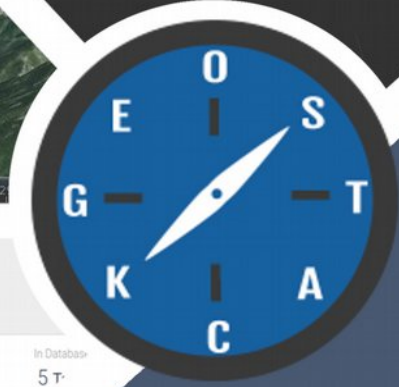
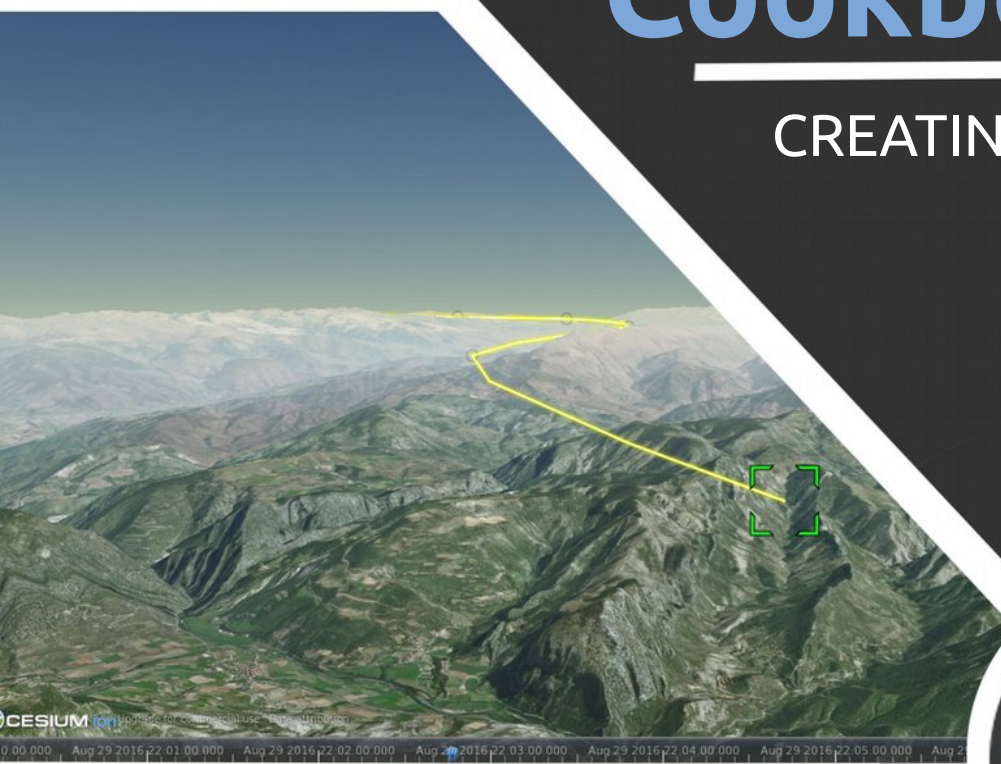


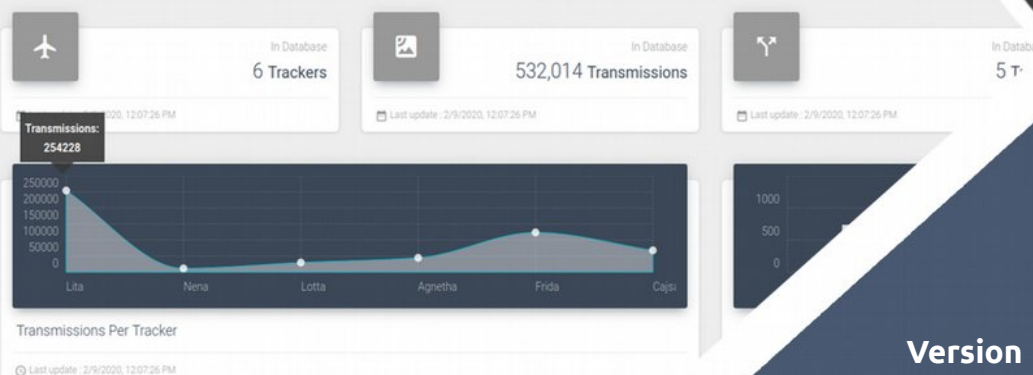


# Cookbook

## CREATING A PYTHON-FLASK WEB APPLICATION



### GPS Dashboard



Categories: <span>TRACKERS</span> <span>TRAILS</span>			
MongoID	Local Identifier	Crane name	Study name
5e3ec20e146786f4f6917b85	9407	Agnetha	GPS
5e3ec2c5146786f4f6940d1a	9472	Cajsa	
5e3ec23c146786f4f692297c	9381	Frida	

Version : 1.0

Date : 09-10-2020

Author : The GeoStack Project

License : CC BY 4.0

# Open Content License

This work is licensed under the Creative Commons Attribution 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

## Purpose of this document

This programming manual serves as an extension for the following documents:

### 1) Cookbook: Creating the GeoStack Course VM.

The datastores we will be using during this programming manual are set up during the cookbook: Creating the GeoStack Course VM.

### 2) Cookbook: Data modeling in MongoDB using MongoEngine.

The data used during this programming manual is modeled, imported and indexed in the cookbook : Data modeling in MongoDB using MongoEngine.

### 3) Cookbook: A Secure NGINX Webserver with Modsecurity.

The secure Webserver and the base of the flask application are created during this cookbook. We will be using these products during this programming manual.

If you have not read these documents yet, please do so before reading this document.

The purpose of this programming manual is to extend the base Python-Flask application created during the cookbook : A Secure NGINX Webserver with Modsecurity.

The Python-Flask application is going to serve as API between our MongoDB datastores (Crane Tracker database and the Trail Database) the PostgreSQL datastore (World Port Index database) and or web applications. The API contains the connection to these datastores and the queries which have to be performed on them.

The Python-Flask application also contains the link to our Tilestache server index.html file which is going to contain the entries in our Tilestache configuration file. We have not created the Tilestache Tileservers yet! This will be done later on in the Course.

The web applications are going to perform API calls to our Python-Flask application after which the API retrieve the requested data by executing queries on the datastore in question. The results of these queries are then returned to our Frontend which are our web applications.

Our flask application will be running behind the secure NGINX webserver.

Again, If you have not created the webserver or the Python-Flask application yet please read the cookbook: "A Secure NGINX Webserver with Modsecurity". This cookbook gives instructions on how to setup the base of our Python-Flask application which we are going to extend during the programming manual.

Note: During this programming manual you will come across a lot of images containing the code which you have to create. In the folder POC, which is located in the same folder as this document, you can find the complete source code which is created during this manual. You can use this in case you get stuck!

# Table of Contents

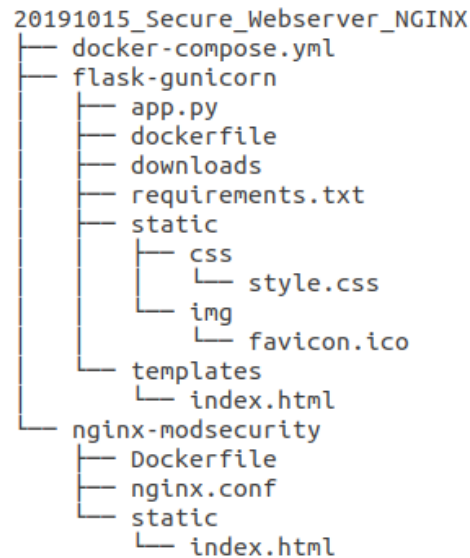
- 1 Small Recap of our Python-Flask base application..... 5
- 2 Updating the requirements.txt file and modules..... 6
- 3 Creating and referencing the configuration file..... 8
- 4 Creating our MongoDB Queries..... 10
- 5 Overview of Geospatial Query options in MongoDB..... 21
- 6 Creating data profiles..... 22
- 7 Connecting to a PostgreSQL database..... 25
- 8 Query a PostgreSQL database..... 27
- 9 Retrieving TileStache configuration entries..... 30
- 10 Adding the GPS Route (Trail) queries..... 32
- 11 Add the Flask API to our LOCAL NGINX config!..... 35
- 12 Dockerizing the Flask API..... 37

# Table of Figures

Illustration 1: Folder and File Structure for a Flask base app.....	5
Illustration 2: Requirements.txt.....	6
Illustration 3: Import flask_pymongo.....	6
Illustration 4: Import psycpg2.....	7
Illustration 5: Import bson.....	7
Illustration 6: Import pandas and pandas_profiling.....	7
Illustration 7: Import urllib, json bs4 and datetime modules.....	7
Illustration 8: Adding the Crane DB URI's.....	8
Illustration 9: Terminal output flask.....	11
Illustration 10: All trackers in database.....	11
Illustration 11: Query for searching tracker by ID.....	12
Illustration 12: values to query on.....	12
Illustration 13: Part of a result returned by query.....	13
Illustration 14: Query for retrieving total transmission count.....	13
Illustration 15: Query all transmissions from a given tracker.....	14
Illustration 16: Query a given amount of transmissions.....	14
Illustration 17: Query all transmission between given DTG.....	15
Illustration 18: Strip DateTime.....	15
Illustration 19: Query transmissions in given polygon.....	17
Illustration 20: Angular log.....	18
Illustration 21: Geospatial operators.....	21
Illustration 22: Part of generated profile.....	22
Illustration 23: Generic function for data profile creation.....	23
Illustration 24: Function for generating data profile.....	24
Illustration 25: Creating a generic function for querying PostgreSQL.....	25
Illustration 26: Connecting and querying a database.....	26
Illustration 27: Creating the WPI Connection data.....	27
Illustration 28: Obtaining the Port Counts.....	28
Illustration 29: Obtaining the port count.....	29
Illustration 30: Function for obtaining the Tilestache Entries.....	30
Illustration 31: Scraping the index.html.....	31
Illustration 32: Obtaining the test in the <p> tags.....	31

# 1 Small Recap of our Python-Flask base application

As mentioned at the beginning of this document, this document is an extension on the flask base application we created in the cookbook : A Secure NGINX Webserver with Modsecurity. So let's start of with a small recap of that cookbook. We ended up with a file structure shown in illustration 1.



