Advanced Web Development
Midterm Report (CM3035)

UNIVERSITY OF LONDON

"natural_history_project"

specimen_catalog

Author: **200409407**

05 January 2024

2000 words not including Figure legends, References or Appendices

Grading Index

- Report is clearly written

- **Discusses the dataset - what's interesting about it?**

    - Abstract;

    - 1. Introduction;

    - 3. Data Mode**;**

- Discuss the endpoints - what's interesting, complexity?

    - 4.1 Rest End points implementation;

    - Appendix 6 - End Points;

- **Explains how application requirements have been met**

    - All topics;

- Application of taught techniques (Django & DRF) evidenced

    - 2. Basic Functionality;

    - 3. Data Model;

- Report explains and code evidences

    - All topics;

- Evidence of critical evaluation of work/approach

    - All topics;

- Report includes necessary run info (OS, Python version, etc)

    - 2.1 Libraries and Tree structure
    - 4.3 Django admin page
    - 5.2 Implementation details
    - Appendix 3 - Requirements - Location: /requirements.txt

****Misc****

- Bonus points for deploying own app using AWS, Digital Ocean

    - Appendix 7 Project AWS EC2 Deployment

Abstract

The Natural history Catalogue Management System is a web-based application designed with the aim to help explorers, researchers, and enthusiasts involved in the preservation of biodiversity to document their specimens findings. Whilst this database is mainly focused on the Natural History Museum and their assets, this database could be expanded to worldwide linked organisations where other specimens found and stored could be mapped and documented. The project utilises Django web framework with a simple front-end design to allow cataloguing, managing and querying various specimens. Potential future contributions would include the implementation of a user authentication to ensure security of contributions.

1. Introduction

The natural history project attempts to fulfil the requirements of the coursework Advanced Web Development assignment. The aim of this project is to create an online platform to efficiently register specimens, categorising by its taxonomy (Appendix_1), geography and catalogue. The platform also allows the users to update and delete records. It uses data from the Natural History Museum Data Portal[13] (Appendix_2), to populate the assignment database, allowing further queries and data manipulation. This data portal stores many categories across its columns; however, for the purpose of this assignment, it was filtered the relevant and more populated categories, maintaining its original format. This project was built using the Django framework with addition of Django Rest Framework for API development and SQLite for the database. The project goes beyond the assignment requirements by incorporating a front-end to better present its features.

The following topics address the accomplished requirements of this assignment.

2. Basic Functionality(R1)

2.1 Libraries and Tree structure[10]

This project was built using **Ubuntu O**S, and to deploy the project, several libraries can be found in the 'requirements.txt' file. The Appendix_3, allows collaboration and replicability of the project dependencies and version. Moreover, the Appendix_4 provides an overview tree structure of the project to better understand and navigate.

2.2 Overview

The Natural History Catalogue application is a modest web application designed to help an efficient exploration and organisation of specimens in a scientific collection. The application works around three modules: Specimen, Taxonomy and expeditions (Figure_1).

Specimens:

1.1. Allows users to input, update and delete specimens using an intuitive interface;
1.2. The system integrates taxonomy classification used in biology;
1.3. There is an interconnectivity between the specimen to a taxonomy and an expedition.

Taxonomy:

1.1. It is structured following the taxonomy classification

1.2. Helps researchers to systematically document their specimens according to their taxonomy and also identify their specimens;

Expeditions

1.1. Allows the users to track the location of their findings by Continent, Country and State.

1.2. Can help give the environment context.

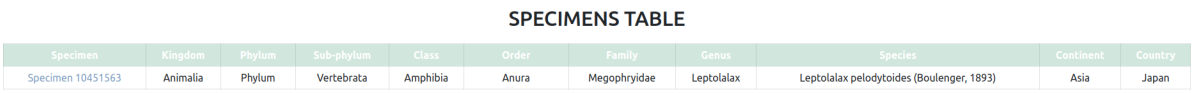


Figure_1 - Specimens table

Moreover, the app includes enhanced search functionalities based on taxonomy attributes and expedition details (Figure_2).

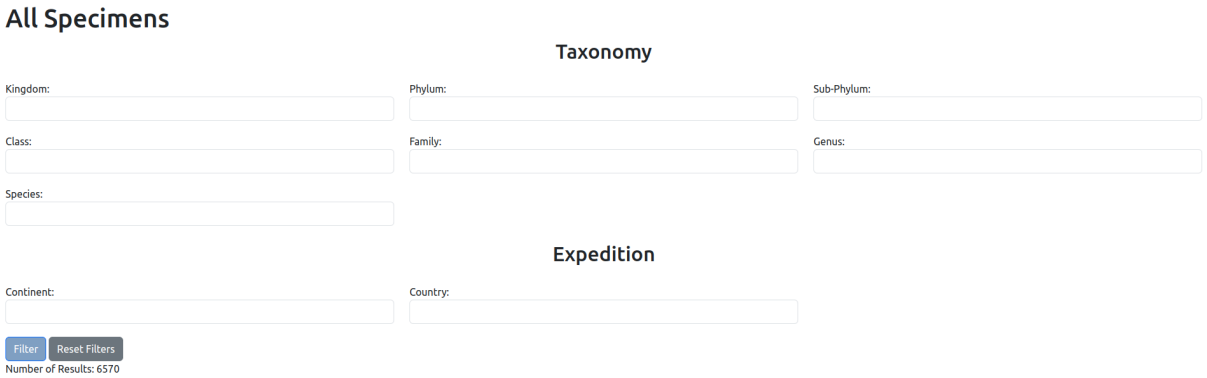


Figure_2 - Taxonomy and Expedition Filter

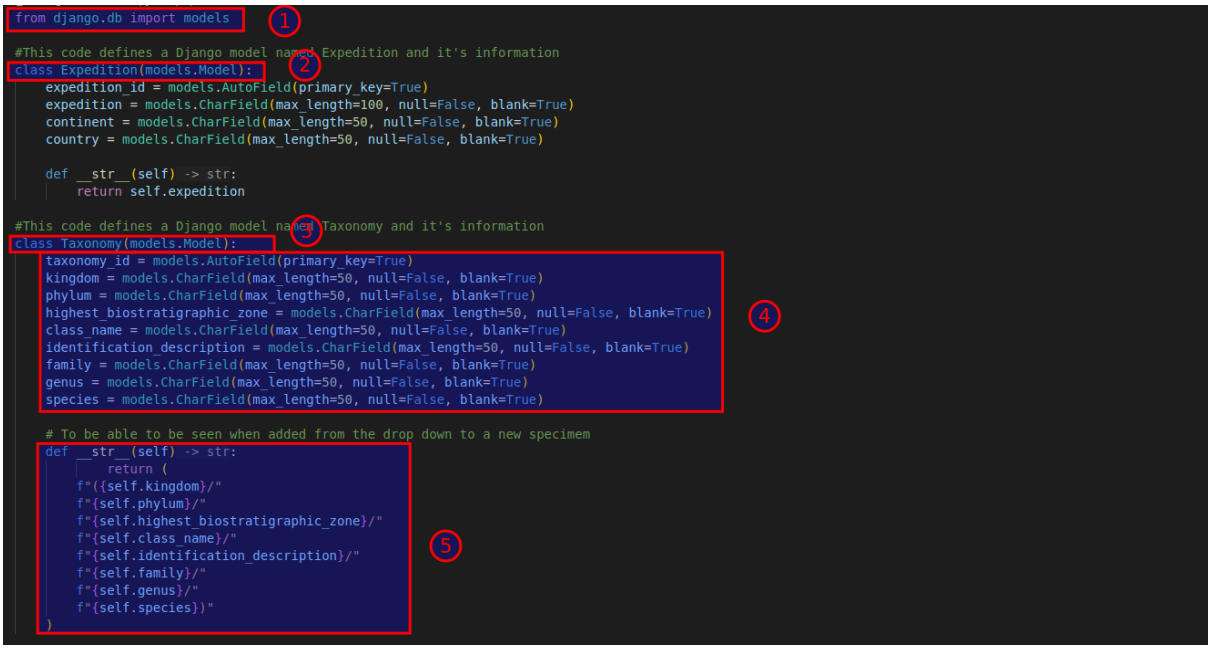
Finally, the application provides the user an efficient pagination[4] allowing the users to handle large datasets (Figure_3).

