1  POLYGON ((454709.766 97354.278, 454704.878 973...   2331038     1300
2  POLYGON ((453810.376 91150.199, 453812.552 911...   4770325     1300
3  POLYGON ((456201.531 98685.274, 456199.109 986...   2304287     1300
4  POLYGON ((453820.737 98574.017, 453816.740 985...   2357574     1300
5  POLYGON ((461723.552 99635.913, 461729.649 996...   2332405     1300

         d_od
0  2000-01-01
1  2000-01-01
2  2000-01-01
3  2000-01-01
4  2000-01-01
5  2000-01-01
```

2.8 Managing Properties of a Collection

[16]:
```
geodb.get_my_collections()
```

[16]:
```
                          owner                      database table_name
0  geodb_9bfgsdfg-453f-445b-a459  geodb_9bfgsdfg-453f-445b-a459   land_use
1             geodb_ci_test_user             geodb_ci_test_user   land_use
2                             tt                             tt      tt300
```

[17]:
```
geodb.get_properties('land_use')
```

[17]:
```
             database table_name  column_name                 data_type
0  geodb_ci_test_user   land_use           id                   integer
1  geodb_ci_test_user   land_use   created_at  timestamp with time zone
2  geodb_ci_test_user   land_use  modified_at  timestamp with time zone
3  geodb_ci_test_user   land_use     geometry              USER-DEFINED
4  geodb_ci_test_user   land_use     raba_pid          double precision
5  geodb_ci_test_user   land_use      raba_id          double precision
6  geodb_ci_test_user   land_use         d_od                      date
```

[18]:
```
geodb.add_property('land_use', "test_prop", 'integer')
```

[18]:
```
Properties added
```

[19]:
```
geodb.get_properties('land_use')
```

[19]:
```
              database table_name  column_name                 data_type
0  geodb_ci_test_user   land_use           id                   integer
1  geodb_ci_test_user   land_use   created_at  timestamp with time zone
2  geodb_ci_test_user   land_use  modified_at  timestamp with time zone
3  geodb_ci_test_user   land_use     geometry              USER-DEFINED
4  geodb_ci_test_user   land_use     raba_pid          double precision
5  geodb_ci_test_user   land_use      raba_id          double precision
6  geodb_ci_test_user   land_use         d_od                      date
7  geodb_ci_test_user   land_use    test_prop                   integer
```

[20]:
```
geodb.drop_property('land_use', 'test_prop')
```

[20]:
```
Properties ['test_prop'] dropped from geodb_ci_test_user_land_use
```

[21]:
```
geodb.get_properties('land_use')
```

[21]:
```
              database table_name  column_name                 data_type
0  geodb_ci_test_user   land_use           id                   integer
1  geodb_ci_test_user   land_use   created_at  timestamp with time zone
2  geodb_ci_test_user   land_use  modified_at  timestamp with time zone
3  geodb_ci_test_user   land_use     geometry              USER-DEFINED
4  geodb_ci_test_user   land_use     raba_pid          double precision
5  geodb_ci_test_user   land_use      raba_id          double precision
6  geodb_ci_test_user   land_use         d_od                      date
```

[22]:
```
geodb.add_properties('land_use', properties={'test1': 'integer', 'test2': 'date'})
```

[22]:
```
Properties added
```

[23]:
```
geodb.get_properties('land_use')
```

[23]:
```
              database table_name  column_name                 data_type
0  geodb_ci_test_user   land_use           id                   integer
1  geodb_ci_test_user   land_use   created_at  timestamp with time zone
2  geodb_ci_test_user   land_use  modified_at  timestamp with time zone
3  geodb_ci_test_user   land_use     geometry              USER-DEFINED
4  geodb_ci_test_user   land_use     raba_pid          double precision
5  geodb_ci_test_user   land_use      raba_id          double precision
6  geodb_ci_test_user   land_use         d_od                      date
7  geodb_ci_test_user   land_use        test1                   integer
8  geodb_ci_test_user   land_use        test2                      date
```

[24]:
```
geodb.drop_properties('land_use', properties=['test1', 'test2'])
```

[24]:
```
Properties ['test1', 'test2'] dropped from geodb_ci_test_user_land_use
```

[8]:
```
geodb.get_properties('land_use')
```

[8]:
```
              table_name  column_name                 data_type
0  geodb_admin_land_use           id                   integer
1  geodb_admin_land_use   created_at  timestamp with time zone
2  geodb_admin_land_use  modified_at  timestamp with time zone
3  geodb_admin_land_use     geometry              USER-DEFINED
4  geodb_admin_land_use     raba_pid          double precision
```

(continues on next page)

(continued from previous page)

```
5  geodb_admin_land_use      raba_id          double precision
6  geodb_admin_land_use         d_od                      date
7  geodb_admin_land_use   testä_prop                   integer
```

[25]:
```
geodb.drop_collection('land_use')
```

[25]:
```
Collection ['geodb_ci_test_user_land_use'] deleted
```

[ ]:
```

```

CHAPTER

THREE

SHARE GEODB COLLECTIONS

The geoDB is a service provided by the Euro Data Cube project (EDC) as a paid service. It comes with a Python client that provides hugh level access to your data, and a certain amount of space in a PostGreSQL database. For managing (as sharing is) your data you will need a management (read/write) account to your database which you can purchase at the EDC market place.

You can access the service in two ways:

- By using the Jupyter Python notebook provided by EOX (configuration free, `geodb = GeoDBClient()`)
- By using you own Jupyter notebook or Python script by providing a client id and secret to the GeoDBClient (`geodb = GeoDBClient(client_id="myid", client_secret="mysecet")`)

The client ID and secret is also provided by EOX in the latter case. You will find them in your EOX hub account section. You can also provide the credentials via system environment variables (GEODB_AUTH_CLIENT_ID and GEODB_AUTH_CLIENT_SECRET). These variables can be supplied via a .env file.

There are two different types of geoDB accounts: a read only, and a management (read/write) access. The system will determine your access right through your authentication credentials.

3.1 Sharing Data

[3]:
```
from xcube_geodb.core.geodb import GeoDBClient
```

3.2 Login from any maschine

Install xcube geoDB with command:

```
conda install xcube_geodb -c conda-forge
```

[4]:
```
### uncomment if not on EDC

#client_id=YourID
#client_secret=YourSecret
#geodb = GeoDBClient(client_id=client_id, client_secret=client_secret, auth_mode="client-
↪credentials")
```

3.3 Login in EDC environment

[5]:
```
### comment if not on EDC

geodb = GeoDBClient()
```

3.4 Get your user name

[4]:
```
geodb.whoami
```

[4]:
```
'geodb_ci_test_user'
```

3.5 Create Collection

[6]:
```
import geopandas

# Have a look at fiona feature schema
collections = {
        "land_use":
        {
            "crs": 3794,
            "properties":
            {
                "RABA_PID": "float",
                "RABA_ID": "float",
                "D_OD": "date"
            }
        }
    }


geodb.create_collections(collections, clear=True)

gdf = geopandas.read_file('data/sample/land_use.shp')
geodb.insert_into_collection('land_use', gdf.iloc[:100,:]) # minimizing rows to 100, if␣
↪you are in EDC, you dont need to make the subset.
```

```
Processing rows from 0 to 100
```

[6]:
```
100 rows inserted into land_use
```

3.6 Publish a Collection to the World

[7]:
```
geodb.list_my_grants()
```

[7]:
```
      Grants
0  No Grants
```

Please change the second positional kwargs to the geodb user you want to grant access to, if you are on EDC, go ahead and use 'geodb_test5' user:

[8]:
```
geodb.grant_access_to_collection("land_use", "geodb_admin")
```

[8]:
```
Access granted on land_use to geodb_admin
```

[9]:
```
geodb.list_my_grants()
```

[9]:
```
             database table_name      grantee privileges
0  geodb_ci_test_user   land_use  geodb_admin     SELECT
```

3.7 Accessing the Collection as a different User

Let's access the collection as a different user (a test user in this case), whose credentials were exported as an environment variable. You should now see a land_use collection.

[10]:
```
import os
```

[11]:
```
test_client_id = os.environ.get("TEST_CLIENT_ID")
test_client_secret = os.environ.get("TEST_CLIENT_SECRET")
test_client_geodb_api_server_url=os.environ.get("TEST_GEODB_API_SERVER_URL")
```

[12]:
```
geodb = GeoDBClient(client_id=test_client_id, client_secret=test_client_secret, auth_
↪mode="client-credentials", server_url=test_client_geodb_api_server_url)
geodb.whoami
```

[12]:
```
'geodb_test5'
```

[13]:
```
geodb.get_my_collections()
```

[13]:
```
                                      owner                         database  \
0  geodb_8c2d3fbe-f7a9-4492-8068-121e47e61a4f                          test_db
1  geodb_8c2d3fbe-f7a9-4492-8068-121e47e61a4f                          test_db
2               geodb_9bfgsdfg-453f-445b-a459  geodb_9bfgsdfg-453f-445b-a459