*Illustration 1: Folder and File Structure for a Flask base app*

We also had 2 entries in our requirements.txt file: a flask entry and a gunicorn entry.

During this programming manual we are going to do extend our application by doing the following:

- ➔ Extend our requirements.txt file;
- ➔ Add the modules we need to import;
- ➔ Add a configuration file which is going to contain all the values and connection strings required to connect to our databases;
- ➔ Add a MongoDB connection and queries;
- ➔ Add a PostgreSQL connection and queries;
- ➔ Add the link to our Tilestache server.

First we need to copy the flask-gunicorn folder, which was created in the cookbook: "A secure NGINX webserver with Modsecurity", to the Geostack root folder.

This is done by opening a terminal and entering the command: "cp -r {path to flask folder} ~/Geostack/".

Where "{path to flask folder}" is the location of the flask-gunicorn folder.

In this case the command would be:

```
cp -r ~/20191017_Secure_Webserver_NGINX/flask-gunicorn ~/Geostack/
```

## 2 Updating the requirements.txt file and modules

To be able to connect to our MongoDB database and our PostgreSQL database we need some extra modules. These modules are as follows:

→ **flask\_pymongo:**

This module is used to connect to our MongoDB database(s) and to perform queries on it.

→ **Psycopg2:**

This module is used to connect to our PostgreSQL database(s) and to perform queries on it.

Some extra modules we are going to need are as follows:

→ **bson:**

This module is used to obtain and query on the the ObjectID of MongoDB records. The module has some build in functions that convert a string to a valid ObjectID.

→ **Datetime:**

This module is used to convert timestamps to human readable date time format.

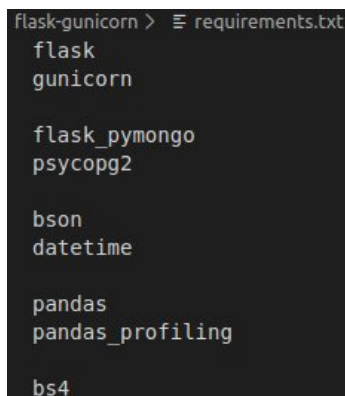
→ **Pandas and Pandas profiling:**

These packages are used to generate dataset profiles of the Crane datasets.

→ **bs4:**

bs4, also called: "BeautifulSoup", is a Python package used to scrape websites. Scraping means extracting all the content from an HTML page. We are going to use this package to scrape all our Tilestache configuration entries.

So let's add these modules to our requirements.txt file. The final file will look the same as shown in illustration.



```
flask-gunicorn > requirements.txt
flask
gunicorn

flask_pymongo
psycopg2

bson
datetime

pandas
pandas_profiling

bs4
```

Illustration 2: Requirements.txt

Next, we want to import some extra modules to our app.py file. We are going to start of with the flask\_pymongo module. How this is done is shown in the illustration below. A description of the module is also given in this illustration.



```
'''
The PyMongo module is used to create MongoDB queries and connections to
our MongoDB databases.
'''
from flask_pymongo import PyMongo
```

Illustration 3: Import flask\_pymongo



Next we want to import the psycopg2 package, this packages enables us to create connections and send queries to the PostgreSQL database. How this is done is shown in illustration 4.

```
'''
The psycopg2 package is used to create PostgreSQL queries and connections
to our PostgreSQL databases
'''
import psycopg2
```

*Illustration 4: Import psycopg2*

Now we want to import the BSON Modules How this is done, is shown in illustration 5.

```
'''
The ObjectId module is used to transform an Id passed by the Angular application
to a valid MongoDB id.
The json_util module is used to convert the BSON, which is returned by MongoDB,
into a valid JSON format
'''
from bson.objectid import ObjectId
from bson import json_util
```

*Illustration 5: Import bson*

The next modules we are going to import are the pandas and pandas\_profiling modules. How this is done is shown in illustration 6.

```
'''
The pandas and pandas_profiling packages are used to create data profiles of
the datasets in our MongoDB datastore
'''
import pandas as pd
import pandas_profiling
```

*Illustration 6: Import pandas and pandas\_profiling*

The last modules we want to import are the urllib, json bs4 and datetime modules. How this is done is shown in illustration 7.

```
'''
The urlopen and BeautifulSoup modules are used to parse the HTML page,
which contains our all tilestache configuration entries, from the NGINX Webserver.
The datetime module is used to convert datetime strings to a valid datetime format.
'''
from urllib.request import urlopen
from bs4 import BeautifulSoup
from datetime import datetime
import json
```

*Illustration 7: Import urllib, json bs4 and datetime modules*

### 3 Creating and referencing the configuration file.

When using databases it's always useful to create a configuration file. This configuration file contains all the connection strings, usernames, database names and passwords related to our database. The configuration file is going to contain the connection strings for both our Dockerized and our local instances of our GeoStack components.

The configuration file will be located in the root of our flask-gunicorn folder and will be called: "config.py".

So let's create this file by running the command:

```
touch ~/Geostack/flask-gunicorn/config.py
```

Let's start off by adding the configuration lines for our local and docker instances of our Crane database. We do this by adding the lines, shown in illustration 8, to this file.

```
# Here we define the Local URI of our Crane Database.
# Uncomment this line if you are going to run the MongoDB instance locally.
CRANE_DATABASE_URI = "mongodb://localhost:27017/Crane_Database"

# Here we define the Docker URI of our Crane Database.
# Uncomment this line if you are going to run the MongoDB instance in Docker.
#CRANE_DATABASE_URI = "mongodb://mongodb-datastore:27017/Crane_Database"
```

Illustration 8: Adding the Crane DB URI's

Now we want to do the same for the instances of our Trail database. We do this by adding the following lines to the configuration file

```
# Here we define the Local URI of our Trail Database.
# Uncomment this line if you are going to run the MongoDB instance locally.
TRAIL_DATABASE_URI = "mongodb://localhost:27017/Trail_Database"

# Here we define the Docker URI of our Trail Database.
# Uncomment this line if you are going to run the MongoDB instance in Docker.
#TRAIL_DATABASE_URI = "mongodb://mongodb-datastore:27017/Trail_Database"
```

Now we want to define the entries required to access the Tilestache Tileservers index.html file. We do this by adding the following lines to the configuration file.

```
# Here we define the Local URI of our Tilestache Server Index file.
# Uncomment this line if you are going to run the Tilestache Tileservers
# instance locally.
TILESTACHE_INDEX = 'http://localhost/tiles/'

# Here we define the Docker URI of our Tilestache Server Index file
# Uncomment this line if you are going to run the Tilestache Tileservers
# instance in Docker.

#TILESTACHE_INDEX = 'http://nginx-webserver/tiles/'
```



Now that we have defined the connection strings, we want to create a reference to our configuration file in our app.py file.

We are going to add this line of code underneath the line, which we already defined in the cookbook: Creating a secure NGINX webserver with Modsecurity”, shown in the illustration below.

```
app = Flask(__name__) # app is now an instance of the Flask class (= WSGI app!).
```

How to reference to the configuration file is shown in illustration. This line of code makes sure our Flask application knows where our connection strings are located.

```
'''
In the line below, the flask configuration file, called: "config.py", is assigned to
our Flask instance. This configuration file contains our database connectionstrings.
'''
app.config.from_pyfile('config.py')
```

Next we want to create a connection to our Crane database. How this is done, is shown in the illustration below.

```
# 3) Connect to the Crane Database
crane_connection= PyMongo(app, uri=app.config["CRANE_DATABASE_URI"])
```

Some explanation concerning the code above is as follows:

- First we want to create a variable called: “crane\_connection”
- Next we want to create a new instance of Pymongo
- As first parameter we pass the instance of our Flask application called: “app”
- As second parameter we want to pass the connection string of our local MongoDB database. This connection string is defined in the configuration file.