Figure_3 - Paginator

2.2 Models[5]

In the Django framework, models and migrations are important to define the structure of the database and manage it. Models are located under the models.py (Figure_4) file and utilise from django.db import models(1) structure using classes(2) to define the attributes and their

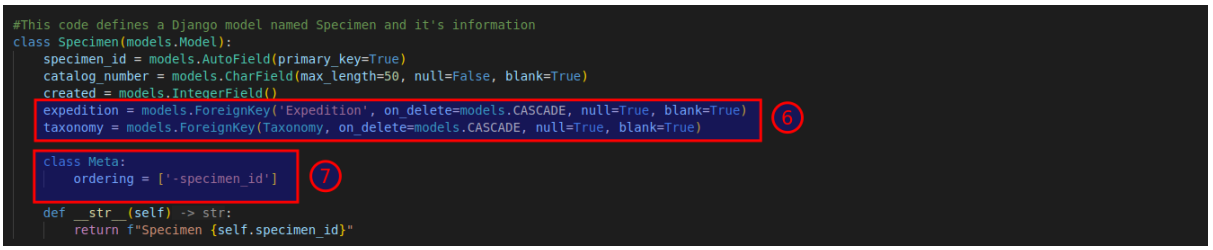
type of data it will store[3 & 4]. Most attributes were given as *Charfield* with a set maximum length and not required to fill.




```python
from django.db import models

#This code defines a Django model named Expedition and it's information
class Expedition(models.Model):
    expedition_id = models.AutoField(primary_key=True)
    expedition = models.CharField(max_length=100, null=False, blank=True)
    continent = models.CharField(max_length=50, null=False, blank=True)
    country = models.CharField(max_length=50, null=False, blank=True)

    def __str__(self) -> str:
        return self.expedition

#This code defines a Django model named Taxonomy and it's information
class Taxonomy(models.Model):
    taxonomy_id = models.AutoField(primary_key=True)
    kingdom = models.CharField(max_length=50, null=False, blank=True)
    phylum = models.CharField(max_length=50, null=False, blank=True)
    highest_biostratigraphic_zone = models.CharField(max_length=50, null=False, blank=True)
    class_name = models.CharField(max_length=50, null=False, blank=True)
    identification_description = models.CharField(max_length=50, null=False, blank=True)
    family = models.CharField(max_length=50, null=False, blank=True)
    genus = models.CharField(max_length=50, null=False, blank=True)
    species = models.CharField(max_length=50, null=False, blank=True)

    # To be able to be seen when added from the drop down to a new specimem
    def __str__(self) -> str:
        return (
        f"({self.kingdom}/"
        f"{self.phylum}/"
        f"{self.highest_biostratigraphic_zone}/"
        f"{self.class_name}/"
        f"{self.identification_description}/"
        f"{self.family}/"
        f"{self.genus}/"
        f"{self.species})"
    )
```

Figure_4 - models.py

Furthermore, models allow to establish relationships between the other models using primary key / foreign key[3 & 6] (Figure_5) and Meta class within a model allows organising the dataset by a specific field[7].

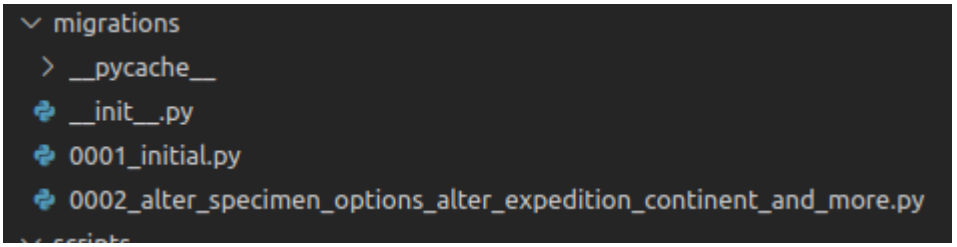



```python
#This code defines a Django model named Specimen and it's information
class Specimen(models.Model):
    specimen_id = models.AutoField(primary_key=True)
    catalog_number = models.CharField(max_length=50, null=False, blank=True)
    created = models.IntegerField()
    expedition = models.ForeignKey('Expedition', on_delete=models.CASCADE, null=True, blank=True)
    taxonomy = models.ForeignKey(Taxonomy, on_delete=models.CASCADE, null=True, blank=True)

    class Meta:
        ordering = ['-specimen_id']

    def __str__(self) -> str:
        return f"Specimen {self.specimen_id}"
```

Figure_5 - models.py attributes

2.3 Migrations[6]

Migrations are used to propagate changes to the models into a database schema. It allows a version control to track changes to the database schema overtime (Figure_6)



Figur_6 - Migrations

2.4 Forms[7]

Forms fields were automatically generated based on the model's fields maintaining consistency and reducing redundancy. Three forms, 'SpecimenForm', 'ExpeditionForm' and 'TaxonomyForm' were created to easily update the database and 'NewSpecimenForm' for creation of a new specimen. Moreover, to maintain accuracy and security of the data being stored in the database, validators and error handling was used throughout the project (Figure_7).

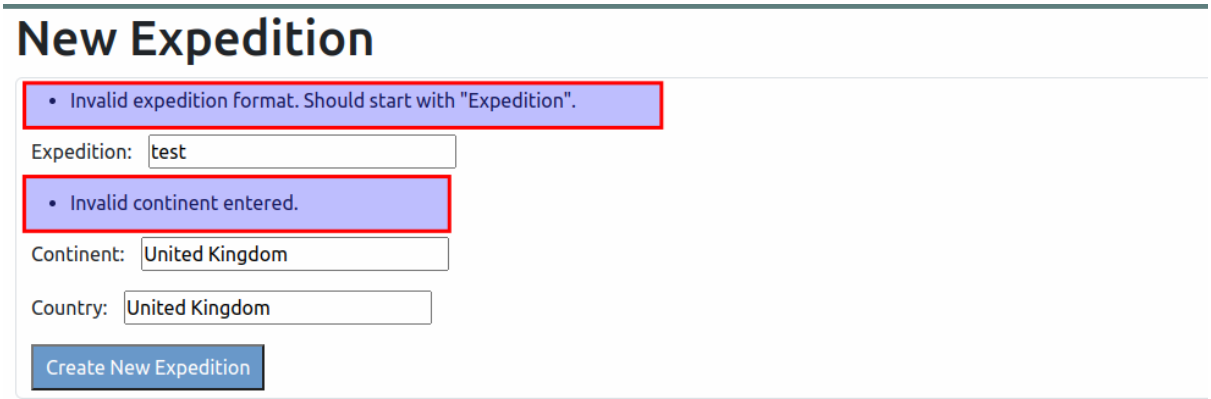


Figure_7 - Form in the webpage

As an example, the form.py there are validators that check the fields "Continent"(1) and "Country"(2) from the form 'Expedition' check against a specific list for validation (Figure_8).

Figure_8 - Expedition form

On the ‘Taxonomy’ form the user is required to type at least 3 characters (Figure_9).

```
def clean_field_length(self, field_name, min_length) -> Any:
    field_value = self.cleaned_data[field_name]
    label = self.fields[field_name].label
    # Ensures field value is at least min_length characters long
    if len(field_value) < min_length:
        raise ValidationError(f'{label} must be at least {min_length} characters long.')

    return field_value

def clean_kingdom(self) -> Any:
    return self.clean_field_length('kingdom', 3)

def clean_phylum(self) -> Any:
    return self.clean_field_length('phylum', 3)

def clean_highest_biostratigraphic_zone(self) -> Any:
    return self.clean_field_length('highest_biostratigraphic_zone', 3)

def clean_class_name(self) -> Any:
    return self.clean_field_length('class_name', 3)

def clean_identification_description(self) -> Any:
    return self.clean_field_length('identification_description', 3)

def clean_family(self) -> Any:
    return self.clean_field_length('family', 3)

def clean_genus(self) -> Any:
    return self.clean_field_length('genus', 3)

def clean_species(self) -> Any:
    return self.clean_field_length('species', 3)
```

Figure 9 - Taxonomy form

Finally, in the ‘Specimen’ form the catalog_number has max_ lengths for its code = [4,2,2,4][(1)]. (Figure_10).

```
# Cleans and validate the catalog_number field
def clean_catalog_number(self) -> Any:

    # Gets the catalog_number from the cleaned_data dictionary
    catalog_number = self.cleaned_data['catalog_number']

    # Validates catalog number format (four parts separated by dots)
    parts = catalog_number.split('.')

    if len(parts) != 4:
        # Raises a ValidationError if the format is not correct
        raise ValidationError('Invalid catalog number format. Should have 4 parts separated by dots.')

    # Defines maximum lengths for each part
    max_lengths = [4, 2, 2, 4]

    for part, max_length in zip(parts, max_lengths):
        try:
            # Checks if each part is a non-negative integer and within the specified maximum length
            value = int(part)
            if value < 0 or len(part) > max_length:
                # Raises a ValueError if the part is not a valid integer or exceeds the maximum length
                raise ValueError
        except ValueError:
            # Raises a ValidationError if the part is not a valid integer
            raise ValidationError(f'Invalid catalog number. Parts must be non-negative integers with a maximum length of {max_length}.')

    # Returns the cleaned catalog_number if it passes validation
    return catalog_number
```