        table_name
0  test_default_db
1    test_deleting
2         land_use
```

3.8 Revoke access

Let's go back to the original user.

[14]:
```
geodb = GeoDBClient()
geodb.whoami
```

[14]:
```
'geodb_ci_test_user'
```

[15]:
```
geodb.list_my_grants()
```

[15]:
```
              database table_name      grantee privileges
0  geodb_ci_test_user   land_use  geodb_admin     SELECT
```

[16]:
```
geodb.revoke_access_from_collection("land_use", 'geodb_admin')
```

[16]:
```
Access revoked from geodb_ci_test_user on land_use
```

[17]:
```
geodb.list_my_grants()
```

[17]:
```
      Grants
0  No Grants
```

3.9 Finally going back to the initial user and deleting the collection:

[20]:
```
geodb.drop_collection('land_use')
```

[20]:
```
Collection ['geodb_ci_test_user_land_use'] deleted
```

[ ]:
```

```

CHAPTER

FOUR

GEODB ACCESS

The geoDB is a service provided by the Euro Data Cube project (EDC) as a payed service. It comes with a Python client that provides hugh level acess to your data and a certain amount of space in a PostGreSQL database. For exploring data you will need at least a read only to the geoDB which you can purchase at the EDC market place.

You can access the service in two ways:

- By using the Jupyter Python notebook provided by EDC Marketplace (configuartion free, `geodb = GeoDBClient()`)
- By using you own Jupyter notebook or Python script by providing a client id and secret to the GeoDBClient (`geodb = GeoDBClient(client_id="myid", client_secret="mysecet")`)

The client ID and secret is also provided by EDC in the latter case. You will find them in your EDC Marketplace account section. You can also provide the credentials via system environment varibles (GEODB_AUTH_CLIENT_ID and GEODB_AUTH_CLIENT_SECRET). These variables can be supplied via a .env file.

4.1 Exploring Data

[1]:
```
from xcube_geodb.core.geodb import GeoDBClient
```

4.2 Login from any maschine

Install xcube geoDB with command:

```
conda install xcube_geodb -c conda-forge
```

[20]:
```
### uncomment if not on EDC
#client_id=YourID
#client_secret=YourSecret
#geodb = GeoDBClient(client_id=client_id, client_secret=client_secret, auth_mode="client-
↪credentials")
```

4.3 Login in EDC environment

[21]:
```
### comment if not on EDC

geodb = GeoDBClient()
```

4.4 Get your user name

[3]:
```
geodb.whoami
```

[3]:
```
'geodb_ci_test_user'
```

[4]:
```
geodb.get_my_collections()
```

[4]:
```
                          owner                         database table_name
0  geodb_9bfgsdfg-453f-445b-a459  geodb_9bfgsdfg-453f-445b-a459   land_use
1                            tt                              tt      tt300
```

[5]:
```
import geopandas

# Have a look at fiona feature schema
collections = {
        "land_use":
        {
            "crs": 3794,
            "properties":
            {
                "RABA_PID": "float",
                "RABA_ID": "float",
                "D_OD": "date"
            }
        }
    }


geodb.create_collections(collections, clear=True)
```

[5]:
```
{'collections': {'geodb_ci_test_user_land_use': {'crs': 3794,
                                                 'properties': {'D_OD': 'date',
                                                                'RABA_ID': 'float',
                                                                'RABA_PID': 'float'}}}}
```

[6]:
```
gdf = geopandas.read_file('data/sample/land_use.shp')
geodb.insert_into_collection('land_use', gdf.iloc[:100,:]) # minimizing rows to 100, if␣
↪you are in EDC, you dont need to make the subset.
```

```
Processing rows from 0 to 100
```

[6]:
```
100 rows inserted into land_use
```

4.5 List Datasets

Step 1: List all datasets a user has access to.

[7]:
```
geodb.get_my_usage() # to be updated so that all available collections are displayed␣
↪includign sensible information ont heir availability, e.g. public, purchased, etc..
```

[7]:
```
{'usage': '96 kB'}
```

[8]:
```
geodb.get_my_collections()
```

[8]:
```
                          owner                       database table_name
0  geodb_9bfgsdfg-453f-445b-a459  geodb_9bfgsdfg-453f-445b-a459   land_use
1              geodb_ci_test_user             geodb_ci_test_user   land_use
2                              tt                             tt      tt300
```

Step 2: Let's get the whole content of a particular data set.

[9]:
```
gdf = geodb.get_collection('land_use') # to be updated, so that namespace is not needed␣
↪or something more suitable, e.g. 'public'
gdf
```

[9]:
```
     id                        created_at modified_at  \
0     1  2021-01-22T10:02:34.390867+00:00        None
1     2  2021-01-22T10:02:34.390867+00:00        None
2     3  2021-01-22T10:02:34.390867+00:00        None
3     4  2021-01-22T10:02:34.390867+00:00        None
4     5  2021-01-22T10:02:34.390867+00:00        None
..  ...                               ...         ...
95   96  2021-01-22T10:02:34.390867+00:00        None
96   97  2021-01-22T10:02:34.390867+00:00        None
97   98  2021-01-22T10:02:34.390867+00:00        None
98   99  2021-01-22T10:02:34.390867+00:00        None
99  100  2021-01-22T10:02:34.390867+00:00        None

                                             geometry  raba_pid  raba_id  \
0   POLYGON ((453952.629 91124.177, 453952.696 911...   4770326     1410
1   POLYGON ((453810.376 91150.199, 453812.552 911...   4770325     1300
2   POLYGON ((456099.635 97696.070, 456112.810 976...   2305689     7000
3   POLYGON ((455929.405 97963.785, 455933.284 979...   2305596     1100
4   POLYGON ((461561.512 96119.256, 461632.114 960...   2310160     1100
..                                                ...       ...      ...
95  POLYGON ((458514.067 93026.352, 458513.306 930...   5960564     1600
96  POLYGON ((458259.239 93110.981, 458259.022 931...   5960569     3000
97  POLYGON ((458199.608 93099.296, 458199.825 930...   5960630     3000
98  POLYGON ((458189.403 93071.618, 458179.669 930...   5960648     1100
99  POLYGON ((454901.777 95801.099, 454889.964 958...   2353305     1321

          d_od
0   2019-03-26
```

(continues on next page)

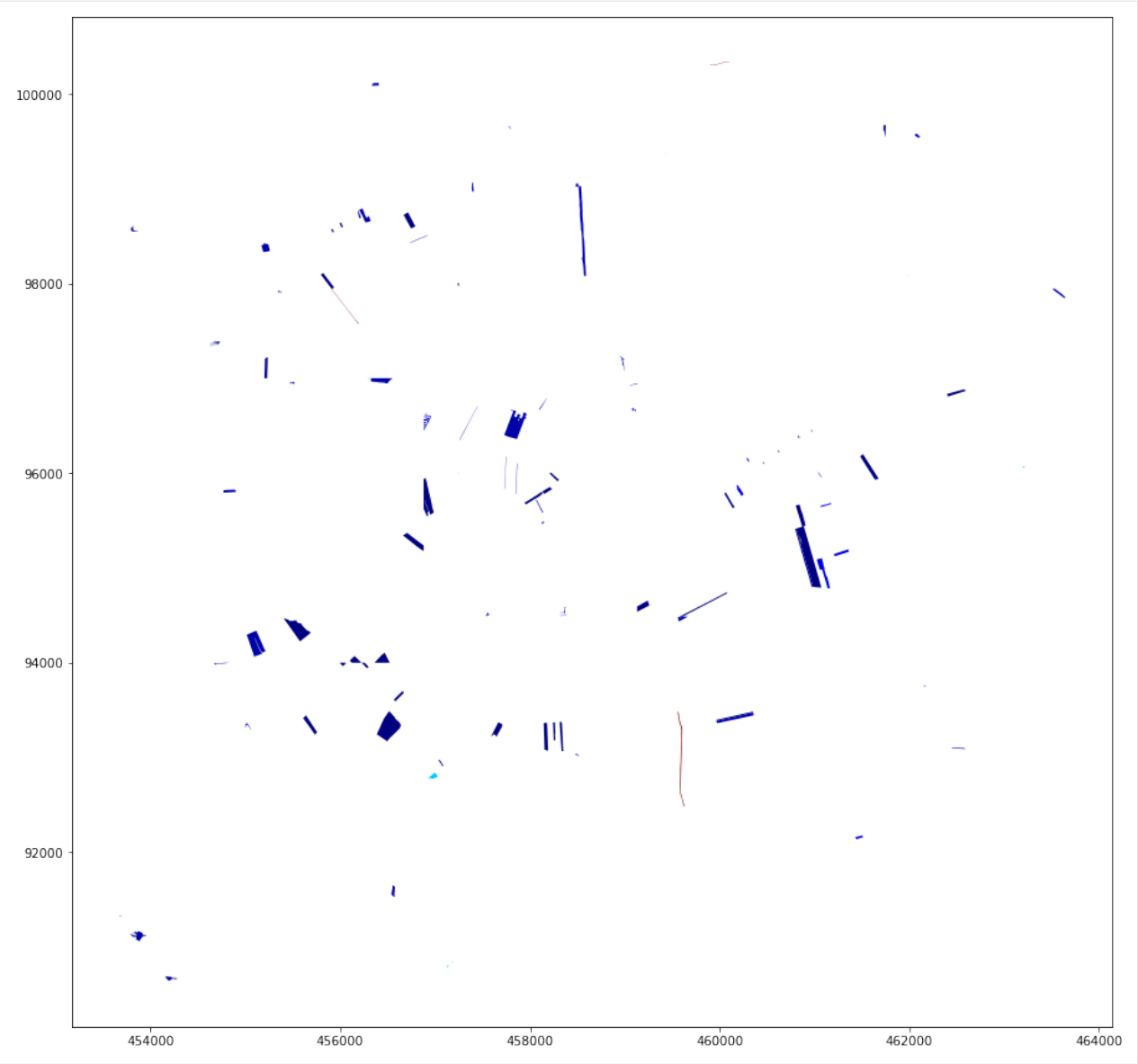
(continued from previous page)