**NOTE: if you are going to build the Docker containers later on in the course, you have to make sure to change the connection strings located in our config.py file.**

- **This is done by removing the ‘#’ in front of the entries which represent our Docker container connection strings.**

**You have to this for the following entries:**

- ➔ **The Crane Database connection string**
- ➔ **The Trail Database connection string**
- ➔ **The Tilestache index URL**
- ➔ **The World-Port-Index Database connection string. (This connection string will be created later on in this document)**

## 4 Creating our MongoDB Queries

In this chapter we are going to create the queries to our Crane database. First is will explain how you should go about constructing a function that is accessible via an URL. In the illustration below you can see an example of a query which returns all trackers.

```
@app.route('/api/trackers/', methods=['GET'])
def get_all_trackers():
    query_result = crane_connection.db.tracker.find()
    return json.dumps(list(query_result), default=json_util.default)
```

To create a function which executes a query and then returns the data, you have to do the following:

- 1) Define the URL and request type to bind to the function (Red):

This is the URL on which the function: "get\_all\_trackers()", will be triggered. If you navigate to the URL: <http://localhost/api/trackers>, it will execute the code which is defined in the function: "get\_all\_trackers()". The request type is assigned to the variable: "methods". In our case this is going to be a GET request, which means the we are only going to retrieve data from the data store.

Other request types are as follows:

- ➔ **POST request:** This request type is used when you want to add data to a database
- ➔ **PUT request:** This request type is used when you want to update existing data in a database.

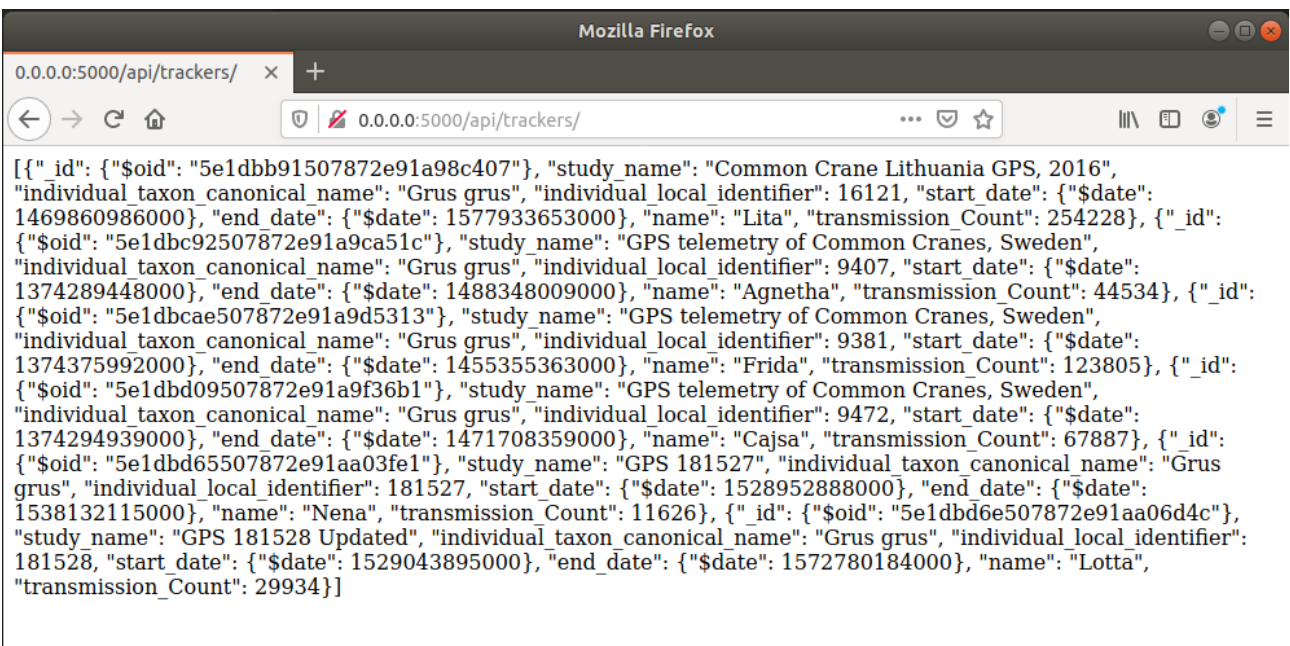
- 2) Define the name of the function (Blue). In this case the name of the function is: "get\_all\_trackers()".
- 3) Create a variable called query\_result (Green). This is the variable to which the data, returned by the MongoDB datastore, is assigned.
- 4) Create the query that should be executed (Yellow). To create a query you have to do the following:
  - ➔ Define which database connection you want to use (Purple). In this case we want to retrieve all the trackers, stored in our crane database. In chapter 2 we created a connection to the crane database and assigned it to a variable called: "crane\_connection", this is also the connection we want to use now.
  - ➔ Define which database collection you want to query on (Pink). In this case we want to retrieve all trackers in the tracker collection.
  - ➔ Define the values on which you want to query (light blue). Since, in this case, we want all trackers we don't need to define any specific values.
- 5) Create a return statement which dumps the MongoDB data results to a valid JSON format. Since the query\_result is still a Pymongo object, we need to convert it to a list (Brown). Then we need to transform it into valid JSON by passing the parameter: "default = json\_util.default" (Orange).

To validate whether the things we have created so far are correct, we are going to start the Flask application by entering the command: "python3 app.py" in the terminal. If everything is working accordingly the terminal should show the output seen in illustration 9.

```
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Illustration 9: Terminal output flask

Now make sure MongoDB is running. If MongoDB is running and you navigate to <http://localhost:5000/api/trackers> you should be greeted with the screen similar to the one shown in illustration 10.



```
[{"_id": {"$oid": "5e1dbb91507872e91a98c407"}, "study_name": "Common Crane Lithuania GPS, 2016", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 16121, "start_date": {"$date": "1469860986000"}, "end_date": {"$date": "1577933653000"}, "name": "Lita", "transmission_Count": 254228}, {"_id": {"$oid": "5e1dbc92507872e91a9ca51c"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9407, "start_date": {"$date": "1374289448000"}, "end_date": {"$date": "1488348009000"}, "name": "Agnetha", "transmission_Count": 44534}, {"_id": {"$oid": "5e1dbcae507872e91a9d5313"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9381, "start_date": {"$date": "1374375992000"}, "end_date": {"$date": "1455355363000"}, "name": "Frida", "transmission_Count": 123805}, {"_id": {"$oid": "5e1dbd09507872e91a9f36b1"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9472, "start_date": {"$date": "1374294939000"}, "end_date": {"$date": "1471708359000"}, "name": "Cajsa", "transmission_Count": 67887}, {"_id": {"$oid": "5e1dbd65507872e91aa03fe1"}, "study_name": "GPS 181527", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 181527, "start_date": {"$date": "1528952888000"}, "end_date": {"$date": "1538132115000"}, "name": "Nena", "transmission_Count": 11626}, {"_id": {"$oid": "5e1dbd6e507872e91aa06d4c"}, "study_name": "GPS 181528 Updated", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 181528, "start_date": {"$date": "1529043895000"}, "end_date": {"$date": "1572780184000"}, "name": "Lotta", "transmission_Count": 29934}]
```

Illustration 10: All trackers in database

Congratulations! You created your first MongoDB query which returns all the trackers in the database.

Next we are going to create the remaining queries. The queries we are going to create are as follows:

- ➔ A Query which returns one tracker by searching on it's ID.
- ➔ A Query which returns the total amount of transmissions in the database.
- ➔ A Query which returns all transmissions belonging to a certain tracker.
- ➔ A Query which returns a given amount of transmissions belonging to a certain tracker.
- ➔ A Query which returns all transmissions between a given DTG (Date time group), belonging to a certain tracker.
- ➔ A Query which returns all transmissions in a given polygon, belonging to a certain tracker.

We are going to start off by creating the function which contains a query that returns a tracker by matching a given ID. In illustration 11 you can see the code for creating that function

```
@app.route('/api/trackers/<id>' methods=['GET'])
def get_one_tracker(id):

    query_result = crane_connection.db.tracker.find({"_id": ObjectId(id)})

    return json.dumps(query_result, default=json_util.default)
```

Illustration 11: Query for searching tracker by ID

The steps required to create this function are similar to the first function we created:

- 1) First we define the URL and request type to bind to the function (Red):

As you can see in illustration 11, the URL which is bound to the function looks a bit different than the first one. In this URL we add <id>. The <id> will be replaced by the ObjectId from the tracker we want to retrieve from the datastore.

For example: Let's say we want to obtain the tracker from the crane: "Frida", which has an objectId that is equal to 1. The URL passed to our API, by our Angular application, will look like this: "<http://localhost/api/trackers/1>". After the URL is passed the function: "get\_one\_tracker(1)". Notice the 1 passed as parameter in the function. This one is obtained from the number representing the <id> in the URL we bound to the function.

- 2) For defining the query we do the same as we did in the first query we created. The only difference is that we need to pass values on which the query has to search (Blue). To define values on which the query should search, the following syntax is used: {dbconnection}.db.{collectionname}.find({"fieldname": value }).

In this syntax the {dbconnection} should be replaced with the database connection instance (crane\_connection in this case), {collectionname} should be replaced with the collection you want to perform the query on (tracker in this case), fieldname should be replaced with the name of the field, in our database, on which you want to search (\_id in this case), value should be replaced with the parameter passed in the function which is triggered (id in this case)

Since we are searching on the ID of the tracker, we need to define the name of the field which represents the trackerID in our Crane database. In illustration 12 we zoom in on the part of the query in which we define the values, on which the query should search. The name of the field, which represents the trackerID in our database is called: "\_id" (Orange). Then we convert the ID passed in the function: "get\_one\_tracker()", to a valid MongoDB ID by using the imported module: "ObjectId"(Green). As parameter we pass the ID.