Figure_10 - Specimen Form

2.5 Serializers[11]

Serialization is an important aspect that facilitates efficient data exchange. The application allows users to convert complex data types into JSON (Figure_11). It used a standard ‘ModeSerializer’ class from Django Rest Framework which automatically generates the classes based on the models. Please see the end points in the Appendix_5.

```python
from rest_framework import serializers
from .models import Expedition, Taxonomy, Specimen

class ExpeditionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Expedition
        fields = '__all__'

class TaxonomySerializer(serializers.ModelSerializer):
    class Meta:
        model = Taxonomy
        fields = '__all__'

class SpecimenSerializer(serializers.ModelSerializer):
    expedition = ExpeditionSerializer()
    taxonomy = TaxonomySerializer()

    class Meta:
        model = Specimen
        fields = '__all__'

    def create(self, validated_data) -> Specimen:
        expedition_data = validated_data.pop('expedition', None)
        taxonomy_data = validated_data.pop('taxonomy', None)

        specimen = Specimen.objects.create(**validated_data)

        if expedition_data:
            Expedition.objects.create(specimen=specimen, **expedition_data)

        if taxonomy_data:
            Taxonomy.objects.create(specimen=specimen, **taxonomy_data)

        return specimen
```

Figure_11 - serializers.py

2.6 Tests[8]

Testing the app was vital to ensure code reliability, functionality and overall stability. It used the factory_boy method to test the models, API tests for CRUD operations and view tests for template rendering and form handling (Figure_12).

```
Found 37 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....................................
----------------------------------------------------------------------
Ran 37 tests in 6.107s

OK
Destroying test database for alias 'default'...
```

Figure_12 - Test output

3. Data Model(R2)

The data model chosen evolves from three main entities: ‘Taxonomy’, ‘Expedition’ and ‘Specimen. ‘Taxonomy’ represents the classification hierarchy for organisms from Kingdom to species; ‘Expedition’ constitutes information about the geography of a certain scientific expedition; Specimen represents unique identifiers and has a relationship using foreign keys with specific expeditions and taxonomies. This relational model guarantees an efficient organisation, retrieval and maintenance of all data but also provides a clear structure of all the data stored (Figure_14).

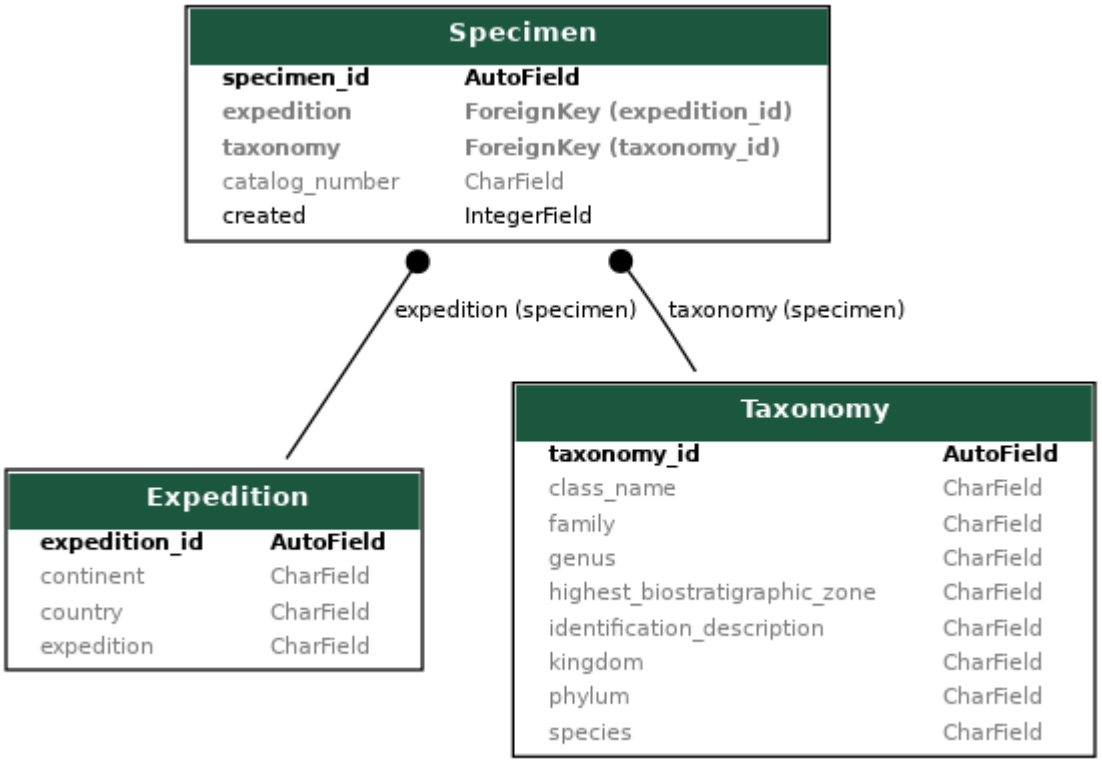


Figure_14 - Database Schema - `python manage.py graph_models specimen_catalog -o models.png`

During the design process I encountered several challenges which included the selection of the most important fields for my model, due to the extensive dataset, consisting of 143 columns by 6529 rows. Furthermore, some fields were not clear, have ambiguous codes. It required careful consideration when designing the structured data, especially when loading the data into the application database that would align with the complexities of the dataset, while maintaining clarity and integrity of the data.

4. Implementation of App Components(R3)

4.1 REST Endpoints Implementation[11]

Defining consistent and clear endpoints are important during the design of an application as it provides a structured way for end users to communicate with the server. Below is a list of all url patterns (Figure_15) used in this application and associated end points. Please also see end points in more detail in the Appendix_6.

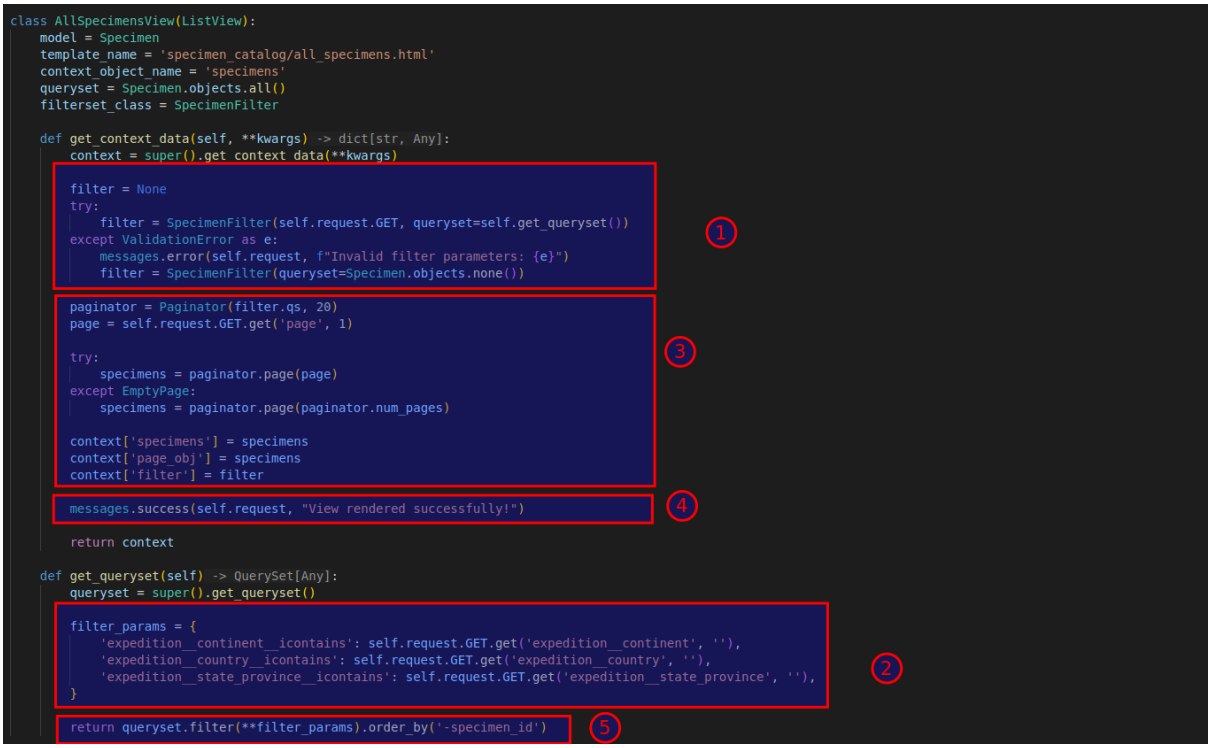
```python
from .views import (AllSpecimensView, SpecimenDetailView, ExpeditionUpdateView, TaxonomyUpdateView,
                    SpecimenDeleteView, NewSpecimenView, NewTaxonomyView, NewExpeditionView,
                    )

# URL patterns defines routes for specimen_catalog app
urlpatterns = [
    # Home page
    path('', views.index, name='index'),
    # Displays all specimens listed in a table
    path('all_specimens/', AllSpecimensView.as_view(), name='all_specimens'),
    # View details of a specific specimen
    path('specimen/detail/<int:pk>/', SpecimenDetailView.as_view(), name='specimen_detail'),
    # Updates a specific specimen record
    path('specimen/<int:pk>/update/', views.SpecimenUpdateView.as_view(), name='specimen_update'),
    # Updates a expediation associated with a specimen
    path('expedition_update/<int:pk>/', ExpeditionUpdateView.as_view(), name='expedition_update'),
    # Updates a taxonomy associated with a specimen
    path('taxonomy_update/<int:specimen_pk>/', TaxonomyUpdateView.as_view(), name='taxonomy_update'),
    # Deletes a specific specimen
    path('specimen/<int:pk>/delete/', SpecimenDeleteView.as_view(), name='specimen_delete'),
    # Creates a new specimen
    path('new_specimen/', NewSpecimenView.as_view(), name='new_specimen'),
    # Creates a new taxonomy
    path('new_taxonomy/', NewTaxonomyView.as_view(), name='new_taxonomy'),
    # Creates a new expedition
    path('new_expedition/', NewExpeditionView.as_view(), name='new_expedition'),

    # API views #
    # SPECIMENS
    path('api/specimens/', views.SpecimenListAPIView.as_view(), name='specimen-list'),
    path('api/specimens/<int:pk>/', views.SpecimenDetailAPIView.as_view(), name='specimen-detail'),
    # EXPEDITION
    path('api/expeditions/', views.ExpeditionListAPIView.as_view(), name='expedition-list'),
    path('api/expeditions/<int:pk>/', views.ExpeditionDetailAPIView.as_view(), name='expedition-detail'),
    # TAXONOMIES
    path('api/taxonomies/', views.TaxonomyListAPIView.as_view(), name='taxonomy-list'),
    path('api/taxonomies/<int:pk>/', views.TaxonomyDetailAPIView.as_view(), name='taxonomy-detail'),
]
```

