
Paradise Paper Search Documentation

Release 1

Neo4j

Feb 07, 2018

CONTENTS:

1	Set up the environment.	3
1.1	Set your virtualenv	3
1.2	Add project dependencies	4
2	Set the NeoModel database	5
3	How to create the base models.	7
3.1	Create a new app for the API:	7
3.2	Create The Models Directory:	8
3.3	Start creating The Models Files:	8
3.4	Modify the __init__.py file:	11
3.5	Create constraints or indexes:	11
4	How to query the Neo4j database with Neomodel.	13
4.1	Intro.	13
4.2	Using your models	13
4.3	Neomodel Query API	14
4.4	Creating some utils to search the PPGDB	16
4.5	Using the search utils	20
5	How to create the API.	23
5.1	Why the API structure	23
5.2	Explain the library	23
5.3	Create the views / endpoints	23
6	How to serialize the data.	27
6.1	Why we need to serialize the data	27
6.2	Create serialize methods	27
6.3	Call the serialize methods on the helpers	29
6.4	Create the serialize relationships methods	29
6.5	Call the serialize relationships methods on the helpers	32
6.6	Return the json to the frontend	32
7	The Search App, consuming the Fetch API.	39

Overview

The aim of this tutorial is to demonstrate how to develop a web application using the Django framework backed by Neo4j, connecting them with the Neomodel driver. Neomodel is an Object Graph Mapper (OGM) for the Neo4j graph database.

The tutorial covers several topics, some of them are: how to configure Neomodel within a Django program, create models, query to the database, create an API. It provides best-practice guidance on implementing Neomodel.

The project is a web application to search for information on a Paradise Paper Graph Database. The data from the Database includes companies and people in more than 200 countries that are part of the Paradise Papers, Panama Papers, Bahama Leaks or the Offshore Leaks investigations.

The search of the program can filter the mentioned data by country, jurisdiction and/or data source. The approach adopted to make the program was to create a single-page web application, from which the data that is displayed will be obtained from an API. The API will fetch the data from the Neo4j database and return it as a JSON. This allows you to retrieve fast segments of data in several smaller requests, instead of making a single large request.

The types of data you can encounter when searching are: Entity, Officer, Intermediary, Address, Other. Each of these data records is displayed as a *Node* within the graph database. If you are not familiar with the definition of a node, it is a graphic data record. Here is an example:



In a graph database nodes are linked with each other with lines; they represent the relationship between them. This is called a *relationship*. On this application there are offshore entities that

have a registered address; Therefore there is a relationship type called *REGISTERED_ADDRESS*.



On this example we can see a offshore entity register to an address on the bahamas.

What's more, each node will have *properties* , which are essentially labels that are apply to a record. For example, a Entity node might have a label country with the value of "Bahamas". Having these labels allow the system to find all the nodes that are in "Bahamas", instead of the ones that are in "Canada" or "Mexico". The properties can be compare as columns in SQL databases.

SET UP THE ENVIRONMENT.

The following section will have the steps to set your local environment, so you can begin to work on your Django-neomodel implementation. If you want a point to start, we provide a base app to begin the tutorial.

First clone the [paradise-papers-django](#) repo.

After you download the repo go to the `start_app` branch and you will get a working app that will be our start point.

1.1 Set your virtualenv

First, you need Python and pip install on your local environment. Here is the official download page of [python](#)

Here is a quick tutorial on how to install [pip](#)

Now we have everything we need to create our virtualenv. On the next part we are going to introduce the basic to get your app running; if you want more information check this [virtualenv](#) tutorial

First go to your app folder on the case of the `paradise-papers-django` project you need to be on the main folder (`localPath/paradise-papers-django`) and run the following command to install virtualenv:

```
pip install virtualenv
```

You can use this pip command to check the dependencies installed:

```
pip freeze
```

After you install the virtualenv; you need to create a virtual environment for your app. On the same folder that you install virtualenv you need to run the following command:

```
virtualenv .
```

The dot means that you going to install your virtual environment on the current folder, but you can specify the path that you need to create your virtual environment.

After your virtual environment is ready your directory should look like this if you are using the example app:

```
├── paradise-papers-django/
│   ├── docs/
│   ├── Include/
│   ├── Lib/
│   ├── paradise_papers_search/
│   ├── LICENSE
│   ├── pip-selfcheck.json
│   ├── PULL_REQUEST_TEMPLATE.md
│   └── README.md
```

Now you need to activate your brand new virtual environment with the following command:

```
.\Scripts\activate
```

On linux an Mac:

```
source bin/activate
```

Now you are in your new virtual environment!!!

1.2 Add project dependencies

The next step is to install the other dependencies that we need. For this you can use our requirement file if you are using the example app otherwise you need to install the following dependencies using the `pip install <name>==<version>` command:

```
Django==1.11.2
django-neomodel==0.0.4
neo4j-driver==1.2.1
neomodel==3.2.5
```

If you're using your own requirement file you can use this command to get all your dependencies:

```
pip install -r /path/to/requirements.txt
```

if you are using our staring app; go to the paradise-papers-search folder and run this command:

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

Now you have all dependencies that you need for this implementation!!!!

SET THE NEOMODEL DATABASE

On this section, we are going to install Neo4j on our local machine and show how you are going to connect it with our app. First, you need to download the desktop app of [Neo4j](#).

Follow the steps of the wizard to install the app. Then you need to set your database; here is the official [guide](#) that you can follow to get Neo4j working on your local machine.

On the app example, we are going to use the LEAKS DATABASE. This is the data from the Panama Papers, the Offshore Leaks, the Bahamas Leaks provided by the ICIJ(International Consortium of Investigative Journalists) here are some information about the [ICIJ](#) and the [offshore](#) leaks.

On this tutorial, we are going to concentrate on replicating the offshore leaks search app. It's important that you know the entities that we are going to handle the app.

Offshore Entity: A company, trust or fund created in a low-tax, offshore jurisdiction by an agent.

Officer: A person or company who plays a role in an offshore entity.

Intermediary: A go-between for someone seeking an offshore corporation and an offshore service provider – usually a law-firm or a middleman that asks an offshore service provider to create an offshore firm for a client.

Address: Contact postal address as it appears in the original databases obtained by ICIJ.

Definitions provided by https://offshoreleaks.icij.org/pages/about#terms_definition

We use the paradise-papers database for the example app. [Here](#) is a quick set up for the Desktop app that will bring the database included

After you have Neo4j up and running we only need to create the connection between Django and the database to do this neomodel will give us a hand.

Go to the settings file on your project and import config from neomodel:

```
from neomodel import config
```

Then you need to delete the default database setting:

```
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
```

```
'ENGINE': 'django.db.backends.sqlite3',
'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
}
```

And now you only need to set the Neo4j database on the config in the settings like this:

```
#Connect to Neo4j Database
config.DATABASE_URL = 'bolt://neo4j:neo4j@localhost:7687'
```

Now you are all set just need to run your local server with the runserver command:

```
python manage.py runserver
```

HOW TO CREATE THE BASE MODELS.

3.1 Create a new app for the API:

Right now, we need to create a Django app which is going to be used to create the search API and, with it, we will define our models classes.

To create this new app, it is needed to run this command:

```
python manage.py startapp fetch_api
```

Where `fetch_api` is the name of the application we are creating, if you prefer you can use any other name. This will create a new folder inside the Django project directory, and the structure of it is:

```
├── fetch_api/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   └── migrations/
│       └── __init__.py
```

Now we need to create a file called `urls.py` inside the application we just created. Right now we are only going to add the following code:

```
from django.conf.urls import url

urlpatterns = []
```

As we go on in this tutorial, we will start adding the endpoint urls, but for now we only need to have the empty urlpattern list.