```
1   2019-03-26
2   2019-02-25
3   2019-02-25
4   2019-03-11
..         ...
95  2019-01-11
96  2019-01-11
97  2019-01-11
98  2019-01-11
99  2019-01-05

[100 rows x 7 columns]
```

Step 3: Plot the GeoDataframe, select a reasonable column to diplay

```
[10]: gdf.plot(column="raba_id", figsize=(15,15), cmap = 'jet')
```

```
[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7f323b4fa190>
```



Step 5: Subselect the data. Here: Select a specific use by defining an ID value to choose

[11]:
```
gdfsub = geodb.get_collection('land_use', query='raba_id=eq.1410')
gdfsub.head()
```

[11]:
```
   id                       created_at modified_at  \
0   1  2021-01-22T10:02:34.390867+00:00        None
1  62  2021-01-22T10:02:34.390867+00:00        None
2  22  2021-01-22T10:02:34.390867+00:00        None
3  28  2021-01-22T10:02:34.390867+00:00        None
4  32  2021-01-22T10:02:34.390867+00:00        None

                                        geometry  raba_pid  raba_id  \
0  POLYGON ((453952.629 91124.177, 453952.696 911...   4770326     1410
1  POLYGON ((457261.001 96349.254, 457256.831 963...   3596498     1410
2  POLYGON ((455384.809 97907.054, 455380.659 979...   3616776     1410
```

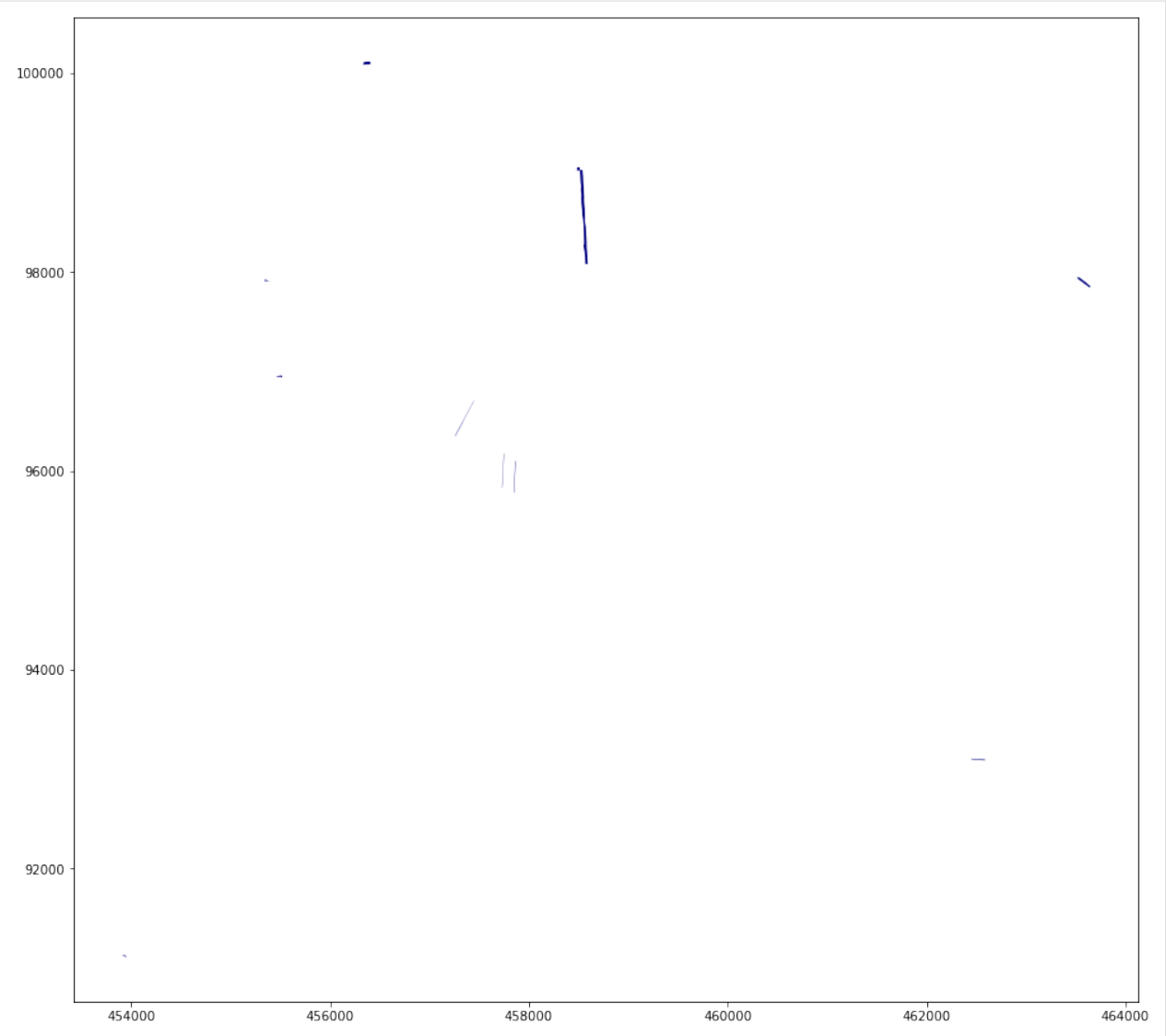
(continues on next page)

(continued from previous page)

```
3  POLYGON ((462585.734 93088.987, 462567.020 930...   3826126     1410
4  POLYGON ((457748.827 96167.354, 457748.394 961...   2309744     1410