```
.find({"_id": ObjectId(id)})
```

Illustration 12: values to query on

To validate whether we correctly created the query, we run the flask application by entering the command: "python3 app.py" in the terminal. First you want to perform the query that retrieves all the trackers in the database. This is done by navigating to the URL: "<http://localhost:5000/api/trackers/5e1dbb91507872e91a98c407>". When zooming in on one of the results it should look similar as the one shown in illustration 13.

```
{"_id": {"$oid": "5e1dbb91507872e91a98c407"}, "study_name": "Common Crane Lithuania GPS, 2016",
```

*Illustration 13: Part of a result returned by query*

Next, you want to copy the string behind the: "\$oid", in my case it's:

"5e1dbb91507872e91a98c407". This is the MongoDB of one of the trackers. Now you want to navigate to the URL:

"<http://localhost:5000/api/trackers/5e1dbb91507872e91a98c407>" (The ID is different in your case). This should return only one tracker.

Now we want to create the query which returns the total amount of transmissions in the database. In illustration 13, you can find the code to do this.

```
@app.route('/api/transmissions_count/', methods=['GET'])
def get_all_transmissions_count():

    query_result = crane_connection.db.transmission.count()

    return str(query_result)
```

*Illustration 14: Query for retrieving total transmission count*

The following applies to the code shown above:

- ➔ The URL on which this function will be triggered is :  
localhost/api/transmission\_count
- ➔ The name of the function is: "get\_all\_transmissions\_count"
- ➔ The query is performed on the transmission collection in the crane database
- ➔ We use the .count() function to perform a count on all the documents stored in the transmission collection.
- ➔ Since the query returns only one value, we don't need to convert it to a list. In this case we convert it to a string.

You can test whether the query is working or not by starting the Flask application and navigating to the URL: [http://localhost:5000/api/transmission\\_count](http://localhost:5000/api/transmission_count)



Next we want to create a query which retrieves all transmissions of a given tracker. How this is done is shown in illustration 15.

```
@app.route('/api/transmissions_by_id/<id>', methods=['GET'])
def get_all_transmissions_by_id(id):

    query_result = crane_connection.db.transmission.find({"tracker": ObjectId(id)})[:100]

    return json.dumps(list(query_result), default=json_util.default)
```

*Illustration 15: Query all transmissions from a given tracker*

The following applies to the code shown above:

- ➔ The URL on which this function will be triggered is :  
localhost/api/transmissions\_by\_id/{the id of the tracker}
- ➔ The name of the function is: "get\_all\_transmissions\_count"
- ➔ The query is performed on the transmission collection in the crane database
- ➔ The ID passed in the function is compared to the tracker ReferenceField in the transmissions documents.
- ➔ [:100] is used to only return the first 100 transmissions. When creating the Angular application, you will see why this is done.
- ➔ The Pymongo object is transformed in a list, which is then returned.

Next we want to create a query which retrieves a given amount of transmissions of a certain tracker. How this is done is shown in illustration 16.

```
@app.route('/api/transmissions_by_amount/<id>/<amount>', methods=['GET'])
def get_all_transmissions_amount(id,amount):

    query_result = crane_connection.db.transmission.find(
        {"tracker": ObjectId(id)})[:int(amount)]

    return json.dumps(list(query_result), default=json_util.default)
```

*Illustration 16: Query a given amount of transmissions*

The following applies to the code shown above:

- ➔ The URL on which this function will be triggered is :  
localhost/api/transmissions\_by\_amount/{the id of the tracker}/{the amount of results to return}
- ➔ The name of the function is: "get\_all\_transmissions\_by\_id"
- ➔ The query is performed on the transmission collection in the crane database
- ➔ The ID passed in the function is compared to the tracker ReferenceField in the transmissions documents.
- ➔ int([:amount]) is used to return a given amount of transmissions. This amount is passed as parameter when the function is triggered. Since the amount is passed as an string, we need to convert it to a string.
- ➔ The Pymongo object is transformed in a list, which is then returned.

Next we want to create a query which retrieves all transmissions between a given time period between 2 Date Time Groups (DTG). How this is done is shown below in illustration 17.

```
@app.route('/api/transmissions_by_dtg/<id>/<dtg_1>/<dtg_2>', methods=['GET'])
def get_all_transmissions_dtg(id,dtg_1,dtg_2):

    dtg_1= datetime.strptime(str(dtg_1), '%Y-%m-%d')
    dtg_2= datetime.strptime(str(dtg_2), '%Y-%m-%d')

    query_result = crane_connection.db.transmission.find(
        {"timestamp": { "$gt": dtg_1, "$lt": dtg_2},"tracker":ObjectId(id)})

    return json.dumps(list(query_result), default=json_util.default)
```

Illustration 17: Query all transmission between given DTG

The code shown in illustration 17 is bit more complicated than the previous queries we created, so hang tight!

The following applies to the code shown above:

- ➔ The URL on which this function will be triggered is :  
localhost/api/transmissions\_by\_amount/{the id of the tracker}/{from dtg}/{to dtg}
- ➔ The name of the function is: "get\_all\_transmissions\_by\_dtg"
- ➔ The query is performed on the transmission collection in the crane database
- ➔ The ID passed in the function is compared to the tracker ReferenceField in the transmissions documents.

The lines shown in illustration 18 is used to convert the DTG's passed by our angular application to a valid datetime format.

```
@app.route('/api/transmissions_by_dtg/<id>/<dtg_1>/<dtg_2>', methods=['GET'])
def get_all_transmissions_dtg(id,dtg_1,dtg_2):

    dtg_1= datetime.strptime(str(dtg_1), '%Y-%m-%d')
    dtg_2= datetime.strptime(str(dtg_2), '%Y-%m-%d')

    query_result = crane_connection.db.transmission.find(
        {"timestamp": { "$gt": dtg_1, "$lt": dtg_2},"tracker":ObjectId(id)})

    return json.dumps(list(query_result), default=json_util.default)
```

Illustration 18: Strip DateTime

The Angular application passes the date-time groups as invalid DTG's to our Flask application.

The format of the string passed is as follows: 2020-01-21, where 2020 is the year, 01 is the month and 21 is the day.

There are 3 steps to make:

1. To convert the values passed by our Angular we need to convert the values to strings using: "str()" (Red).
2. Then we have to define what the order is in which the year, month and date are defined, in the value passed by our Angular application.  
In this case its year-month-day (Blue).
3. The function: ".strptime()" (Green), is used to strip the order of the year, month and date and put it in the correct order.

In the query as shown in the illustration below, we search the collection on the field: "timestamp" (Green) to find all the transmissions of the specified GPS tracker ID (Orange) within a certain time frame indicated by the two date-time groups (dtg).

For the specification of the time frame (time period) we use:

1. The paramter "\$gt" (greater than) (Red), to define that we want all the transmissions after the value of dtg\_1.
2. The parameter "\$lt" (lower than) (Blue), to define that we want all the transmissions below the value of dtg\_2.

```
@app.route('/api/transmissions_by_dtg/<id>/<dtg_1>/<dtg_2>', methods=['GET'])
def get_all_transmissions_dtg(id,dtg_1,dtg_2):

    dtg_1= datetime.strptime(str(dtg_1), '%Y-%m-%d')
    dtg_2= datetime.strptime(str(dtg_2), '%Y-%m-%d')

    query_result = crane_connection.db.transmission.find(
        {"timestamp": { "$gt": dtg_1, "$lt": dtg_2}, "tracker": ObjectId(id)})

    return json.dumps(list(query_result), default=json_util.default)
```

The last query we are going to create is used to search for transmissions in a given polygon, belonging to a certain GPS tracker on a ringed crane.

A polygon is a plane figure with at least three straight sides and angles, and typically five or more.

In the world of GIS (Geographical information systems), a polygon is defined by a set of coordinates.

In this case the query will return all the transmissions from which the coordinates are inside the given polygon coordinates.

The code for creating such a query is shown in illustration 19.

```
@app.route('/api/transmissions_in_polygon/<id>/<coords>', methods=['GET'])
def get_all_transmissions_in_polygon(id,coords):

    splitted_coords= coords.split(',')

    float_coords = [float(i) for i in splitted_coords]

    final_coords = [float_coords[k:k+2] for k in range(0, len(float_coords), 2)]

    query_result = crane_connection.db.transmission.find(
        {
            "geometry.coord":{
                "$geoWithin":{
                    "$geometry":{
                        "type":"Polygon",
                        "coordinates":[final_coords],
                        "crs":{
                            "type":"name",
                            "properties":{"name":"urn:x-mongodb:crs:strictwinding:EPSG:4326"}
                        }
                    }
                }
            },
            "tracker":ObjectId(id)
        })

    return json.dumps(list(query_result), default=json_util.default)
```

*Illustration 19: Query transmissions in given polygon*

The following applies to the code shown above:

- ➔ The URL on which this function will be triggered is :  
localhost/api/transmissions\_in\_polygon/{the id of the tracker}/{a list of coordinates}
- ➔ The name of the function is: "get\_all\_transmissions\_in\_polygon"
- ➔ The query is performed on the transmission collection in the crane database
- ➔ The ID passed in the function is compared to the tracker ReferenceField in the transmissions documents.
- ➔ The Pymongo object is transformed in a list, which is then returned.



The Angular application passes a list of coordinates. The problem is that these coordinates are not valid for MongoDB, so we need to transform these coordinates first. In illustration 20 below you can see a part of the Angular log.