The next step is to register the `fetch_api` URLconf into the root project URLconf. We need to open the file `paradise_papers_search/urls.py`. Afterwards, we have to add `url(r'^fetch/', include('fetch_api.urls', namespace='fetch_api'))`, inside the `urlpatterns` list. After completing these steps the file should look like this:

```
from django.conf.urls import url, include from django.contrib import admin from
django.views.generic import TemplateView

urlpatterns = [ url(r'^admin/', admin.site.urls), url(r'^$',
TemplateView.as_view(template_name='index.html', name='index'), url(r'^fetch/',
include('fetch_api.urls', namespace='fetch_api')), ]
```

3.2 Create The Models Directory:

On The `fetch_api` folder, first, we need to remove the `models.py` file. Then we need to create a new folder called `models`. Inside the folder, you will need to add the files with the structure of the nodes. Each file will have the essential fields and methods of the data stored in Neo4j database.

3.3 Start creating The Models Files:

To start we will create an `entity.py` file:

1. The first step is to add the imports needed:

```
from neomodel import (
    StringProperty,
    StructuredNode,
    RelationshipTo,
    RelationshipFrom
)
```

2. Create a class for the node called `Entity`. We are connecting to a Neo4j database instead of the regular Django database; therefore, we are going to use `StructuredNode`. This is the equivalent of `models.Model` which is usually used when creating a `Models` class in Django. When using `StructuredNode`, `neomodel` automatically creates a label for each class using it with the corresponding indexes and constraints.

If you need `ModelForm`, you will need to change `StructuredNode` to `DjangoNode`. Also, you will need to add a 'Meta' class. For more reference see the documentation for [django-neomodel](#)

```
class Entity(StructuredNode):
```

3. The next step is to add the properties for the node. Each node property in the Neo4j database should be a property in the model class.

The properties of the `Entity` node in the database have the following scheme:

```
{
  "sourceID": "Panama Papers",
  "address": "MEI SERVICES LIMITED ROOM E; 6TH FLOOR; EASTERN COMMERCIAL CENTRE; 395-
  ↪399 HENNESSY ROAD HONG KONG",
  "jurisdiction": "SAM",
  "service_provider": "Mossack Fonseca",
  "countries": "Hong Kong",
  "jurisdiction_description": "Samoa",
  "valid_until": "The Panama Papers data is current through 2015",
  "ibcRUC": "25475",
  "name": "JIE LUN INVESTMENT LIMITED",
  "country_codes": "HKG",
  "incorporation_date": "10-APR-2006",
  "node_id": "10000020",
  "status": "Active",
}
```

Therefore, the structure of the `Entity` class should be:

```
class Entity(StructuredNode):
    sourceID = StringProperty()
    address = StringProperty()
    jurisdiction = StringProperty()
```

```

service_provider      = StringProperty()
countries             = StringProperty()
jurisdiction_description = StringProperty()
valid_until           = StringProperty()
ibcRUC               = StringProperty()
name                 = StringProperty()
country_codes        = StringProperty()
incorporation_date    = StringProperty()
node_id              = StringProperty(index = True)
status               = StringProperty()

```

On our case, the database we are using only have string values for the properties. However, there are several types of properties available; such as, IntegerProperty, ArrayProperty, DateProperty. In case you need to know more about this, go to this [link](#)

4. Add the relationships for the node:

```

officers              = RelationshipFrom('.officer.Officer', 'OFFICER_OF')
intermediaries        = RelationshipFrom('.intermediary.Intermediary',
    ↳ 'INTERMEDIARY_OF')
addresses             = RelationshipTo('.address.Address', 'REGISTERED_ADDRESS
    ↳ ')
others                = RelationshipFrom('.other.Other', 'CONNECTED_TO')

```

How **RelationshipFrom** and **RelationshipTo** works:

1. The first parameter is the type of node you want to connect. e.g `.officer.Officer`
2. The second parameter is the relationship type. e.g. `OFFICER_OF`
 - **RelationshipFrom** is an INCOMING relationship
 - **RelationshipTo** is an OUTGOING relationship
 - Also, there is one call **Relationship** which can be either

If **RelationshipFrom** be illustrated, the output would be something like:

```

Found django_neomodel.DjangoNode
! Skipping class django_neomodel.DjangoNode is abstract
Found fetch_api.models.Entity.Entity
+ Creating index node_id on label Entity for class fetch_api.models.Entity.Entity
Found fetch_api.models.Address.Address
+ Creating index node_id on label Address for class fetch_api.models.Address.Address
Found fetch_api.models.Intermediary.Intermediary
+ Creating index node_id on label Intermediary for class fetch_api.models.Intermediary.Intermediary
Found fetch_api.models.Officer.Officer
+ Creating index node_id on label Officer for class fetch_api.models.Officer.Officer
Found fetch_api.models.Other.Other
+ Creating index node_id on label Other for class fetch_api.models.Other.Other
Finished 6 classes.

```

Repeat these steps for each node class you wish to create. On this program, those would be: `address.py`, `intermediary.py`, `officer.py`, and `other.py`. You must add the following code to each of the files:

`address.py`

```

from neomodel import (
    StringProperty,
    StructuredNode,
    RelationshipFrom

```

```
)

class Address(StructuredNode):
    sourceID      = StringProperty()
    country_codes = StringProperty()
    valid_until   = StringProperty()
    address       = StringProperty()
    countries     = StringProperty()
    node_id       = StringProperty(index = True)
    officers      = RelationshipFrom('.officer.Officer', 'REGISTERED_ADDRESS')
    intermediaries = RelationshipFrom('.intermediary.Intermediary', 'REGISTERED_
↪ADDRESS')
```

intermediary.py:

```
from neomodel import (
    StringProperty,
    StructuredNode,
    RelationshipTo
)

class Intermediary(StructuredNode):
    sourceID      = StringProperty()
    valid_until   = StringProperty()
    name          = StringProperty()
    country_codes = StringProperty()
    countries     = StringProperty()
    node_id       = StringProperty(index = True)
    status        = StringProperty()
    entities      = RelationshipTo('.entity.Entity', 'INTERMEDIARY_OF')
    addresses     = RelationshipTo('.address.Address', 'REGISTERED_ADDRESS')
```

officer.py:

```
from neomodel import (
    StringProperty,
    StructuredNode,
    RelationshipTo,
)

class Officer(StructuredNode):
    sourceID      = StringProperty()
    name          = StringProperty()
    country_codes = StringProperty()
    valid_until   = StringProperty()
    countries     = StringProperty()
    node_id       = StringProperty(index = True)
    addresses     = RelationshipTo('.address.Address', 'REGISTERED_ADDRESS')
    entities      = RelationshipTo('.entity.Entity', 'OFFICER_OF')
```

other.py:

```
from neomodel import (
    StringProperty,
    StructuredNode,
    RelationshipTo,
)
```

```
class Other(StructuredNode):
    sourceID = StringProperty()
    name = StringProperty()
    valid_until = StringProperty()
    node_id = StringProperty(index = True)
    countries = StringProperty()
    addresses = RelationshipTo('.address.Address', 'REGISTERED_ADDRESS')
```

3.4 Modify the `__init__.py` file:

Now we are going to add code to the `__init__.py`.

```
from .entity import Entity
from .address import Address
from .intermediary import Intermediary
from .officer import Officer
from .other import Other
```

This is done to simplify the imports. If you are familiar with Django and having only one model file, you might remember the imports are done like : `from .models import *` or any class you might need. Nevertheless, we have the classes inside a package call *models*, because of this if we need to import a class we have to say, for example *from .models.officer import Officer*. This is because we are one level down.