         d_od
0  2019-03-26
1  2019-01-05
2  2019-02-25
3  2019-01-23
4  2019-01-05
```

[12]:
```
gdfsub.plot(column="raba_id", figsize=(15,15), cmap = 'jet')
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f323b199190>
```



Step 6: Filter by bbox, limit it to 200 entries

[13]:
```
gdf = geodb.get_collection_by_bbox(collection="land_use",  bbox = (452750.0, 88909.549,␣
↪464000.0, 102486.299), comparison_mode="contains", bbox_crs=3794, limit=200, offset=10)
```

(continues on next page)

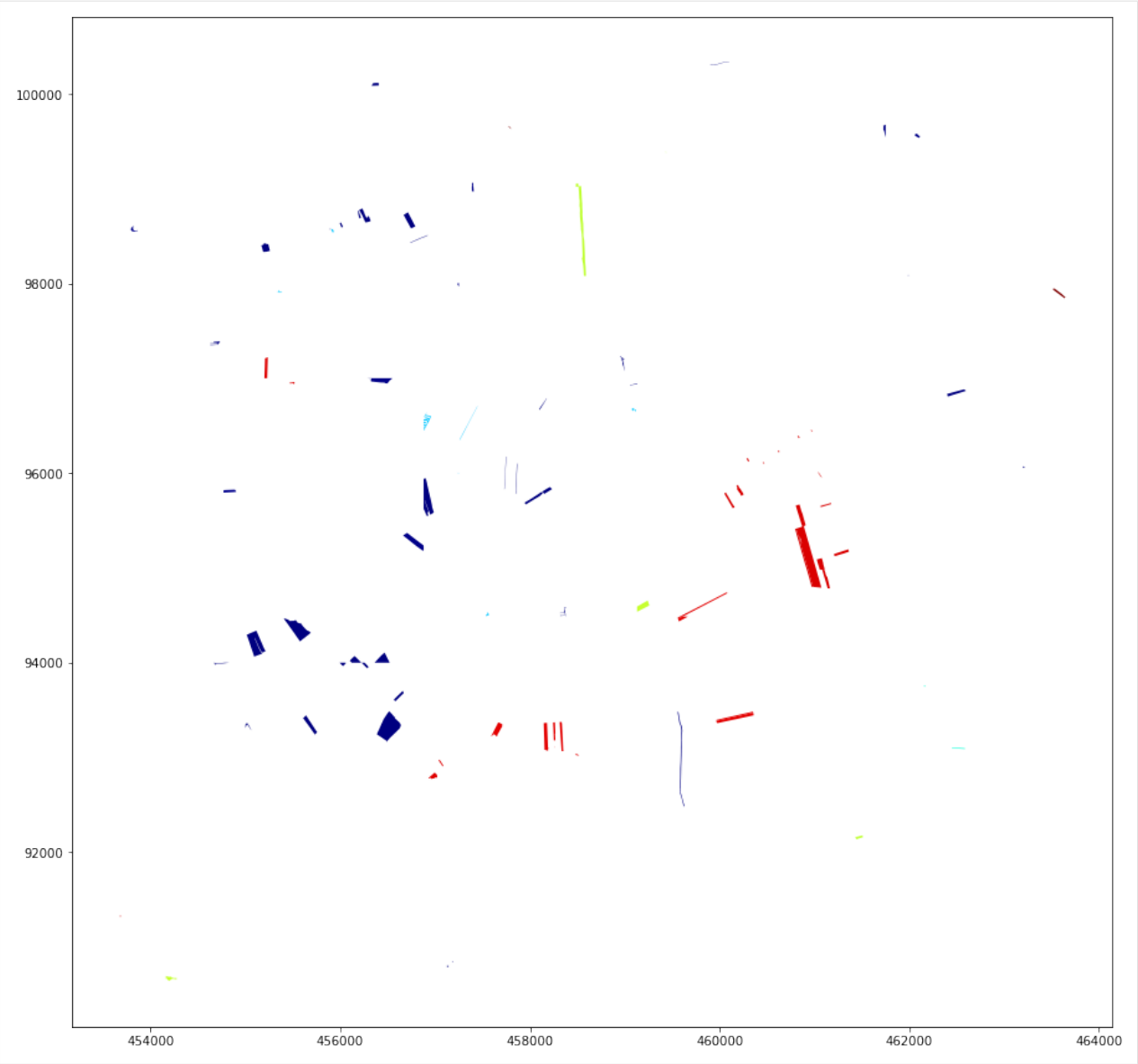
(continued from previous page)

```
gdf
```

[13]:
```
     id                        created_at modified_at  \
0    11  2021-01-22T10:02:34.390867+00:00        None
1    12  2021-01-22T10:02:34.390867+00:00        None
2    13  2021-01-22T10:02:34.390867+00:00        None
3    14  2021-01-22T10:02:34.390867+00:00        None
4    15  2021-01-22T10:02:34.390867+00:00        None
..  ...                               ...         ...
85   96  2021-01-22T10:02:34.390867+00:00        None
86   97  2021-01-22T10:02:34.390867+00:00        None
87   98  2021-01-22T10:02:34.390867+00:00        None
88   99  2021-01-22T10:02:34.390867+00:00        None
89  100  2021-01-22T10:02:34.390867+00:00        None

                                            geometry  raba_pid  raba_id  \
0   POLYGON ((460137.998 95628.898, 460111.001 956...   5983161     1100
1   POLYGON ((453673.609 91328.224, 453678.929 913...   5983074     1600
2   POLYGON ((460312.295 96127.114, 460300.319 961...   5983199     1600
3   POLYGON ((460459.445 96117.356, 460470.516 961...   5983217     1100
4   POLYGON ((457798.753 99628.982, 457783.076 996...   6299143     1600
..                                                ...       ...      ...
85  POLYGON ((458514.067 93026.352, 458513.306 930...   5960564     1600
86  POLYGON ((458259.239 93110.981, 458259.022 931...   5960569     3000
87  POLYGON ((458199.608 93099.296, 458199.825 930...   5960630     3000
88  POLYGON ((458189.403 93071.618, 458179.669 930...   5960648     1100
89  POLYGON ((454901.777 95801.099, 454889.964 958...   2353305     1321

          d_od
0   2019-03-11
1   2019-03-26
2   2019-03-11
3   2019-03-11
4   2019-03-04
..         ...
85  2019-01-11
86  2019-01-11
87  2019-01-11
88  2019-01-11
89  2019-01-05

[90 rows x 7 columns]
```

[14]:
```
gdf.plot(column="raba_pid", figsize=(15,15), cmap = 'jet')
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f323b11b090>
```



Step 6: Fltering using PostGres Syntax; see https://www.postgresql.org/docs/9.1/index.html for details

[15]:
```
gdf = geodb.get_collection_pg(collection='land_use', where='raba_id=1410')
gdf.head()
```

[15]:
```
   id                        created_at modified_at  \
0   1  2021-01-22T10:02:34.390867+00:00        None
1  62  2021-01-22T10:02:34.390867+00:00        None
2  22  2021-01-22T10:02:34.390867+00:00        None
3  28  2021-01-22T10:02:34.390867+00:00        None
4  32  2021-01-22T10:02:34.390867+00:00        None

                                        geometry  raba_pid  raba_id  \
0  POLYGON ((453952.629 91124.177, 453952.696 911...   4770326     1410
1  POLYGON ((457261.001 96349.254, 457256.831 963...   3596498     1410
2  POLYGON ((455384.809 97907.054, 455380.659 979...   3616776     1410
```

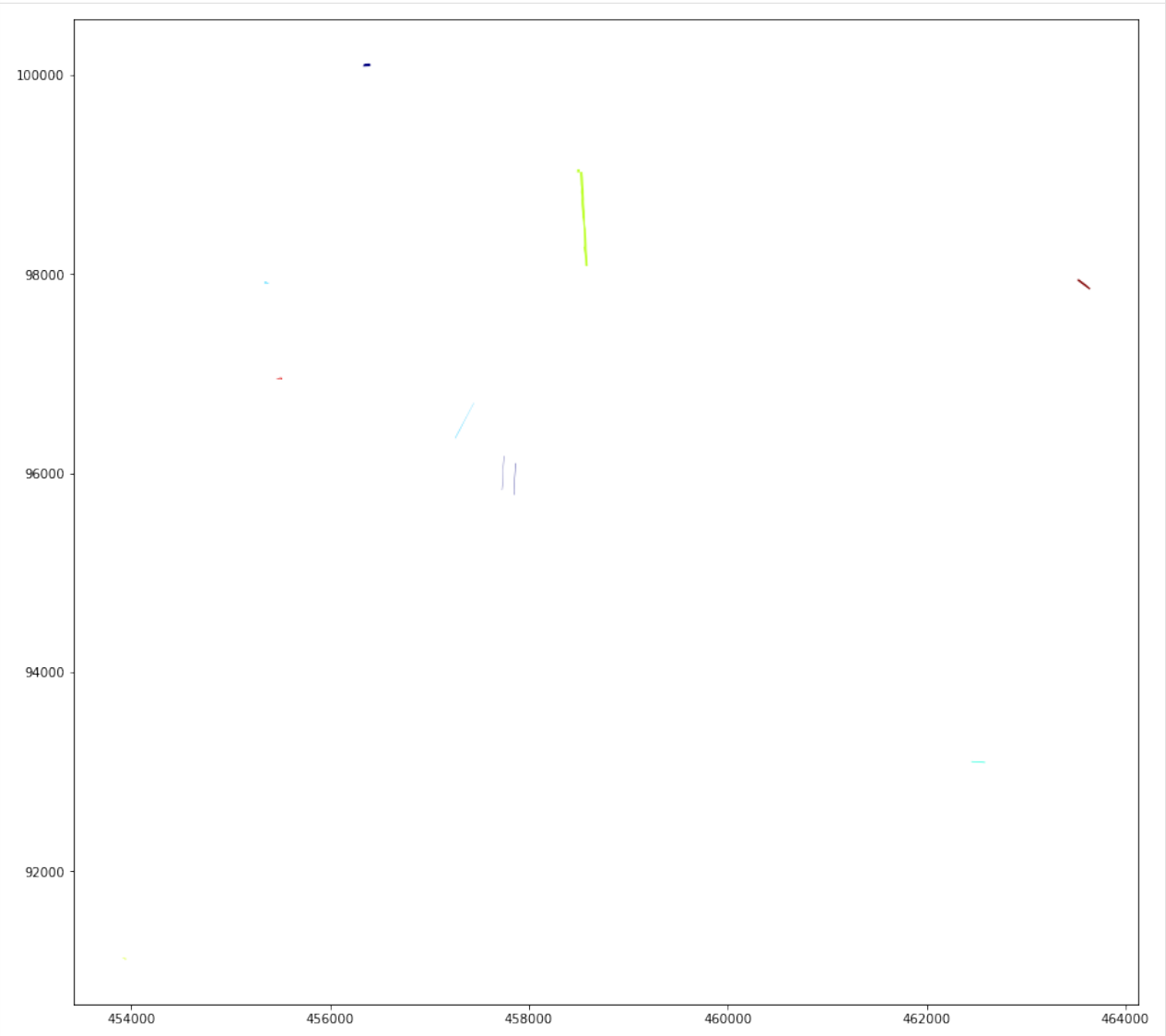
(continues on next page)

(continued from previous page)