```
[HPM] GET /api/transmissions_by_country/5e1dbb91507872e91a98c407/3.1750488,53.6055441,2.9992676,50.8336977,6.3391113,50.7086344,7.3168945,53.5794615,3.1750488,53.6055441
```

Illustration 20: Angular log

In this illustration it is shown what an URL that is passed to the Flask API looks like when querying on transmissions in a polygon.

All the numbers as visualized in the illustration above represent GPS coordinate pairs.

The problem here is, that the coordinates are in one long string and are such useless for a query in a MongoDB database:

"3.1750488,53.6055441,2.9992676,50.8336977,6.3391113,50.7086344,7.3168945,53.5794615,3.1750488,53.6055441"

We need 3 steps as framed in the source code below in Blue, Orange and Red to transform this long string into the following list of lists with coordinate pairs as floats:

```
[[3.1750488,53.6055441],[2.9992676,50.8336977],[6.3391113,50.7086344],  
[7.3168945,53.5794615],[3.1750488,53.6055441]]
```

```
@app.route('/api/transmissions_in_polygon/<id>/<coords>', methods=['GET'])  
def get_all_transmissions_in_polygon(id,coords):  
    splitted_coords= coords.split(',')  
    float_coords = [float(i) for i in splitted_coords]  
    final_coords = [float_coords[k:k+2] for k in range(0, len(float_coords), 2)]  
  
    query_result = crane_connection.db.transmission.find(  
        {  
            "geometry.coord":{  
                "$geoWithin":{  
                    "$geometry":{  
                        "type":"Polygon",  
                        "coordinates":[final_coords],  
                        "crs":{  
                            "type":"name",  
                            "properties":{"name":"urn:x-mongodb:crs:strictwinding:EPSG:4326"}  
                        }  
                    }  
                }  
            },  
            "tracker":ObjectId(id)  
        })  
  
    return json.dumps(list(query_result), default=json_util.default)
```

Now let's zoom in on some of the more difficult parts of the code on the next 2 pages.



### Step 1: Blue

The Python statement, shown in the illustration above in Blue is step 1 and it shows the required code to transform the coordinate list in string format to a coordinate list format.

The line in Blue splits, the string at each: “,” and then it appends the remaining values to a variable called: “splitted\_coords”.

The result of the code above is the transformation of the string into a list of strings:  
['3.1750488', '53.6055441', '2.9992676', '50.8336977', '6.3391113', '50.7086344', '7.3168945', '53.5794615', '3.1750488', '53.6055441']

### Step 2: Orange

Since a coordinate is only valid if it has the type: “float”, we need to transform all the strings to float values in step 2, indicated in Orange.

Here we use a statement with a for-loop for the data type conversion to convert each string in the list to a float value.

This line of code will result in the following list of floats:

[3.1750488, 53.6055441, 2.9992676, 50.8336977, 6.3391113, 50.7086344, 7.3168945, 53.5794615, 3.1750488, 53.6055441]

### Step 3: Red

Now for the final step 3 as framed in Red we need to create a list for each coordinate pair to hold the 2 GPS coordinates as pairs for which we also use a for-loop. This statement alternately appends the GPS coordinate pairs to a list.

This line results in a list of lists with valid coordinate pairs for MongoDB that are floats:  
[[3.1750488, 53.6055441], [2.9992676, 50.8336977], [6.3391113, 50.7086344], [7.3168945, 53.5794615], [3.1750488, 53.6055441]]

### The next step

Now that we have a valid list of coordinates, we want to use the list in a geospatial MongoDB query.

The part of the source code is shown in the illustration below for the MongoDB database query to get the set of crane location points sent by a GPS tracker of a ringed crane within a geospatial polygon area described by the list of GPS coordinates in `final_coords`:

```
query_result = crane_connection.db.transmission.find(
{
  "geometry.coord":{
    "$geoWithin":{
      "$geometry":{
        "type":"Polygon",
        "coordinates":[final_coords],
        "crs":{
          "type":"name",
          "properties":{
            "name":"urn:x-mongodb:crs:strictwinding:EPSG:4326"
          }
        }
      }
    }
  }
},
{"tracker":ObjectId(id)}
)
```

The following 6 steps explain the Python source code:

1. Do a query on the `coord` field in the collection of Transmission Documents (Red). Remember, when creating a data model we created an `EmbeddedDocument` called: `"geometry"`, which contains the `coord` field. This is also the reason why in this case we use: `"geometry.coord"`.
2. Then we use the geospatial operator: `"$geoWithin"` (Blue). This operator is used to tell MongoDB that we want to search for all transmissions within a certain geometry type.
3. Then we use the geospatial operator: `"$geometry"` (Green). This operator is used to tell MongoDB that a type of geometry is going to be used.
4. Then we tell MongoDB what type of geometry we are going to use in the query (purple). Since we are going to query the transmissions in a polygon we define a polygon as geometry type.
5. Then we tell MongoDB what type of coordinate reference system (`crs`) is going to be used (Orange). In this case we specify the custom MongoDB coordinate reference system.
6. Finally, we also want to query on a given tracker, by using the `ReferenceField`: `"tracker"` in our transmission documents (Pink).

That's it! You have created your first geospatial query!

# 5 Overview of Geospatial Query options in MongoDB

MongoDB offers a wide range of queries related to geospatial data. If you want more info related to geospatial queries, you can click on the following link: "<https://docs.mongodb.com/manual/reference/operator/query-geospatial/>". In the illustration below, you can find an overview of all geospatial queries offered by MongoDB.

<code>\$geoIntersects</code>	Selects geometries that intersect with a <code>GeoJSON</code> geometry. The <code>2dsphere</code> index supports <code>\$geoIntersects</code> .
<code>\$geoWithin</code>	Selects geometries within a bounding <code>GeoJSON</code> geometry. The <code>2dsphere</code> and <code>2d</code> indexes support <code>\$geoWithin</code> .
<code>\$near</code>	Returns geospatial objects in proximity to a point. Requires a geospatial index. The <code>2dsphere</code> and <code>2d</code> indexes support <code>\$near</code> .
<code>\$nearSphere</code>	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The <code>2dsphere</code> and <code>2d</code> indexes support <code>\$nearSphere</code> .
<code>\$box</code>	Specifies a rectangular box using legacy coordinate pairs for <code>\$geoWithin</code> queries. The <code>2d</code> index supports <code>\$box</code> .
<code>\$center</code>	Specifies a circle using legacy coordinate pairs to <code>\$geoWithin</code> queries when using planar geometry. The <code>2d</code> index supports <code>\$center</code> .
<code>\$centerSphere</code>	Specifies a circle using either legacy coordinate pairs or <code>GeoJSON</code> format for <code>\$geoWithin</code> queries when using spherical geometry. The <code>2dsphere</code> and <code>2d</code> indexes support <code>\$centerSphere</code> .
<code>\$geometry</code>	Specifies a geometry in <code>GeoJSON</code> format to geospatial query operators.
<code>\$maxDistance</code>	Specifies a maximum distance to limit the results of <code>\$near</code> and <code>\$nearSphere</code> queries. The <code>2dsphere</code> and <code>2d</code> indexes support <code>\$maxDistance</code> .
<code>\$minDistance</code>	Specifies a minimum distance to limit the results of <code>\$near</code> and <code>\$nearSphere</code> queries. For use with <code>2dsphere</code> index only.
<code>\$polygon</code>	Specifies a polygon to using legacy coordinate pairs for <code>\$geoWithin</code> queries. The <code>2d</code> index supports <code>\$center</code> .

Illustration 21: Geospatial operators

# 6    Creating data profiles

In this chapter is the explanation on how to create data profiles using the python packages: “Pandas” and “Pandas\_Profiling”.

A part of such a profile is shown below in illustration 22:

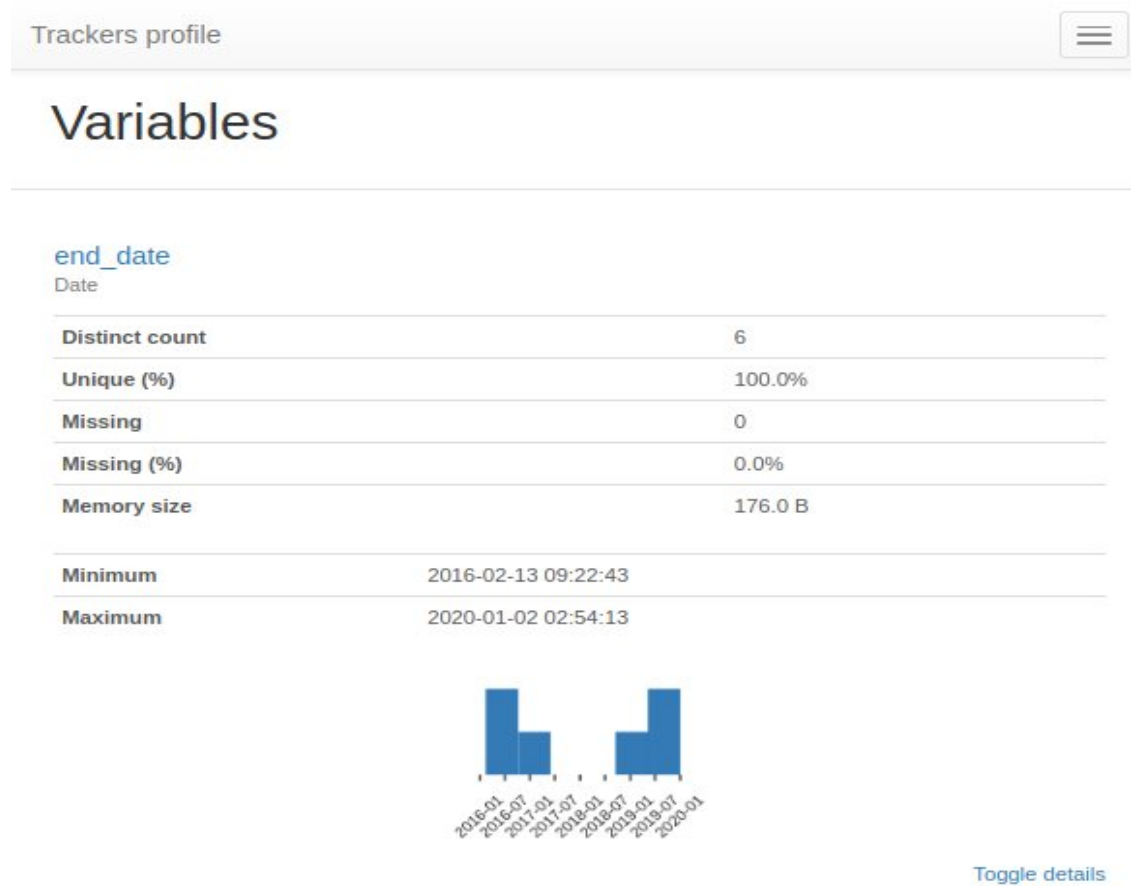


Illustration 22: Part of generated profile

The following information is reported in a data profile:

- ➔ Type inference: detect the types of columns in a dataframe.
- ➔ Essentials: type, unique values, missing values
- ➔ Quantile statistics like minimum value, Q1, median, Q3, maximum, range, interquartile range
- ➔ Descriptive statistics like mean, mode, standard deviation, sum, median absolute deviation, coefficient of variation, kurtosis, skewness
- ➔ Most frequent values
- ➔ Histogram
- ➔ Correlations highlighting of highly correlated variables, Spearman, Pearson and Kendall matrices
- ➔ Missing values matrix, count, heatmap and dendrogram of missing values
- ➔ Text analysis learn about categories (Uppercase, Space), scripts (Latin, Cyrillic) and blocks (ASCII) of text data.

To be able to create these profiles, we first have to define a generic function. In illustration 23 is shown how this is done.