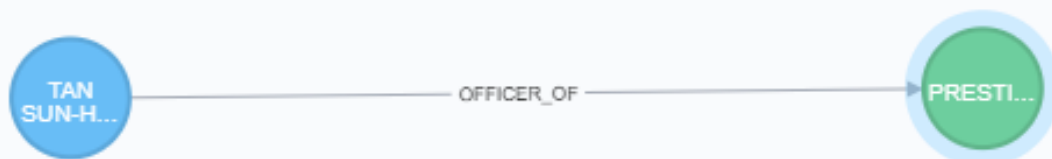
However, since we took the approach of having the imports on the `__init__.py` file, now we can import them as *from .models import Officer*

3.5 Create constraints or indexes:

Creating constraints and labels have to be done after you add/change the node definitions. The command that you will need to use is:

```
python manage.py install_labels
```

In this case, since we added `index=True` on the `node_id` property the output would create indexes on each of the property mentioned:



If it were a completely new database, Neomodel would also have created the node labels, properties and relationships.

After doing these steps, the structure folder of the project changed. Right now the structure of the `fetch_api` app should be:

```
├── fetch_api/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   └── urls.py
```

```
├── migrations/
│   └── __init__.py
├── models/
│   ├── __init__.py
│   ├── address.py
│   ├── entity.py
│   ├── intermediary.py
│   ├── officer.py
│   └── other.py
```


HOW TO QUERY THE NEO4J DATABASE WITH NEOMODEL.

4.1 Intro.

I assume that you know how to create a Django project and how to init and setup a Django app inside this project. If you don't have a clue of what I said, I recommend to come back after reading the Django intro tutorial and start again from part one of this tutorial: <https://docs.djangoproject.com/en/1.11/intro/>

At this point of the tutorial, you should have already created and setup the `paradise_paper_search` Django project and the `fetch_api` Django app (*Part 1*). You learned how to integrate the Neomodel OGM into the Django project (*Part 2*). Also, at *Part 3* you learned the way a Neo4j Graph Database is modeled using Neomodel and ended with a group of python class definitions that represent the nodes, properties and relationships of the Paradise Paper Graph Database (PPGDB).

Current project structure:

```
paradise_papers_search/
├── paradise_papers_search/
│   └── +
├── fetch_api/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   └── __init__.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── address.py
│   │   ├── entity.py
│   │   ├── intermediary.py
│   │   ├── officer.py
│   │   └── other.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
└── manage.py
```

Now, how you actually query a graph database inside your Django project or apps?

4.2 Using your models

Using your models is pretty standard. You usually just import the ones you need and use them, for example, in your Django views. Before we get to that, we need to learn the Neomodel Query API. This API will allow us to express

queries to the database without having to write them in plain Cypher.

We will learn to query using the Paradise Paper models we did before. To do that we will first use an instance of the python interpreter.

4.2.1 The project python interpreter:

Let's open the python interpreter through our Django project `manage.py` which will import our project settings(remember you set the `DATABASE_URL` in there, this is needed to connect to the db). Also, it will try to use `ipython` or `bpython` if available.

First let's start opening the console, if necessary.

Then make sure you are at the `paradise_papers_search` root directory where you created the Django project(where the `manage.py` module is).

Run the command:

```
python manage.py shell
```

With the python interpreter at hand, we can import our models and start to use them as soon as we execute the following python import commands:

```
from fetch_api.models import Entity
from fetch_api.models import Intermediary
from fetch_api.models import Officer
from fetch_api.models import Address
from fetch_api.models import Other
```

Each of the models we just imported maps to a specific structure of a node label, property keys and relationship types in the PPGDB. Now we are actually ready to start exploring the Paradise Paper Graph Database through the Neomodel Query API.

4.3 Neomodel Query API

Each of your models has some properties and methods(inherited from `StructureNode` or `DjangoNode`) that help us to express queries to the Neo4j database.

4.3.1 NodeSet and Nodes Neomodel objects

A *NodeSet* object represents a set of nodes matching common query parameters or filters.

A *Node* object is an instance of one of our models. That means we can access all the properties and methods defined on the model class. Each instance represents a single node in the database.

The `<Model>.nodes` class property of each model store a *NodeSet* object. Each time we access this `.nodes` property we get a brand new nodeset object, which means we get nodeset without any filters applied. Initially, before applying any filters, this nodeset represents all the nodes mapped under a model(nodes labeled with the same class name). For instance, `Entity.nodes` contains all the nodes with the label `Entity` on the database.

Later we will see how we can apply filters in order to match a specific subset of nodes.

4.3.2 Length of a NodeSet

If we wanted to count all the Entity nodes that are stored in the database, we just call the `len` python function over the `Entity.nodes` nodeset.

Example:

```
len(Entity.nodes)
```

When we call `len(Entity.nodes)`, Neomodel will generate a cypher query that counts all the nodes with the label `Entity`. Then that query is executed in the Neo4j database and we get back the count. The cypher query string that is generated by Neomodel behind the scene is:

```
MATCH (n:Entity) RETURN COUNT(n)
```

Note: We are not retrieving all the nodes from the database and then count them. The actual counting is done by the Neo4j database engine which is faster.

Another example, to get a count of all the nodes that exist in the PPGDB database:

```
len(Entity.nodes) \
+ len(Officer.nodes) \
+ len(Intermediary.nodes) \
+ len(Address.nodes) \
+ len(Other.nodes)
```

If nodeset is filtered, only nodes that fulfill the filters will be counted.

4.3.3 Fetching nodes

In order to retrieve the nodes, read their properties and relationships, an actual cypher query needs to be executed by Neomodel. This is handled completely by Neomodel and we just need to use its query API.

A call to the NodeSet method `.all()`, would return all the nodes of a nodeset; nevertheless this would result in an expensive query. The reason is that Neomodel will actually try to retrieve all the nodes at once. It is recommended to use `.all()` when the nodeset is small. We can reduce the size by filtering the nodeset as will see in the later.

It is better to fetch the nodes in batches from a nodeset. The NodeSet objects support the same operators for indexing and slicing just like the normal python lists.

To get the first element of the `Entity.nodes` nodeset, we can reference its index:

```
Entity.nodes[0]
```

To get a subset of nodes, we can use the python slice syntax. This is convenient for writing code that retrieves the nodes in batches. For example to get the first 10 nodes in a list:

```
Entity.nodes[0:10]
```

Note: Neomodel will generate and execute cypher query only to retrieve the nodes we are asking for. So we are not actually retrieving all the nodes at once from the database. An example of a cypher query string generated by new model would be `MATCH (n:Entity) RETURN n SKIP 10 LIMIT 10`

4.3.4 Finding nodes

If we know exactly what node we are looking for, for instance we have the `node_id` or the exact name property value, we can use the `.get()` or `.get_or_none()` nodeset methods. The difference is that if no match, the first one will raise a `DoesNotExist` exception and the second will return `None`.

To get the node which `node_id` is 160380 in a given nodeset:

```
Entity.nodes.get_or_none(node_id=160380)
Entity.nodes.get(node_id=160380)
```

Warning: These methods will raise `MultipleNodesReturned` exception if the property value used to get the node is not unique.

4.3.5 Filtering nodes

It is very probable that we want to get a subset of nodes that fulfill a specified condition. For example, getting all the Entity nodes which name property contains a specific word.

In order to filter nodes in a nodeset, we use the NodeSet method `.filter``. The filter method borrows the same django filter format with double underscore prefixed operators.

To get Entity nodes which name property has the word “financial”, we use the operator *contains*:

```
Entity.nodes.filter(name__contains='financial')
```

The above statement will return a filtered nodeset, in order to actually retrieve the data see the Fetching Nodes section. For more prefixed operators refer to this page: <http://neomodel.readthedocs.io/en/latest/queries.html#node-sets-and-filtering>

4.4 Creating some utils to search the PPGDB

The purpose of this tutorial is to show you how we can use Neomodel with Django. In order to do that we will build an app that will search the Paradise Paper Graph Database. With what we have learned so far is enough for our purpose.