Figure_15 - urls.py

4.2 Views[11]

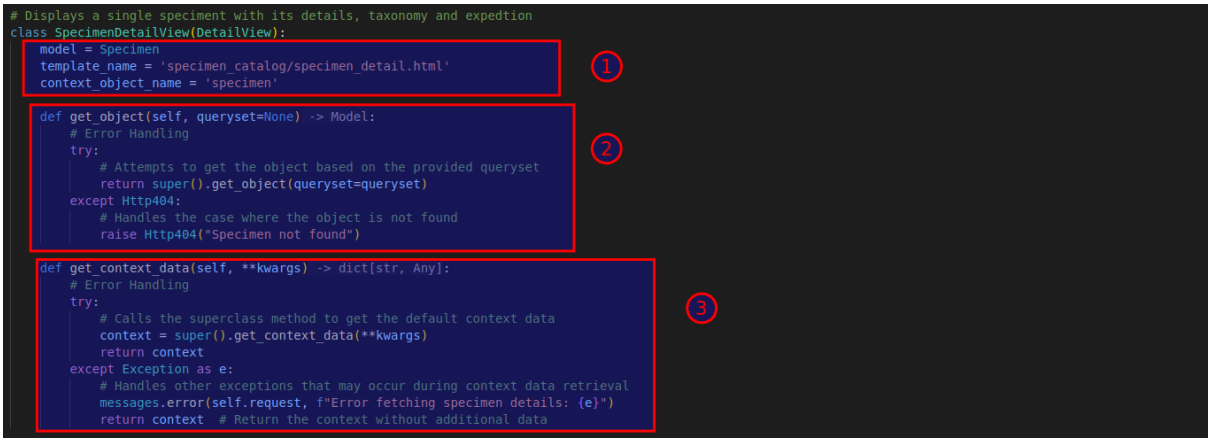
The design of this Django project showcases an effective use of Django features utilising generic views, form handling and error management. The use of generic views as ListView, DetailView, and UpdateView greatly simplifies the codebase keeping it legible and easy to maintain. Additionally, the distinct separation of the views creates an organised structure, promoting modularity laying a strong foundation for future scalability.

AllSpecimensView



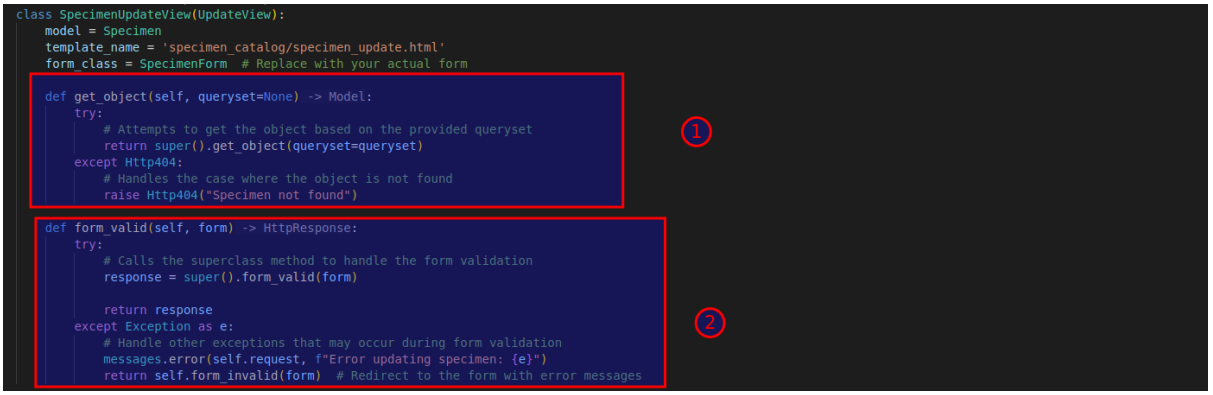
The 'AllSpecimensView' class is a 'ListView' class and is associated with the model 'Specimen'. It handles a get method that initialises a queryset using the 'Specimen.objects.all() to retrieve all the 'Specimen' objects. Also, applies filtering to the database using the 'SpecimenFilter' class(1) and makes request attributes to the expedition module(2). This class also adds pagination(3) to the template 'all_specimens.html' that renders all the data. Finally, there is a message(4) that confirms that the data was rendered successfully, and the table content will display in a descendent order(5) based on 'specimen_id'.

SpecimenDetailView



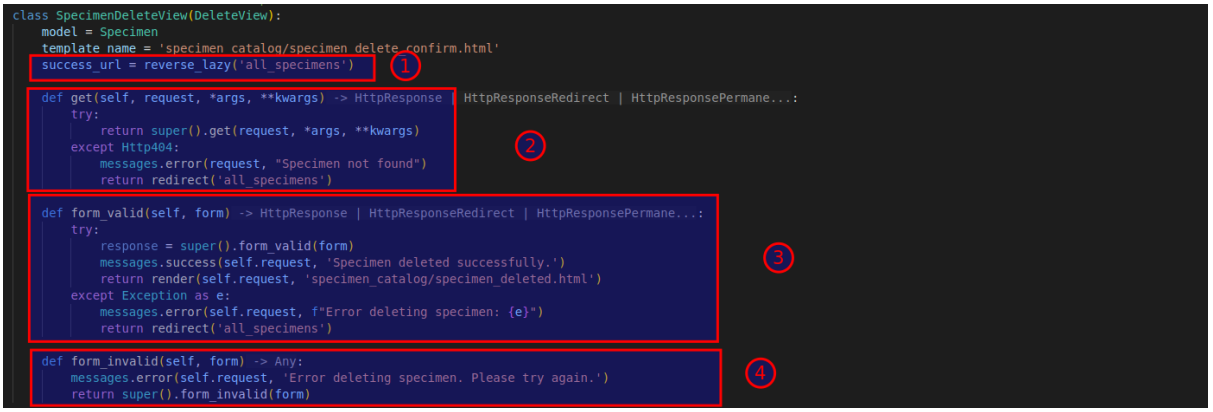
SpecimenDetailView implements a model ‘Specimen’ instance and renders the view into the specimen_detail.html template(1). When accessed through the URL at the end is the primary key that will be provided as a parameter. ‘super().get_object(queryset=queryset)’{2} is called to the default behaviour of ‘DetailView’ retrieving the object based on the ‘pk’ translating the query into the corresponding object. If the object queried is not found, it raises an “HTTP404” exception indicating that the specimen was not found in the database(3).