```
def create_profile(data,title,output):  
  
    df = pd.DataFrame(list(data))  
  
    profile = df.profile_report(title=title)  
  
    profile.to_file(output_file=output)
```

*Illustration 23: Generic function for data profile creation*

The following applies to the code shown above:

- ➔ The name of the function is: "create\_profile" (Red)
- ➔ The function expects the following parameters (Blue):
  - 1) Data from which a Pandas data frame will be created.
  - 2) A title, which will be displayed at the top of the profiles.
  - 3) A output, which is the location where the profile will be generate.
- ➔ Convert the data passed in the query, which is an Pymongo object, into a list and append that list to a dataframe (Green). We assign the results to a variable called: "df".
- ➔ Call the function: "profile\_report()", on the data frame and pass the title, passed in the function:"create\_profile()", as parameter. Then assign the results to a variable called: "profile". (Purple)
- ➔ Export the profile to a .HTML file using the function: ".to\_file()". Pass the output location, passed in the function:"create\_profile()", as parameter in the function: ".to\_file()". (Orange)

Now that we have created the generic function, we need to put it to use.



The code for generating a data profile for the crane data sets is shown in illustration 24.

```
@app.route('/api/generate-trackers-profile', methods=['GET'])
def generate_trackers_profile()

    query_result = crane_connection.db.tracker.find({}, {"_id":0})

    create_profile(query_result, "Trackers profile", "templates/tracker.html")

    return render_template('tracker.html')
```

Illustration 24: Function for generating data profile

The following applies to the code shown above:

- ➔ The route which is bound to the function is: "<http://localhost/api/generate-trackers-profile>" (Purple)
- ➔ The name of the function is: "generate\_trackers\_profile" (Red)
- ➔ Retrieve all the trackers in the database (Blue), and make sure the MongoDB ID is not retrieved from the database (Green). We do this because the MongoDB ID isn't seen as a valid variable by Pandas\_Profiling. To skip a certain variable we use the syntax: `.find({}, {fieldname : 0})`, where "fieldname" is the variable you want to skip.
- ➔ Call the function: "create\_profile()" (Orange), and pass the following parameters:
  - 1) The variable: "query\_result", which contains the data returned by the query.
  - 2) "Trackers profile", which will be the title at the top of the data profile.
  - 3) The location where the tracker profile will be generated. In this case an HTML file called: "tracker.html", will be generated in the templates folder.
- ➔ The function: "generate\_profile", will then return a rendered HTML file containing our tracker data profile. (Brown)

That's it!

To test whether you correctly created the function, start the Flask application and navigate to the URL: "<http://localhost:5000/api/generate-trackers-profile>".

## 7 Connecting to a PostgreSQL database

Now that we are able to connect to our MongoDB database(s), we need to be able to connect and send queries to our PostgreSQL datastores. To be able to do this we are going to write a generic function.

This function takes in the connection parameters which will be defined in the configuration file. The function also takes the query which has to be performed on the PostgreSQL database.

The code for creating such a function is shown in the image below:

```
def query_pgsq(_database, _host, _user, _password, query):  
    try:  
        conn = psycopg2.connect(database=_database,  
                                host=_host,  
                                user=_user,  
                                password=_password)  
  
        curs = conn.cursor()  
  
        curs.execute(query)  
  
        res = curs.fetchall()  
  
        conn.close()  
  
        return res  
    except:  
        print("Failed to connect to:" + _database)
```

Illustration 25: Creating a generic function for querying PostgreSQL

The following applies to the code shown in illustration 25:

- ➔ The name of the function is: "query\_pgsq"
- ➔ The parameters that will be passed when calling the function are as follows (Red):
  - ➔ \_database: The name of the database on which the query will be performed
  - ➔ \_host: the server on which the database is running.
  - ➔ \_user: The name of the user which owns the database.
  - ➔ \_password: The password of the database.
  - ➔ query: The query that will be performed on the database.
- ➔ A try except statement is used in the function. This statement first tries to connect to the database (Blue) . If this fails it will return the code defined in the except statement (Green).

Now let's zoom in on the source code from illustration 25 above in the illustration below:

```
def query_psql(_database, _host, _user, _password, query):  
    try:  
        conn = psycopg2.connect(database=_database,  
                                host=_host,  
                                user=_user,  
                                password=_password)  
        curs = conn.cursor()  
        curs.execute(query)  
        res = curs.fetchall()  
        conn.close()  
        return res  
    except:  
        print("Failed to connect to:" + _database)
```

*Illustration 26: Connecting and querying a database*

The following applies to the code shown in illustration 26:

- ➔ First we connect to the database using the function: ".connect()" on the psycopg2 instance. As parameters in the function: "connect()", we pass the parameters passed in the function: "query\_psql()". We assign the connection to a variable called: conn. (Red)
- ➔ Next, we assign the cursor of the connection to a variable called curs. We use the function: ".cursor()" on the connection. (Blue)
- ➔ Next, we are going to perform the query passed in the function: "query\_psql()" (Green). This is done by calling the function: ".execute()" on the cursor. As parameter we are going to pass the query.
- ➔ After the query is executed in the database, we are going to fetch all the records and assign them to a variable called: res. (Orange)
- ➔ Then we want to close the connection. This is done by calling the function: ".close()" on the connection. (Purple)
- ➔ Finally we want to return all the results stored in the variable: "res". (Brown)

## 8 Query a PostgreSQL database

To perform a query we first need a PostgreSQL database instance connection. Below is described how to perform a query to retrieve the amount of ports in the World Port Index data set.

First we want to add the connection information related to the World Port Index database. The database name, user and password are the same in both the Dockerized and the local version of the World Port Index database. The only difference is the location on which the database is hosted. We do this by defining the connection parameters for this database in our config.py file. How this is done is shown in illustration 27.

```
# Here we define the Local URI of our World Port Index database.
# Uncomment this line if you are going to run the PostgreSQL instance locally.
WPI_HOST = "localhost"

# Here we define the Docker URI of our World Port Index database.
# Uncomment this line if you are going to run the PostgreSQL instance in Docker.
#WPI_HOST = "postgresql-datastore"

WPI_DATABASE = "World_Port_Index_Database"
WPI_USER = "postgres"
WPI_PASS = "geostack"
```

Illustration 27: Creating the WPI Connection data

The following applies to the code, shown in illustration 27:

- ➔ The name of the database is assigned to a variable called: "WPI\_DATABASE"
- ➔ The name of the user is assigned to a variable called: "WPI\_USER"
- ➔ The location of the server when the database is running in Docker is assigned to a variable called: "WPI\_HOST".
- ➔ The location of the server when the database is running locally, is assigned to a variable called: "WPI\_HOST".
- ➔ The password of the database is assigned to a variable called: "WPI\_PASS".

Now that we have defined the connection parameters we want to use them in a function that queries the database. We are going to create this function in the app.py file.

The function `get_all_ports_count()` returns the total amount of ports and the source code is shown in illustration 28 :

```
@app.route('/api/ports_count/', methods=['GET'])
def get_all_ports_count():

    # Create the query to execute
    query = "SELECT COUNT(*) FROM wpi;"

    # Call the function: "query_pgsql()" and pass the connection
    # parameters of the WPI database. Also pass the query defined
    # above.
    query_result = query_pgsql(app.config["WPI_DATABASE"],
                               app.config["WPI_HOST"],
                               app.config["WPI_USER"],
                               app.config["WPI_PASS"],
                               query)

    # Return the results as valid JSON.
    return json.dumps(query_result)
```

Illustration 28: Obtaining the Port Counts

The following applies to the code shown illustration 28:

- ➔ The URL on which this function will be triggered is : `localhost/api/ports_count/` (Green)
- ➔ The name of the function is: `"get_all_ports_count"`
- ➔ The query that is performed is the following: `"SELECT COUNT(*) FROM wpi"`. (Red)
- ➔ The function `query_pgsql` is called. We pass the connection parameters as first 4 parameters in this function. Then we pass the query created in the previous step as last parameter. We assign the results of the query to a variable called: `"query_result"`. (Blue)
- ➔ The query results are dumped and returned as valid JSON. (Orange)

You can test whether or not the query is working by starting the Flask application and navigating to the URL: [http://localhost:5000/api/ports\\_count/](http://localhost:5000/api/ports_count/)



Now we want to create a function which returns all the ports and their names and coordinates. The code for this is shown in illustration 29.

```
@app.route('/api/ports/', methods=['GET'])
def get_all_ports():

    # Create the query to execute
    query = "SELECT longitude, latitude, PORT_NAME, COUNTRY FROM wpi;"

    # Call the function: "query_pgsql()" and pass the connection
    # parameters of the WPI database. Also pass the query defined
    # above.
    query_result = query_pgsql(app.config["WPI_DATABASE"],
                               app.config["WPI_HOST"],
                               app.config["WPI_USER"],
                               app.config["WPI_PASS"],
                               query)

    # Return the results as valid JSON.
    return json.dumps(query_result)
```

Illustration 29: Obtaining the port count

The following applies to the code shown illustration 29:

- ➔ The URL on which this function will be triggered is : localhost/api/ports/ (Green)
- ➔ The name of the function is: "get\_all\_ports".
- ➔ The query that is performed is the following: ""SELECT longitude, latitude, PORT\_NAME, COUNTRY FROM wpi;". (Red)
- ➔ The function query\_pgsql is called. We pass the connection parameters as first 4 parameters in this function. Then we pass the query created in the previous step as last parameter. We assign the results of the query to a variable called: "query\_result". (Blue)
- ➔ The query results are dumped and returned as valid JSON. (Orange)

You can test whether the query is working by starting the Flask application and navigating to the URL: <http://localhost:5000/api/ports/>

Running the flask application is done by opening a terminal and entering the following command in a terminal:

```
gunicorn3 -b :5000 app:app --chdir ~/Geostack/flask-gunicorn
```

## 9 Retrieving TileStache configuration entries.

Now we want to be able to obtain all entries from our Tilestache configuration. Obtaining the entries is needed because one of the functionalities of the Map-viewer applications, is that you can switch between available web map servers. The angular application will send an request to the Flask-API. The Flask-API then returns a list with available map provides. We want to create a function which scrapes the index page of our Tilestache Tileservers. This index page is located at: "<http://localhost/tiles/>" or "<http://nginx-webserver/tiles/>".

```
@app.route('/api/get_tilestache_entries/')
def get_all_tilestache_entries():

    # Scrape the HTML page located at: "http://localhost/tiles/"
    # or "http://nginx-webserver/tiles/" if the nginx webserver is running as
    # docker container.
    scraped_entries = BeautifulSoup(
        urlopen(app.config["TILESTACHE_INDEX"]), features="html.parser")

    # Create an empty list called: "entries"
    entries = []

    # Loop through all the HTML elements on the page.
    # Execute the code in the FOR loop if the HTML element tag
    # matches 'p'.
    for tag in scraped_entries.find_all('p'):

        # If an HTML element matches with the <p> tag, the content
        # of the <p> tag will be encoded and decoded. Then we append
        # it to the list of entries.
        entries.append(tag.text.encode('utf-8').decode('utf-8'))

    # Return the entries in JSON format
    return json.dumps(entries, default=json_util.default)
```

Illustration 30: Function for obtaining the Tilestache Entries

This page is called: "entries.html", and contains all our layer entries in our tilestache configuration file. The code needed to create this function is shown in illustration 30. The following applies to the code, shown in illustration 30:

- ➔ The route which is bound to the function is: "[http://localhost/api/get\\_all\\_tilestache\\_entries/](http://localhost/api/get_all_tilestache_entries/)" (Red)
- ➔ The name of the function is: "get\_all\_tilestache\_entries()" (Blue)
- ➔ This function scrapes all the <p> tags from the HTML file located at: "<http://localhost/tiles/>", which is the homepage of our Tilestache Tileservers running behind the NGINX Web server.

Let's zoom in on some parts of the code. First we want to scrape the whole HTML file. How this is done is shown in the illustration below.

```
scraped_entries = BeautifulSoup(  
    urlopen(app.config["TILESTACHE_INDEX"]), features="html.parser")
```

*Illustration 31: Scraping the index.html*

First we want to create a new instance of BeautifulSoup (Red). As first parameter we pass the function: "urlopen()" (Blue). In this function we pass the URL on which the index page of our Tilestache server is located, which is the URI assigned to the variable: "TILESTACHE\_INDEX" located in our config.py file (Green). As second parameter, in the BeautifulSoup instance, we assign the type of parser that needs to be used (Orange). This parser is called: "html.parser", which makes sure that the HTML page is parsed in HTML format.

Now that we are able to get the content of the HTML page located at: "<http://localhost/tiles/>", we want to extract the text which is defined in <p> tags. How this is done is shown in illustration 31.

```
entries = []  
  
for tag in scraped_entries.find_all('p'):  
    entries.append(tag.text.encode('utf-8').decode('utf-8'))
```

*Illustration 32: Obtaining the text in the <p> tags.*

First we need to create an empty list. This list is called: "entries" (Red). We are going to append all the text in between the <p> tags, from the scraped HTML page, to this list.

To be able to do this we need to create a FOR loop, which loops through all the HTML elements on the scraped page (Blue). To find all <p> tags, we are going to use the function: "find\_all()" on the scraped HTML page (Green). As parameter we pass the value: "p". This will make sure the code, in the for loop, is only performed on the <p> tags.

Then we are going to append all the elements that match tag <p>, to the entries list. For this, we will use the function .append() on the list (Purple). As parameter we pass the tag. We need to encode and decode the text for it to be append-able to a list (Orange).

Now we have a list of TileStache entries, which will be served to and used in the web application to switch between map providers.

## 10 Adding the GPS Route (Trail) queries

Now that we have most of the code in place, we should also add the queries related to obtaining the GPS Route (Trail) data from our MongoDB datastore: "Trail\_Database".

The functions and queries we are going to create for this database are very similar to the ones which we create for the Crane (Tracker) data.

We want to start of by creating the connection to our Trail database. This is done by adding the following code below the function: "get\_all\_tilestache\_entries()":

```
# Here we connect to the Trial Database and assign the connection to a variable
# Called: "trail_connection"
trail_connection = PyMongo(app, uri=app.config["TRAIL_DATABASE_URI"])
```

We are going to create 6 new functions which are as follows:

- ➔ get\_all\_trails(), which is used to obtain all trails in our Trail database;
- ➔ get\_one\_trail(), which is used to obtain one trail using it's MongoDB;
- ➔ get\_all\_signals\_count(), which is used to obtain the total amount of signals in the database;
- ➔ get\_all\_signals\_by\_id(), which is used to obtain all the signals belonging to a certain trail. This is done by passing the MongoDB of the trail as input parameter;
- ➔ get\_all\_signals\_amount(), which is used to obtain a certain amount of signals from a certain trail. This is done by passing the MongoDB of a trail and an amount as input parameters.
- ➔ get\_all\_signals\_dtg(), which is used to obtain all signals between a timeframe from a certain trail. This is done by passing the MongoDB of a trail, a start date and an end date as input parameters.

Let's start of by adding the function called:"get\_all\_trails()". We do this by adding the following code below the code related to connecting to the Trail database which we created earlier in this document:

```
@app.route('/api/trails/', methods=['GET'])
def get_all_trails():

    # We transform the query result into a list and assign the result to a
    # variable called: "query_result".
    query_result = list(trail_connection.db.trail.find())