We will create some function utils that will help us search the PPGDB. Later, we will find ourselves importing and using these helper functions to fetch data from the DB in our Django views.

To start coding, first let’s create a new python module under our `fetch_api/` directory. Name the file as `utils.py`

Now, as we will want to query the Neo4j database, we will import our models. Put the below import statements at the start of the *utils.py*:

```
from .models import Entity
from .models import Intermediary
from .models import Officer
from .models import Address
from .models import Other
```

In order to easily access each of the model classes programmatically, let’s create a key-value map. The key will be the model class name and the value will be the model class itself:

```
MODEL_ENTITIES = {
    'Entity': Entity,
    'Address': Address,
    'Intermediary': Intermediary,
    'Officer': Officer,
    'Other': Other
}
```

4.4.1 Filter Nodes Helper

We will create a function that receives a model class and some filter parameters like *name*, *country* *jurisdiction* and *source_id*. Then this function will return a filtered nodeset containing only the model nodes that pass our filters.

Let's add this helper function to the `utils.py`, with the name `filter_nodes`:

```
def filter_nodes(node_type, search_text, country, jurisdiction, source_id):
    node_set = node_type.nodes

    # On Address nodes we want to check the search_text against the address property
    # For any other we check against the name property
    if node_type.__name__ == 'Address':
        node_set.filter(address__icontains=search_text)
    else:
        node_set.filter(name__icontains=search_text)

    # Only entities store jurisdiction info
    if node_type.__name__ == 'Entity':
        node_set.filter(jurisdiction__icontains=jurisdiction)

    node_set.filter(countries__icontains=country)
    node_set.filter(sourceID__icontains=source_id)

    return node_set
```

4.4.2 Count Nodes Helper

We will create a function that return the length of the nodeset returned by the `filter_nodes` helper we created before. It will receive a dictionary of filters.

Here a representation of the required dictionary keys:

```
{
    'node_type': '',
    'name': '',
    'country': '',
    'jurisdiction': '',
    'sourceID': ''
}
```

Let's add this helper function to the `utils.py`, with the name `count_nodes`:

```
def count_nodes(count_info):
    count = {}
    node_type = count_info['node_type']
    search_word = count_info['name']
```

```
country          = count_info['country']
jurisdiction     = count_info['jurisdiction']
data_source      = count_info['sourceID']
node_set        = filter_nodes(MODEL_ENTITIES[node_type], search_word,
↪country, jurisdiction, data_source)
count['count']   = len(node_set)

return count
```

4.4.3 Fetch Nodes Helper

We will create a function that returns a subset of nodes filtered by the `filter_nodes` helper that we created previously. It will receive a dictionary of filters.

Here a representation of the required dictionary keys:

```
{
    'node_type': '',
    'name': '',
    'country': '',
    'jurisdiction': '',
    'sourceID': ''
    'limit': 10,
    'page': 1
}
```

The `limit` and `page` filters are necessary to calculate the `start` and `end` values that we will use to get a subset of nodes from a nodeset. Just like we learned in the Fetching Nodes section, we will return the nodes in batches using slice python syntax on the nodeset.

Let's add this helper function to the `utils.py`, with the name `fetch_nodes`:

```
def fetch_nodes(fetch_info):
    node_type      = fetch_info['node_type']
    search_word    = fetch_info['name']
    country        = fetch_info['country']
    limit          = fetch_info['limit']
    start          = ((fetch_info['page'] - 1) * limit)
    end            = start + limit
    jurisdiction   = fetch_info['jurisdiction']
    data_source    = fetch_info['sourceID']
    node_set       = filter_nodes(MODEL_ENTITIES[node_type], search_word, country,
↪jurisdiction, data_source)
    fetched_nodes  = node_set[start:end]

    return fetched_nodes
```

4.4.4 Fetch Node Details Helper

We will create a function that return a single node. It will receive a dictionary of filters with the `node_type` and the `node_id`.

Here a representation of the required dictionary keys:

```
{
    'node_type': '',
    'node_id': ''
}
```

Let's add this helper function to the `utils.py`, with the name `fetch_node_details`:

```
def fetch_node_details(node_info):
    node_type = node_info['node_type']
    node_id = node_info['node_id']
    node = MODEL_ENTITIES[node_type].nodes.get(node_id=node_id)
    node_details = node

    return node_details
```

4.4.5 Fetch countries, jurisdictions and data source helpers

As we are filtering the nodes by countries, jurisdictions and data source, we will need a list of valid filtering values.

First let's create a new python module under our `paradise_papers_search/` directory. Name the file as `constants.py`.

We will fetch this data from the database, however, we are not going to use the models. Sometimes it is convenient to make raw cypher queries to the database. Neomodel allows you to do that.

In your `constant.py` module, import the database util 'db' from `neomodel`:

```
from neomodel import db
```

Now you can use the `cypher_query` method, to execute raw cypher queries and get the results.

For example, we will query the countries, jurisdictions and data sources in the `constants.py`:

```
countries = db.cypher_query(
    '''
    MATCH (n)
    WHERE NOT n.countries CONTAINS ';'
    RETURN DISTINCT n.countries AS countries
    '''
)[0]

jurisdictions = db.cypher_query(
    '''
    MATCH (n)
    RETURN DISTINCT n.jurisdiction AS jurisdiction
    '''
)[0]

data_sources = db.cypher_query(
    '''
    MATCH (n)
    RETURN DISTINCT n.sourceID AS dataSource
    '''
)[0]
```

With the results we will make sorted lists of COUNTRIES, JURISDICTIONS and DATASOURCE:

```
COUNTRIES = sorted([country[0] for country in countries])
JURISDICTIONS = sorted([jurisdiction[0] for jurisdiction in jurisdictions if
↳ isinstance(jurisdiction[0], str)])
DATASOURCE = sorted([data_source[0] for data_source in data_sources if
↳ isinstance(data_source[0], str)])
```

In the `utils.py`, import `COUNTRIES`, `JURISDICTIONS`, `DATASOURCE`:

```
from paradise_papers_search.constants import COUNTRIES, JURISDICTIONS, DATASOURCE
```

Then create `fetch_countries`, `fetch_jurisdictions` and `fetch_data_source` helpers:

```
def fetch_countries():
    return COUNTRIES

def fetch_jurisdictions():
    return JURISDICTIONS

def fetch_data_source():
    return DATASOURCE
```

We created the `constants.py` module because we want to make the cypher queries once. Not each time we call `fetch_countries`, `fetch_jurisdictions` or `fetch_data_source` helpers.

Since these queries might take some time to execute, we want them ready at the start of the application. We can do that by executing the `constants.py` code.

Open the `fetch_api/app.py` module. Add a new method to the Django application class with the name `ready` and import the `constants.py` module. That will be enough to initialize `COUNTRIES`, `JURISDICTIONS` and `DATASOURCE` constants.

Here how the `fetch_api/app.py` would look like:

```
from django.apps import AppConfig

class FetchApiConfig(AppConfig):
    name = 'fetch_api'

    def ready(self):
        from paradise_papers_search import constants
```

4.5 Using the search utils

To use the search utils, we just need to import them into the module they will be used. In this case, we will need to import them into the `fetch_api/views.py` module. Later they will be used to create our application views.

Here the import statement, place this code in the `fetch_api/views.py` module:

```
from .utils import (
    count_nodes,
    fetch_nodes,
    fetch_node_details,
    fetch_countries,
    fetch_jurisdictions,
```



```
    fetch_data_source,  
)
```

In the next section, you will build the rest of this code.

4.5.1 Testing utils in the console

Just like we did models, we can the utils in the project python interpreter and play around.

Make sure you are at the `paradise_papers_search` root directory where you created the Django project (where the `manage.py` module is).