```
3  POLYGON ((462585.734 93088.987, 462567.020 930...   3826126     1410
4  POLYGON ((457748.827 96167.354, 457748.394 961...   2309744     1410

         d_od
0  2019-03-26
1  2019-01-05
2  2019-02-25
3  2019-01-23
4  2019-01-05
```

[16]:
```
gdf.plot(column="raba_pid", figsize=(15,15), cmap = 'jet')
```

[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f323b13a9d0>
```



Step 7: Fltering using PostGres Syntax Allowing Aggregation Here according to data, note that the data set has been reduced to 200 entries above

[17]:
```
df = geodb.get_collection_pg('land_use', where='raba_id=1410', group='d_od', select=
↪'COUNT(d_od) as ct, d_od')
df.head()
```

[17]:
```
   ct        d_od
0   1  2019-03-04
1   1  2019-03-20
2   1  2019-03-26
3   1  2019-04-01
4   1  2019-02-25
```

[18]:
```
geodb.drop_collection('land_use')
```

[18]:
```
Collection ['geodb_ci_test_user_land_use'] deleted
```

[ ]:

[ ]:

CHAPTER

FIVE

PUBLISH YOUR COLLECTION USING THE EDC / BC GEOSERVICE

Publish you collections as WMS service. The BC geoservice provides access to publishing your collection as a WMS service using the geoDB Python client. Please refer to the geoserver documentation for configuring the rest API call to your WMS/geojson service.

You will have access to this service if you have purchased a large management account. This service is in beta mode. If you find any issues, please report them on our GitHub Issues page.

5.1 Init the GeoDB client

[7]:
```
from xcube_geodb.core.geodb import GeoDBClient
geodb = GeoDBClient(raise_it=False, server_url='https://xcube-geodb.brockmann-consult.de
↪', gs_server_url='https://xcube-geodb.brockmann-consult.de')
geodb._gs_server_url
```

[7]:
```
'https://xcube-geodb.brockmann-consult.de'
```

5.2 Create a Database and Copy some Public Data Across

[8]:
```
my_database_name = 'my-urban-eea-subset-db13'
```

In case your database does not exist, you need to uncomment the following line and create the database. Please note, that database names must be unique, which means if another user has already used a database name you cannot create a database with the same name.

[9]:
```
geodb.create_database(my_database_name)
```

[9]:
```
<xcube_geodb.core.message.Message at 0x7f47d17014c0>
```

Here we copy a collection from a public database, so we can publish as a geoservice.

[10]:
```
geodb.copy_collection(collection='SI001L2_LJUBLJANA_UA2018', new_collection='SI001L2_
↪LJUBLJANA_UA2018', database='eea-urban-atlas', new_database=my_database_name)
```

[10]:
```
<xcube_geodb.core.message.Message at 0x7f47cfa4e100>
```

You can check if the colleciton was successfully added to your database:

[11]:
```
geodb.get_my_collections(database=my_database_name)
```

[11]:
```
                                      owner                    database  \
0  geodb_965a53df-6c09-4de2-b9ec-1052a2a9534a  my-urban-eea-subset-db13

                  collection                table_name
0  SI001L2_LJUBLJANA_UA2018  SI001L2_LJUBLJANA_UA2018
```

[12]:
```
## List Collections Published on the BC WMS Service

#geodb.get_published_gs(database=my_database_name)
```

5.3 Publish your Collection to the BC WMS Service

[13]:
```
geodb.publish_gs(collection='SI001L2_LJUBLJANA_UA2018', database=my_database_name)
```

[13]:
```
<xcube_geodb.core.message.Message at 0x7f47d0983670>
```

You can now click on the preview link which will give you an unstyled view of your WMS output.

[14]:
```
## List Collections Published on the BC WMS Service

geodb.get_published_gs(database=my_database_name)
```

[14]:
```
<xcube_geodb.core.message.Message at 0x7f47d09912b0>
```

5.4 View the Data

Once you have published you collection, you can use the WMS service to visualise your collection. For this using ipyleaflet. In this example we have used a pre-defined style. You can always provide a custom style using the parameter `sld`. We have provided a class that will pass that parameter to the WMS service. Just replace WMSLayer with SldWMSLayer. The `sld`parameter is given as a URL to an SLD file.

[15]:
```
url = f"http://xcube-geodb.brockmann-consult.de/geoserver/{my_database_name}/wms?"
```

[16]:
```
from ipyleaflet import Map, WMSLayer, basemaps
```

[17]:
```
from traitlets import Unicode

class SldWMSLayer(WMSLayer):

    sld = Unicode('').tag(sync=True, o=True)
```

[17]:
```
wmsn = WMSLayer(
    url=url,
    layers=f'{my_database_name}:{my_database_name}_SI001L2_LJUBLJANA_UA2018',
    format='image/png',
    opacity=0.5,
```

(continues on next page)

(continued from previous page)

```
    attribution='Copernicus',
    styles='ljubilana'
#     sld='https://raw.githubusercontent.com/dzelge/test-gha/main/ljubilana.sld'
)
```

[18]:
```
m = Map(basemap=basemaps.CartoDB.Positron, center=(46.0, 14.5), zoom=13)
m.add_layer(wmsn)

m
```

```
Map(center=[46.0, 14.5], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪'zoom_in_title', 'zoom_out...
```

[19]:
```
geodb.unpublish_gs(collection='SI001L2_LJUBLJANA_UA2018', database=my_database_name)
```

[19]:
```
True
```

[ ]:
```
geodb.drop_collection('SI001L2_LJUBLJANA_UA2018', database=my_database_name)
```

CHAPTER

SIX

PYTHON CLIENT

6.1 GeoDBClient

class xcube_geodb.core.geodb.**GeoDBClient**(*server_url: Optional[str] = None*, *server_port: Optional[int] = None*, *client_id: Optional[str] = None*, *client_secret: Optional[str] = None*, *username: Optional[str] = None*, *password: Optional[str] = None*, *access_token: Optional[str] = None*, *dotenv_file: str = '.env'*, *auth_mode: Optional[str] = None*, *auth_aud: Optional[str] = None*, *config_file: str = '/home/docs/.geodb'*, *database: Optional[str] = None*, *access_token_uri: Optional[str] = None*, *gs_server_url: Optional[str] = None*, *gs_server_port: Optional[int] = None*, *raise_it: bool = True*)

Constructing the geoDB client. Depending on the setup it will automatically setup credentials from environment variables. The user can also pass credentials into the constructor.