SpecimenUpdateView, ExpeditionUpdateView & TaxonomyUpdateView



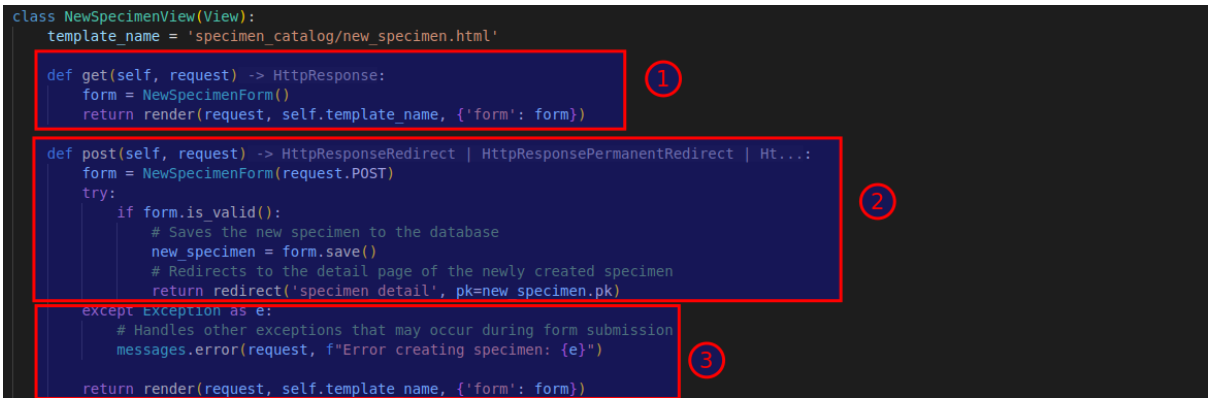
This class is designed to handle updating ‘Specimen’ objects and render into the specimen_update.html template. It works the same way as the SpecimenDetailView to retrieve the ‘specimen_id’{1} object but displays as in a form due the Django ‘updateView’. It will raise an exception if it does not find the specimen in the database. The class also has a form validation that checks if the data has been inserted appropriately{2}.

SpecimenDeleteView



This class inherits from ‘DeleteView’ and renders into the specimen_delete_confirm.html template. If the ‘Specimen’ is successfully deleted it renders a success message page ‘specimen_deleted.html’[3] template and the view redirects to ‘all_specimens.html[2] template using the Django utility function reverse_lazy[1]. If the form is not valid it will prompt to try to delete again[4].

NewSpecimenView, NewTaxonomyView & NewExpeditionView

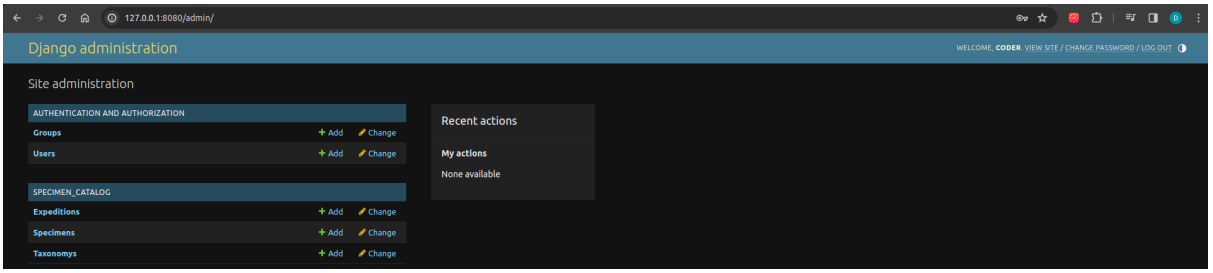


These three classes' views are designed to create new instances, using the HTTP GET method, and based on the form fields and validation. It renders the corresponding template and passes the form as context data[1]. The post method is called when the user submits the form. If the form.is_valid() it will be saved to the database using the ‘form.save()’ function and the page redirects to the ‘specimen_detail.html’ page that takes that ‘pk’ parameter[2].

Once again, there is exception handling that checks if the form was submitted properly and if not, will re-render displaying errors in the corresponding form fields[3].

4.3 Django admin page[12]

The django admin aims to simplify data management using a user-friendly interface to perform administrative tasks such as view, add, modify and delete records. It serves as a quick and convenient way to interact with the database without the need to build a custom interface for each model.



It automatically generates form based on model definitions (Figure_16)

```python
# Imports the models
from django.contrib import admin
from .models import Expedition, Taxonomy, Specimen

# Register the Expedition model with the Django Admin interface
@admin.register(Expedition)
class ExpeditionAdmin(admin.ModelAdmin):
    list_display = ('expedition_id', 'expedition', 'continent', 'country')

# Register the Taxonomy model with the Django Admin interface
@admin.register(Taxonomy)
class TaxonomyAdmin(admin.ModelAdmin):
    list_display = ('taxonomy_id', 'kingdom', 'phylum','highest_biostratigraphic_zone',
                    'class_name', 'family', 'genus', 'species')

# Register the Specimen model with the Django Admin interface
@admin.register(Specimen)
class SpecimenAdmin(admin.ModelAdmin):
    list_display = ('specimen_id', 'catalog_number', 'created', 'taxonomy', 'expedition')
```

Figure_16 - admin forms

To access the Django admin panel with superuser privileges, use the following credentials:

Database superuser login: *coder*

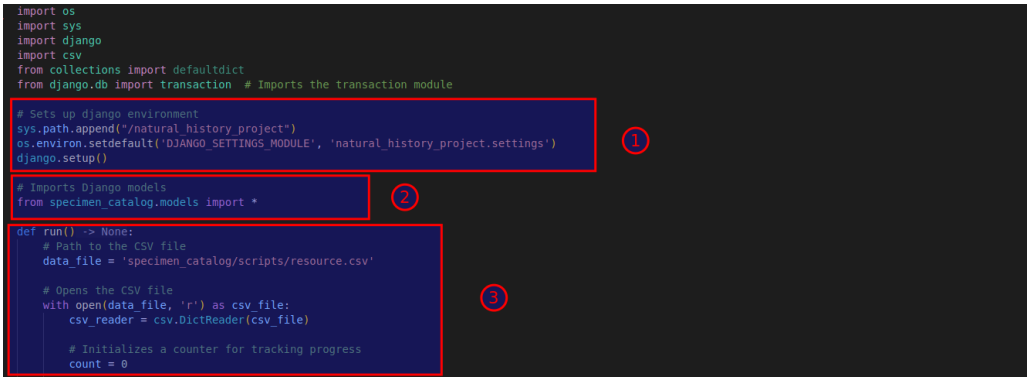
Password: *coder12*

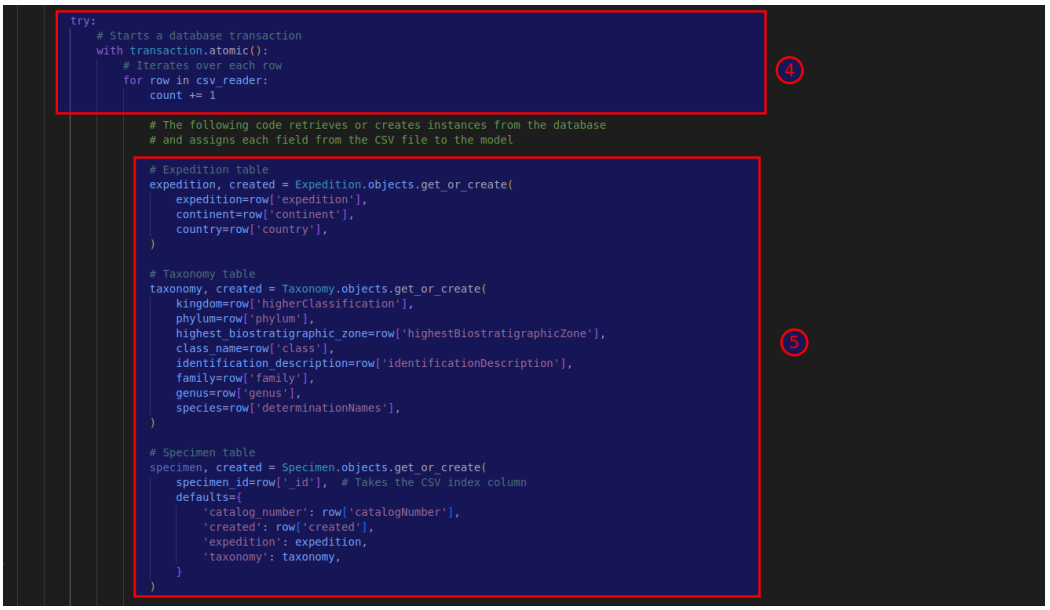
Email: *coder@email.com*

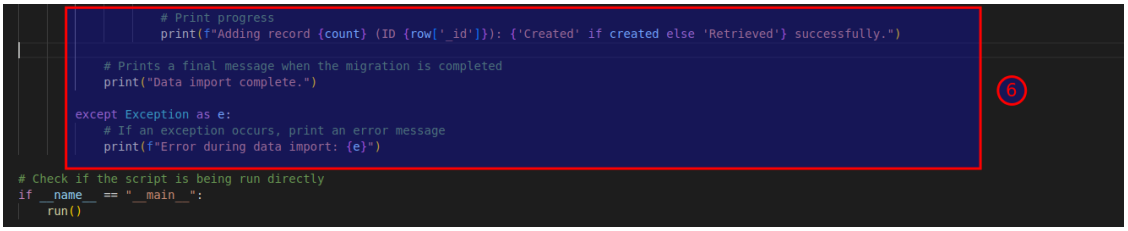
5. Bulk Loading Data(R4)

5.1 Method of Bulk Loading Data:

The dataset consists of 6543 rows spread over 143 columns and was efficiently migrated using a specific build script tailored for one-time or infrequent data loading. The script uses the column name, instead of column numbers, to find the data, making it more precise in case of new columns being added to the database. It also uses a 'get_or_create' method(5) for each model allowing an efficient batch processing that minimises the number of database queries. Therefore this script interacts with the project models by appending(1) the project path to the system path(2). Then, it reads the csv file using 'csv.DictReader'(3) to interpret rows as dictionaries. Afterwards, the script iterates(4) over each row retrieving the instances from the Django models assigning the selected columns into the corresponding model fields(5). The process completes displaying a final message indication that the import was completed(6).