Run the command:

```
python manage.py shell
```

With the python interpreter at hand, we can import our utils and start to use them as soon as we execute the following python import command:

```
from .utils import (  
    count_nodes,  
    fetch_nodes,  
    fetch_node_details,  
    fetch_countries,  
    fetch_jurisdictions,  
    fetch_data_source,  
)
```

Now, for example, you can count all the nodes that pass some filters:

```
count_nodes({  
    'node_type': 'Entity',  
    'name': 'Junior',  
    'country': '',  
    'jurisdiction': '',  
    'sourceID': ''  
})
```

Or you can fetch a subset of nodes that pass some filters:

```
fetch_nodes({  
    'node_type': 'Entity',  
    'name': 'Junior',  
    'country': '',  
    'jurisdiction': '',  
    'sourceID': ''  
    'limit': 10,  
    'page': 1,  
})
```


HOW TO CREATE THE API.

5.1 Why the API structure

We decided to use APIs to fetch the data to make queries more effective, as sometimes queries get too big. We decided it was better to fetch the data in little chunks rather than one big request. The API structure allows us to do so and to have a single page application. We will be returning JSON to the calls since it is easily usable on the frontend side.

5.2 Explain the library

In this project, we are using the `rest_framework` library because it has some simple non-limiting ways to implement an API structure. If you desire you can use any of your choosing, but the examples presented ahead will be based on the behavior of this specific library.

You should have the library already installed since it is in the requirements file that we provided for the project. If not just run:

```
pip install rest_framework
```

On your console and restart your Django server.

5.3 Create the views / endpoints

We are going to need a couple different endpoints to make our application work, but for now, we will focus on one to explain the process of making them, we are going to be building the `GetNodesData / nodes` that will serve us to fetch the search matches nodes data. We'll be using a type of class-based view called `APIView` from the framework to make them, this will allow us to have an endpoint rather than a page view.

Clean your `fetch_api/views.py` file and add the following code:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class GetNodesData(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)
```

Now on your `fetch_api/urls.py` add the endpoint to the urls:

```
from django.conf.urls import url
from .views import GetNodesData

urlpatterns = [
    url(r'^nodes[/]?$', GetNodesData.as_view(), name='get_nodes_data'),
]
```

Now you can hit the endpoint with a method of your choosing, for this example, we are going to use curl, on your console:

```
curl http://127.0.0.1:8000/fetch/nodes
```

You should see:

```
"Temporary Data"
```

The other 5 endpoints with its classes/URLs respectively are:

GetNodesCount / count, to fetch the count of matches

GetNodeData / node, to fetch the info for a specific node

GetCountries / countries, to fetch the list of countries

GetJurisdictions / jurisdictions, to fetch the list of jurisdictions

GetDataSource / datasource, to fetch the list of data sources

When you finish your `fetch_api/views.py` file should look like this:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class GetNodesCount(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)

class GetNodesData(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)

class GetNodeData(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)

class GetCountries(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)

class GetJurisdictions(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)

class GetDataSource(APIView):
    def get(self, request):
        return Response('Temporary Data', status=status.HTTP_200_OK)
```

And the `fetch_api/urls.py` should look like this:

```
from django.conf.urls import url
from .views import (
    GetNodesCount,
    GetNodesData,
    GetNodeData,
    GetCountries,
    GetJurisdictions,
    GetDataSource,
)

urlpatterns = [
    url(r'^count[/]?$', GetNodesCount.as_view(), name='get_nodes_count'),
    url(r'^nodes[/]?$', GetNodesData.as_view(), name='get_nodes_data'),
    url(r'^node[/]?$', GetNodeData.as_view(), name='get_node_data'),
    url(r'^countries[/]?$', GetCountries.as_view(), name='get_countries'),
    url(r'^jurisdictions[/]?$', GetJurisdictions.as_view(), name='get_jurisdictions'),
    url(r'^datasource[/]?$', GetDataSource.as_view(), name='get_data_source'),
]
```


HOW TO SERIALIZE THE DATA.

6.1 Why we need to serialize the data

The nodes that we are fetching so far are just instances of the model classes we defined (Entity, Officer, etc), we want to return JSON files to the fetch calls that are going to be made in the frontend. Python objects structure is not JSON serializable and the nodes don't support the already build in serializers on the library, so we have to create our own serialize methods and manage the logic of those since sometimes we would need to serialize only the directly related nodes to a specific node, to serialize the data is nothing more than getting it from the object format to a parseable format.

6.2 Create serialize methods

The way we are going to serialize the data is using property methods on each model, to then call them on the helper functions. Adding a serialize method to the model classes is a convenient way to serialize the nodes objects because each node will know how to translate itself. We are going to be using the Entity model for these examples since this one presents all the cases contemplated in this tutorial.

In your `fetch_api/models/entity.py` add the following as a method of the Entity class:

```
@property
def serialize(self):
    return {
        'node_properties': {
            'sourceID': self.sourceID,
            'address': self.address,
            'jurisdiction': self.jurisdiction,
            'service_provider': self.service_provider,
            'countries': self.countries,
            'jurisdiction_description': self.jurisdiction_description,
            'valid_until': self.valid_until,
            'ibcRUC': self.ibcRUC,
            'name': self.name,
            'country_codes': self.country_codes,
            'incorporation_date': self.incorporation_date,
            'node_id': self.node_id,
            'status': self.status,
        },
    }
```

If you look at this closely you'll notice this is just all the properties on the Entity class mapped to a dictionary but the relationships. We'll explain later why we don't serialize the relationships here and how to do it, you will need to do the same for the other model classes:

Address:

```
@property
def serialize(self):
    return {
        'node_properties': {
            'sourceID': self.sourceID,
            'country_codes': self.country_codes,
            'valid_until': self.valid_until,
            'address': self.address,
            'countries': self.countries,
            'node_id': self.node_id,
        },
    }
```

Intermediary:

```
@property
def serialize(self):
    return {
        'node_properties': {
            'sourceID': self.sourceID,
            'valid_until': self.valid_until,
            'name': self.name,
            'country_codes': self.country_codes,
            'countries': self.countries,
            'node_id': self.node_id,
            'status': self.status,
        },
    }
```

Officer:

```
@property
def serialize(self):
    return {
        'node_properties': {
            'sourceID': self.sourceID,
            'name': self.name,
            'country_codes': self.country_codes,
            'valid_until': self.valid_until,
            'countries': self.countries,
            'node_id': self.node_id,
        },
    }
```

Other:

```
@property
def serialize(self):
    return{
        'node_properties': {
            'sourceID': self.sourceID,
            'name': self.name,
            'countries': self.countries,
            'valid_until': self.valid_until,
            'node_id': self.node_id,
        },
    }
```


6.3 Call the serialize methods on the helpers

Now instead of your `fetch_nodes` function on your `fetch_api/utils.py` looking like this:

```
def fetch_nodes(fetch_info):
    node_type      = fetch_info['node_type']
    search_word    = fetch_info['name']
    country        = fetch_info['country']
    limit          = fetch_info['limit']
    skip           = ((fetch_info['skip'] - 1) * limit)
    jurisdiction    = fetch_info['jurisdiction']
    node_set       = filter_nodes(MODEL_ENTITIES[node_type], search_word, country,
    ↪ jurisdiction)
    node_set.limit = limit
    node_set.skip  = skip
    fetched_nodes  = node_set.all()

    return fetched_nodes
```

Change the return statement to:

```
return [node.serialize for node in fetched_nodes]
```

That's just going to create a new array of dictionaries with the values of the serialized nodes.

And your `fetch_node_details` function looking like this:

```
def fetch_node_details(node_info):
    node_type      = node_info['node_type']
    node_id        = node_info['node_id']
    node           = MODEL_ENTITIES[node_type].nodes.get(node_id=node_id)

    return node
```

Should look like this:

```
def fetch_node_details(node_info):
    node_type      = node_info['node_type']
    node_id        = node_info['node_id']
    node           = MODEL_ENTITIES[node_type].nodes.get(node_id=node_id)
    node_details   = node.serialize

    return node_details
```

Basically doing the same as before but for a single node instead of a set.