Parameters

- **server_url** (*str*) – The URL of the PostGrest Rest API service
- **server_port** (*str*) – The port to the PostGrest Rest API service
- **dotenv_file** (*str*) – Name of the dotenv file [.env] to set client IDs and secrets
- **client_secret** (*str*) – Client secret (overrides environment variables)
- **client_id** (*str*) – Client ID (overrides environment variables)
- **auth_mode** (*str*) – Authentication mode [silent]. Can be the oauth2 modes 'client-credentials', 'password',
- **and 'none' for no authentication** (*'interactive'*) –
- **auth_aud** (*str*) – Authentication audience
- **config_file** (*str*) – Filename that stores config info for the geodb client

Raises

- **GeoDBError** – if the auth mode does not exist
- **NotImplementedError** – on auth mode interactive

Examples

```
>>> geodb = GeoDBClient(auth_mode='client-credentials', client_id='***', client_
↪secret='***')
>>> geodb.whoami
my_user
```

get_collection_info(*collection: str*, *database: Optional[str] = None*) → Dict

Parameters

- **collection** (*str*) – The name of the collection to inspect
- **database** (*str*) – The database the collection resides in [current
- **database]** –

Returns A dictionary with collection information

Raises **GeoDBError** – When the collection does not exist

Examples

```
>>> geodb = GeoDBClient(auth_mode='client-credentials', client_id='***', client_
↪secret='***')
>>> geodb.get_collection_info('my_collection')
{
    'required': ['id', 'geometry'],
    'properties': {
    'id': {
        'format': 'integer', 'type': 'integer',
        'description': 'Note:This is a Primary Key.'
    },
    'created_at': {'format': 'timestamp with time zone', 'type': 'string'},
    'modified_at': {'format': 'timestamp with time zone', 'type': 'string'},
    'geometry': {'format': 'public.geometry(Geometry,3794)', 'type': 'string'},
    'my_property1': {'format': 'double precision', 'type': 'number'},
    'my_property2': {'format': 'double precision', 'type': 'number'},
    'type': 'object'
}
```

get_collection_bbox(*collection: str*, *database: Optional[str] = None*, *exact: Optional[bool] = False*) → Union[None, Sequence]

Retrieves the bounding box for the collection, i.e. the union of all rows' geometries.

Parameters

- **collection** (*str*) – The name of the collection to return the bounding box for.
- **database** (*str*) – The database the collection resides in. Default: current database.
- **exact** (*bool*) – If the exact bbox shall be computed. Warning: This may take much longer. Default: False.

Returns the bounding box given as tuple xmin, ymin, xmax, ymax or None if collection is empty

Examples

```
>>> geodb = GeoDBClient(auth_mode='client-credentials',
↪            client_id='***',
                        client_secret='***')
>>> geodb.get_collection_bbox('my_collection')
(-5, 10, 5, 11)
```

get_my_collections(*database: Optional[str] = None*) → Sequence

Parameters **database** (*str*) – The database to list collections from

Returns A Dataframe of collection names

Examples

```
>>> geodb = GeoDBClient(auth_mode='client-credentials', client_id='***', client_
↪secret='***')
>>> geodb.get_my_collections()
    owner                           database                          
↪collection
0   geodb_9bfgsdfg-453f-445b-a459   geodb_9bfgsdfg-453f-445b-a459   land_use
```

property raise_it: bool
Returns: The current error message behaviour

property database: str
Returns: The current database

property whoami: str
Returns: The current database user

property capabilities: Dict
Returns: A dictionary of the geoDB PostGrest REST API service's capabilities

refresh_config_from_env(*dotenv_file: str = '.env'*, *use_dotenv: bool = False*)
Refresh the configuration from environment variables. The variables can be preset by a dotenv file. :param dotenv_file: A dotenv config file :type dotenv_file: str :param use_dotenv: Whether to use GEODB_AUTH_CLIENT_ID a dotenv file. :type use_dotenv: bool

get_my_usage(*pretty=True*) → Union[Dict, xcube_geodb.core.message.Message]
Get my geoDB data usage.

Parameters **pretty** (*bool*) – Whether to return in human readable form or in bytes

Returns A dict containing the usage in bytes (int) or as a human readable string

Example

```
>>> geodb = GeoDBClient()
>>> geodb.get_my_usage(True)
{'usage': '6432 kB'}
```

create_collection_if_not_exists(*collection: str*, *properties: Dict*, *crs: Union[int, str] = 4326*, *database: Optional[str] = None*, ***kwargs*) → Union[Dict, xcube_geodb.core.message.Message]

Creates a collection only if the collection does not exist already.

Parameters

- **collection** (*str*) – The name of the collection to be created
- **properties** (*Dict*) – Properties to be added to the collection
- **crs** (*int, str*) – projection
- **database** (*str*) – The database the collection is to be created in [current database]
- **kwargs** – Placeholder for deprecated parameters

Returns Collection info id operation succeeds None: If operation fails

Return type Collection

Examples

See create_collection for an example

create_collections_if_not_exist(*collections: Dict*, *database: Optional[str] = None*, ***kwargs*) → Dict

Creates collections only if collections do not exist already.

Parameters

- **collections** (*Dict*) – The name of the collection to be created
- **database** (*str*) – The database the collection is to be created in [current database]
- **kwargs** – Placeholder for deprecated parameters

Returns List of informations about created collections

Return type List of Collections

Examples

See create_collections for examples

create_collections(*collections: Dict*, *database: Optional[str] = None*, *clear: bool = False*) → Union[Dict, xcube_geodb.core.message.Message]

Create collections from a dictionary :param clear: Delete collections prioer to creation :type clear: bool :param collections: A dictionalry of collections :type collections: Dict :param database: Database to use for creating the collection :type database: str

Returns Success

Return type bool

Examples

```
>>> geodb = GeoDBClient()
>>> collections = {'[MyCollection]': {'crs': 1234, 'properties':
↪     {'[MyProp1]': 'float', '[MyProp2]': 'date'}}}
>>> geodb.create_collections(collections)
```

create_collection(*collection: str*, *properties: Dict*, *crs: Union[int, str] = 4326*, *database: Optional[str] = None*, *clear: bool = False*) → Dict

Create collections from a dictionary

Parameters

- **collection** (*str*) – Name of the collection to be created
- **clear** (*bool*) – Whether to delete existing collections
- **properties** (*Dict*) – Property definitions for the collection
- **database** (*str*) – Database to use for creating the collection
- **crs** – sfdv

Returns Success

Return type bool

Examples

```
>>> geodb = GeoDBClient()
>>> properties = {'[MyProp1]': 'float', '[MyProp2]': 'date'}
>>> geodb.create_collection(collection='[MyCollection]', crs=3794,
↪properties=properties)
```

drop_collection(*collection: str*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Parameters

- **collection** (*str*) – Name of the collection to be dropped
- **database** (*str*) – The database the colections resides in [current database]

Returns Success

Return type bool

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.drop_collection(collection='[MyCollection]')
```

drop_collections(*collections: Sequence[str]*, *cascade: bool = False*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Parameters

- **database** (*str*) – The database the colections resides in [current database]

- **collections** (*Sequence[str]*) – Collections to be dropped
- **cascade** (*bool*) – Drop in cascade mode. This can be necessary if e.g. sequences have not been deleted properly

Returns Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.drop_collections(collections=['[MyCollection1]', '[MyCollection2]'])
```

grant_access_to_collection(*collection: str*, *usr: str*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Parameters

- **collection** (*str*) – Collection name to grant access to
- **usr** (*str*) – Username to grant access to
- **database** (*str*) – The database the collection resides in

Returns Success

Return type bool

Raises **HttpError** – when http request fails

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.grant_access_to_collection('[Collection]', '[User who gets access]')
Access granted on Collection to User who gets access}
```

rename_collection(*collection: str*, *new_name: str*, *database: Optional[str] = None*)

Parameters

- **collection** (*str*) – The name of the collection to be renamed
- **new_name** (*str*) – The new name of the collection
- **database** (*str*) – The database the collection resides in

Raises **HttpError** – When request fails

move_collection(*collection: str*, *new_database: str*, *database: Optional[str] = None*)

Move a collection from one database to another.

Parameters

- **collection** (*str*) – The name of the collection to be moved
- **new_database** (*str*) – The database the collection will be moved to
- **database** (*str*) – The database the collection resides in

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.move_collection('collection', 'new_database')
```

copy_collection(*collection: str*, *new_collection: str*, *new_database: str*, *database: Optional[str] = None*)

Parameters

- **collection** (*str*) – The name of the collection to be copied
- **new_collection** (*str*) – The new name of the collection
- **database** (*str*) – The database the collection resides in [current database]
- **new_database** (*str*) – The database the collection will be copied to

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.copy_collection('col', 'col_new', 'target_db',
                          'source_db')
```

publish_collection(*collection: str*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Publish a collection. The collection will be accessible by all users in the geoDB.

Parameters

- **database** (*str*) – The database the collection resides in [current database]
- **collection** (*str*) – The name of the collection that will be made public

Returns Message whether operation succeeded

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.publish_collection('[Collection]')
```

unpublish_collection(*collection: str*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Revoke public access to a collection. The collection will not be accessible by all users in the geoDB.

