

# 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 Model](#);
  - **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](#)

<b>Abstract.....</b>	<b>5</b>
<b>1. Introduction.....</b>	<b>6</b>
<b>2. Basic Functionality(R1).....</b>	<b>6</b>
2.1 Libraries and Tree structure[10].....	6
2.2 Overview.....	6
2.2 Models[5].....	7
2.3 Migrations[6].....	8
2.4 Forms[7].....	9
2.5 Serializers[11].....	10
2.6 Tests[8].....	11
<b>3. Data Model(R2).....</b>	<b>12</b>
<b>4. Implementation of App Components(R3).....</b>	<b>13</b>
4.1 REST Endpoints Implementation[11].....	13
4.2 Views[11].....	13
4.3 Django admin page[12].....	17
<b>5. Bulk Loading Data(R4).....</b>	<b>18</b>
5.1 Method of Bulk Loading Data:.....	18
5.2 Implementation Details.....	19
5.3 Ensuring Data Integrity:.....	19
<b>6. Conclusion.....</b>	<b>20</b>
<b>7. References.....</b>	<b>21</b>
<b>8. Appendices.....</b>	<b>23</b>



# 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<sup>[13]</sup> ([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<sup>[10]</sup>

This project was built using **Ubuntu OS**, 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.

SPECIMENS TABLE

Specimen	Kingdom	Phylum	Sub-phylum	Class	Order	Family	Genus	Species	Continent	Country
Specimen 10451563	Animalia	Phylum	Vertebrata	Amphibia	Anura	Megophryidae	Leptolalax	Leptolalax pelodytoides (Boulenger, 1893)	Asia	Japan

Figure\_1 - Specimens table

Moreover, the app includes enhanced search functionalities based on taxonomy attributes and expedition details ([Figure\\_2](#)).

#### All Specimens

#### Taxonomy

Kingdom:

Phylum:

Sub-Phylum:

Class:

Family:

Genus:

Species:

#### Expedition

Continent:

Country:

Filter

Reset Filters

Number of Results: 6570

Figure\_2 - Taxonomy and Expedition Filter

Finally, the application provides the user an efficient pagination<sup>[4]</sup> allowing the users to handle large datasets ([Figure\\_3](#)).

Page 1 of 327 next last »

Figure\_3 - Paginator

## 2.2 Models<sup>[5]</sup>

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<sup>(1)</sup> structure using classes<sup>(2)</sup> to define the attributes and their

type of data it will store<sup>(3 & 4)</sup>. Most attributes were given as *Charfield* with a set maximum length and not required to fill.

```

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 specimen
    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<sup>(3 & 6)</sup> (Figure\_5) and Meta class within a model allows organising the dataset by a specific field<sup>(7)</sup>.

```

#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}"

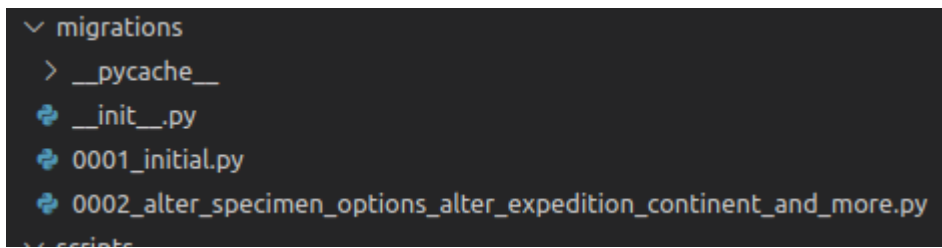
```

Figure\_5 - models.py attributes

## 2.3 Migrations<sup>[6]</sup>

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 <sup>[7]</sup>

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"<sup>(1)</sup> and "Country"<sup>(2)</sup> from the form 'Expedition' check against a specific list for validation ([Figure 8](#)).