5.2 Implementation Details

The command needed to run the script is:

Script location :

natural_history_project/specimen_catalog/scripts/populate_specimen_catalog.py

[python manage.py runscript --verbosity 2
specimen_catalog.scripts.populate_specimen_catalog]

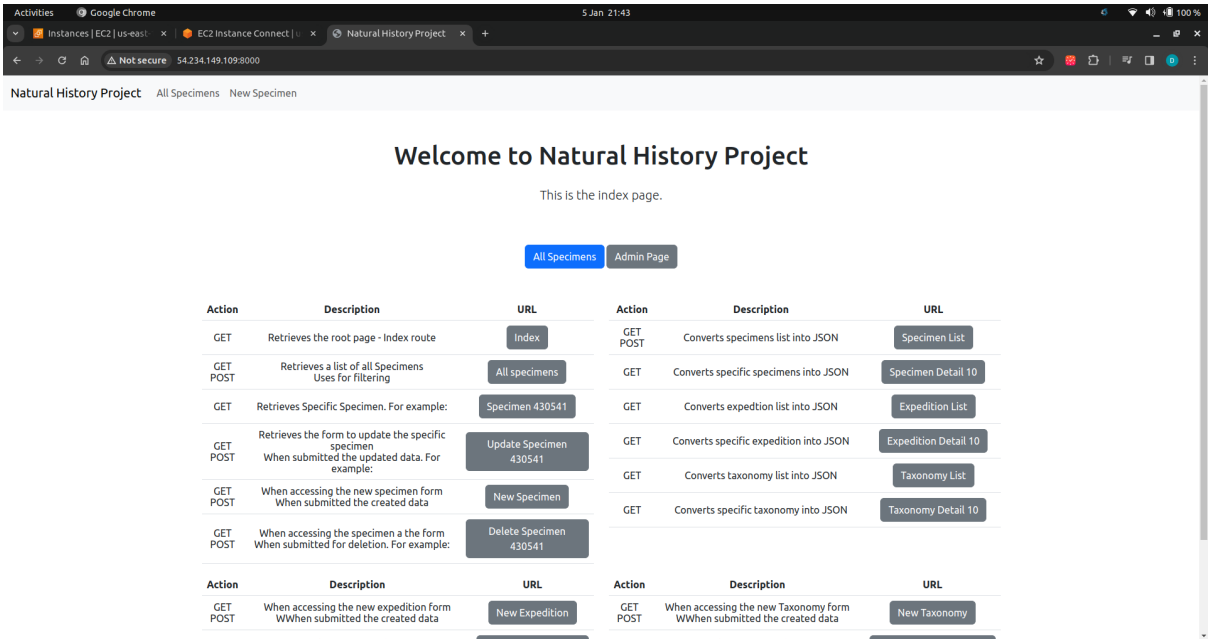
5.3 Ensuring Data Integrity:

The designed script relies on Django models to define the schema, and any issues with the schema would be caught during the database migration. It also uses a ‘transaction.atomic()’ insertion block process, ensuring that the entire process of reading the CSV file and inserting records into the database is treated as a single transaction. If any part of the process fails, the entire transaction is rolled back and the database remains unchanged, maintaining the integrity and consistency of the data.

5.4 Deployment

The project was successfully deployed into AWS cloud using EC2 instance. This allowed us to test the application using a production environment and test its responsiveness. Although the database has 6543 rows, using the paginator helps the page to load faster and display information using modularity. Please use the following link to visit the webpage:

Appendix 7 - http://54.234.149.109:8000/



6. Conclusion

In conclusion, the Specimen_catalog App effectively addresses the requirements outlined within the report, offering a powerful platform for exploring and organising specimen collections. The developed code guarantees integrity and efficiency, with key functionalities focused around Specimen, Taxonomy, and Expeditions modules. Despite challenges, including integrating records loading scripts, the mission's objectives were met through strategic making plans and hassle-solving. The Specimen_catalog App was proudly a result of many hours of work and troubleshooting, and undoubtedly a significant contribution to the expansion of my knowledge.

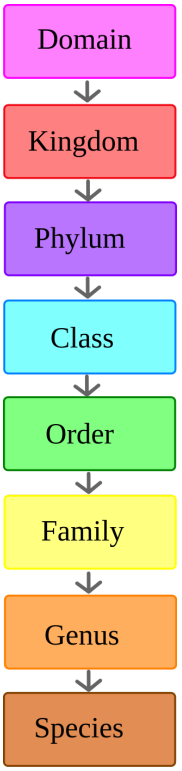
7. References

8. Deploying - Tanveer, S. (2023). *Deploying a Django Project on AWS EC2: A Step-by-Step Guide*. [online] Medium. Available at: https://medium.com/@awseng12/deploying-a-django-project-on-aws-ec2-a-step-by-step-guide-787f646b5e77 [Accessed 3 Jan. 2024].
9. *Django*. (n.d.). Django Project Models. https://docs.djangoproject.com/en/5.0/topics/db/models/
10. django-filter.readthedocs.io. (n.d.). *django-filter — django-filter 22.1 documentation*. [online] Available at: https://django-filter.readthedocs.io/en/stable/.
11. Django Forum. (2020). *django-filter & pagination*. [online] Available at: https://forum.djangoproject.com/t/django-filter-pagination/1556 [Accessed 6 Jan. 2024].
12. Django Project Models. (n.d.). *Django*. [online] Available at: https://docs.djangoproject.com/en/5.0/topics/db/models/ [Accessed 6 Jan. 2024].
13. Django Project Migrations. (n.d.). *Django*. [online] Available at: https://docs.djangoproject.com/en/5.0/topics/migrations/ [Accessed 6 Jan. 2024].
14. Django Project Forms. (n.d.). *Django*. [online] Available at: https://docs.djangoproject.com/en/5.0/topics/forms/.
15. Django Project Tests. (n.d.). *Django*. [online] Available at: https://docs.djangoproject.com/en/5.0/topics/testing/overview/ [Accessed 6 Jan. 2024].
16. *Django*. (n.d.). Django Project. https://docs.djangoproject.com/en/5.0/topics/migrations/#:~:text=Migrations%20are%20Django's%20way%20of,problems%20you%20might%20run%20into
17. Django Packages. (n.d.). *Django Packages : Reusable apps, sites and tools directory for Django*. [online] Available at: https://djangopackages.org/.
18. EndPoints - www.django-rest-framework.org. (n.d.). *Quickstart - Django REST framework*. [online] Available at: https://www.django-rest-framework.org/tutorial/quickstart/
19. Django Project. (n.d.). *Django*. [online] Available at: https://docs.djangoproject.com/en/5.0/ref/contrib/admin/ [Accessed 6 Jan. 2024].

20. Natural History Museum (2023). Data Portal query on 1 resources created at 2023-12-05 18:02:10.728085 PID https://doi.org/10.5519/qd.i5aq5l5f