Parameters

- **database** (*str*) – The database the collection resides in [current database]
- **collection** (*str*) – The name of the collection that will be removed from public access

Returns Message whether operation succeeded

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.unpublish_collection('[Collection]')
```

revoke_access_from_collection(*collection: str*, *usr: str*, *database: Optional[str] = None*, ***kwargs*) → xcube_geodb.core.message.Message

Revoke access from a collection.

Parameters

- **collection** (*str*) – Name of the collection
- **usr** (*str*) – User to revoke access from
- **database** (*str*) – The database the collection resides in [current database]

Returns Whether operation has succeeded

Return type Message

list_my_grants() → Union[pandas.DataFrame, xcube_geodb.core.message.Message]

List the access grants the current user has granted.

Returns A list of the current user's access grants

Return type DataFrame

Raises **GeoDBError** – If access to geoDB fails

add_property(*collection: str*, *prop: str*, *typ: str*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Add a property to an existing collection.

Parameters

- **collection** (*str*) – The name of the collection to add a property to
- **prop** (*str*) – Property name
- **typ** (*str*) – The data type of the property (Postgres type)
- **database** (*str*) – The database the collection resides in [current database]

Returns Success message

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.add_property(collection='[MyCollection]',                name=
↪'[MyProperty]', type='[PostgresType]')
```

add_properties(*collection: str*, *properties: Dict*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Add properties to a collection.

Parameters

- **collection** (*str*) – The name of the collection to add properties to
- **properties** (*Dict*) – Property definitions as dictionary

- **database** (*str*) – The database the collection resides in [current database]

Returns Whether the operation succeeded

Return type Message

Examples

```
>>> properties = {'[MyName1]': '[PostgresType1]',
↪'[MyName2]': '[PostgresType2]'}
>>> geodb = GeoDBClient()
>>> geodb.add_property(collection='[MyCollection]',
↪       properties=properties)
```

drop_property(*collection: str*, *prop: str*, *database: Optional[str] = None*, ***kwargs*) → xcube_geodb.core.message.Message

Drop a property from a collection.

Parameters

- **collection** (*str*) – The name of the collection to drop the property from
- **prop** (*str*) – The property to delete
- **database** (*str*) – The database the collection resides in [current database]

Returns Whether the operation succeeded

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.drop_property(collection='[MyCollection]',
↪         prop='[MyProperty]')
```

drop_properties(*collection: str*, *properties: Sequence[str]*, *database: Optional[str] = None*) → xcube_geodb.core.message.Message

Drop properties from a collection.

Parameters

- **collection** (*str*) – The name of the collection to delete properties from
- **properties** (*List*) – A list containing the property names
- **database** (*str*) – The database the collection resides in [current database]

Returns Whether the operation succeeded

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.drop_properties(collection='MyCollection',
↪           properties=['MyProperty1',
↪          'MyProperty2'])
```

get_properties(*collection: str*, *database: Optional[str] = None*, ***kwargs*) → pandas.DataFrame
Get a list of properties of a collection.

Parameters

- **collection** (*str*) – The name of the collection to retrieve a list of properties from
- **database** (*str*) – The database the collection resides in [current database]

Returns A list of properties

Return type DataFrame

create_database(*database: str*) → xcube_geodb.core.message.Message
Create a database.

Parameters **database** (*str*) – The name of the database to be created

Returns A message about the success or failure of the operation

Return type Message

truncate_database(*database: str*) → xcube_geodb.core.message.Message
Delete all tables in the given database.

Parameters **database** (*str*) – The name of the database to be created

Returns A message about the success or failure of the operation

Return type Message

get_my_databases() → pandas.DataFrame
Get a list of databases the current user owns.

Returns A list of databases the user owns

Return type DataFrame

database_exists(*database: str*) → bool
Checks whether a database exists.

Parameters **database** (*str*) – The name of the database to be checked

Returns database exists

Return type bool

Raises **HttpError** – If request fails

delete_from_collection(*collection: str*, *query: str*, *database: Optional[str] = None*) →
xcube_geodb.core.message.Message
Delete rows from collection.

Parameters

- **collection** (*str*) – The name of the collection to delete rows from
- **database** (*str*) – The name of the database to be checked
- **query** (*str*) – Filter which records to delete. Follow the

- **http** – //postgrest.org/en/v6.0/api.html query convention.

Returns Whether the operation has succeeded

Return type Message

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.delete_from_collection('[MyCollection]', 'id=eq.1')
```

update_collection(*collection: str*, *values: Dict*, *query: str*, *database: Optional[str] = None*, ***kwargs*) → xcube_geodb.core.message.Message

Update data in a collection by a query.

Parameters

- **collection** (*str*) – The name of the collection to be updated
- **database** (*str*) – The name of the database to be checked
- **values** (*Dict*) – Values to update
- **query** (*str*) – Filter which values to be updated. Follow the http://postgrest.org/en/v6.0/api.html query
- **convention.** –

Returns Success

Return type Message

Raises **GeoDBError** – if the values is not a Dict or request fails

Example:

insert_into_collection(*collection: str*, *values: geopandas.GeoDataFrame*, *upsert: bool = False*, *crs: Optional[Union[int, str]] = None*, *database: Optional[str] = None*, *max_transfer_chunk_size: int = 1000*) → xcube_geodb.core.message.Message

Insert data into a collection.

Parameters

- **collection** (*str*) – Collection to be inserted to
- **database** (*str*) – The name of the database the collection resides in [current database]
- **values** (*GeoDataFrame*) – Values to be inserted
- **upsert** (*bool*) – Whether the insert shall replace existing rows (by PK)
- **crs** (*int, str*) – crs (in the form of an SRID) of the geometries. If not present, the method will attempt guessing it from the GeoDataFrame input. Must be in sync with the target collection in the GeoDatabase
- **max_transfer_chunk_size** (*int*) – Maximum number of rows per chunk to be sent to the geodb.

Raises

- **ValueError** – When crs is not given and cannot be guessed from the GeoDataFrame
- **GeoDBError** – If the values are not in format Dict

Returns Success

Return type bool

Example:

static transform_bbox_crs(*bbox: Tuple[float, float, float, float]*, *from_crs: Union[int, str]*, *to_crs: Union[int, str]*, *wsg84_order: str = 'lat_lon'*)

This function can be used to reproject bboxes particularly with the use of GeoDBClient.get_collection_by_bbox.

Parameters

- **bbox** – Tuple[float, float, float, float]: bbox to be reprojected, given as MINX, MINY, MAXX, MAXY
- **from_crs** – Source crs e.g. 3974
- **to_crs** – Target crs e.g. 4326
- **wsg84_order** (*str*) – WSG84 (EPSG:4326) is expected to be in Lat Lon format ("lat_lon"). Use "lon_lat" if Lon Lat is used.

Returns The reprojected bounding box

Return type Tuple[float, float, float, float]

Examples

```
>>> bbox = GeoDBClient.transform_bbox_crs(bbox=(450000, 100000, 470000, 110000),
↪ from_crs=3794, to_crs=4326)
>>> bbox
(49.36588643725233, 46.012889756941775, 14.311548793848758, 9.834303086688251)
```

get_collection_by_bbox(*collection: str*, *bbox: Tuple[float, float, float, float]*, *comparison_mode: str = 'contains'*, *bbox_crs: Union[int, str] = 4326*, *limit: int = 0*, *offset: int = 0*, *where: Optional[str] = 'id>-1'*, *op: str = 'AND'*, *database: Optional[str] = None*, *wsg84_order='lat_lon'*, ***kwargs*) → Union[geopandas.GeoDataFrame, pandas.DataFrame]

Query the database by a bounding box. Please be careful with the bbox crs. The easiest is using the same crs as the collection. However, if the bbox crs differs from the collection, the geoDB client will attempt to automatially transform the bbox crs according to the collection's crs. You can also directly use the method GeoDBClient.transform_bbox_crs yourself before you pass the bbox into this method.

Parameters

- **collection** (*str*) – The name of the collection to be quried
- **bbox** (*Tuple[float, float, float, float]*) – minx, miny, maxx, maxy
- **comparison_mode** (*str*) – Filter mode. Can be 'contains' or 'within' ['contains']
- **bbox_crs** (*int, str*) – Projection code. [4326]
- **op** (*str*) – Operator for where (AND, OR) ['AND']
- **where** (*str*) – Additional SQL where statement to further filter the collection
- **limit** (*int*) – The maximum number of rows to be returned
- **offset** (*int*) – Offset (start) of rows to return. Used in combination with limit.
- **database** (*str*) – The name of the database the collection resides in [current database]

- **wsg84_order** (*str*) – WSG84 (EPSG:4326) is expected to be in Lat Lon format ("lat_lon"). Use "lon_lat" if
- **Lat is used.** (*Lon*) –

Returns A GeoPandas Dataframe

Raises **HttpError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.get_collection_by_bbox(table="[MyCollection]", bbox=(452750.0, 88909.
↪549, 464000.0,                  102486.299), comparison_mode="contains", bbox_
↪crs=3794, limit=10, offset=10)
```

count_collection_rows(*collection: str*, *database: Optional[str] = None*, *exact_count: Optional[bool] = False*) → Union[int, xcube_geodb.core.message.Message]

Return the number of rows in the given collection. By default, this function returns a rough estimate within the order of magnitude of the actual number; the exact count can also be retrieved, but this may take much longer. Note: in some cases, no estimate can be provided. In such cases, -1 is returned if exact_count == False.