```
ALLOWED_CONTINENTS = ['Africa', 'Antarctica', 'Asia', 'Europe', 'North America', 'Oceania', 'South America']

def clean_continent(self) -> Any:
    continent = self.cleaned_data['continent']
    # Validates that the continent is in the list of allowed continents (case-insensitive)
    if continent.title() not in self.ALLOWED_CONTINENTS:
        raise ValidationError('Invalid continent entered.')
    return continent

def clean_country(self) -> Any:
    country = self.cleaned_data['country']
    # Validates that the country is a valid country code or name
    try:
        pycountry.countries.lookup(country)
    except LookupError:
        raise ValidationError('Invalid country entered.')
    return country
```

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]<sup>(1)</sup>. ([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] ①

    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 ②
```

Figure\_10 - Specimen Form

## 2.5 Serializers<sup>[11]</sup>

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](#).

```

1 from rest_framework import serializers
2 from .models import Expedition, Taxonomy, Specimen
3
4 class ExpeditionSerializer(serializers.ModelSerializer):
5     class Meta:
6         model = Expedition
7         fields = '__all__'
8
9 class TaxonomySerializer(serializers.ModelSerializer):
10     class Meta:
11         model = Taxonomy
12         fields = '__all__'
13
14 class SpecimenSerializer(serializers.ModelSerializer):
15     expedition = ExpeditionSerializer()
16     taxonomy = TaxonomySerializer()
17
18     class Meta:
19         model = Specimen
20         fields = '__all__'
21
22     def create(self, validated_data) -> Specimen:
23         expedition_data = validated_data.pop('expedition', None)
24         taxonomy_data = validated_data.pop('taxonomy', None)
25
26         specimen = Specimen.objects.create(**validated_data)
27
28         if expedition_data:
29             Expedition.objects.create(specimen=specimen, **expedition_data)
30
31         if taxonomy_data:
32             Taxonomy.objects.create(specimen=specimen, **taxonomy_data)
33
34     return specimen

```

Figure\_11 - serializers.py

## 2.6 Tests<sup>[8]</sup>

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

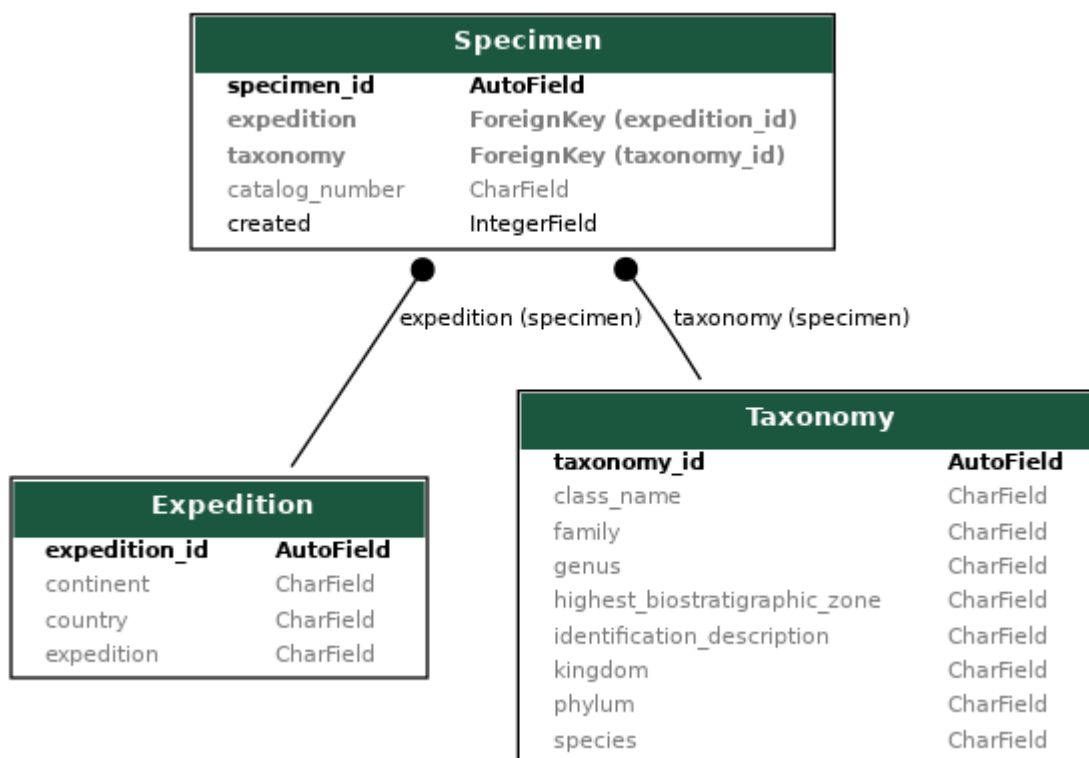
OK
Destroying test database for alias 'default'...

```

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<sup>[11]</sup>

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](#).