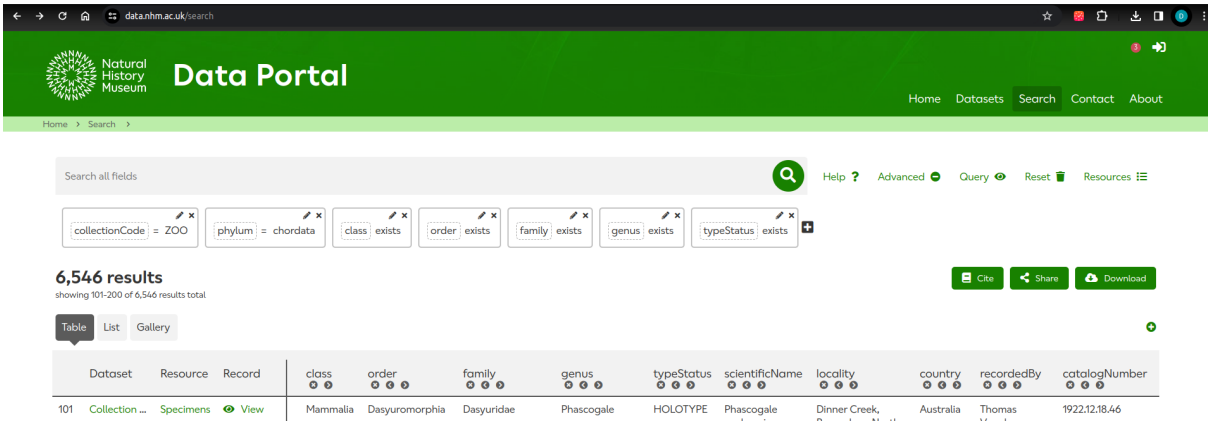
8. Appendices

Appendix 1 - Wikipedia Taxonomy Classification



Source - https://simple.wikipedia.org/wiki/Taxonomy

Appendix 2 - Natural History Museum - Data Portal



Source - https://data.nhm.ac.uk/doi/10.5519/qd.i5aq5l5f

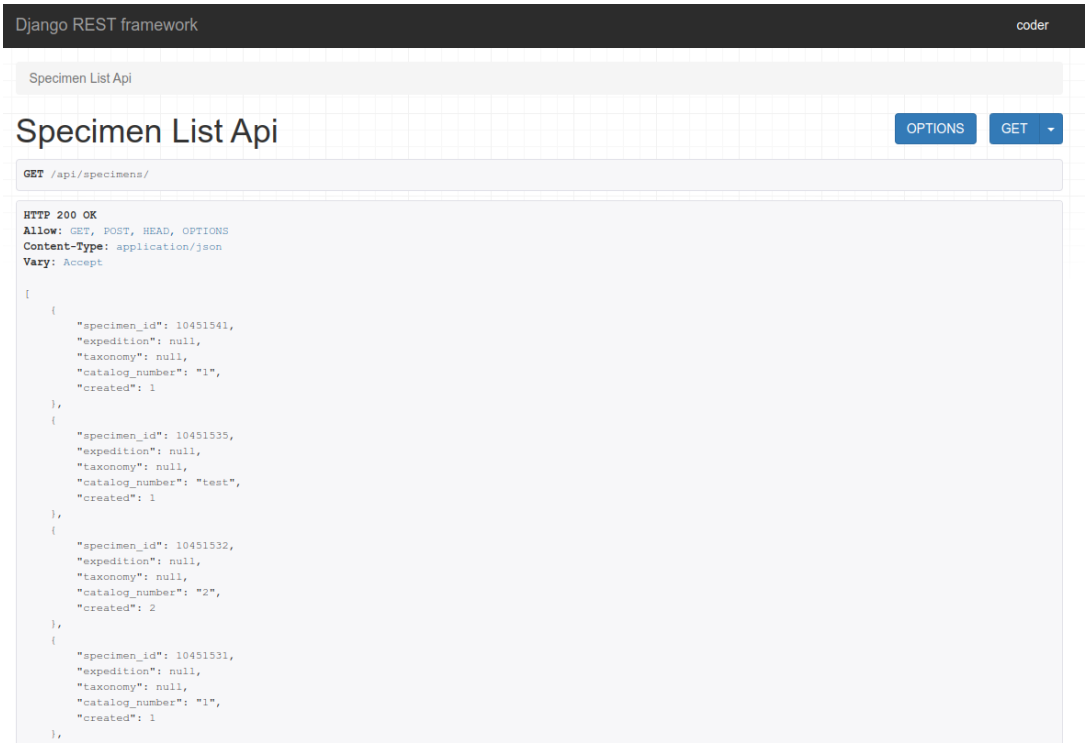
Appendix 3 - Requirements - Location: /requirements.txt

```
asgiref==3.7.2
certifi==2023.7.22
cffi==1.15.1
charset-normalizer==3.2.0
crispy-bootstrap5==0.7
cryptography==41.0.4
defusedxml==0.7.1
Django==4.2.3
django-crispy-forms==2.0
django-extensions==3.2.3
django-filter==23.2
django-tables2==2.7.0
django-taggit==4.0.0
djangorestframework==3.14.0
djangoviz==0.1.1
factory-boy==3.3.0
Faker==22.0.0
geographiclib==2.0
graphqlclient==0.2.4
gunicorn==20.1.0
idna==3.4
importlib-metadata==6.8.0
Markdown==3.4.3
oauthlib==3.2.2
pycountry==23.12.11
pycparser==2.21
pydotplus==2.0.2
PyJWT==2.7.0
pyparsing==3.1.1
python-dateutil==2.8.2
python3-openid==3.2.0
pytz==2023.3
requests==2.31.0
requests-oauthlib==1.3.1
six==1.16.0
social-auth-app-django==5.2.0
social-auth-core==4.4.2
sqlparse==0.4.4
typing_extensions==4.7.1
urllib3==2.0.7
whitenoise==6.4.0
zipp==3.15.0
```

Appendix 4 - Tree - Location: /tree.txt

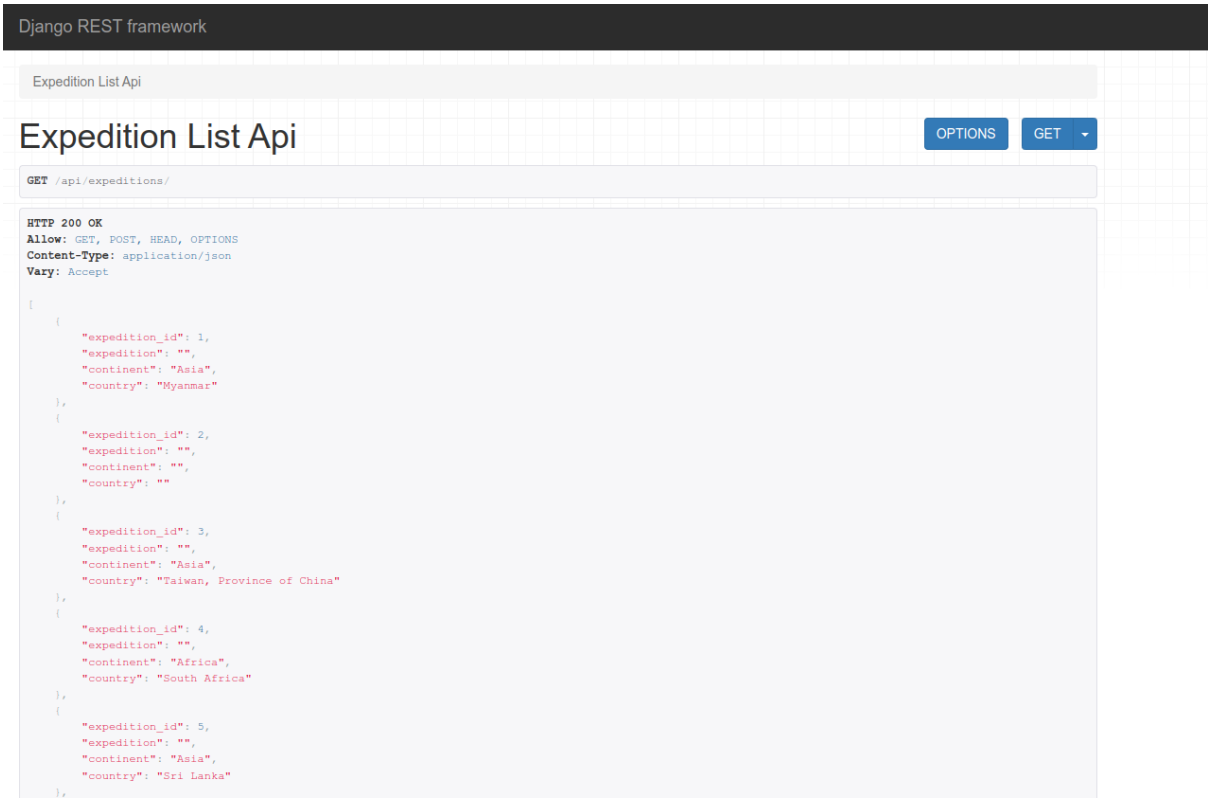
```
.
├── db.sqlite3
├── manage.py
├── models.png
├── natural_history_project
│   ├── asgi.py
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── requirements.txt
├── specimen_catalog
│   ├── admin.py
│   ├── apps.py
│   ├── filters.py
│   ├── forms.py
│   ├── __init__.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_alter_specimen_created.py
│   │   ├── 0003_remove_specimen_created.py
│   │   ├── 0004_specimen_continent_specimen_country.py
│   │   ├── 0005_remove_specimen_continent_remove_specimen_country.py
│   │   └── __init__.py
│   ├── model_factories.py
│   ├── models.py
│   ├── scripts
│   │   ├── populate_specimen_catalog.py
│   │   └── resource.csv
│   ├── serializers.py
│   ├── static
│   │   └── css
│   │       └── styles.css
│   ├── templates
│   │   └── specimen_catalog
│   │       ├── all_specimens.html
│   │       ├── api_urls.html
│   │       ├── base.html
│   │       ├── expedition_list.html
│   │       ├── expedition_update.html
│   │       ├── filters.html
│   │       ├── footer.html
│   │       ├── header.html
│   │       ├── index.html
│   │       ├── master.html
│   │       ├── new_expedition.html
│   │       ├── new_specimen.html
│   │       ├── new_taxonomy.html
│   │       ├── pagination.html
│   │       ├── specimen_delete_confirm.html
│   │       ├── specimen_deleted.html
│   │       ├── specimen_detail.html
│   │       ├── specimen_list.html
│   │       ├── specimen_table.html
│   │       ├── specimen_update.html
│   │       ├── taxonomy_update.html
│   │       └── title.html
│   ├── templatetags
│   │   └── tags.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
└── tree.txt
9 directories, 53 files
```