    # We return the query_result as json.
    return json.dumps(query_result, default=json_util.default)
```



Now we want to create the function related to querying a trail using it's MongoDB. This function is called: "get\_one\_trail()". Adding the function is done by adding the following code below the function: "get\_all\_trails()":

```
@app.route('/api/trails/<id>', methods=['GET'])
def get_one_trail(id):

    # We assign the result of the query to a variable called: "query_result".
    query_result = trail_connection.db.trail.find({"_id": ObjectId(id)})

    # We return the query_result as JSON.
    return json.dumps(query_result, default=json_util.default)
```

Next up is the function related to obtaining the total amount of signals in the Trail database. This function is called: "get\_all\_signals\_count()". Adding the function is done by adding the following code below the function: "get\_one\_trail()":

```
@app.route('/api/signals_count/', methods=['GET'])
def get_all_signals_count():

    # we assign transform the result of the query to a string and assign it to
    # a variable called: "query_result".
    query_result = str(trail_connection.db.signal.count())

    # We return the query_result which is a string.
    return query_result
```

Now we want to create the function related to querying all signals belonging to a trail. This function is called: "get\_all\_signals\_by\_id()". Adding the function is done by adding the following code below the function: "get\_all\_signals\_count()":

```
# -----
# 21) Create the function which retrieves all the signals belonging to a certain
# trail. This is done by passing the MongoDB of the trail as input parameter.
@app.route('/api/signals_by_id/<id>', methods=['GET'])
def get_all_signals_by_id(id):

    # The query is performed on the signal collection
    # the value of the Trail ReferenceField is compared to the
    # id passed in the function. the [:3000] is used to only return
    # the first 3000 results which are assigned to a variable called:
    # "query_result"
    query_result = list(trail_connection.db.signal.find(
        {"trail": ObjectId(id)})[:3000])

    # The result of the query is returned as JSON.
    return json.dumps(query_result, default=json_util.default)
```



Next up is creating the function related to querying a certain amount of signals belonging to a trail. This function is called: "get\_all\_signals\_amount()". Adding the function is done by adding the following code below the function: "get\_all\_signals\_by\_id()":

```
# - - - - -
# 22) Create the function which retrieves a certain amount of signals from a
#      certain trail. This is done by passing the MongoDB of a trail and an
#      amount as input parameters.
@app.route('/api/signals_by_amount/<id>/<amount>', methods=['GET'])
def get_all_signals_amount(id,amount):

    # Here we create a query which obtains a certain amount of signals belonging
    # end date which were passed as input parameters and transformed above.
    # We transform the result to a list and assign the result to a variable
    # called:"query_result".
    query_result = list(trail_connection.db.signal.find(
        {"trail": ObjectId(id)})[:int(amount)]
    )

    # Here we return the query_result as JSON.
    return json.dumps(query_result, default=json_util.default)
```

The last function we need to add is the function related to querying a certain amount of signals belonging to a trail. This function is called: "get\_all\_signals\_dtg()". Adding the function is done by adding the following code below the function: "get\_all\_signals\_amount()":

```
# - - - - -
# 23) Create the function which retrieves all signals between a timeframe from
#      a certain trail. This is done by passing the MongoDB of a trail, a start
#      date (dtg_1) and an end date (dtg_2) as input parameters.
@app.route('/api/signals_by_dtg/<id>/<dtg_1>/<dtg_2>', methods=['GET'])
def get_all_signals_dtg(id,dtg_1,dtg_2):

    # Here we convert the strings (representing the datetimes) to a valid
    # datetime format.
    dtg_1= datetime.strptime(str(dtg_1), '%Y-%m-%d')
    dtg_2= datetime.strptime(str(dtg_2), '%Y-%m-%d')

    # Here we create a query which obtains the signals between the start and
    # end date which were passed as input parameters and transformed above.
    # We transform the result to a list and assign the result to a variable
    # called:"query_result".
    query_result = list(trail_connection.db.signal.find(
        {"time": { "$gt": dtg_1, "$lt": dtg_2},"trail":ObjectId(id)})
    )

    # Here we return the query_result as JSON.
    return json.dumps(query_result, default=json_util.default)
```

That's it! Now we are able to obtain all the Trail data from our MongoDB datastore!

## 11 Add the Flask API to our LOCAL NGINX config!

At this point we have a working Flask API. Now we want to add the API to our local NGINX configuration file called: "nginx-local.conf", which is located in the folder: "~/Geostack/nginx-modsecurity".

So let's open the file. We first want to create the upstream server for our Flask API. This is done by adding the following code below the line: "modsecurity\_rules\_file /etc/nginx/modsec/main.conf;"

```
upstream flask-api { server localhost:5000; }
```

Next we want to add the location on which the Flask API will become available when running behind the NGINX web server. We do this by adding the following line below the section where we have defined the default index.html location of the NGINX web server:

```
location /api/ { proxy_pass http://flask-api; }
```

That's it! Now we need to copy the nginx-local.conf file to the folder /etc/nginx/ by running the command:

```
sudo cp ~/Geostack/nginx-modsecurity/nginx-local.conf /etc/nginx/nginx.conf
```

Now we need to restart NGINX by running the command: `sudo service nginx restart`

For ease of use we are also going to create 2 desktop shortcuts which are used to start and stop the Flask API. This is done by performing the following steps:

- 1) Create a desktop shortcut to start the Flask API by running the following command: `touch ~/Desktop/Start-Flask-API.desktop`
- 2) Add the following code to the file that is created on the desktop and save it.

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=true
Exec=sh -c "gunicorn3 -b :5000 app:app --chdir ~/Geostack/flask-gunicorn"
Icon=gnome-panel-launcher
Name[en_US]=Start-Flask-API
```

- 3) Create a desktop shortcut to stop the Flask API by running the following command.

```
touch ~/Desktop/Stop-Flask-API.desktop
```

- 4) Add the following code to the file that is created on the desktop and save it.

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Version=1.0
Type=Application
Terminal=false
Exec=sh -c "fuser -k 5000/tcp"
Icon=gnome-panel-launcher
Name[en_US]=Stop-Flask-API
```

- 5) Set the desktop shortcuts to trusted by running the following command:

```
for i in ~/Desktop/*.desktop; do gio set "$i" "metadata::trusted" yes ;done
```

- 6) Make sure the shortcuts are launch-able by running the following command:

```
sudo chmod +x ~/Desktop/Stop-Flask-API.desktop && sudo chmod +x \
~/Desktop/Start-Flask-API.desktop
```

Now start the Flask API by running the desktop shortcut that we created above. See the illustration below for the shortcut.



Once the Flask API is started you should navigate to the URL:

<http://localhost/api/trackers> in you web browser.

You will be greeted with all the trackers in the MongoDB datastore as shown in the illustration below.

```
[{"_id": {"$oid": "5e834749341c351a68f7da86"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9407, "start_date": {"$date": "1374289448000"}, "end_date": {"$date": "1488348009000"}, "name": "Agnetha", "transmission_Count": 44534}, {"_id": {"$oid": "5e834776341c351a68f8887d"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9381, "start_date": {"$date": "1374375992000"}, "end_date": {"$date": "1455355363000"}, "name": "Frida", "transmission_Count": 123805}, {"_id": {"$oid": "5e8347dc341c351a68fa6c1b"}, "study_name": "GPS telemetry of Common Cranes, Sweden", "individual_taxon_canonical_name": "Grus grus", "individual_local_identifier": 9472, "start_date": {"$date": "1374294939000"}, "end_date": {"$date": "1471708359000"}, "name": "Cajsa", "transmission_Count": 67887},
```

## 12 Dockerizing the Flask API

Now that we have a fully functional Flask API we need to add it as service to our docker-compose.yml file located in the Geostack root folder (~ / Geostack / docker-compose.yml).

So let's open this file and add the following service below the entry of our nginx-webserver service:

```
flask-api:
  # Here we set the name of the Flask-API / App container
  container_name: flask-api
  # The line below makes sure the Flask container restarts when stopping
  # accidentally
  restart: always
  # Set the directory in which the dockerfile is located
  build: ./flask-gunicorn
  # Here we set the port on which the Tileservers will be running
  ports:
    - "5000"
  # Here we set the command that will be executed when the Flask-API
  # service starts.
  command: gunicorn -b :5000 app:app
  # Here we add the Downloads volume of the docker container
  volumes:
    - ../downloads
  # Below we define the services on which the Flask-API depends.
  # These services will be started before the Flask-API starts.
  depends_on:
    - mongodb-datastore
    - postgresql-datastore
    - tilestache-server
```

After you have added the lines above you should save the file.

We also need to edit the Docker NGINX configuration which is called: "nginx-docker.conf" and is located in the folder: "~ / Geostack / nginx-modsecurity /".

So let's open this file and add the following line below the line: "modsecurity\_rules\_file / etc / modsecurity.d / include.conf;"

```
# Create the upstream server for the flask-API
upstream flask-api { server flask-api:5000; }
```

Then also add the following below the line where we defined the index.html location:

```
# Create a new location on which the Flask-API is accessible
location /api/ { proxy_pass http://flask-api; }
```

After you have added the lines above you should save the file.

NOTE: as mentioned before, when you are building the Flask API docker container you have to make sure that all the connection strings in the config.py file are changed to the **DOCKER** connection strings instead of the **LOCAL** connection strings.