```
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 expedition 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<sup>[11]</sup>

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

```
class AllSpecimensView(ListView):
    model = Specimen
    template_name = 'specimen_catalog/all_specimens.html'
    context_object_name = 'specimens'
    queryset = Specimen.objects.all()
    filterset_class = SpecimenFilter

    def get_context_data(self, **kwargs) -> dict[str, Any]:
        context = super().get_context_data(**kwargs)

        filter = None
        try:
            filter = SpecimenFilter(self.request.GET, queryset=self.get_queryset())
        except ValidationError as e:
            messages.error(self.request, f"Invalid filter parameters: {e}")
            filter = SpecimenFilter(queryset=Specimen.objects.none())

        paginator = Paginator(filter.qs, 20)
        page = self.request.GET.get('page', 1)

        try:
            specimens = paginator.page(page)
        except EmptyPage:
            specimens = paginator.page(paginator.num_pages)

        context['specimens'] = specimens
        context['page_obj'] = specimens
        context['filter'] = filter

        messages.success(self.request, "View rendered successfully!")

        return context

    def get_queryset(self) -> QuerySet[Any]:
        queryset = super().get_queryset()

        filter_params = {
            'expedition_continent__icontains': self.request.GET.get('expedition_continent', ''),
            'expedition_country__icontains': self.request.GET.get('expedition_country', ''),
            'expedition_state_province__icontains': self.request.GET.get('expedition_state_province', ''),
        }

        return queryset.filter(**filter_params).order_by('-specimen_id')
```

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<sup>(1)</sup> and makes request attributes to the expedition module<sup>(2)</sup>. This class also adds pagination<sup>(3)</sup> to the template 'all\_specimens.html' that renders all the data. Finally, there is a message<sup>(4)</sup> that confirms that the data was rendered successfully, and the table content will display in a descendent order<sup>(5)</sup> based on 'specimen\_id'.

## SpecimenDetailView

```
# Displays a single specimen with its details, taxonomy and expedition
class SpecimenDetailView(DetailView):
    model = Specimen
    template_name = 'specimen_catalog/specimen_detail.html'
    context_object_name = 'specimen'

    def get_object(self, queryset=None) -> Model:
        # Error Handling
        try:
            # Attempts to get the object based on the provided queryset
            return super().get_object(queryset=queryset)
        except Http404:
            # Handles the case where the object is not found
            raise Http404("Specimen not found")

    def get_context_data(self, **kwargs) -> dict[str, Any]:
        # Error Handling
        try:
            # Calls the superclass method to get the default context data
            context = super().get_context_data(**kwargs)
            return context
        except Exception as e:
            # Handles other exceptions that may occur during context data retrieval
            messages.error(self.request, f"Error fetching specimen details: {e}")
            return context # Return the context without additional data
```

SpecimenDetailView implements a model 'Specimen' instance and renders the view into the specimen\_detail.html template<sup>(1)</sup>. When accessed through the URL at the end is the primary key that will be provided as a parameter. 'super().get\_object(queryset=queryset)'<sup>(2)</sup> 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<sup>(3)</sup>.

## SpecimenUpdateView, ExpeditionUpdateView & TaxonomyUpdateView

```
class SpecimenUpdateView(UpdateView):
    model = Specimen
    template_name = 'specimen_catalog/specimen_update.html'
    form_class = SpecimenForm # Replace with your actual form

    def get_object(self, queryset=None) -> Model:
        try:
            # Attempts to get the object based on the provided queryset
            return super().get_object(queryset=queryset)
        except Http404:
            # Handles the case where the object is not found
            raise Http404("Specimen not found")

    def form_valid(self, form) -> HttpResponseRedirect:
        try:
            # Calls the superclass method to handle the form validation
            response = super().form_valid(form)
            return response
        except Exception as e:
            # Handle other exceptions that may occur during form validation
            messages.error(self.request, f"Error updating specimen: {e}")
            return self.form_invalid(form) # Redirect to the form with error messages
```

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'<sup>(1)</sup> 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<sup>(2)</sup>.

## SpecimenDeleteView

```
class SpecimenDeleteView(DeleteView):
    model = Specimen
    template_name = 'specimen_catalog/specimen_delete_confirm.html'
    success_url = reverse_lazy('all_specimens') ①

    def get(self, request, *args, **kwargs) -> HttpResponseRedirect | HttpResponseRedirect | HttpResponseRedirect...:
        try:
            return super().get(request, *args, **kwargs)
        except Http404:
            messages.error(request, "Specimen not found")
            return redirect('all_specimens') ②

    def form_valid(self, form) -> HttpResponseRedirect | HttpResponseRedirect | HttpResponseRedirect...:
        try:
            response = super().form_valid(form)
            messages.success(self.request, 'Specimen deleted successfully.')
            return render(self.request, 'specimen_catalog/specimen_deleted.html') ③
        except Exception as e:
            messages.error(self.request, f"Error deleting specimen: {e}")
            return redirect('all_specimens')

    def form_invalid(self, form) -> Any:
        messages.error(self.request, 'Error deleting specimen. Please try again.')
        return super().form_invalid(form) ④
```

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'<sup>(3)</sup> template and the view redirects to 'all\_specimens.html'<sup>(2)</sup> template using the Django utility function reverse\_lazy<sup>(1)</sup>. If the form is not valid it will prompt to try to delete again<sup>(4)</sup>.

## NewSpecimenView, NewTaxonomyView & NewExpeditionView

```
class NewSpecimenView(View):
    template_name = 'specimen_catalog/new_specimen.html'

    def get(self, request) -> HttpResponseRedirect | HttpResponseRedirect | HttpResponseRedirect...:
        form = NewSpecimenForm()
        return render(request, self.template_name, {'form': form}) ①

    def post(self, request) -> HttpResponseRedirect | HttpResponseRedirect | HttpResponseRedirect...:
        form = NewSpecimenForm(request.POST)
        try:
            if form.is_valid():
                # Saves the new specimen to the database
                new_specimen = form.save()
                # Redirects to the detail page of the newly created specimen
                return redirect('specimen_detail', pk=new_specimen.pk) ②
            else:
                # Handles other exceptions that may occur during form submission
                messages.error(request, f"Error creating specimen: {e}") ③
        except Exception as e:
            messages.error(request, f"Error creating specimen: {e}")
        return render(request, self.template_name, {'form': form})
```

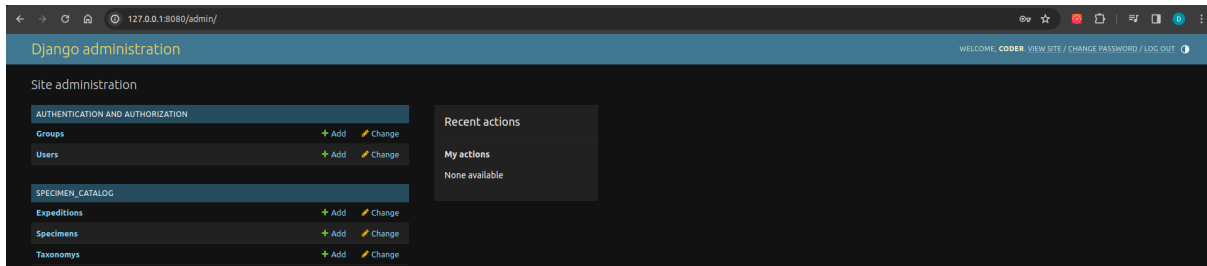
These three classes' views are designed to create new instances, using the HTTP GET method, and based on the form fields and validation. It renders the corresponding template and passes the form as context data<sup>(1)</sup>. The post method is called when the user submits the form. If the form.is\_valid() 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<sup>(2)</sup>.

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<sup>(3)</sup>.



## 4.3 Django admin page<sup>[12]</sup>

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](#))

```
# 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](mailto: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<sup>(5)</sup> 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<sup>(1)</sup> the project path to the system path<sup>(2)</sup>. Then, it reads the csv file using 'csv.DictReader'<sup>(3)</sup> to interpret rows as dictionaries. Afterwards, the script iterates<sup>(4)</sup> over each row retrieving the instances from the Django models assigning the selected columns into the corresponding model fields<sup>(5)</sup>. The process completes displaying a final message indication that the import was completed<sup>(6)</sup>.

```
import os
import sys
import django
import csv
from collections import defaultdict
from django.db import transaction # Imports the transaction module

# Sets up django environment
sys.path.append("/natural_history_project")
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'natural_history_project.settings')
django.setup()

# Imports Django models
from specimen_catalog.models import *

def run() -> None:
    # Path to the CSV file
    data_file = 'specimen_catalog/scripts/resource.csv'

    # Opens the CSV file
    with open(data_file, 'r') as csv_file:
        csv_reader = csv.DictReader(csv_file)

    # Initializes a counter for tracking progress
    count = 0

    try:
        # Starts a database transaction
        with transaction.atomic():
            # Iterates over each row
            for row in csv_reader:
                count += 1

        # The following code retrieves or creates instances from the database
        # and assigns each field from the CSV file to the model

        # Expedition table
        expedition, created = Expedition.objects.get_or_create(
            expedition=row['expedition'],
            continent=row['continent'],
            country=row['country'],
        )

        # Taxonomy table
        taxonomy, created = Taxonomy.objects.get_or_create(
            kingdom=row['higherClassification'],
            phylum=row['phylum'],
            highest_biostratigraphic_zone=row['highestBiostratigraphicZone'],
            class_name=row['class'],
            identification_description=row['identificationDescription'],
            family=row['family'],
            genus=row['genus'],
            species=row['determinationNames'],
        )

        # Specimen table
        specimen, created = Specimen.objects.get_or_create(
            specimen_id=row['_id'], # Takes the CSV index column
            defaults={
                'catalog number': row['catalogNumber'],
                'created': row['created'],
                'expedition': expedition,
                'taxonomy': taxonomy,
            }
        )
```

```

# Print progress
print(f'Adding record {count} (ID {row['_id']}): {'Created' if created else 'Retrieved'} successfully.')

# Prints a final message when the migration is completed
print("Data import complete.")

except Exception as e:
    # If an exception occurs, print an error message
    print(f"Error during data import: {e}")

# Check if the script is being run directly
if __name__ == "__main__":
    run()

```

## 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.

