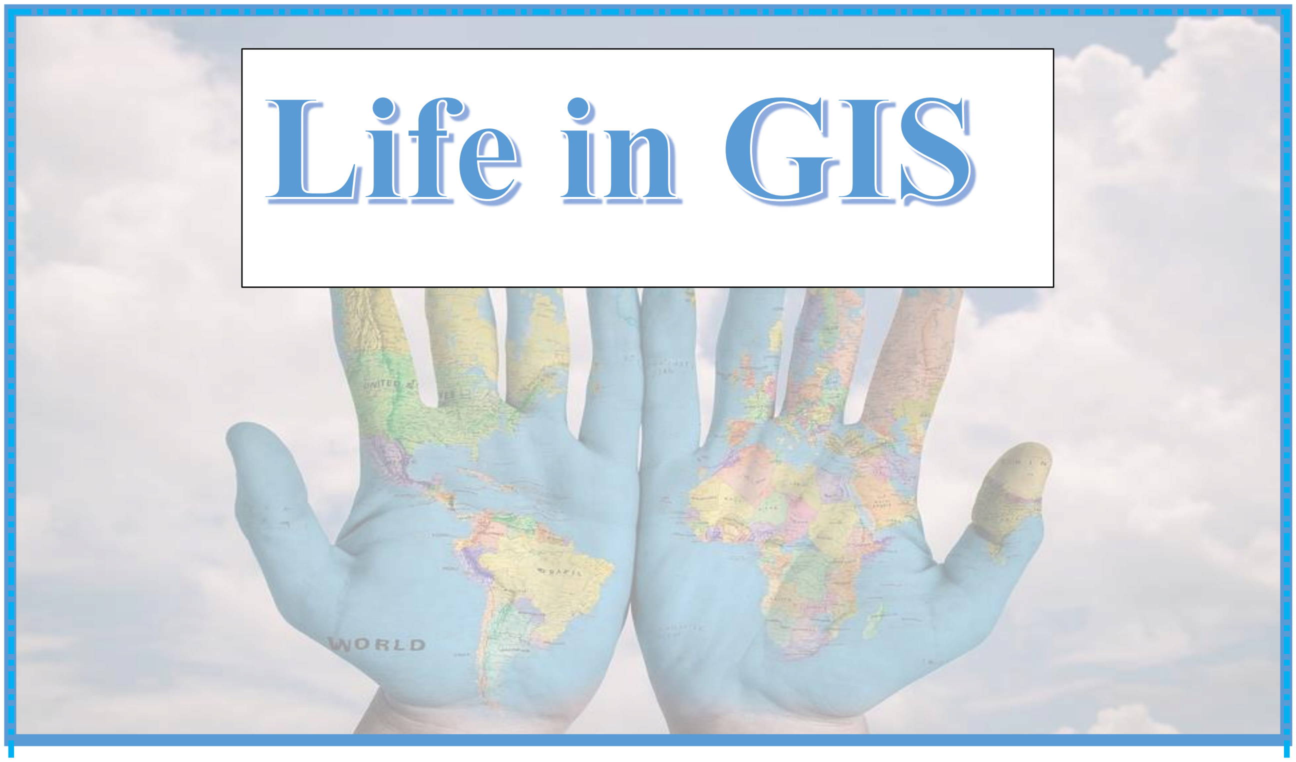
Life in GIS



GeoDjango Tutorial Guide

March 2019

By:

Wanjohi Kibui

swanjohi@lifeingis.com

+254 719 696 921

www.lifeingis.com

Table of Contents

Introduction

Hello there, my name is Wanjohi Kibui, the owner of the Life in GIS website and Initiative. I work as a GIS consultant. My focus is on Web GIS development and GIS software training. Python is my language, GIS is my life.

I started some years ago, while in college, practicing on different aspects of GIS such as system development, training, blogging and much more. I got my first GIS jobs while in college and consulted for a number of companies back then.

A few years later, being in the industry, providing solutions for different entities and individuals has always been my goal. I spend most of my days doing GIS stuff in different organizations and regions.

Life in GIS is a phrase that came with time, signifying my day-to-day works with GIS. It comes along with the slogan, ***GIS for a better today***, which I can't really tell its source but, it's self-explanatory.

I started my YouTube channel to share my knowledge in GIS programming and specifically to help fellow developers out there who might be struggling with problems that I went through or that exist in this field. It's been a great channel to me; it brings joy, connections, jobs and much more.

Make sure you subscribe to the YouTube Channel and for a newsletter at the Life in GIS website.

Disclaimer

The contents of this eBook are for training and development purposes only. This is intended to serve as a starting point to anyone who wishes to learn web mapping in GeoDjango. The goals of every developer may vary hence this should act as a guide to your goals.

The content of this document is a property of **Wanjohi Kibui** of **Life in GIS**. Please use, share, improve but don't claim its ownership.

This guide is an attempt to build a replica of the YouTube channel at Life in GIS

Informer

To start it all, let me share this short paragraph from HowtoForge

Django is a web application framework written in python that follows the MVC (Model-View-Controller) architecture, it is available for free and released under an open source license. It is fast and designed to help developers get their application online as quickly as possible. Django helps developers to avoid many common security mistakes like SQL Injection, XSS, CSRF and clickjacking. Django is maintained by the Django Software Foundation and used by many big technology companies, government, and other organizations. Some large websites like Pinterest, Mozilla, Instagram, Discuss, The Washington Post etc. are developed with Django.

GeoDjango is a geographic web framework that helps users build GIS web applications and handle spatially enabled data in their applications. It's a module within the Django framework. GeoDjango has grown over time and has enabled many developers embed mapping solutions within their project. A number of APIs have been released such as the GeoDjango Database API, forms API, model API e.t.c

In this guide, I demonstrate how to develop a web application that supports spatial data using the Django framework and GeoDjango module in specific.

We will be using;

1. **Framework** - Django/Geodjango
2. **Database** - PostgreSQL/PostGIS
3. **Code Editor** - Microsoft VS Code
4. **Browser** - Google Chrome

The requirements 3 and 4 above are not mandatory. Choose your own weapon.

This guide is built with some assumptions;

a) You have basics in Python programming language

b) Have handled spatial data before or have a hint what spatial data entails

c) Have introduced yourself to Django framework

d) You are passionate to learn how GeoDjango works

e) You are an expert programmer, NOPE – I'm kidding, you actually need to know how stuff works

I hope we are into this. Right? Let's proceed!!

Chapter One

Setting up

In this chapter, we will setup our development environment and start a Django project that we will build on throughout this guide. As explained in the Django project site, Django is fast, scalable and flexible. In this case, setting up a Django development environment is no rocket science. We will begin with this.

There are two ways for setting up a Django environment on your localhost.

a) Install Django on the base operating system
b) Install Django on a virtual-environment

I prefer the later as it keeps your OS clean and free from possible corruption by Python dependencies installed and configured when working with Django. It also allows multiple projects on the same system.

Install Python

It's necessary to check whether Python is installed on your system and if not, go ahead and install it.

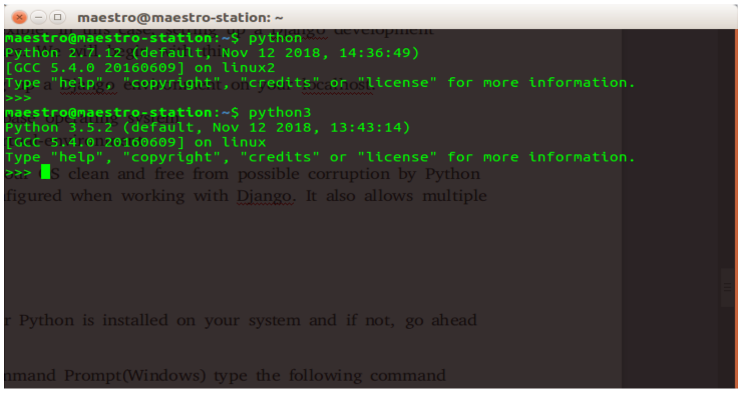
On the Terminal (Linux) type the following command

```
python     #This command outputs the version of Python installed on your system
```

On Windows, use

```
python --version
```

The command should return the version of Python installed on your computer or an error meaning that Python is not installed or the paths are not well set.



```
maestro@maestro-station:~$ python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
maestro@maestro-station:~$ python3
Python 3.5.2 (default, Nov 12 2018, 13:43:14)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Check to ensure you have the version of Python that you need. In this guide, we will be using Python 3.x

In Linux, Python is installed by default. On Windows, you can follow the guide at How To Geek to install Python on your system.

We will be installing Django in a virtual environment. So, we install **virtualenv** first; we will install virtualenv using pip, which is not yet installed. Here is the procedure;

Install Pip(Ubuntu)

```
apt-get install python-pip #For python 2

apt-get install python3-pip #for python 3
```

Install Pip(Windows)

1. Download get-pip.py to a folder on your computer.
2. Open a command prompt and navigate to the folder containing get-pip.py.
3. Run the following command:
4. python get-pip.py
5. Pip is now installed. You can confirm by running pip -V on cmd

Install Virtualenv and Virtualenvwrapper

These will help us create dedicated environment for our project. You can install both or just the virtualenv.

On Linux,

```
pip install virtualenv

pip install virtualenvwrapper
```

On Windows,

```
pip install virtualenvwrapper-win
```

Create a virtual environment

Navigate to the folder where you want your envs to live

```
virtualenv geo     #Give a name to your virtualenv. Just replace geo with your favorite name
```

On Windows, can do,

```
mkvirtualenv geo
```

Activate your virtual environment

Source *geo/bin/activate* or . *geo/bin/activate #don't forget the period*

In you installed virtualenvwrapper, use

```
workon geo
```

To deactivate the virtual environment, use

```
deactivate
```

We have the virtual environment set and activate. Let's install Django.

```
pip install django   #This will install the latest version of Django available.
```

To restrict the version use

```
pip install Django==2.0.5    #replace the number with the version you need
```

To check version of Django installed

```
django-admin --version
```

Spatial Databases

In order to store spatial data for our project, we will use PostGIS, an extension in PostgreSQL. PostGIS extends PostgreSQL database and ensures spatial data can be stored, queried and manipulated.

To work with Django and PostgreSQL, we need pyscopg2 installed. Psycopg2 is a database adapter for the Python language. In this guide we will be working with version 2.7.7 of the Psycopg2.

To install psycopg2;

```
pip install psycopg2==2.7.7
```

After installing Psycopg2, we have to install **PostgreSQL** and **PostGIS**.

We will be using PostgreSQL 9.6. I like tutoring with versions that I have used over time and comfortable with. You can use any version from 9.5 that you are comfortable with.

We have to add the official PostgreSQL Apt Repository to our sources.list:

```
sudo add-apt-repository "deb http://apt.postgresql.org/pub/repos/apt/ xenial-pgdg main"
```

Import the relevant signing key:

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
```

Update your packages:

```
sudo apt update
```

Start installing PostgreSQL 9.6 and the “contrib” package to add additional utilities and functionality to the database:

```
sudo apt install postgresql-9.6 postgresql-contrib-9.6
```

Check your PostgreSQL Version:

```
psql --version
```

The output should look somehow like this:

```
psql (PostgreSQL) 9.6.2
```

Create a new database user (replace “wanjohi” with your name):

```
sudo -u postgres createuser -P wanjohi
```

You will be prompted for a password. As always: Use a strong password here!

Create a new database (replace “wanjohi” with your username and “geo” with whatever you want to name your database):

```
sudo -u postgres createdb -O wanjohi geo
```

Test if your database works correctly:

```
psql -h localhost -U wanjohi geo
```

Exit psql:

```
\q
```

Now, let’s add PostGIS support to our database:

Add UbuntuGIS-unstable repository and update packages:

```
sudo add-apt-repository ppa:ubuntugis/ubuntugis-unstable

sudo apt update
```

Install PostGIS:

```
sudo apt install postgis postgresql-9.6-postgis-2.3
```

Create extensions for your postgres database:

```
sudo -u postgres psql -c "CREATE EXTENSION postgis; CREATE EXTENSION
postgis_topology;" geo
```

Kudos, we have set up everything required for this project. Let’s get into the real deal. Follow Me!

Chapter Two

Starting a Project

Excerpt

In this guide, we will be developing a system that can be used to report ***incidences*** *in the agriculture sector (Same as the one on the YouTube Channel Playlist). The incidences range from crop diseases, animal disease outbreaks, natural calamities etc. These incidences will have location data, hence will build a map portal to display the incidences as reported. We will not focus much on the working logic of the system but on how we can use GeoDjango to build such systems. We will utilize OSM API, Leaflet JS and many more. We will also add counties data so as to show the distribution of incidences in different counties (This dataset is available for Kenya but you can use any administrative boundary of interest)*

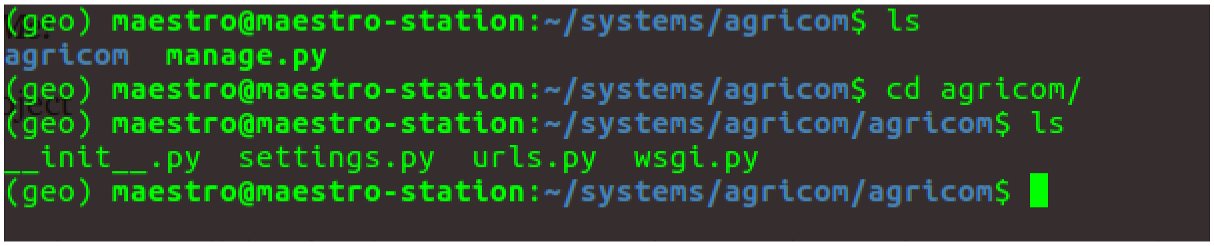
We will call the project **Agricom**

If everything is clear, let's proceed.

To create a django project run the following command.

```
django-admin startproject agricom #Remember to change the name for your project if need be
```

Now, we have a project called **agricom**. Inside the **agricom** directory there is another **agricom** directory and a **manage.py** file. This file acts as a command-line utility that lets you interact with this Django project in various ways.



```
(geo) maestro@maestro-station:~/systems/agricom$ ls
agricom  manage.py
(geo) maestro@maestro-station:~/systems/agricom$ cd agricom/
(geo) maestro@maestro-station:~/systems/agricom/agricom$ ls
__init__.py  settings.py  urls.py  wsgi.py
(geo) maestro@maestro-station:~/systems/agricom/agricom$
```

The agricom folder has four(4) files in it. We will know the use for each in details as we build on this project later in this guide.

Things to note

1. ***settings.py*** - Holds the configurations of the Agricom project
2. ***__init__.py*** - This tells Python to consider this directory as a Python Package.
3. ***urls.p***y - This carries the URL declarations for the Agricom project
4. ***wsgi.py*** - This file serves as entry-point for WSGI-compatible web servers to serve our project

These files will change as the project grows

The default database is set to sqlite3 (Will change this later in this guide)

At this time, we can check the awesome Django admin and front-end.

But, first, we need to link up the database for this project with the different Django apps(Available by default like auth). To migrate the project to the db, run

```
python manage.py migrate
```

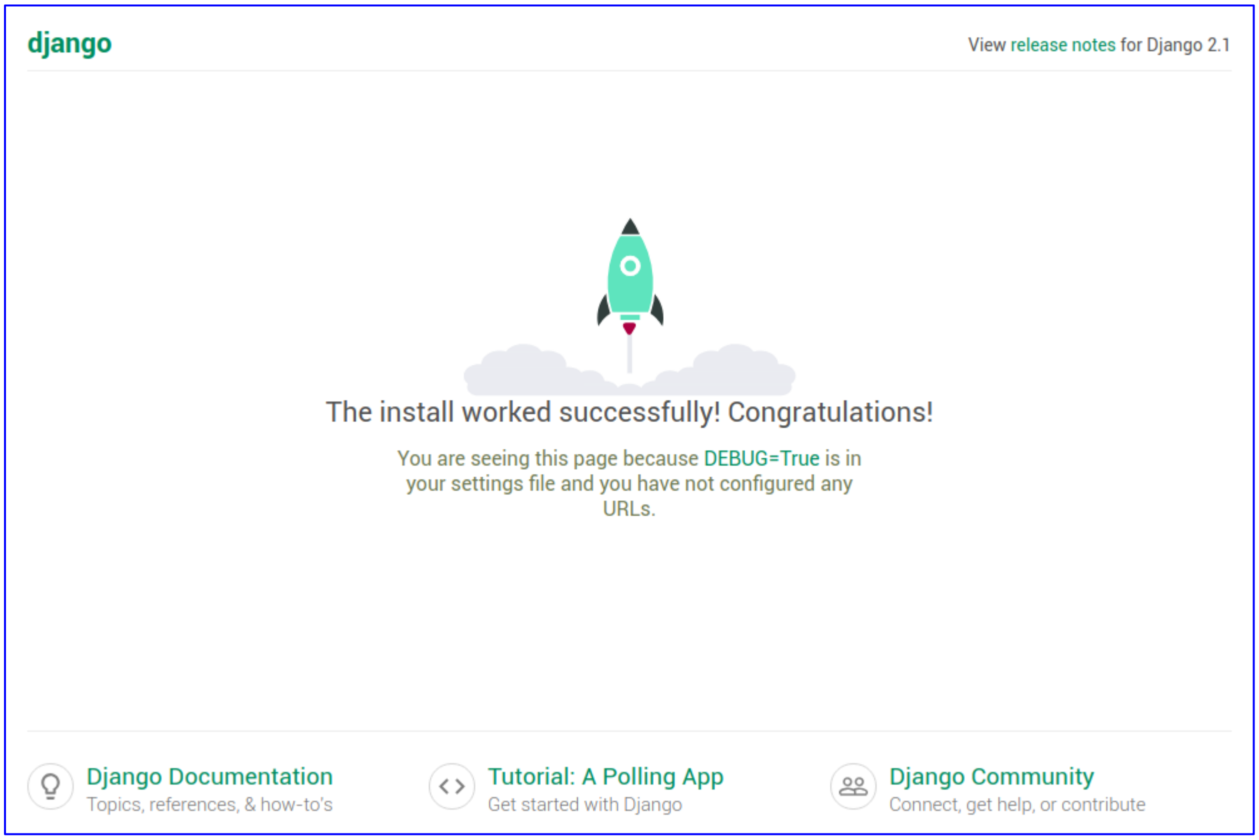
Also, we need an admin user to manage the project;

```
python manage.py createsuperuser
```

Input username, email and password. Then;

```
python manage.py runserver
```

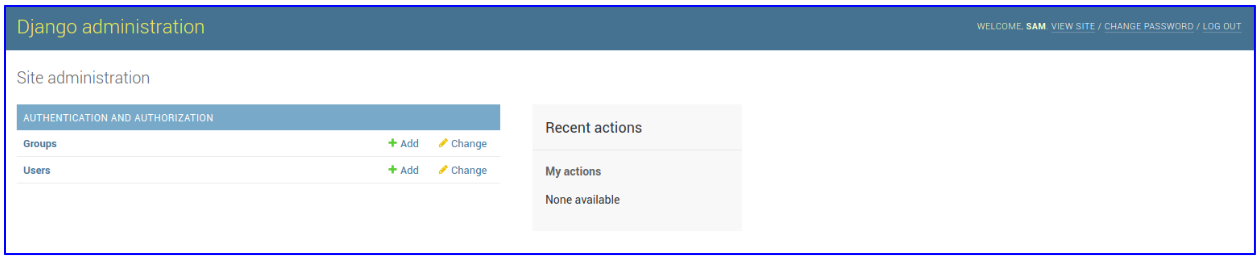
On the browser, open **localhost:8000** and check the goodness in there


django
View release notes for Django 2.1
Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Also **localhost:8000/admin** and check what Django has done for you



Start a Django Application

One amazing thing about Django is that it comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories. Let's explore this in the next steps;

Ensure you are in the agricom folder(On Terminal). If not,

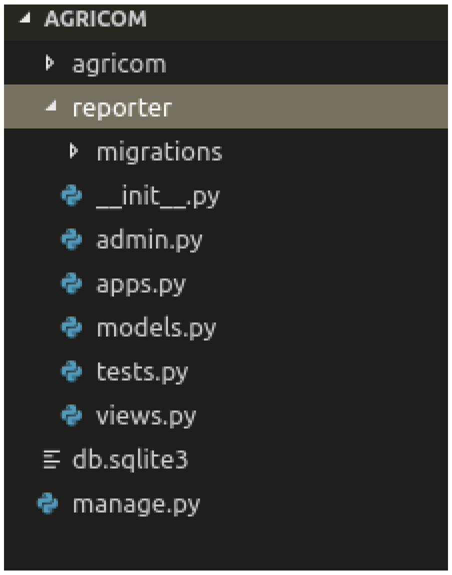
```
cd agricom
```

Then, run the command to create the app ***reporter***

```
python manage.py startapp reporter
```

Edit the **settings.py** file and add 'reporter' under the **INSTALLED_APPS** section.

The **reporter** directory has six(6) files and one directory.



```
AGRICOM
  agricom
  reporter
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  db.sqlite3
  manage.py
```

a) **__init__.py** - This file tells Python to consider this directory as a Python package.

b) **admin.py** - Basically, this is the file where you store the configuration of the Django built-in admin.

c) **apps.py** - This file is used in Django’s internal app registery and is mainly used to store meta data.

d) **models.py** - This is the file that stores all models related to this app. Models are basically how Django interprets your database.

e) **tests.py** - This is basically where you add unit tests and integration tests.

f) **views.py** - This is where you store the business logic of the app.

g) **migrations folder** - These are basically illustrations of how Django builds your database.

At this point, we are interested in changing the **models.py** and **admin.py** files.

We need to create a model that will be used to store incidences information.

Under models.py;

```
from __future__ import unicode_literals

from django.db import models

from django.contrib.gis.db import models as gis_models

from django.db.models import Manager as GeoManager

# Create your models here.

class Incidence(models.Model):

        title = models.CharField(max_length=20)

        description = models.TextField(max_length=250, null=True)

        date_reported = models.DateField(auto_now_add=True)

        location = gis_models.PointField(srid=4326)

        objects = GeoManager()

        def __str__(self):

                return self.title

        class Meta:

                verbose_name_plural =" Incidences"
```

In the script above, we import models from **django.contrib.gis.db** as **gis_models** so as to differentiate them from the usual models from **django.db**.

Also we import a Manager from models so as to be used in the incidence class. This is a workaround in Django 2.x on using the model managers. In Django < 2 you would use the manager directly as shown below;

```
objects = models.GeoManager()
```

To effect the changes in the database, we need to create migrations which will be migrated to the database.But before this, we need to add GIS support in our settings.py file.

Add the following under INSTALLED_APPS.

```
'django.contrib.gis'
```

Remember that we have our database settings set to **sqlite**. In this project, we need to use PostGIS database. We need to make some changes to the settings.py file.

In settings.py, change;

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),

    }

}
```

To

```
DATABASES = {

    'default': {

        'ENGINE': 'django.contrib.gis.db.backends.postgis',

        'NAME': 'geo', #Name of the database

        'USER': 'postgres', #Name of the user

        'HOST': 'localhost', #Change if the database lives in a system different from your local system.

        'PASSWORD': 'xxxxxxx',

        'PORT': '5432',    }}
```

This will change our database from sqlite to PostGIS.

After this change, we need to migrate our project to the new database

```
python manage.py migrate
```

Then **makemigrations** to track the changes in the reporter app.

```
python manage.py makemigrations
```

Then

```
python manage.py migrate
```

Remember to create a super user to use in your project (Refer to section above)

At this point, we have a project with one app, configured with a PostGIS database. Our app has one model, **Incidences**. Make sure you understand every bit before proceeding.

If all is well, let's proceed.

If you log in into the django admin, there are no changes in the interface. We need to add, view and edit Incidences in our database from the admin. To do this, we need to register the models in the admin.py file.

Edit **admin.py** file to reflect the following;

```
from django.contrib import admin

from .models import Incidence

class IncidenceAdmin(admin.ModelAdmin):

    list_display = ('title','date_reported','location')

    search_fields = ('title',)

    filter_fields = ('title','date_reported')

admin.site.register(Incidence, IncidenceAdmin)
```

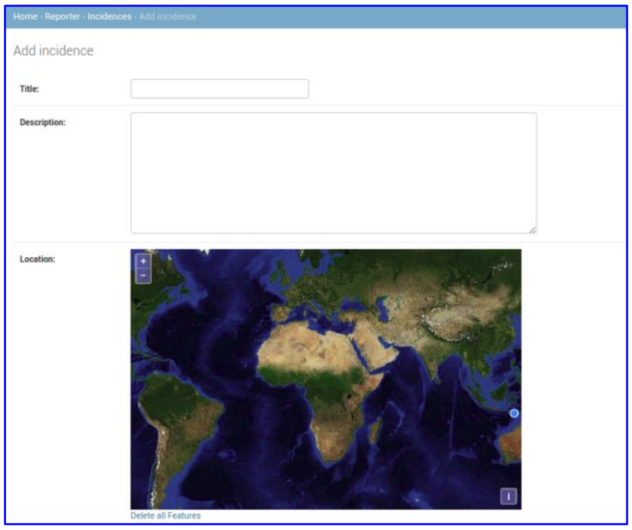
In the script above, we import the Incidence model from models.py. We also define an admin that will register the Incidence model. We also define;

a) **list_display** - which holds the fields that will be shown on the object list under this admin.

b) **search_fields** - Which define the fields that will be used to find an object from the list.

c) **filter_fields** - These are fields that will be used to categorize the objects in Django admin.

After the changes, access the admin and click on the Incidence link then **Add Incidence**. An interface resembling the figure below will be shown;



The new interface shows a form with two text fields and one GIS field. By default, a base layer is added to the GIS field as shown in the figure above. In this case, the base layer is NASA Worldview.

Summary

At this point of the guide, we have;

a) One model in the models.py

b) One admin class for the model in admin.py

c) A super user in the database

d) The admin and front-end.

We are also able to add a new incidence from the admin back-end and view a list of all added incidences at one instance.

We can also search, filter and view the objects list.

Cool! Right? Take a break if need be!

Chapter Three

GIS Data and GIS Models

In this guide, we will be using GIS data in all our models and operations. In this regard, we need to understand how to handle this kind of dataset. In the previous chapter, we have added a GIS model and constructed its admin class.

In this chapter we shall really understand what's happening.

The GIS data we are handling includes four (4) types of extensions;

1. **.shp -** Holds the vector data for the world borders geometries.
2. .**shx** - Spatial index file for geometries stored in the .shp.
3. **.dbf** - Database file for holding non-geometric attribute data (e.g., integer and character fields).
4. **.prj** - Contains the spatial reference information for the geographic data stored in the shapefile

We will work with these files in QGIS in later chapters of this guide.

Note:

The words ***shapefile*** *and* ***layer*** *are used interchangeably to mean a GIS file(which has the four files described above).*

Scenario 1:
We have a shapefile of all incidences that occurred in 2018 in Kenya and we need to build a GIS system to manage this data. We also need to add boundary layer (Counties) to show extents. The system should have two models.

In this chapter, we are handling **Scenario 1**. Unlike in the previous chapter where we only had one model that we were managing through the django admin.

In our project, we need to think of a way in which we can load this existing data into the database, without omitting a single record and in the simplest way possible.

In this regard, we will generate geographical model from existing **county** shapefile and use the model in our project.

In this project we are dealing with vector data. There are many ways in which you can use to generate models from shapefiles and import the data into the database;

Generating Geographical Models

This section illustrates the process of automating the generation of geographical models from the spatial data.

We will utilize the **ogrinspect management command**. This command introspects a vector data source and generates a model definition and its respective LayerMapping dictionary automatically.

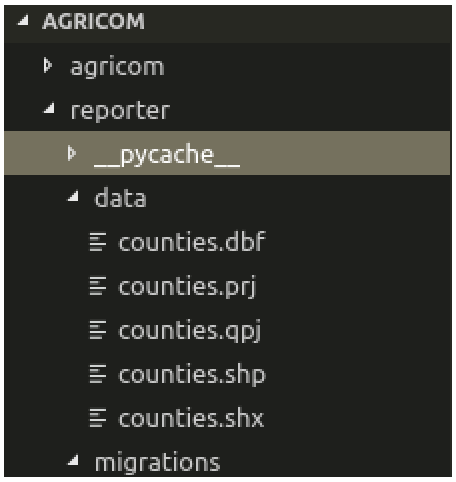
To achieve this, we need the County shapefile. In the reporter directory, create another directory named **data**. This is where the all layers will live.

To create the directory, use;

```
mkdir data
```

Download the Kenya Administrative Data, extract it and copy the contents into the data directory. The data can be extracted from any source or if you don't have any, try Diva-GIS portal.

Your structure should resemble the one shown in the figure below;



```
AGRICOM
  agricom
  reporter
    __pycache__
    data
      counties.dbf
      counties.prj
      counties.qpj
      counties.shp
      counties.shx
    migrations
```

Run the following command to generate the model from the file.

```
python manage.py ogrinspect reporter/data/counties.shp County --srid=4326 --multi --mapping
```

You need to specify the path, where the shapefile is located and the name of the output model (**County**). This path depends on your current location on terminal. The command above assumes you are in the Agricom project folder.

--srid=4326 option sets the Spatial Reference Identifier for the geographic field.

--mapping option tells ogrinspect to also generate a mapping dictionary for use with LayerMapping.

--multi option is specified so that the geographic field is a MultiPolygonField instead of just a PolygonField.

The outputs are as follows;

```
# This is an auto-generated Django model module created by ogrinspect.

from django.contrib.gis.db import models

class County(models.Model):

    counties = models.CharField(max_length=25)

    codes = models.IntegerField()

    cty_code = models.CharField(max_length=24)

    dis = models.IntegerField()

    geom = models.MultiPolygonField(srid=4326)

# Auto-generated `LayerMapping` dictionary for County model

county_mapping = {

    'counties': 'Counties',

    'codes': 'Codes',

    'cty_code': 'Cty_CODE',

    'dis': 'dis',

    'geom': 'MULTIPOLYGON',

}
```

The first section is the model representing the County shapefile while the second part is a mapping instance of the generated model.

Copy the generated model into your models.py file.

Then inside the reporter directory, create a file e.g ***loader.py***

Copy the second part of the output above and paste it in loader.py

Then, edit the file to resemble the contents shown below;

#loader.py

```
import os

from django.contrib.gis.utils import LayerMapping

from .models import County

county_mapping = {

    'counties' : 'Counties',

    'codes' : 'Codes',

    'cty_code' : 'Cty_CODE',

    'dis' : 'dis',

    'geom' : 'MULTIPOLYGON',

}

county_shp = os.path.abspath(os.path.join(os.path.dirname(__file__), 'data', 'counties.shp'),)

def run(verbose=True):

                lm = LayerMapping(County, county_shp, county_mapping, transform= False,
encoding='iso-8859-1')

                lm.save(strict=True,verbose=verbose)
```

In the script above, we are using the LayerMapping data import utility. In the import utility, we need to

1) Define the path to the shapefile. As shown above, the path will work in any environment provided the data folder remains(The path is not absolute)
2) Add the mapping dictionary which corresponds to the fields in the County model
3) Transform the data if the shapefile is in a different SRID from the one defined in the geom field of the County model. In our example above, transform is false since the data is in the same SRID as defined in the model.

Next, we need to effect the changes of our models.py into the database. To do this, run **makemigrations** then **migrate** commands.

After this, we register the County model into our admin. So, in the admin.py add the County details. Your admin.py should now be as follows;

```
from django.contrib import admin

from .models import Incidence, County

# Register your models here.

class IncidenceAdmin(admin.ModelAdmin):

    list_display = ('title','date_reported','location')

    search_fields = ('title',)

    filter_fields = ('title','date_reported')

class CountyAdmin(admin.ModelAdmin):

    list_display = ('counties','cty_code')

    search_fields = ('counties',)

    filter_fields = ('counties','cty_code')

admin.site.register(Incidence, IncidenceAdmin)

admin.site.register(County, CountyAdmin)
```

We now have two models in our project, **Incidence** and **County**. Incidences will be added from the admin interface while counties will be imported from the counties.shp file.

Import Data

From the terminal, run;

```
python manage.py shell
```

This command opens the Python/Django shell.

Next, we import the loader module then we run the module.

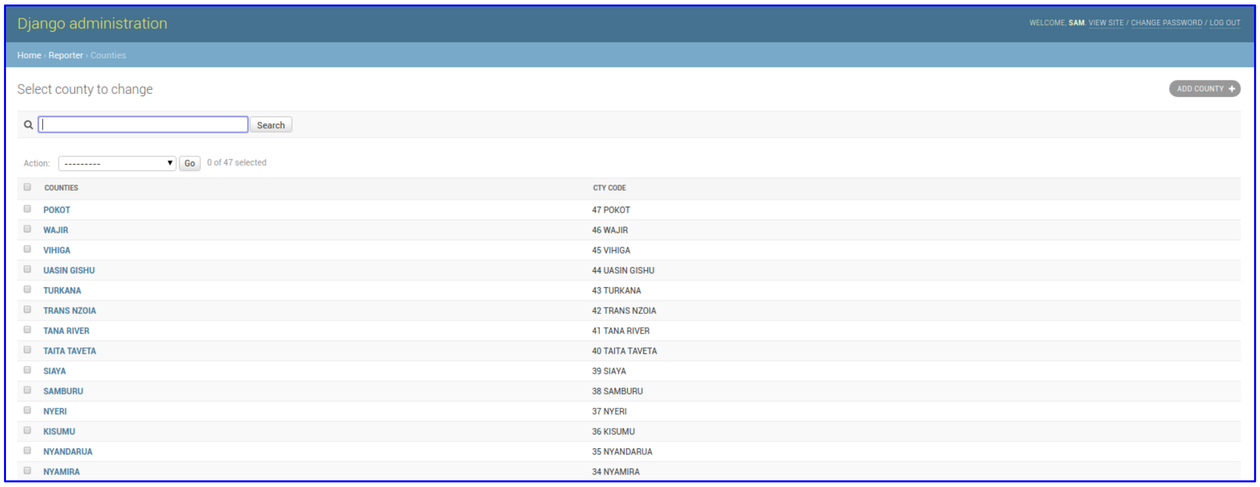
```
from reporter import loader #this is the name of the file you created above

loader.run()
```

If successful, the outputs will be as follows;

```
Saved: BARINGO

Saved: BOMET

Saved: BUNGOMA

…………………
```

Ctrl + D to exit shell. Then runserver.

Login into the django admin on any browser and check the newly imported county objects.



Django administration

WELCOME, SAM. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Reporter › Counties

Select county to change

ADD COUNTY

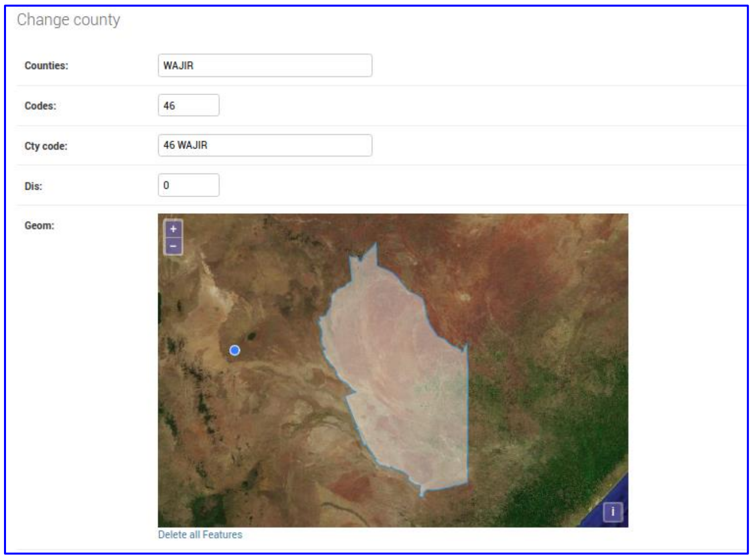
Search

Action: --------- Go 0 of 47 selected

COUNTIES	CTY CODE
POKOT	47 POKOT
WAJIR	46 WAJIR
VIHIGA	45 VIHIGA
UASIN GISHU	44 UASIN GISHU
TURKANA	43 TURKANA
TRANS NZOIA	42 TRANS NZOIA
TANA RIVER	41 TANA RIVER
TAITA TAVETA	40 TAITA TAVETA
SIAYA	39 SIAYA
SAMBURU	38 SAMBURU
NYERI	37 NYERI
KISUMU	36 KISUMU
NYANDARUA	35 NYANDARUA
NYAMIRA	34 NYAMIRA

If you are able to view such a list, it means your system is working as expected. If not, try to re-check your code and fix existing issues.

Try to click on one of the counties and view the changes. Something awesome is shown now. *Try to play around with the polygon shape.*



Change county

Counties:	WAJIR
Codes:	46
Cty code:	46 WAJIR
Dis:	0
Geom:	

Delete all Features

Notes

In this section, we’ve managed to generate geographic models from a shapefile and imported the data into the model.

We’ve also explored the various options that exist while handling spatial data in GeoDjango.

If your application doesn’t work; GDAL or GEOS failure, add the paths of GDAL and GEOS to the settings.py

```
GDAL_PATH_LIBRARY =    “your directory/gdal.dll”
```

Chapter Four

The Front-End (Views, URLs, Static and Templates)

In the previous chapters we've imported County data into the database that can be edited from the Django admin. Our project has two models, Incidence and County, which can be visualized from the Django admin. We need to develop a front-end in which the data in these models can be visualized. The admin is basically for staff but we also need a "Portal" where anyone who visits the site can visualize the data.

In this regard, we need to develop a mapping dashboard for visualizing these layers. In this chapter we will focus on creating views, urls, templates and handling static files to develop a visualization dashboard.

First of all, we have only updated the **models.py** and **admin.py** in our reporter app. Let's explore the other files in this app.

Let us create a homepage for our project. To do this, let's update the **views.py** file.

#views.py

```
from django.shortcuts import render

from django.views.generic import TemplateView

from django.http import HttpResponse

from models import County,Incidence

# Create your views here.

class HomePageView(TemplateView):

        template_name = 'index.html'
```

In this script, we have one view, in which we inherit the TemplateView view. We also define the template name in which this view will render.

Next, we add a url which will render the view that we just added above. Inside the reporter directory, create a file **urls.py**. This is the file that will hold our url declarations for the reporter app.

Once created, edit it as follows;

#reporter/urls.py

```
from django.urls import path

from .views import *

urlpatterns = [

   path('', HomePageView.as_view(), name= 'home'),

  ]
```

There are many ways of achieving the task above in Django. You can change to these ways once you get to know them.

At this point we have url and a view. But, we need a template named **index.html**.

To add this, inside the **agricom** directory, create a directory named **templates**.

```
mkdir templates
```

Inside the templates directory, create a file named **index.html**. Then, edit as follows;

#index.html

```
<html>

   <head>

      <title>Agricom Map Portal</title>

   </head>

   <body>

      <p> This is our map portal page</p>

   </body>

</html>
```

This is just a basic html template that displays "***This is our map portal page"*** on the browser.

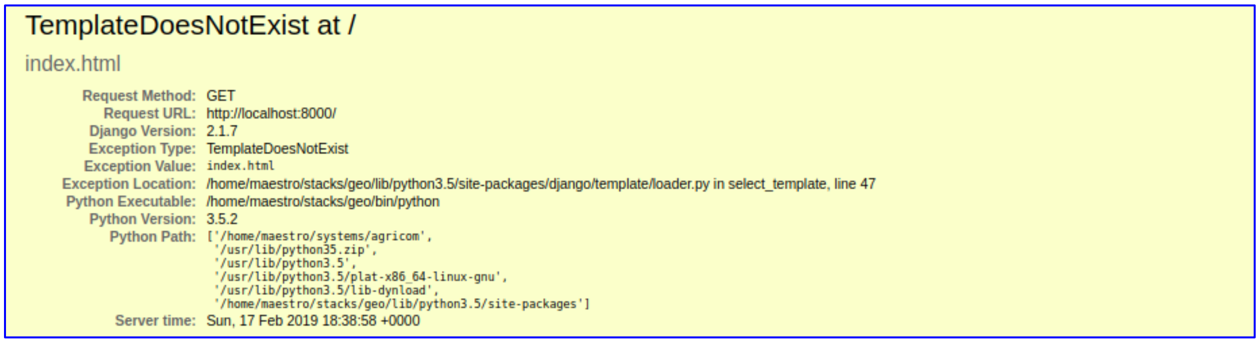
We now have our template, a view and url. If you try to access the project on the browser, there will be no changes. Why? This is because the project doesn't recognize the **reporter app** url. To enable their use, include the urls in the base url declarations file (urls.py in the agricom directory in our project)

Your base **urls.py** file should look as shown below;

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('reporter.urls'))

]
```

This means that if we try to access **localhost:8000/** it will search for the relevant url in **reporter/urls.py**

Refresh the page on your browser to view the changes. At this point, a new error will appear on your browser. A **TemplateDoesNotExist** error will appear.



```
TemplateDoesNotExist at /
index.html

    Request Method:  GET
       Request URL:  http://localhost:8000/
    Django Version:  2.1.7
    Exception Type:  TemplateDoesNotExist
   Exception Value:  index.html
Exception Location:  /home/maestro/stacks/geo/lib/python3.5/site-packages/django/template/loader.py in select_template, line 47
 Python Executable:  /home/maestro/stacks/geo/bin/python
    Python Version:  3.5.2
       Python Path:  ['/home/maestro/systems/agricom',
                      '/usr/lib/python35.zip',
                      '/usr/lib/python3.5',
                      '/usr/lib/python3.5/plat-x86_64-linux-gnu',
                      '/usr/lib/python3.5/lib-dynload',
                      '/home/maestro/stacks/geo/lib/python3.5/site-packages']
       Server time:  Sun, 17 Feb 2019 18:38:58 +0000
```

At this point, you might ask yourself why the system isn't working. Right? Well, there is something we're missing out. We need to "tell" django where to find our templates.

We have one template directory for our entire project. This convection will differ from user to user. One should work with the approach they prefer. Another approach on this templates issue would be to place a templates directory inside every app in the project. Well, choose you approach.

How do we let Django know where to find templates? We need to set this in **settings.py** file.

Under the **templates dictionary** in the settings.py file, edit the line that starts with **'DIRS'** to;

```
'DIRS': [os.path.join(BASE_DIR, 'templates'),],
```

This line tells django where to find templates. If your path is different from mine, make sure to edit the section with **'templates'** accordingly.

Well, you've done it again. You got things under control.

Refresh your page and see the statement ***This is our map portal page*** displayed on your page.

Wow! wow! wow!. Well done. A break maybe good!

We just displayed a single statement on the page. This wasn't our objective. We need to display a map with Incidences and County layers. But first, let me show you something in the admin. We can make some adjustments to change our displays. In the **admin.py** file, we used **admin.ModelAdmin** to define the classes. In this scenario all geographical fields are shown as a map but with limited functionality.

Let's tweak this bit!

By default, django ships with a class known as **OSMGeoAdmin**. Which displays any geographical field as a map with extra tools, features and options for customization? Let's have a look.

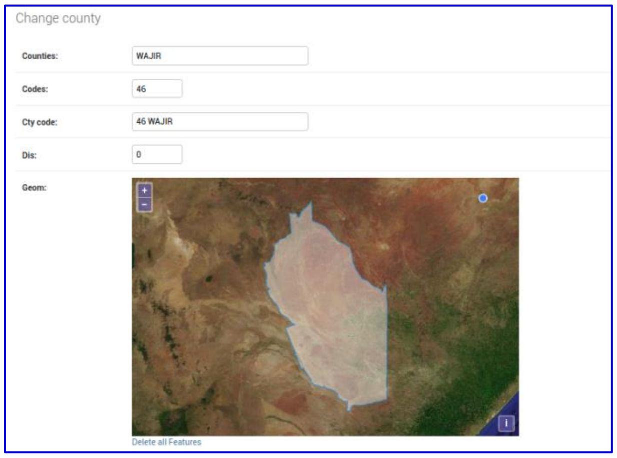
In the admin.py file, change CountyAdmin to;

```
class CountyAdmin(OSMGeoAdmin):

   list_display = ('counties','cty_code')

   search_fields = ('counties',)

   filter_fields = ('counties','cty_code')
```

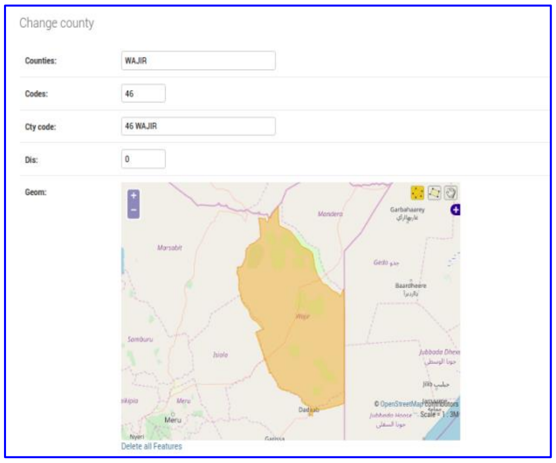
In this scenario we use **OSMGeoAdmin** instead of **admin.ModelAdmin**

For you to use OSMGeoAdmin, you must import it from Django's GIS Contrib. On the import statements add ***from django.contrib.gis.admin import OSMGeoAdmin***

Compare the new geom display and note the differences.


Change county

Counties:	WAJIR
Codes:	46
Cty code:	46 WAJIR
Dis:	0
Geom:	Delete all Features


Change county

Counties:	WAJIR
Codes:	46
Cty code:	46 WAJIR
Dis:	0
Geom:	Delete all Features

Note the differences?

You've made your displays beautiful and added additional tools.

Let's continue with our front-end things.

We have one template, **index.html**, which displays one statement on the browser. Let' customize this template to display a map.

As indicated earlier, we will be using Leaflet JS, which is a JavaScript library for interactive maps. To use this library, we have to install it into our project. To do this, run this command (Make sure your virtualenv is activated)

```
pip install django-leaflet
```

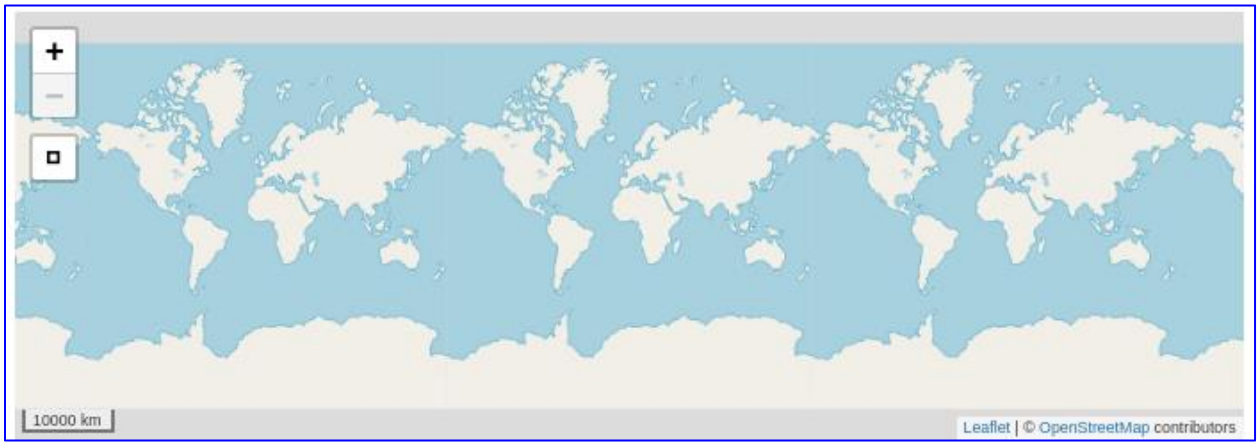
Once it's installed, remember to add **leaflet** under **INSTALLED_APPS** in **settings.py**

Now, we have Leaflet installed. We need to display a map on our page. To do this, we edit the index.html file to;

```
{% load leaflet_tags %}

<html>

   <head>

      <title>Agricom Map Portal</title>

      {% leaflet_js %}

      {% leaflet_css %}

   </head>

   <body>

      {% leaflet_map "agrimap" %}

   </body>

</html>
```

For us to use leaflet on our page, we have to load it into the template. In addition, we load the CSS and JS files under the head section. Lastly, we define a map inside the <body> section. This will display a leaflet map on our page. We also assign the map a name, just in case we want to reference it later in our code.

You should have a map shown as below;



The map is has a base layer, a full-screen tool, an attribution control and zoom in, zoom out controls.

Let’s add sauce to our map.

First, we need to focus our map to our area of interest i.e when the map loads, it should automatically zoom to our area of interest.

Before doing this, we will make some changes to our code. We will write more code in the next sections, thus the change in structure.

Copy and paste the code below. The explanation follows right after the script.

#index.html

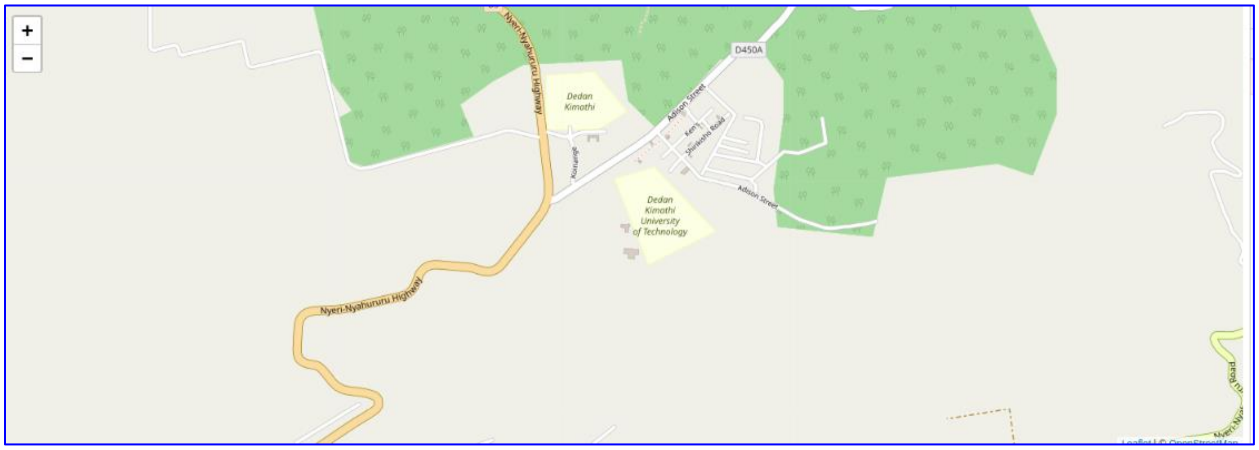
```
{% load leaflet_tags %}

<html>

  <head>

    <title>Agricom Map Portal</title>

    {% leaflet_js %}

    {% leaflet_css %}

  </head>

  <body><div id="agrimap" style="height: 600px; width: 100%;"></div></body>

<script>

  var map = L.map(‘agrimap’, {

    center: [-0.397783, 36.960936],
    zoom: 15,

  });

  var osm = L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {

        maxZoom: 19,attribution: '&copy; <a
href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'

    });

  var topo = L.tileLayer('http://{s}.tile.opentopomap.org/{z}/{x}/{y}.png', {

    maxZoom: 17, attribution: 'Map data: &copy; <a
href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>, <a
href="http://viewfinderpanoramas.org">SRTM</a> | Map style: &copy; <a
href="https://opentopomap.org">OpenTopoMap</a> (<a
href="https://creativecommons.org/licenses/by-sa/3.0/">CC-BY-SA</a>)'

  });

  osm.addTo(map);

</script>

</html>
```

In the script above, we separated the JavaScript code from the rest of the code [we restructured the code]. In this way, we can handle similar codes together. We have defined a map component which holds the leaflet map declarations. In this case, we have a center, which is the point in which the map should be centered-to once loaded and zoom which is the zoom level in which our map will be assigned once loaded.

We have also introduced a div in the body section of the file. This is similar to what we had before, ***{% leaflet_map "agrimap" %}***. Not the style we’ve defined for the div. If you remove these style options, you page will just be blank. Remember it worked with leaflet_map tag before. This is because this is already defined in the leaflet css linked to in the file. You can also write your own css to control the appearance. Feel free!

We have added two base layers, **osm** and **topo**, the added osm to the map. We shall be able to interchange the base layers at a later stage in this guide.

The script above should output the following;



Hoping you understand stuff in this section, let’s move on.

Remember we have a map, but we can’t see our two layers, Incidences and County. Let’s work on them.

Let’s start with rendering data into a format we can consume in our map. In this case, we will be using GeoJSON. Let’s create a view in the views.py file.

We are creating a view that will serialize our data into GeoJSON format and render this data into a url. Add the following views.

#views.py

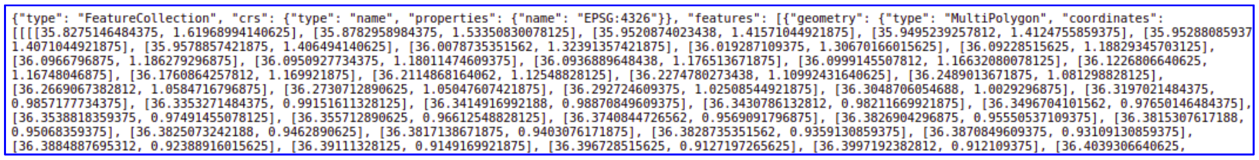
```
from django.core.serializers import serialize

def IncidenceData(request):

        points= serialize('geojson', Incidence.objects.all())

        return HttpResponse(points,content_type='json')

def CountyData(request):

        counties = serialize('geojson', County.objects.all())

        return HttpResponse(counties,content_type='json')
```

In the views above, we serialize all objects from each model into GeoJSON and we return the GeoJSON object. Technically, this will be rendered to the url that we will define in a few.

Once this is done, we need to create urls that will channel this data to wherever we want the data to be. Change your urls.py file to the following.

```
from django.urls import path

from .views import *

urlpatterns = [

    path('', HomePageView.as_view(), name='home'),

    path('incidences/', IncidenceData, name='incidences'),

    path('county/', CountyData, name='counties')

]
```

That was easy. Right? Let’s see what we have. Open one of the paths in your browser and see what you got. This is how the County GeoJSON looks like.



```
{"type": "FeatureCollection", "crs": {"type": "name", "properties": {"name": "EPSG:4326"}}, "features": [{"geometry": {"type": "MultiPolygon", "coordinates":
[[[[35.8275146484375, 1.61968994140625], [35.8782958984375, 1.53350830078125], [35.9520874023438, 1.41571044921875], [35.9495239257812, 1.4124755859375], [35.95288085937
1.4071044921875], [35.9578857421875, 1.406494140625], [36.0078735351562, 1.32391357421875], [36.019287109375, 1.30670166015625], [36.09228515625, 1.18829345703125],
[36.0966796875, 1.186279296875], [36.0950927734375, 1.18011474609375], [36.0936889648438, 1.176513671875], [36.0999145507812, 1.16632080078125], [36.1226806640625,
1.167480046875], [36.1760864257812, 1.169921875], [36.2114868164062, 1.12548828125], [36.2274780273438, 1.10992431640625], [36.2489013671875, 1.081298828125],
[36.2669067382812, 1.0584716796875], [36.2730712890625, 1.05047607421875], [36.292724609375, 1.02508544921875], [36.3048706054688, 1.0029296875], [36.3197021484375,
0.9857177734375], [36.3353271484375, 0.99151611328125], [36.3414916992188, 0.98870849609375], [36.3430786132812, 0.98211669921875], [36.3496704101562, 0.97650146484375],
[36.3538818359375, 0.97491455078125], [36.355712890625, 0.96612548828125], [36.3740844726562, 0.9569091796875], [36.3826904296875, 0.95550537109375], [36.3815307617188,
0.95068359375], [36.3825073242188, 0.9462890625], [36.3817138671875, 0.9403076171875], [36.3828735351562, 0.9359130859375], [36.3870849609375, 0.93109130859375],
[36.3884887695312, 0.92388916015625], [36.39111328125, 0.9149169921875], [36.396728515625, 0.9127197265625], [36.3997192382812, 0.912109375], [36.4039306640625,
```

We have our views and urls ready. Let’s add this data into our map.

In the <script> section, add the following code after map component.

```
var incidences = L.geoJson(null, {

  pointToLayer: function (feature, latlng) {

    return L.marker(latlng, {

      icon: L.icon({

        iconUrl: "static/img/red.png",

        iconSize: [28, 32],

        iconAnchor: [12, 28],

        popupAnchor: [0, -25]

      }),

      title: feature.properties.title,
      riseOnHover: true
    });
  },
}).addTo(map);
var incidencesUrl = 'incidences';
$.getJSON(incidencesUrl, function (data) {
  incidences.addData(data);
});
```

In this script;

We define an **incidences** variable which holds the Leaflet GeoJSON feature. In this feature, we pass various options such as the pointToLayer option where we define the icon to be shown on the map, its size, anchor etc.

We then add the incidences to the map.

In the next section, we define a variable incidencesUrl which hold the url/path where the GeoJSON object is found. We then use JQuery to get the GeoJSON object and add the data to the incidences layer.

Refresh your map on the browser to view the changes. Noted something? The icons are not displayed on the map. Why? If you followed my steps keenly, we defined **iconUrl** under **pointToLayer**. This should be the cause. We told the system to find a **.png** file inside a folder named **static** then **img**. These directories and the file do not exist. Let’s work on this issue.

Inside the project directory, create a directory named **static**. Inside static, create another directory named **img**. Then copy the **red.png** file into the img folder.

Note:

The tasks above may vary based on how you structure your project. I’m using the most basic structure in this guide.

We also need to change our settings.py to ensure the Django finds static files in the right location.

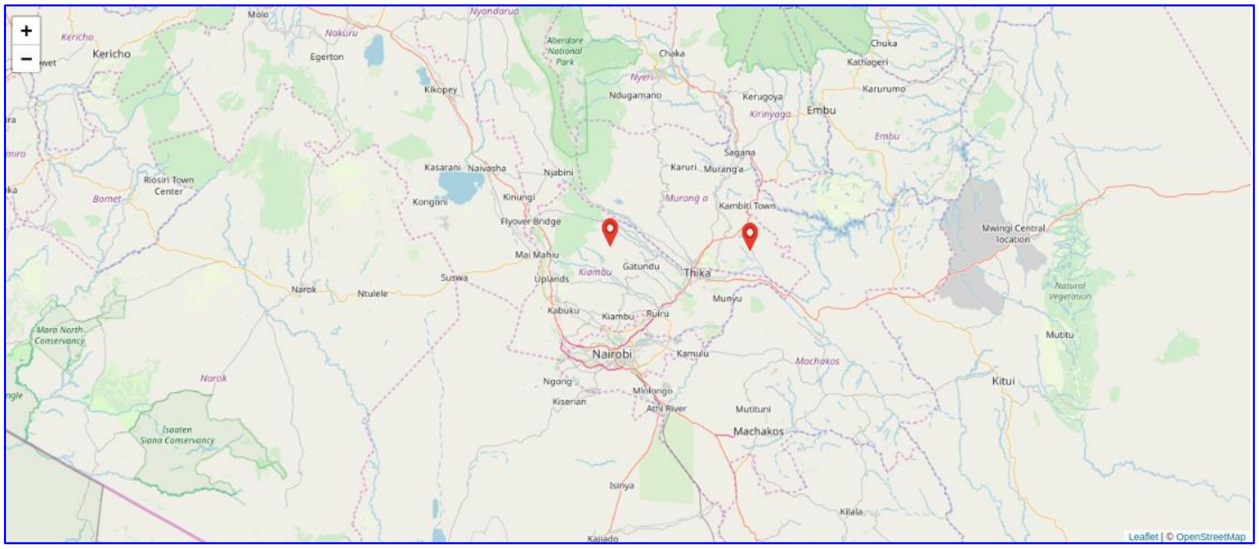
Under settings.py, add at the bottom.

```
STATICFILES_DIRS = (

    os.path.join(BASE_DIR,'static'),

)
```

This adds an absolute path for the static files. This would work in any system and environment.

We cool now. Refresh your map now.

Can you see something on the map? [Remember to add some incidences from the Django admin]



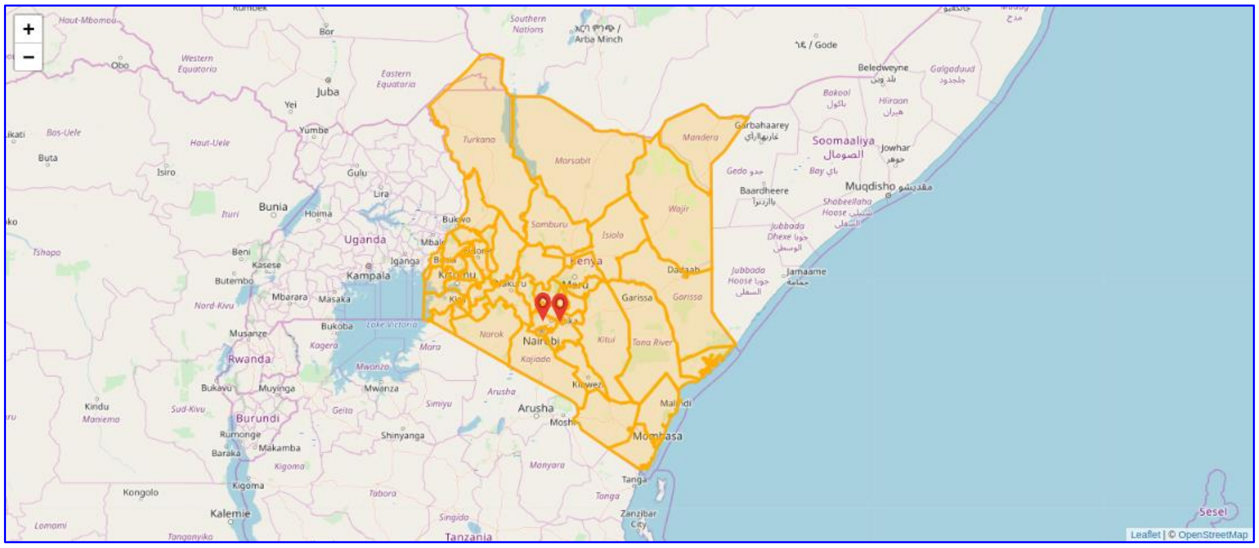
Well! Well! Well! That was easy!

Task 1:

Repeat the procedure described above for the County layer. Remember this is a polygon.

Tip: *You must add style for the polygon layer.*

After this task, your map should have two layers, Incidences and County. See below;



So far so good!

Note:

The approach used in this section may differ from user to user. To make things easier in the future, try to use absolute paths on static files. This will ease code re-use and editing easy.

Summary

You have a map which displays three layers, Incidences, Counties and Base Layer. We will work with these layers further in the next chapter.

You are able to set the display parameters of the map using style or if you went further, using custom css files

We've utilized the Leaflet JS library in detail in this chapter.

Chapter Five

Popup, Layer Styles and Controls

In chapter 4, we worked so much on the front-end. In this chapter we will build on the map portal. So far, our map doesn't have much functionality compared to what we wanted to achieve.

We need a map that;

a) A user can switch between different layers

b) Can differentiate between base layers and other overlay layers

c) One can get more information on a feature displayed on the map

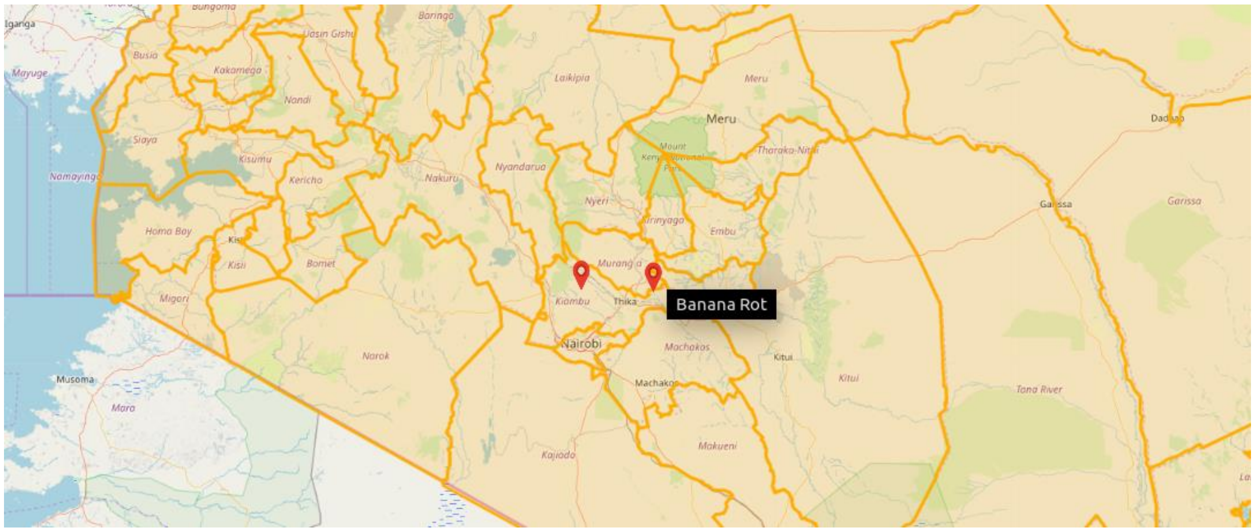
d) Can contain our own signature (The developer of the map)

Let's dive in.

Popup

In most cases, when using a map, you would like to know what that icon represents or what that line on the map represents. In regard to this, there are ways in which this can be achieved.

In our map, if you try to hover over the incidence icons, the title of the incidence is shown. This is because we have defined this feature in our ***pointToLayer*** section.



In this guide, we will implement popups that show information about a specific feature [One feature at a time].Popups are useful if you want to show more details about a feature. How? Follow along;

We will create popups for the Incidences layer. We will display ***title***, ***description*** and ***date_reported*** fields.

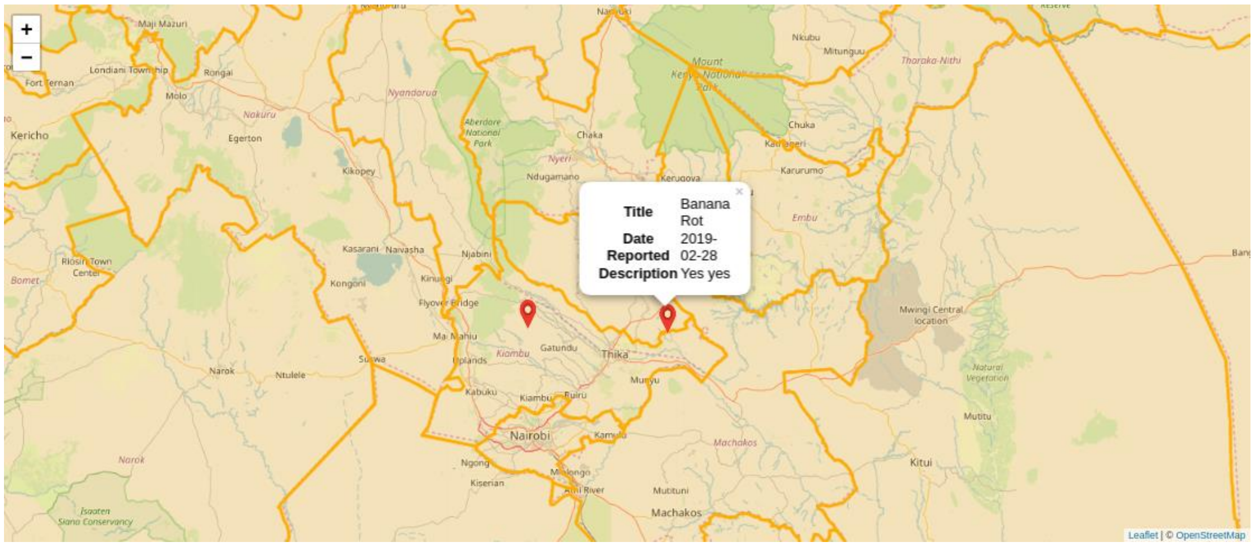
In Leaflet, there is a ***onEachFeature*** method used to trigger various actions on each feature on the map. This is what we will use.

Add the following script in your ***incidences*** layer section, after ***pointToLayer*** part.

```
onEachFeature: function (feature, layer) {

   if (feature.properties) {

     var content = "<table class='table table-striped table-bordered table-condensed'>" +
"<tr><th>Title</th><td>" + feature.properties.title + "</td></tr>" + "<tr><th>Date
Reported</th><td>" + feature.properties.date_reported + "</td></tr>" +
"<tr><th>Description</th><td>" + feature.properties.description + "</td></tr>" +
"<table>";

     layer.on({

       click: function (e) {

         layer.bindPopup(content).openPopup(e.latlng);

       }

     });

   }

 }
```

In this script, we define an onEachFeature method which is triggerred when a feature is clicked. Once clicked, the popup will be shown with content added to it and structured in a table format. The popup is anchored to the location where the feature is [latlng].

Once this is added, you should have something like this;



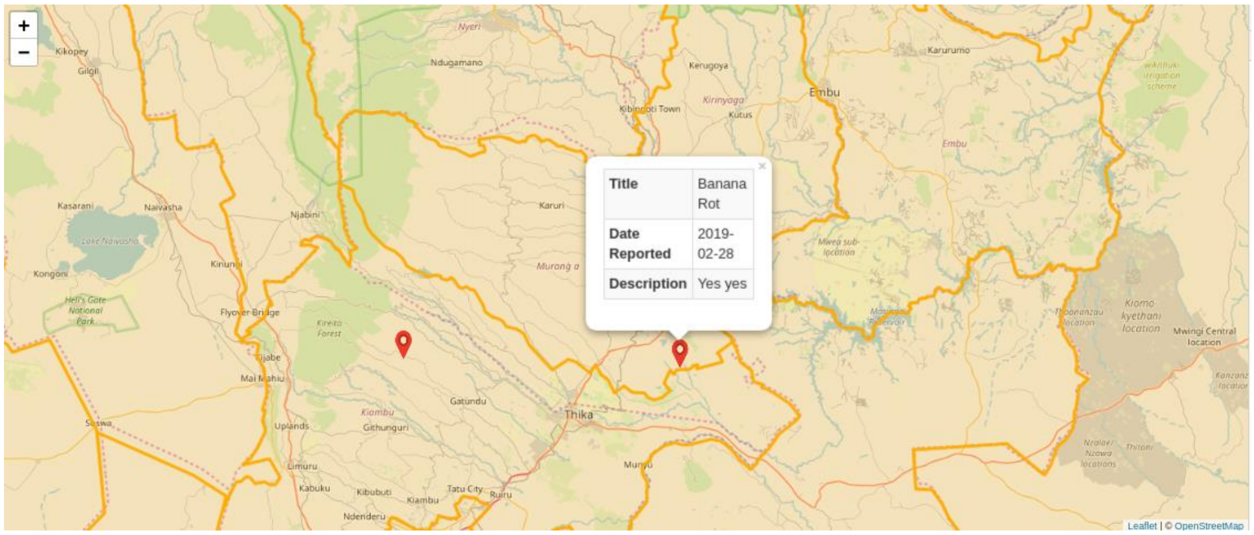
Title	Banana Rot
Date Reported	2019-02-28
Description	Yes yes

However, our popup doesn’t look good. The texts are a bit misaligned and confusing. Let’s fix this. The display is due to some styling issues. We have defined a table with classes such as ***table table-striped table-bordered table-condensed*** which is not defined. These classes belong to BootStrap, which we haven’t added to our code yet. This process is optional btw.

Add this line in your <head></head> section. This will add the sytles we require in our popup.

```
<link rel="stylesheet"
ref="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
```

Refresh your map and compare the popups.



Title	Banana Rot
Date Reported	2019-02-28
Description	Yes yes

The popup is a bit better, with distinctions between rows and columns. Cool. Right?

Task 2:

Add a popup to the County layer. Display at least three fields from the model.

Layer Styles

In this project, we have two overlay layers. Ideally, we have different approaches to data displays in terms of colors, appearance, orientation etc. In this map, the County layer has an Orange color and has a continuous boundary lines.

Let’s explore more in this section.

Scenario:

We want to style our polygon layer in such a way that we can show different colors based on the category. We will use the **codes** field to create 2 categories of the Counties. This will be 1-25, 26 and above.

Our County layer has been styled using this section.

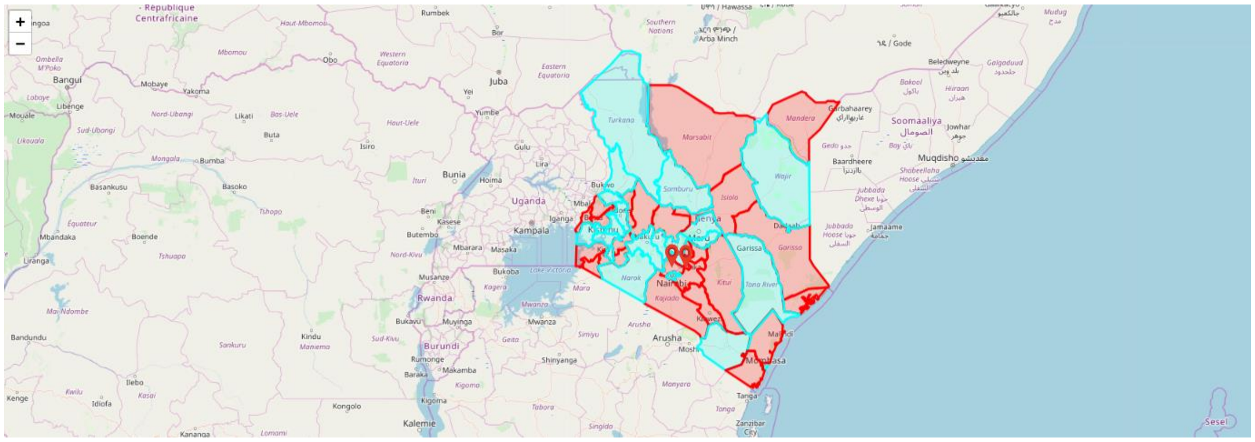
```
style: function colors(feature){

      return{

         color: 'orange',

         fillOpacity: 0.2

      };

},
```

Let’s advance this. We will have to write a looping function to check the codes in the Counties and classify them under a class. In this example, we will use if statements.

Edit the style function to.

```
style: function(feature){

      value = feature.properties.codes;

      if(value <= 25){

         return {color: "#ff0000"};

      }else if(value > 25){

         return {color: "#00ffff"};

      }

   },
```

The output of this style will be as shown below;



Wow! You can alter this map to resemble your styles.

You can also add more options to the style section as shown below;

```
return {
    weight: 2,
    opacity: 0.4,
    color: 'grey',
    Dashboard: '3',
    fillOpacity: 0.7,
    fillColor: 'orange'
};
```

Have questions so far? Let me know!

Controls

This is the third component in this chapter. Apparently, as the name itself indicates, this is where we manage controls on the layer that we have on our map.

This is where we add functionality such as layer switching, attribution, mouse position etc.

Layer Switching Control

In our map, we have two overlay layers and two base layers. We want to create a control that enables us to switch between layer i.e you display a layer based on need.

To add this control, we need to define these groups first; base layers and overlay layers.

To do this, add this after the map definition (After var map = L.map())

```
var baseLayers = {

  "Open Street Map": osm,

  "Topo Map": OpenTopoMap

};

var groupedOverlays = {

  "Point Data": {

    "incidences": incidences,

  },

  "Other": {

    "Administrative Boundaries": counties;

  },

};

var layerControl = L.control.groupedLayers(baseLayers, groupedOverlays).addTo(map);
```

In this script, we define two variables, baseLayers and groupedOverlays. This helps us to categorize the layers in different groups and definitions.

We then define layerControl which holds the control that adds the layers into the map. All these layers are added in one control with different groupings.

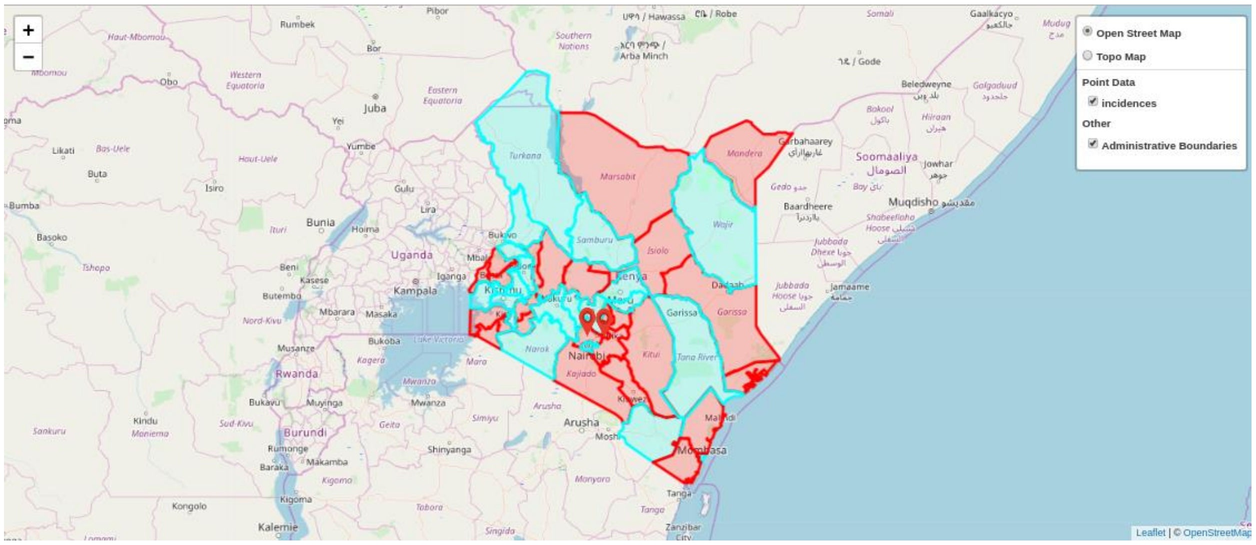
If you refresh your map, there will be no changes. Why? This is because the groupedLayers control inherits from a class that isn't available on our project. To add this class, we need to load its css and js files.

Inside the <head> section, add

```
<link rel="stylesheet" href="{% static 'leaflet-groupedlayercontrol/leaflet.groupedlayercontrol.css' %}">

<script src="{% static 'leaflet-groupedlayercontrol/leaflet.groupedlayercontrol.js' %}"></script>
```

But again, we will receive a new error if we try to run this. In our lines above, we have {% static ... %} which is not recognized. To handle this, add `{% load static %}` at the top of the file(maybe before <html>)

Refresh the map page and see the changes.



You can now switch between different layers in the map.

Mouse Position

When using a map, you might find an interesting feature on it and need to note that location (Coordinates). What do you do?

Let's build a control that will display the location for cursor on the map.

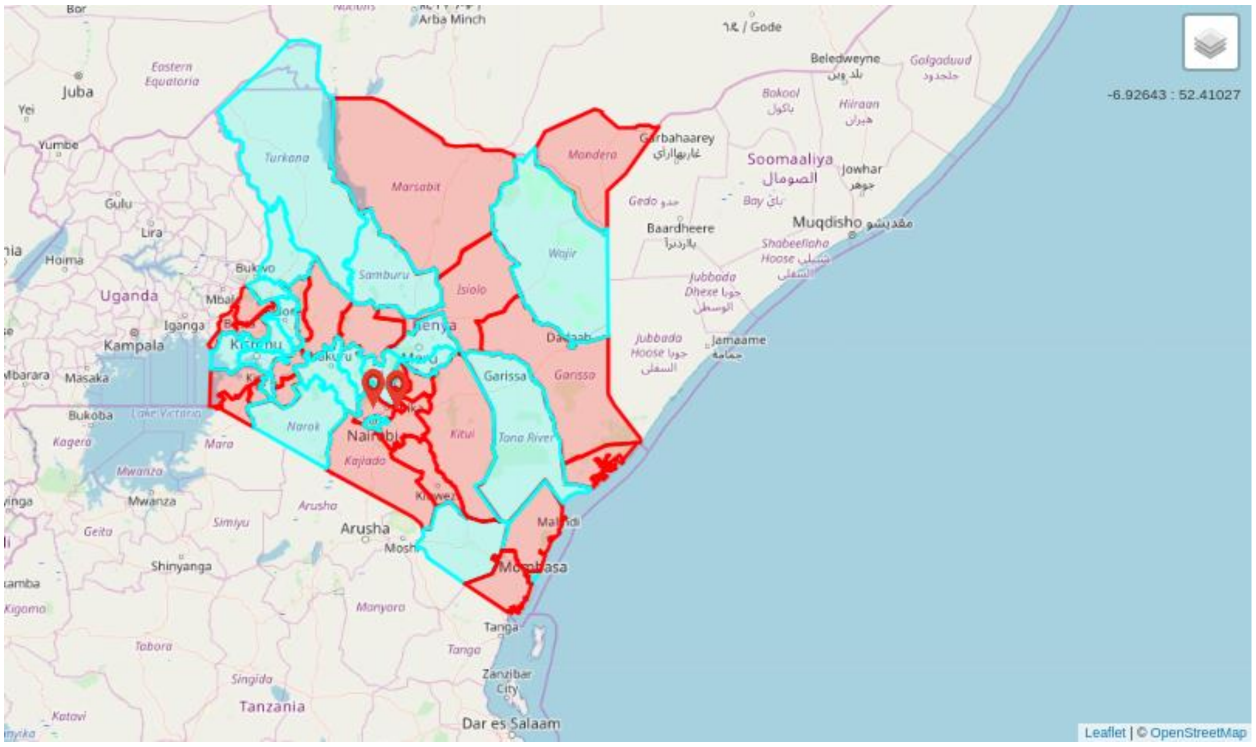
To do this, add this below your grouperLayers control.

```
L.control.mousePosition({position: "topright",}).addTo(map);
```

Just as we did with the layer control, we need to reference a JS file. Add this in the <head> section and after JQuery.

```
<script src="{% static 'js/mousePosition.js' %}"></script>
```

Refresh your map and hover the map. You should see the coordinates on the topright section.



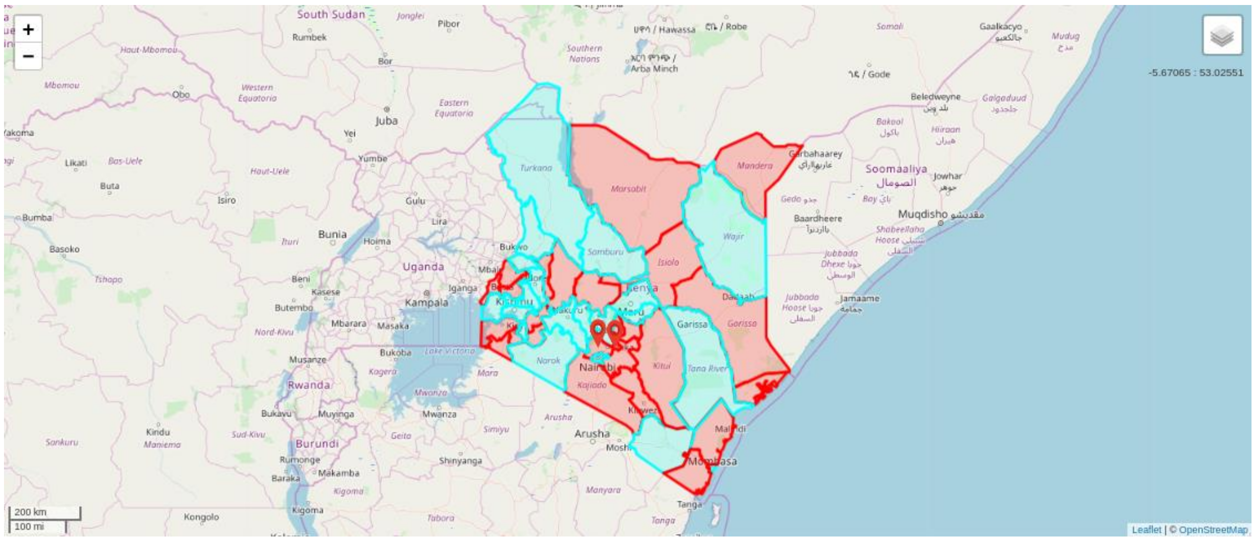
Scale

Scale is one of the vital elements that make up a quality map. In our project, we will add a scale bar that displays the scale based on the interaction of user with the map.

To add a scale bar, add this below this the MousePosition definition.

```
L.control.scale().addTo(map);
```

By default, the bar added at the bottom left position on the map. You can change this to your desired position.



Basically, there are quite a number of controls that one can work with in any Leaflet project. I encourage you to check out for more controls from the Leaflet JS website.

Bonus Tips and Tweaks

In the guide, we have handled many aspects of Leaflet and GeoDjango. I want to add one more bit.

Remember where we added OSMGeoAdmin to our app? Let's add something else.

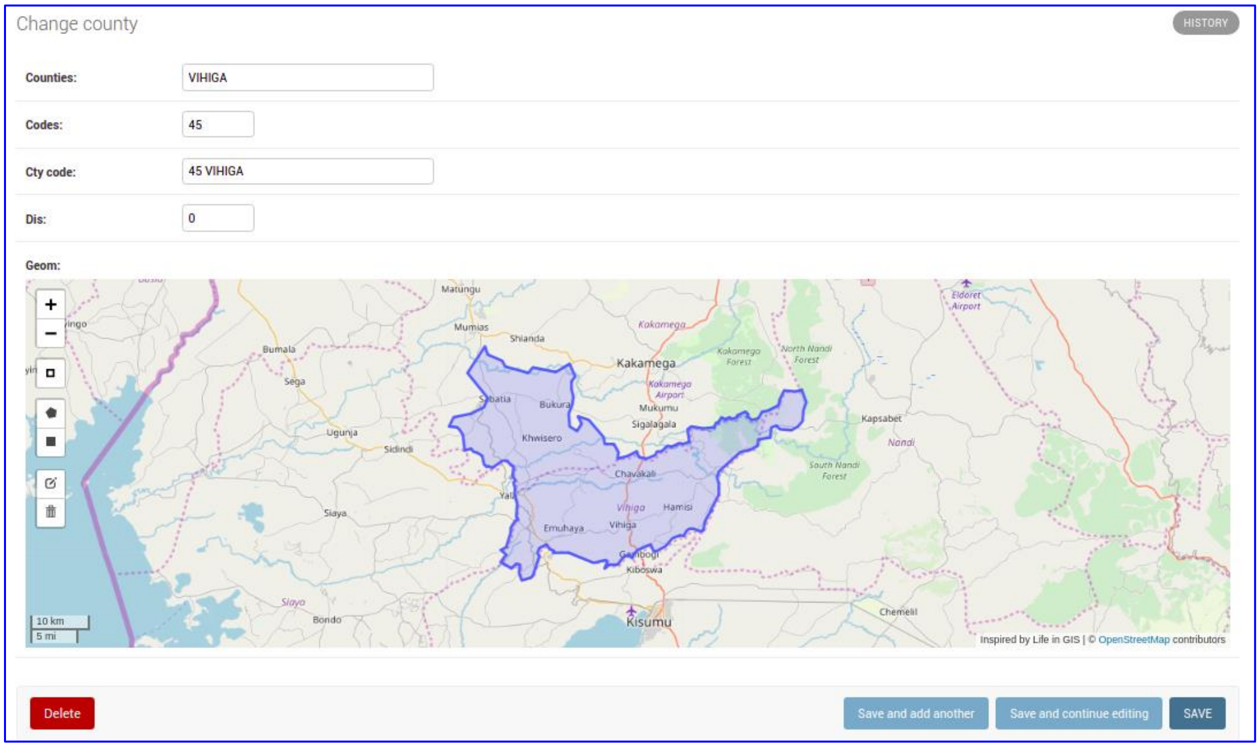
We can advance this bit using Leaflet (In the Django admin). Follow along.

Replace OSMGeoAdmin with LeafletGeoAdmin. Then remember to import the LeafletGeoAdmin class from the leaflet module. Your admin.py file should be as follow;

```
from django.contrib import admin

from .models import Incidence, County

from django.contrib.gis.admin import OSMGeoAdmin

from leaflet.admin import LeafletGeoAdmin



# Register your models here.

class IncidenceAdmin(LeafletGeoAdmin):

    list_display = ('title','date_reported','location')

    search_fields = ('title',)

    filter_fields = ('title','date_reported')

class CountyAdmin(LeafletGeoAdmin):

    list_display = ('counties','cty_code', 'codes')

    search_fields = ('counties',)

    filter_fields = ('counties','cty_code')

admin.site.register(Incidence, IncidenceAdmin)

admin.site.register(County, CountyAdmin)
```

Try accessing either County or Incidence in Django admin. Open one record and see the display of the geo field.

Do you have something like this?



Change county

Counties: VIHIGA

Codes: 45

Cty code: 45 VIHIGA

Dis: 0

Geom:

Inspired by Life in GIS | © OpenStreetMap contributors
Kakamega
Vihiga
Kisumu
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

Looking better? Nice!

One more thing;

Try adding a new record of the Incidence model from the admin. You will not that the geo field is zoomed to [0,0] by default and displays multiple worlds(If there is something like this). It also doesn't limit the zoom levels on the map.

Let's work on this. To customize this bit, let's add a configuration in our settings.py file.

Add this at the bottom of the file.

```
LEAFLET_CONFIG = {

   'DEFAULT_CENTER': (-.023, 36.87),

   'DEFAULT_ZOOM': 5,

   'MAX_ZOOM': 20,

   'MIN_ZOOM':3,

   'SCALE': 'both',

   'ATTRIBUTION_PREFIX': 'Inspired by Life in GIS'

}
```

In this config, we define the center, where the map will be focused on load, maximum and minimum zoom etc. We also add our own attribution.

Try adding a new record again and see the difference.

Cool? Isn’t it?

The bonus ends here! HA!

Conclusion

I believe that this guide has been of help to you. We've done a lot. At this point, Agricom can be hosted and used by any of the interested parties.

The purpose of this guide was to help you learn and understand GeoDjango and other complimenting platforms and libraries such as Leaflet JS. Some technologies might have been discussed shallowly but they fit the purpose. Having practiced with this guide, you are now ready to explore the horizons of web-GIS using Python and through the Django framework, you can achieve much.

I don't have much to talk about this guide, but would encourage you to practice more in the field of web mapping and even share knowledge with the world.

Remember this guide is a written version of what can be found from the GeoDjango Tutorial Series Playlist on YouTube.

Also, remember to subscribe to my YouTube channel and also on the Life in GIS website to receive updates and exciting materials such as this guide.

The code for this project is available on this GitHub Repo.

Cheers and happy hacking!