Parameters

- **collection** (*str*) – The name of the collection
- **database** (*str*) – The name of the database the collection resides in [current database]
- **exact_count** (*bool*) – If True, the actual number of rows will be counted. Default value: false.

Returns the number of rows in the given collection, or -1 if exact_count is False and no estimate could be provided.

Raises **GeoDBError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.count_collection_rows('my_collection'], exact_count=True)
```

count_collection_by_bbox(*collection: str*, *bbox: Tuple[float, float, float, float]*, *comparison_mode: str = 'contains'*, *bbox_crs: Union[int, str] = 4326*, *where: Optional[str] = 'id>-1'*, *op: str = 'AND'*, *database: Optional[str] = None*, *wsg84_order='lat_lon'*) → Union[geopandas.GeoDataFrame, pandas.DataFrame]

Query the database by a bounding box and return the count. Please be careful with the bbox crs. The easiest is using the same crs as the collection. However, if the bbox crs differs from the collection, the geoDB client will attempt to automatially transform the bbox crs according to the collection's crs. You can also directly use the method GeoDBClient.transform_bbox_crs yourself before you pass the bbox into this method.

Parameters

- **collection** (*str*) – The name of the collection to be quried
- **bbox** (*Tuple[float, float, float, float]*) – minx, miny, maxx, maxy
- **comparison_mode** (*str*) – Filter mode. Can be 'contains' or 'within' ['contains']

- **bbox_crs** (*int, str*) – Projection code. [4326]
- **op** (*str*) – Operator for where (AND, OR) ['AND']
- **where** (*str*) – Additional SQL where statement to further filter the collection
- **database** (*str*) – The name of the database the collection resides in [current database]
- **wsg84_order** (*str*) – WSG84 (EPSG:4326) is expected to be in Lat Lon format ("lat_lon"). Use "lon_lat" if
- **Lat is used.** (*Lon*) –

Returns A GeoPandas Dataframe

Raises **HttpError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.count_collection_by_bbox(table="[MyCollection]", bbox=(452750.0,␣
↪88909.549, 464000.0,                 102486.299), comparison_mode="contains",␣
↪bbox_crs=3794)
```

head_collection(*collection: str*, *num_lines: int = 10*, *database: Optional[str] = None*) → Union[geopandas.GeoDataFrame, pandas.DataFrame]

Get the first num_lines of a collection.

Parameters

- **collection** (*str*) – The collection's name
- **num_lines** (*int*) – The number of line to return
- **database** (*str*) – The name of the database the collection resides in [current database]

Returns results

Return type GeoDataFrame or DataFrame

Raises **HttpError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.head_collection(collection='[MyCollection]', num_lines=10)
```

get_collection(*collection: str*, *query: Optional[str] = None*, *database: Optional[str] = None*, *limit: Optional[int] = None*, *offset: int = 0*) → Union[geopandas.GeoDataFrame, pandas.DataFrame]

Query a collection.

Parameters

- **collection** (*str*) – The collection's name.
- **query** (*str*) – A query. Follow the http://postgrest.org/en/v6.0/api.html query convention.
- **database** (*str*) – The name of the database the collection resides in [current database].
- **limit** (*int*) – The maximum number of rows to be returned.

- **offset** (*int*) – Offset (start) of rows to return. Used in combination with limit.

Returns results

Return type GeoDataFrame or DataFrame

Raises **HttpError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> geodb.get_collection(collection='[MyCollection]', query='id=ge.1000')
```

get_collection_pg(*collection: str*, *select: str = '*'*, *where: Optional[str] = None*, *group: Optional[str] = None*, *order: Optional[str] = None*, *limit: Optional[int] = None*, *offset: Optional[int] = None*, *database: Optional[str] = None*) → Union[geopandas.GeoDataFrame, pandas.DataFrame]

Parameters

- **collection** (*str*) – The name of the collection to query
- **select** (*str*) – Properties (columns) to return. Can contain aggregation functions
- **where** (*Optional[str]*) – SQL WHERE statement
- **group** (*Optional[str]*) – SQL GROUP statement
- **order** (*Optional[str]*) – SQL ORDER statement
- **limit** (*Optional[int]*) – Limit for paging
- **offset** (*Optional[int]*) – Offset (start) of rows to return. Used in combination with limit.
- **database** (*str*) – The name of the database the collection resides in [current database]

Returns results

Return type GeoDataFrame or DataFrame

Raises **HttpError** – When the database raises an error

Examples

```
>>> geodb = GeoDBClient()
>>> df = geodb.get_collection_pg(collection='[MyCollection]', where='raba_
↪id=1410', group='d_od',                 select='COUNT(d_od) as ct, d_od')
```

property server_url: str

Get URL of the geoDB server.

Returns The URL of the geoDB REST service

Return type str

get_collection_srid(*collection: str*, *database: Optional[str] = None*) → Union[str, None, xcube_geodb.core.message.Message]

Get the SRID of a collection.

Parameters

- **collection** (*str*) – The collection's name
- **database** (*str*) – The name of the database the collection resides in [current database]

Returns The name of the SRID

publish_gs(*collection: str*, *database: Optional[str] = None*)
Publishes collection to a BC geoservice (geoserver instance). Requires access registration.

Parameters

- **collection** (*str*) – Name of the collection
- **database** (*Optional[str]*) – Name of the database. Defaults to user database

Returns Dict

get_all_published_gs(*database: Optional[str] = None*) → Union[Sequence, xcube_geodb.core.message.Message]

Parameters **database** (*str*) – The database to list collections from a database which are published via geoserver

Returns A Dataframe of collection names

get_published_gs(*database: Optional[str] = None*) → Union[Sequence, xcube_geodb.core.message.Message]

Parameters **database** (*str*) – The database to list collections from a database which are published via geoserver

Returns A Dataframe of collection names

Examples

```
>>> geodb = GeoDBClient(auth_mode='client-credentials', client_id='***', client_
↪secret='***')
>>> geodb.get_published_gs()
    owner                          database                          ␣
↪collection
0   geodb_9bfgsdfg-453f-445b-a459   geodb_9bfgsdfg-453f-445b-a459   land_use
```

unpublish_gs(*collection: str*, *database: str*)
'Unpublishes' collection from a BC geoservice (geoserver instance). Requires access registration.

Parameters

- **collection** (*str*) – Name of the collection
- **database** (*Optional[str]*) – Name of the database. Defaults to user database

Returns Dict

property auth_access_token: str
Get the user's access token.

Returns The current authentication access_token

Raises **GeoDBError on missing ipython shell** –

refresh_auth_access_token()
Refresh the authentication token.

collection_exists(*collection: str*, *database: str*) → bool
Checks whether a collection exists

Parameters

- **collection** (*str*) – The collection's name
- **database** (*str*) – The name of the database the collection resides in [current database]

Returns Whether the collection exists

get_event_log(*collection: Optional[str] = None*, *database: Optional[str] = None*, *event_type: Optional[xcube_geodb.core.geodb.EventType] = None*) → pandas.DataFrame

Parameters

- **collection** (*str*) – The name of the collection for which to get the event log; if None, all collections are returned
- **database** (*str*) – The database of the collection
- **event_type** (*EventType*) – The type of the events for which to get the event log; if None, all events are returned

Returns Whether the stored procedure exists

Raises **GeoDBError if the stored procedure does not exist** –

static setup(*host: Optional[str] = None*, *port: Optional[str] = None*, *user: Optional[str] = None*, *passwd: Optional[str] = None*, *dbname: Optional[str] = None*, *conn: Optional[any] = None*)
Sets up the database. Needs DB credentials and the database user requires CREATE TABLE/FUNCTION grants.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX