Introduction

The RESTful Django web application aims to provide access to South Africa's protected areas, catering to potential investors and environmentalists. Leveraging the World Database on Protected Areas (WDPA), this project facilitates exploration, contribution, and management of protected area data. The WDPA, which is the only global database of protected areas, is one of the component databases of the Protected Planet Initiative. Protected Planet is a joint product of UNEP and IUCN, managed by UNEP-WCMC and the IUCN working with governments, communities and collaborating partners. The dataset includes diverse attributes such as location, size, designation type, and management status. By analyzing this dataset, we can identify trends, gaps, and conservation priorities specific to South Africa.

Application

Prerequisites

For the project, the following packages are required to run the application.

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

```
asgiref==3.8.1
Django==5.0.6
djangorestframework==3.15.2
numpy==2.0.0
pandas==2.2.2
python-dateutil==2.9.0.post0
pytz==2024.1
six==1.16.0
sqlparse==0.5.0
tzdata==2024.1
```

Running the Application

Django has an automatic admin interface which I have set up with the following login details.

Username: **admin** Password: **admin**

To run the application, in the command prompt, navigate to the directory containing your Django project's manage.py file, run ***python manage.py runserver***. Open a web browser and navigate to **http://127.0.0.1:8000/**

Data Source

The data was downloaded from https://www.protectedplanet.net/en/search-areas?search_term=south+africa&geo_type=site . The zip contains WDPA_WDOECM_Jun2024_Public_ZAF_csv file, which is the main data source, the file has 1693 rows. The locations file is a simple file that include all the province codes and names in South Africa.

Data Model

The project contains two data models, **Location**, with province information and **ProtectedArea** which contains all the protected areas and the relevant information. There is a one-to-many relationship between the Location and ProtectedArea table.

Each protected area is represented by a model with various attributes.

- wdpaid (int): The unique identifier for the protected area.
- wdpa_pid (str): The PID (Protected Area ID) of the protected area.
- name (str): The name of the protected area.
- desig_eng (str): The English designation of the protected area.
- desig_type (str): The type of designation for the protected area.
- marine (str): Indicates whether the protected area is marine or not.
- rep_m_area (float): The reported marine area of the protected area.
- gis_m_area (float): The GIS (Geographic Information System) marine area of the protected area.
- rep_area (float): The reported area of the protected area.
- gis_area (float): The GIS area of the protected area.
- status (str): The status of the protected area.
- status_yr (int): The year of the status of the protected area.
- gov_type (str): The type of government responsible for the protected area.
- own_type (str): The type of ownership for the protected area.
- mang_auth (str): The managing authority of the protected area.
- sub_loc (Location): The sub-location of the protected area (foreign key).
- parent_iso3 (str): The ISO3 code of the parent location.

Loading Data

If the data is not available in the database, users can access the following links to load the data.

Model	URL	Database Table
Location	http://127.0.0.1:8000/import-locations	core_location
ProtectedArea	http://127.0.0.1:8000/import-protectedareas	core_protectedarea

If the data is available in the database and a user wishes to test the data load, run ***delete from <database table>*.** The application uses a SQLite backend where the 2 tables are populated by the csv files. The data tables are loaded with data using functions defined in the views.py file in the **core** application.

The code demonstrates how to read data from a CSV file and use it to update a database in a Django application, ensuring that duplicate entries are not created. It demonstrates the use of file handling, CSV parsing, and Django's ORM capabilities in Python. The files are stored in the csv-data folder. The file is opened in read mode and csv.DictReader reads each row of the file into a dictionary, using the first row's values as keys. This makes it convenient to access column values by their headers.

```
def import_wdoecm_data(request):
    file_path = 'core\csv-data\WDOECM_data.csv'
    c = 0
    with open(file_path, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            # Create or get the Location instance
            location, created = Location.objects.get_or_create(
            sub_loc=row['SUB_LOC']
            )

            ProtectedArea.objects.create(
```

Django REST Framework

The Django REST Framework provides a powerful toolkit for building RESTful APIs in Django. I used ModelSerializer which simplifies creating serializers based on the existing data models. ModelSerializer automatically generates fields in the serializer that correspond to the fields in your model. This saves you time and effort compared to manually defining each field in a serializer class. The LocationSerializer, ProtectedAreaSerializer and NationalParksSerializer return all fields in the model.

```
class LocationSerializer(serializers.ModelSerializer):
    class Meta:
        model = Location
        fields = '__all__'
```

API Endpoints

Users can use the following links to access the APIs

1. **GET** List all protected areas? Provides an overview of all protected areas within South Africa. http://127.0.0.1:8000/protected-areas/
2. **GET** List all locations? Provides an overview of all provinces within South Africa. http://127.0.0.1:8000/locations/

3. **GET** Return protected areas that have been designated in each province? Allows modifications to existing protected area information. http://127.0.0.1:8000/location-protected-areas/<sub_loc>/
4. **GET** List of national parks, from largest to smallest? Offers sorted data, aiding statistical analysis or prioritization. http://127.0.0.1:8000/national-parks/
5. **POST** Update locations. http://127.0.0.1:8000/location/update/<str:sub_loc>/<str:province_name>'

Unit Tests

There are several test cases in the Django project. The tests cover some essential aspects of the application, including creating locations via POST requests, validating relationships between models, and checking the correctness of the serializers. The tests can be kicked off using the ***python manage.py test*** command in the application directory.

- **PostRequestTestCase** the class evaluates a POST request functionality, updating an existing sub_loc in the locations model. The test_post_api simulates a POST request to the /locations/ endpoint with specific data for sub_loc and province. It then checks if a new location object with the provided sub_loc is updated in the database.

```
class PostRequestTestCase(TestCase):
  def test_post_api(self):
    data = {
      'sub_loc': 'ZA-TEST',
      'province': 'Awesomeland'
    }
    headers = {'Content-type': 'application/x-www-form-urlencoded', 'Accept': 'text/plain'}
    response = self.client.post('/locations/',data)
    location = Location.objects.get(sub_loc='ZA-TEST')
    # Post request to create new location and check if the new location exists
    self.assertEqual(location.sub_loc, 'ZA-TEST')
```

- ProtectedAreaTestCase evaluates the ProtectedArea model. The setup method creates a Location object and a ProtectedArea object associated with that location for use in the subsequent tests.
    - test_protected_area test verifies that the location field of the created ProtectedArea object points to the correct Location object based on sub_loc.
    - test_protected_area_fail is a negative test that ensure the location field is not accidentally equal to a different value (Pampanga in this case).
- ProtectedAreaSerializerTestCase class evaluates the behaviour of the ProtectedAreaSerializer. The setup is similar to ProtectedAreaTestCase,
    - test_contains_expected_fields asserts that the serializer data contains the expected fields (id, name, and location).
    - test_location verifies that the serialized location data matches the actual sub_loc of the associated Location object.

- test_location_fail: This negative test ensures that the serialized location is not accidentally set to a different value (Pampanga).
- test_protected_area: This test checks if the serialized data for the name field matches the actual name of the ProtectedArea object.
- test_protected_area_fail: This negative test ensures that the serialized name is not accidentally different (Pampanga Protected Area).

Critical Evaluation

The tests have limited scope and they could be expanded to cover more functionalities. Including the following checks

- Retrieving existing locations and protected areas.
- Updating locations or protected areas.
- Authentication and authorization (if applicable).
- Error handling scenarios (e.g., invalid data in POST requests).