Appendix 5 - Serializers



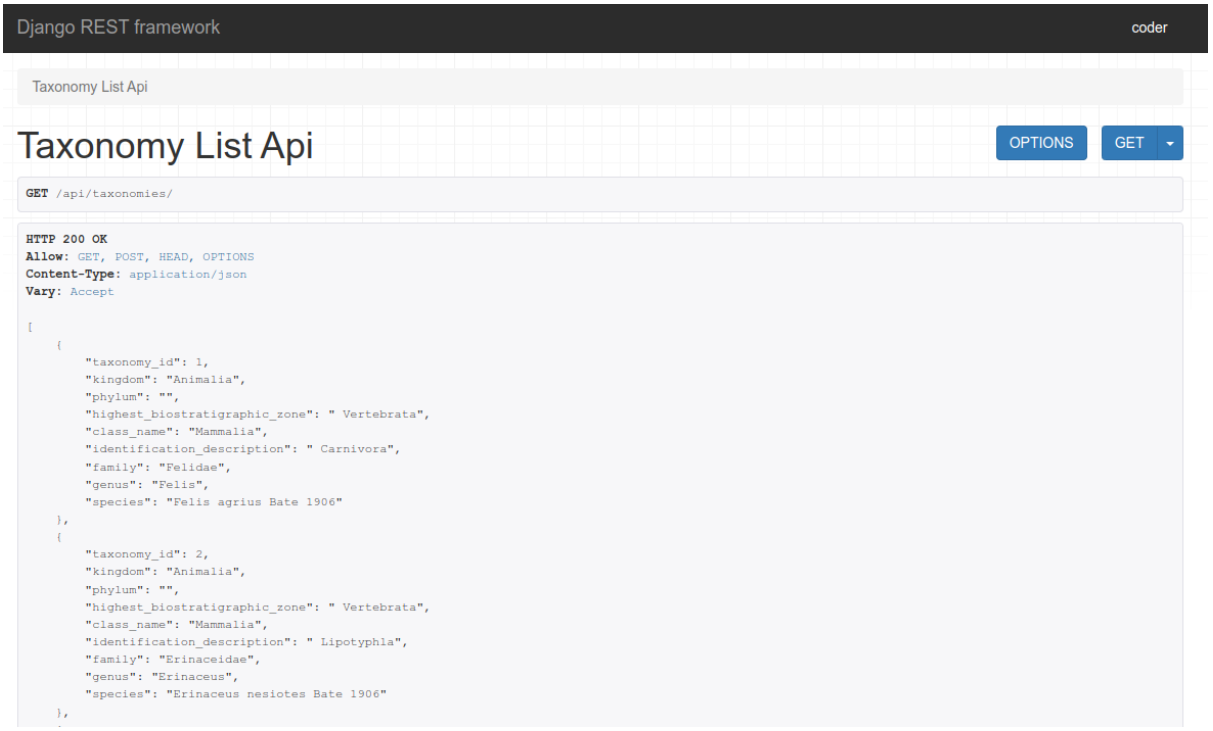
- Specimen List: http://127.0.0.1:8080/api/specimens/

  Specimen Detail: http://127.0.0.1:8080/api/specimens/{specimen_id}/



- Expedition List: http://127.0.0.1:8080/api/expeditions/

  Expedition Detail: http://127.0.0.1:8080/api/expeditions/{expedition_id}/



- Taxonomy List: http://127.0.0.1:8080/api/taxonomies/
- Taxonomy Detail: http://127.0.0.1:8080/api/taxonomies/{taxonomy_id}/

Appendix 6 - End Points

Index

Root

```
path('', views.index, name='index'),
```

HTTP Methods:
GET - Retrieves the root page

End Point - http://127.0.0.1:8080/

Specimens API

List

```
path('all_specimens/', AllSpecimensView.as_view(), name='all_specimens'),
```

HTTP Methods:
GET - Retrieves a list of all Specimens
POST - Uses for filtering
End point - /all_specimens/

View

```
path('specimen/detail/<int:pk>/', SpecimenDetailView.as_view(), name='specimen_detail'),
```

HTTP Methods:
GET - Retrieves a Specimen

End point Ex. - /specimen/detail/430541/

Updates

```
path('specimen/<int:pk>/update/', views.SpecimenUpdateView.as_view(),
name='specimen_update'),
```

HTTP Methods:
GET - Retrieves the form to update the specific specimen
POST - When submitted the updated data

EndPoint Ex.- /specimen/10438851/update/

Creates

```
path('new_specimen/', NewSpecimenView.as_view(), name='new_specimen'),
```

HTTP Methods:
GET - When accessing the new specimen form
POST - When submitted the created data

End Point - /new_specimen/

Deletes

```
path('specimen/<int:pk>/delete/', SpecimenDeleteView.as_view(), name='specimen_delete'),
```

HTTP Methods:
GET - When accessing the specimen a the form
POST - When submitted for deletion

End Point - /specimen/10438851/delete/

Expedition API

Creates

```
path('new_expedition/', NewExpeditionView.as_view(), name='new_expedition'),
```

HTTP Methods:

GET - When accessing the new expedition form
POST - When submitted the created data

End Point - /new_expedition/

Updates

```
    path('expedition_update/<int:pk>/', ExpeditionUpdateView.as_view(),
name='expedition_update'),
```

HTTP Methods:

GET - Retrieves the form to update a specific expedition
POST - When submitted the updated data

End Point Ex. - /expedition_update/1544/

Taxonomy API

Creates

```
path('taxonomy_update/<int:specimen_pk>/', TaxonomyUpdateView.as_view(), name='taxonomy_update'),
```

HTTP Methods:
- GET - When accessing the new Taxonomy form
- POST - When submitted the created data

End Point - /new_taxonomy/

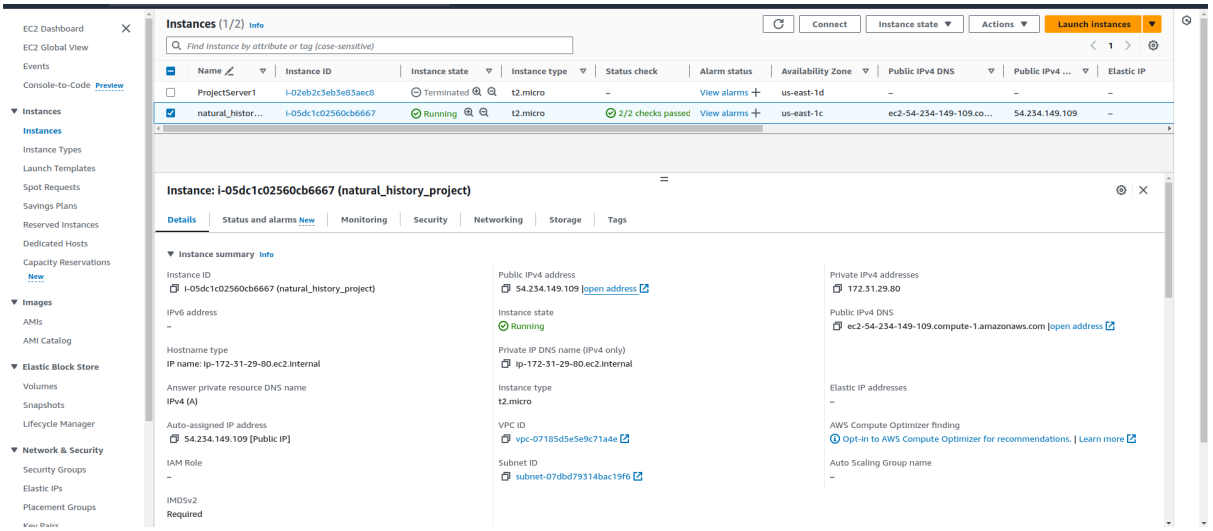
Updates

```
path('new_taxonomy/', NewTaxonomyView.as_view(), name='new_taxonomy'),
```

HTTP Methods:
- GET - When accessing the Taxonomy form
- POST - When submitted the updated data

End Point Ex. - taxonomy_update/10438851/

Appendix 7 - Project AWS EC2 Deployment



AWS EC2 Cloud address:

http://54.234.149.109:8000/