6.4 Create the serialize relationships methods

We are not serializing the relationships along with the properties because:

- That would create a loophole between nodes
- You don't always need the relationships

So, with this said, this is how we are gonna serialize the relationships, add the following to the file `fetch_api/models/entity.py`:

```
from .nodeutils import NodeUtils
```

And to your `Entity` class:

```
class Entity(StructuredNode, NodeUtils):

    ...

    @property
    def serialize_connections(self):
        return [
            {
                'nodes_type': 'Officer',
                'nodes_related': self.serialize_relationships(self.officers.all(),
↪ 'OFFICER_OF'),
            },
            {
                'nodes_type': 'Intermediary',
                'nodes_related': self.serialize_relationships(self.intermediaries.all(),
↪ 'INTERMEDIARY_OF'),
            },
            {
                'nodes_type': 'Address',
                'nodes_related': self.serialize_relationships(self.addresses.all(),
↪ 'REGISTERED_ADDRESS'),
            },
            {
                'nodes_type': 'Other',
                'nodes_related': self.serialize_relationships(self.others.all(),
↪ 'CONNECTED_TO'),
            },
            {
                'nodes_type': 'Entity',
                'nodes_related': self.serialize_relationships_of_type('Entity'),
            },
        ]
```

You will need to do the same for the other model classes:

Address:

```
@property
def serialize_connections(self):
    return [
        {
            'nodes_type': 'Officer',
            'nodes_related': self.serialize_relationships(self.officers.all(),
↪ 'REGISTERED_ADDRESS'),
        },
        {
            'nodes_type': 'Intermediary',
            'nodes_related': self.serialize_relationships(self.intermediaries.all(),
↪ 'REGISTERED_ADDRESS'),
        },
    ]
```

Intermediary:

```
@property
def serialize_connections(self):
    return [
        {
            'nodes_type': 'Entity',
            'nodes_related': self.serialize_relationships(self.entities.all(),
↪ 'INTERMEDIARY_OF'),
        },
        {
            'nodes_type': 'Address',
            'nodes_related': self.serialize_relationships(self.addresses.all(),
↪ 'REGISTERED_ADDRESS'),
        },
        {
            'nodes_type': 'Officer',
            'nodes_related': self.serialized_realtionships_of_type('Officer'),
        },
    ]
```

Officer:

```
@property
def serialize_connections(self):
    return [
        {
            'nodes_type': 'Address',
            'nodes_related': self.serialize_relationships(self.addresses.all(),
↪ 'REGISTERED_ADDRESS'),
        },
        {
            'nodes_type': 'Entity',
            'nodes_related': self.serialize_relationships(self.entities.all(),
↪ 'OFFICER_OF'),
        },
        {
            'nodes_type': 'Officer',
            'nodes_related': self.serialized_realtionships_of_type('Officer'),
        },
    ]
```

Other:

```
@property
def serialize_connections(self):
    return [
        {
            'nodes_type': 'Officer',
            'nodes_related': self.serialized_realtionships_of_type('Officer'),
        },
        {
            'nodes_type': 'Entity',
            'nodes_related': self.serialized_realtionships_of_type('Entity'),
        },
        {
            'nodes_type': 'Address',
            'nodes_related': self.serialize_relationships(self.addresses.all(),
↪ 'REGISTERED_ADDRESS'),
        },
    ]
```

```
    },  
]
```

6.5 Call the serialize relationships methods on the helpers

On your `fetch_api/utils.py` the `fetch_node_details` function change put this above the return statement:

```
# Make sure to return an empty array if not connections  
node_details['node_connections'] = []  
if (hasattr(node, 'serialize_connections')):  
    node_details['node_connections'] = node.serialize_connections
```

So the function should be looking something like this:

```
def fetch_node_details(node_info):  
    node_type      = node_info['node_type']  
    node_id        = node_info['node_id']  
    node           = MODEL_ENTITIES[node_type].nodes.get(node_id=node_id)  
    node_details   = node.serialize  
  
    # Make sure to return an empty array if not connections  
    node_details['node_connections'] = []  
    if (hasattr(node, 'serialize_connections')):  
        node_details['node_connections'] = node.serialize_connections  
  
    return node_details
```

6.6 Return the json to the frontend

Now if you call your functions `fetch_node_details` and `fetch_nodes` should be returning the same data but in a way that is JSON parsable, so let's change a couple things in order of returning this data that we need.

In your project settings file for us it is `paradise_papers_search/settings/base.py` but if you are using a costume project it may be different for you, add the following:

```
# Rest-Framework settings  
REST_FRAMEWORK = {  
    'DEFAULT_RENDERER_CLASSES': (  
        'rest_framework.renderers.JSONRenderer',  
    )  
}
```

This is going to cause that when we invoke the render method on our APIs views the method will call the JSON renderer rather than just the HTTP one.

So once again we will clean our `fetch_api/views.py` file and leave it like this:

```
from rest_framework.views import APIView  
from rest_framework.response import Response  
from .models.helpers import fetch_nodes  
  
class GetNodesData(APIView):
```

```
def get(self, request):
    fetch_info = {
        'node_type': request.GET.get('t', 'Entity'),
        'name': request.GET.get('q', ''),
        'country': request.GET.get('c', ''),
        'jurisdiction': request.GET.get('j', ''),
        'sourceID': request.GET.get('s', ''),
        'limit': 10,
        'skip': int(request.GET.get('p', 1)),
    }
    nodes = fetch_nodes(fetch_info)
    data = {
        'response': {
            'status': '200',
            'rows': len(nodes),
            'data': nodes,
        },
    }
    return Response(data)
```

Here we are just taking the query parameters off the request and parsing them to pass them to the respective fetch function.

This lines here:

```
'limit': 10,
'skip': int(request.GET.get('p', 1)),
```

The 'limit' property is in charge of determinating how much nodes are going to be fetched from the database, we set that value to 10 since we decided it was a good balance between performance and enough information. The 'skip' property is how many sets nodes(10 nodes) are going to be skip to start fetching, this is basically working as a pagination here.

Now if you try this:

```
curl http://127.0.0.1:8000/fetch/nodes?q=maria&t=Entity
```