```

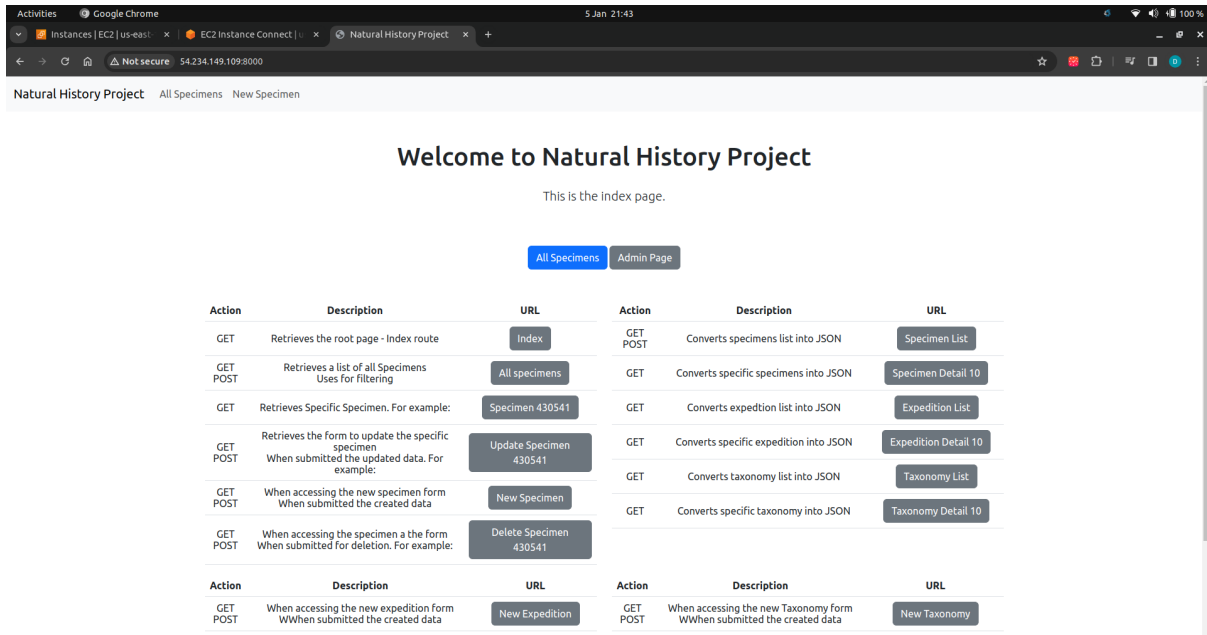
try:
    # Starts a database transaction
    with transaction.atomic():
        # Iterates over each row
        for row in csv_reader:

```

## 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

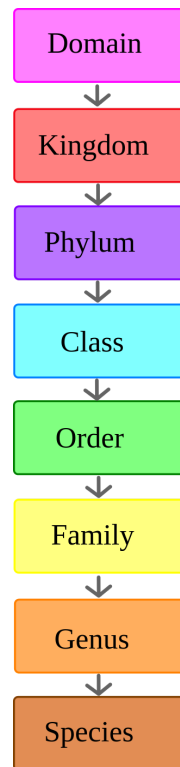
In conclusion, the Specimen\_catalog App effectively addresses the requirements outlined within the report, offering a powerful platform for exploring and organising specimen collections. The developed code guarantees integrity and efficiency, with key functionalities focused around Specimen, Taxonomy, 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

The screenshot shows the Natural History Museum Data Portal search results page. The header is green with the museum logo and 'Data Portal' text. Navigation links include Home, Datasets, Search, Contact, and About. The search bar contains the text 'Search all fields' and a magnifying glass icon. Below the search bar, there are filters for 'collectionCode = ZOO', 'phylum = chordata', 'class exists', 'order exists', 'family exists', 'genus exists', and 'typeStatus exists'. The results section shows '6,546 results' and 'showing 101-200 of 6,546 results total'. There are buttons for 'Cite', 'Share', and 'Download'. Below the results, there are tabs for 'Table', 'List', and 'Gallery'. The table has columns for Dataset, Resource, Record, class, order, family, genus, typeStatus, scientificName, locality, country, recordedBy, and catalogNumber. The first row of data shows '101 Collection ... Specimens View' with details for Mammalia, Dasyuromorphia, Dasyuridae, Phascogale, HOLOTYPE, Phascogale, Dinner Creek, Australia, Thomas, and 1922.12.18.46.

Search all fields

collectionCode = ZOO phylum = chordata class exists order exists family exists genus exists typeStatus exists

6,546 results  
showing 101-200 of 6,546 results total

Cite Share Download

Table List Gallery

Dataset	Resource	Record	class	order	family	genus	typeStatus	scientificName	locality	country	recordedBy	catalogNumber
101 Collection ...	Specimens	View	Mammalia	Dasyuromorphia	Dasyuridae	Phascogale	HOLOTYPE	Phascogale	Dinner Creek, ...	Australia	Thomas	1922.12.18.46

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
django-rest-framework==3.14.0
djanguviz==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

Django REST framework coder

Specimen List Api

Specimen List Api OPTIONS GET

GET /api/specimens/

HTTP 200 OK  
Allow: GET, POST, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept

```
[
  {
    "specimen_id": 10451541,
    "expedition": null,
    "taxonomy": null,
    "catalog_number": "1",
    "created": 1
  },
  {
    "specimen_id": 10451535,
    "expedition": null,
    "taxonomy": null,
    "catalog_number": "test",
    "created": 1
  },
  {
    "specimen_id": 10451532,
    "expedition": null,
    "taxonomy": null,
    "catalog_number": "2",
    "created": 2
  },
  {
    "specimen_id": 10451531,
    "expedition": null,
    "taxonomy": null,
    "catalog_number": "1",
    "created": 1
  }
]
```

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

Specimen Detail: [http://127.0.0.1:8080/api/specimens/{specimen\\_id}/](http://127.0.0.1:8080/api/specimens/{specimen_id}/)

Django REST framework

Expedition List Api

Expedition List Api OPTIONS GET

GET /api/expeditions/

HTTP 200 OK  
Allow: GET, POST, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept

```
[
  {
    "expedition_id": 1,
    "expedition": "",
    "continent": "Asia",
    "country": "Myanmar"
  },
  {
    "expedition_id": 2,
    "expedition": "",
    "continent": "",
    "country": ""
  },
  {
    "expedition_id": 3,
    "expedition": "",
    "continent": "Asia",
    "country": "Taiwan, Province of China"
  },
  {
    "expedition_id": 4,
    "expedition": "",
    "continent": "Africa",
    "country": "South Africa"
  },
  {
    "expedition_id": 5,
    "expedition": "",
    "continent": "Asia",
    "country": "Sri Lanka"
  }
]
```

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

Expedition Detail: [http://127.0.0.1:8080/api/expeditions/{expedition\\_id}/](http://127.0.0.1:8080/api/expeditions/{expedition_id}/)

Django REST framework coder

Taxonomy List Api

Taxonomy List Api OPTIONS GET

GET /api/taxonomies/

HTTP 200 OK  
Allow: GET, POST, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept

```
[
  {
    "taxonomy_id": 1,
    "kingdom": "Animalia",
    "phylum": "",
    "highest_biostratigraphic_zone": "Vertebrata",
    "class_name": "Mammalia",
    "identification_description": "Carnivora",
    "family": "Felidae",
    "genus": "Felis",
    "species": "Felis agrius Bate 1906"
  },
  {
    "taxonomy_id": 2,
    "kingdom": "Animalia",
    "phylum": "",
    "highest_biostratigraphic_zone": "Vertebrata",
    "class_name": "Mammalia",
    "identification_description": "Lipotyphla",
    "family": "Erinaceidae",
    "genus": "Erinaceus",
    "species": "Erinaceus nesiotis Bate 1906"
  }
]
```

- Taxonomy List: <http://127.0.0.1:8080/api/taxonomies/>
- Taxonomy Detail: [http://127.0.0.1:8080/api/taxonomies/{taxonomy\\_id}/](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/](/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/](/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

The screenshot displays the AWS Management Console interface for EC2 instances. The top navigation bar includes 'EC2 Dashboard', 'EC2 Global View', 'Events', 'Console-to-Code', and 'Preview'. The left sidebar lists various AWS services like 'Instances', 'Instance Types', 'Launch Templates', 'Spot Requests', 'Savings Plans', 'Reserved Instances', 'Dedicated Hosts', 'Capacity Reservations', 'Images', 'AMI Catalog', 'Elastic Block Store', 'Volumes', 'Snapshots', 'Lifecycle Manager', 'Network & Security', 'Security Groups', 'Elastic IPs', and 'Placement Groups'. The main content area shows a table of instances with columns for Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, Public IPv4 DNS, Public IPv4..., and Elastic IP. The instance 'natural\_hist... i-05dc1c02560cb6667' is highlighted in blue and marked as 'Running'. Below the table, the details for this instance are shown, including its public IP address 54.234.149.109, private IP address 172.31.29.80, and various other configuration details. Below the console screenshot, a terminal window shows the command 'curl http://54.234.149.109:8000' and its output, which includes '2024-01-05 21:42:35 +0000 [3480] [INFO] Starting unicorn 20.1.0' and '2024-01-05 21:42:35 +0000 [3480] [INFO] Listening at: http://0.0.0.0:8000 (3480)'.

AWS EC2 Cloud address:

<http://54.234.149.109:8000/>