You should see something like this:

```
{
  "response": {
    "data": [
      {
        "node_properties": {
          "valid_until": "The Panama Papers data is current through 2015",
          "name": "MARIANTHI LIMITED",
          "jurisdiction_description": "Seychelles",
          "service_provider": "Mossack Fonseca",
          "incorporation_date": "14-JUL-2009",
          "countries": "United Arab Emirates",
          "country_codes": "ARE",
          "ibcRUC": "063736",
          "address": "OMNI MANAGEMENT CONSULTANCY FZE OFFICE NUMBER 425;
↪RAS AL KHAIMAH FREE TRADE ZONE AUTHORITY GOVERNMENT OF RAS AL KHAIMAH; P.O. BOX
↪10055 RAS AL KHAIMAH; UNITED ARAB EMIRATES",
          "status": "Defaulted",
          "node_id": "10026610",
          "jurisdiction": "SEY",
          "sourceID": "Panama Papers"
```

```

    }
  },
  {
    "node_properties": {
      "valid_until": "The Panama Papers data is current through 2015",
      "name": "LUZMARIA S.A.",
      "jurisdiction_description": "Seychelles",
      "service_provider": "Mossack Fonseca",
      "incorporation_date": "24-JAN-2013",
      "countries": "Luxembourg",
      "country_codes": "LUX",
      "ibcRUC": "118634",
      "address": "EFG BANK (LUXEMBOURG9 S.A. ATTN: PIERRE AVIRON-VIOLET;
↪ 14, ALLÉE MARCONI; L - 2013 LUXEMBOURG LUXEMBOURG",
      "status": "Active",
      "node_id": "10027827",
      "jurisdiction": "SEY",
      "sourceID": "Panama Papers"
    }
  },
  {
    "node_properties": {
      "valid_until": "The Panama Papers data is current through 2015",
      "name": "NUMMARIA LIMITED",
      "jurisdiction_description": "Niue",
      "service_provider": "Mossack Fonseca",
      "incorporation_date": "23-OCT-1997",
      "countries": "United Kingdom",
      "country_codes": "GBR",
      "ibcRUC": "002351",
      "address": "HOLLINGWORTH CONSULTANTS LTD. PARKVIEW HOUSE,
↪ BUCCLEUCH ROAD HAWICK; ROXBURGHSHIRE SCOTLAND; TD9 0EL",
      "status": "Defaulted",
      "node_id": "10036241",
      "jurisdiction": "NIUE",
      "sourceID": "Panama Papers"
    }
  },
  {
    "node_properties": {
      "valid_until": "The Panama Papers data is current through 2015",
      "name": "GRUPO NUMMARIA S.L.",
      "jurisdiction_description": "Niue",
      "service_provider": "Mossack Fonseca",
      "incorporation_date": "12-FEB-1996",
      "countries": "United Kingdom",
      "country_codes": "GBR",
      "ibcRUC": "000737",
      "address": "HOLLINGWORTH CONSULTANTS LTD. PARKVIEW HOUSE,
↪ BUCCLEUCH ROAD HAWICK; ROXBURGHSHIRE SCOTLAND; TD9 0EL",
      "status": "Defaulted",
      "node_id": "10036779",
      "jurisdiction": "NIUE",
      "sourceID": "Panama Papers"
    }
  },
  {
    "node_properties": {

```

```

        "valid_until": "The Panama Papers data is current through 2015",
        "name": "MARIACHI CORP.",
        "jurisdiction_description": "Niue",
        "service_provider": "Mossack Fonseca",
        "incorporation_date": "09-AUG-1999",
        "countries": "Belize",
        "country_codes": "BLZ",
        "ibcRUC": "004700",
        "address": "BOND & COMPANY 35 BARRACK ROAD BELIZE CITY BELIZE*S.I.
↪*",
        "status": "Defaulted",
        "node_id": "10040810",
        "jurisdiction": "NIUE",
        "sourceID": "Panama Papers"
    }
},
{
    "node_properties": {
        "valid_until": "The Panama Papers data is current through 2015",
        "name": "M.P. MARIANNE S.A.",
        "jurisdiction_description": "Panama",
        "service_provider": "Mossack Fonseca",
        "incorporation_date": "02-AUG-2007",
        "countries": "Switzerland",
        "country_codes": "CHE",
        "ibcRUC": "51",
        "address": "UNION BANCAIRE PRIVÉE UBP (SWITZERLAND) ATTN: MR.
↪FABIEN DE FRAIPONT RUE DU RHÔNE 96-98 CP 1320 CH-1211 GENEVA 1 SWITZERLAND GENEVE,
↪SWITZERLAND",
        "status": "Changed agent",
        "node_id": "10053581",
        "jurisdiction": "PMA",
        "sourceID": "Panama Papers"
    }
},
{
    "node_properties": {
        "valid_until": "The Panama Papers data is current through 2015",
        "name": "MARIADA HOLDINGS LIMITED",
        "jurisdiction_description": "British Virgin Islands",
        "service_provider": "Mossack Fonseca",
        "incorporation_date": "21-JUL-1995",
        "countries": "Switzerland",
        "country_codes": "CHE",
        "ibcRUC": "156189",
        "address": "PRIMEWAY S.A. 7, RUE DU RHÔNE 1204 GENEVE SWITZERLAND
↪",
        "status": "Changed agent",
        "node_id": "10064371",
        "jurisdiction": "BVI",
        "sourceID": "Panama Papers"
    }
},
{
    "node_properties": {
        "valid_until": "The Panama Papers data is current through 2015",
        "name": "MARIANNE PROPERTIES LIMITED",
        "jurisdiction_description": "British Virgin Islands",

```

```

        "service_provider": "Mossack Fonseca",
        "incorporation_date": "02-JUL-1992",
        "countries": "Guernsey",
        "country_codes": "GGY",
        "ibcRUC": "65048",
        "address": "KLEINWORT BENSON (GUERNSEY) TRUSTEES LIMITED P.O. BOX
↪44 WESTBOURNE; THE GRANGE ST. PETER PORT; GUERNSEY GY1 3BG CHANNEL ISLANDS ATTN:
↪MS. TINA BROWNING",
        "status": "Defaulted",
        "node_id": "10057577",
        "jurisdiction": "BVI",
        "sourceID": "Panama Papers"
    }
},
{
    "node_properties": {
        "valid_until": "The Panama Papers data is current through 2015",
        "name": "GRUPPO NUMMARIA LTD.",
        "jurisdiction_description": "British Virgin Islands",
        "service_provider": "Mossack Fonseca",
        "incorporation_date": "25-NOV-1993",
        "countries": "United Kingdom",
        "country_codes": "GBR",
        "ibcRUC": "101294",
        "address": "AUSKERRY INVESTMENTS LIMITED 1, PARK ROAD LONDON NW1
↪6XN ENGLAND",
        "status": "Defaulted",
        "node_id": "10060886",
        "jurisdiction": "BVI",
        "sourceID": "Panama Papers"
    }
},
{
    "node_properties": {
        "valid_until": "The Panama Papers data is current through 2015",
        "name": "F.S.C. LTD.-MARIAH OVERSEAS LIMITED",
        "jurisdiction_description": "British Virgin Islands",
        "service_provider": "Mossack Fonseca",
        "incorporation_date": "04-JAN-1994",
        "countries": "Switzerland",
        "country_codes": "CHE",
        "ibcRUC": "103935",
        "address": "GOTTHARDSTRASSE 57 6045 MEGGEN SWITZERLAND",
        "status": "Active",
        "node_id": "10064221",
        "jurisdiction": "BVI",
        "sourceID": "Panama Papers"
    }
}
],
"rows": 10,
"status": "200"
}
}

```

You will need to do the same for each of the other endpoints with its corresponding fetch function.

Your file should look like this at the end:


```

from rest_framework.views import APIView
from rest_framework.response import Response
from .models.helpers import (
    count_nodes,
    fetch_nodes,
    fetch_node_details,
    fetch_countries,
    fetch_jurisdictions,
    fetch_data_source,
)

class GetNodesCount(APIView):
    def get(self, request):
        count_info = {
            'node_type': request.GET.get('t', 'Entity'),
            'name': request.GET.get('q', ''),
            'country': request.GET.get('c', ''),
            'jurisdiction': request.GET.get('j', ''),
            'sourceID': request.GET.get('s', ''),
        }
        count = count_nodes(count_info)
        data = {
            'response': {
                'status': '200',
                'data': count,
            },
        }
        return Response(data)

class GetNodesData(APIView):
    def get(self, request):
        fetch_info = {
            'node_type': request.GET.get('t', 'Entity'),
            'name': request.GET.get('q', ''),
            'country': request.GET.get('c', ''),
            'jurisdiction': request.GET.get('j', ''),
            'sourceID': request.GET.get('s', ''),
            'limit': 10,
            'skip': int(request.GET.get('p', 1)),
        }
        nodes = fetch_nodes(fetch_info)
        data = {
            'response': {
                'status': '200',
                'rows': len(nodes),
                'data': nodes,
            },
        }
        return Response(data)

class GetNodeData(APIView):
    def get(self, request):
        node_info = {
            'node_type': request.GET.get('t', 'Entity'),
            'node_id': int(request.GET.get('id')),
        }
        node_details = fetch_node_details(node_info)
        data = {

```

```
        'response': {
            'status': '200',
            'data': node_details,
        },
    }
    return Response(data)

class GetCountries(APIView):
    def get(self, request):
        countries = fetch_countries()
        data = {
            'response': {
                'status': '200',
                'data': countries,
            },
        }
    return Response(data)

class GetJurisdictions(APIView):
    def get(self, request):
        jurisdictions = fetch_jurisdictions()
        data = {
            'response': {
                'status': '200',
                'data': jurisdictions,
            },
        }
    return Response(data)

class GetDataSource(APIView):
    def get(self, request):
        data_source = fetch_data_source()
        data = {
            'response': {
                'status': '200',
                'data': data_source,
            },
        }
    return Response(data)
```

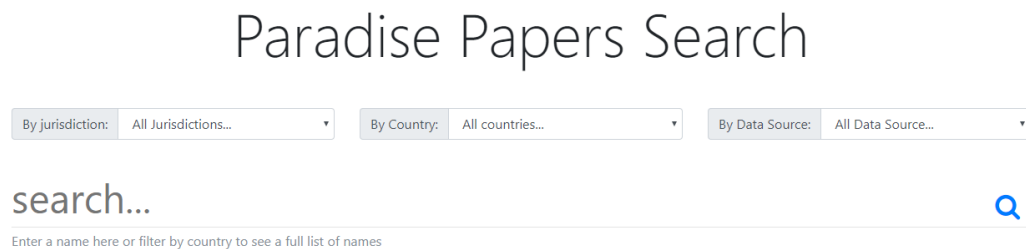
THE SEARCH APP, CONSUMING THE FETCH API.

At this point, the API is ready. Congratulation!

The purpose of this tutorial was to only show you how to use Django + Neomodel to query a Neo4j database. We have covered that while making the API backend. Now it is time to consume the API.

We have provided a single web application that will search the Paradise Paper Graph Database. This application basically just makes requests to our API and display the response data in a convenient way.

Here are some screenshots of the app.



The screenshot shows the 'Paradise Papers Search' web application. At the top, there are three filter dropdown menus: 'By jurisdiction: All Jurisdictions...', 'By Country: All countries...', and 'By Data Source: All Data Source...'. Below these is a large search input field with the placeholder text 'search...'. Underneath the search field is a smaller line of text: 'Enter a name here or filter by country to see a full list of names'. To the right of the search field is a blue magnifying glass icon.

Fig. 7.1: That is Search Home, where we can input the search text and filters.

The tools used to build the Paradise Paper Search App frontend are probably pretty familiar to you (JavaScript, KnockoutJS, JQuery, HTML, Bootstrap). We will not give many details about its workings, but you can take a look if you want to.

The search app communicates with the API through GET HTTP requests. We are using the JQuery `$.getJSON` ajax shorthand method to construct these requests. That way we load JSON-encoded data from our Fetch API.

Here are some examples we actually use in the app.

Getting some nodes:

Entity 254 Officer 106 Intermediary 25 Address 150 Other 5

	Incorporation	Jurisdiction	Linked To	Data From
BIG APPLE INC.	05-OCT-2000	PMA	Costa Rica	Panama Papers
APPLEBY HOLDINGS CORP.	10-APR-2001	PMA	Switzerland	Panama Papers
APPLEBY LTD.	25-APR-2005	SEY	Monaco	Panama Papers
APPLEBY HOUSE LTD.	24-OCT-2005	SEY	Monaco	Panama Papers
APPLEGATE CORPORATE HOLDING LTD.	19-JUN-2002	SEY	Samoa	Panama Papers
APPLEWHITE CORP.	05-JAN-2007	SEY	Monaco	Panama Papers
GREEN APPLE CORP.	08-JUL-2009	SEY	Luxembourg	Panama Papers
APPLETON LIMITED	20-FEB-1998	NIUE	Guernsey	Panama Papers
APPLEBLOSSOM S. A.	02-SEP-1982	PMA	Monaco	Panama Papers
APPLEBY BUSINESS INC.	26-JAN-2005	PMA	Monaco	Panama Papers

Load More

Fig. 7.2: That is Search Results. There, we can see nodes that matched our search filters. They are presented in tabular format.

Entity 254 Officer 106 Intermediary 25 Address 150 Other 5

Entity

BIG APPLE INC.

Connected to 1 Intermediary

Incorporated: 05-OCT-2000

Status: Defaulted

Registered in: Panama

Linked countries: Costa Rica

Data from: Panama Papers

Agent: Mossack Fonseca

The Panama Papers data is current through 2015

Connections

Intermediary

	Linked To	Data From
VOYAGEUR FOUNDATION	Costa Rica	Panama Papers

Close

Load More

Fig. 7.3: That is modal with more details of a specific node. We will see, for example, the connections of Entity node with others Officer or Intermediary nodes.

```
$.getJSON(
  'fetch/nodes',
  {
    'q': 'apple',      // search text
    'c': 'Monaco',     // country
    'j': 'PMA',        // jurisdiction
    's': '',           // source
    'p': 1             // page
  }
)
.done(nodes => {
  // Do something with each node
  nodes.response.data.forEach(node => {
    console.log(node.node_properties);
  });
})
.fail(() => {
  // Handle errors
  console.log("Fetch error");
})
.always(() => {
  // Always do something at end
});
```

Getting node details:

```
$.getJSON(
  'fetch/node',
  {
    'id': 10033457,    // node id
    't': 'Entity',     // node type
  }
)
.done(node => {
  // Do something with node
  console.log(node);
})
.fail(() => {
  // Handle errors
  console.log("Fetch error");
})
.always(() => {
  // Always do something at end
});
```

Currently the Search App is not making any requests. Instead, it is using some mockup data. But now that we have Fetch API working, we can use that.

The code that made the requests is already in place, but It is not executed because we load or return mockup data first. In order to change that, go ahead and remove or comment lines of code that load or return the mockup data.

The file that you need to change is the `paradise_papers_search/static/js/search.js`.

Check this screenshots to know what lines you need to remove. They are highlighted in green.

```
207     fetch () {
208 +       // Test code
209 +       let mock = mockup_data[this._node_type].data;
210 +       mock.forEach(row => {
211 +         this._nodeSearchData.push(new Node(this._node_type, row.node_properties));
212 +       });
213 +       return;
214 +       // /Test code
215     this._fetchState(true);
216
217     $.getJSON(
218
219
220
221
222
223
224
225
226   }
227
228   fetchCount() {
229 +     // Mock Data
230 +     return this._nodeSearchDataCount(0);
231 +
232
233     $.getJSON(
234       this._search_api + 'count',
235       this._search_filters
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253   }
254
```

```
362     fetchCountries() {
363 +         // Test code
364 +         mockup_data.countries.forEach(country => {
365 +             this._countryList.push(country);
366 +         });
367 +
368 +         return this._countryList;
369 +         // /Test code
370     $.getJSON(
371         'fetch/countries'
372     )
373
374
375
376
377
378 }
379
380
381
382
383
384     fetchJurisdictions() {
385 +         // Test code
386 +         mockup_data.jurisdictions.forEach(jurisdiction => {
387 +             this._jurisdictionList.push(jurisdiction);
388 +         });
389 +         // /Test code
390 +         return this._jurisdictionList;
391 +
392     $.getJSON(
393         'fetch/jurisdictions'
394     )
395
396
397
398
399
400 }
401
402
403
404 }
405
406     fetchDataSource() {
407 +         // mock data
408 +         mockup_data.dataSource.forEach(dataSource => {
409 +             this._dataSourceList.push(dataSource);
410 +         });
411 +
412 +         return this._dataSourceList;
413 +
414     $.getJSON(
415         'fetch/datasource'
416     )
```

After that, the application should work.

Go ahead and play around!