BASIC PROJECT DEMONSTRATION

(Django with REST Framework Integration for Managing Auction)

This project demonstrates the seamless integration of Django, a powerful web framework for building web applications, with Django REST Framework, a toolkit for building Web APIs. The primary objective is to showcase how Django's Object-Relational Mapping (ORM) capabilities can be utilised alongside the RESTful architecture provided by Django REST Framework to efficiently manage data related to coins. Through this integration, developers can leverage Django's robust features for database interaction while also exposing CRUD operations via RESTful API endpoints, thereby enabling seamless data management for coin-related information. This documentation provides a step-by-step guide on setting up the project, defining models, creating API endpoints, and deploying the application for real-world use cases.

1. *Object Relational Mapping*

We defined your database structure, and we can use Django's ORM (Object-Relational Mapping) to interact with your database and then display the data in the Django admin panel.

Table: coins

Field	Type	Description
coin_id	Primary Key	Unique identifier for the coin
coin_image	String	URL or file path to the coin image
coin_name	String	Name or title of the coin
coin_desc	Text	Description or details about the coin
coin_year	Integer	Year the coin was minted
coin_country	String	Country where the coin was minted
coin_material	String	Material of the coin (e.g., gold, silver, copper)

coin_weight	Float	Weight of the coin in grams
starting_bid	Float	Starting bid price for the coin
coin_status	String	Status of the coin (e.g., available, sold)

NOTE : WHILE FOLLOWING BELOW CODES, MAKE SURE THE INDENTATION IS VERY IMPORTANT IN PYTHON

First, you'll need to create a Django model for the "coins" table. Here's how you can do it:

- Activate the virtual environment:

  source myenv/bin/activate (Replace 'myenv' with actual environment)

- Open your Django app's models.py file. If not, create the app within the main directory by command :

  python manage.py startapp appName

- Define a Python class for the "coins" table, representing each field as a class attribute.
- Use Django model fields to map each database field to a corresponding Python data type.

```
from django.db import models

class Coin(models.Model):
    # Define choices for coin status
    STATUS_CHOICES = (
        ('select', 'Select'),  # Placeholder option
        ('available', 'Available'),
        ('sold', 'Sold'),
        ('pending', 'Pending'),
    )


    coin_id = models.AutoField(primary_key=True)  # Auto-incrementing primary key
```

```
    coin_image = models.ImageField(upload_to='coin_images/', null=True, blank=True)  # Image field to store coin image

    coin_name = models.CharField(max_length=100)  # Char field for coin name

    coin_desc = models.TextField()  # Text field for coin description

    coin_year = models.IntegerField()  # Integer field for coin year

    coin_country = models.CharField(max_length=50)  # Char field for coin country

    coin_material = models.CharField(max_length=50)  # Char field for coin material

    coin_weight = models.FloatField()  # Float field for coin weight

    starting_bid = models.FloatField()  # Float field for starting bid

    coin_status = models.CharField(max_length=50,

choices=STATUS_CHOICES)  # Char field with choices for coin status

    def __str__(self):

        return self.coin_name  # Return the coin name as its string representation
```

- You should install Pillow using pip, either globally or within your virtual environment. Here's how to do it within your virtual environment:

  Install Pillow using pip:

  ```
  python -m pip install Pillow
  ```

- Once installed Pillow you'll be able to manage images, next you need to run Django migrations to create the corresponding table in the database: (Do within your project directory)

  ```
  python manage.py makemigrations
  ```

  ```
  python manage.py migrate
  ```

- The coin_images/ directory should be created within your Django project's media directory. By default, Django looks for media files (including user-uploaded files like images) in the media/ directory of your project.

  Create a media/ directory in your Django project if you haven't already:

  ```
  mkdir media
  ```

mkdir media/coin_images

- Ensure that your Django project's settings.py file is configured to serve media files during development. Add or modify the MEDIA_URL and MEDIA_ROOT settings accordingly:

```
# settings.py
# Add this at the top of the file
import os
# Add this to the bottom of the file
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

- Make sure you have configured the necessary URL patterns to serve media files during development. In your project's urls.py, add the following import and URL configuration:

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
urlpatterns = [
    path('admin/', admin.site.urls),
]
# Add this at the end of the file
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

- Once the table is created, you can register the model with the Django admin site to view and manage the data. Here's how you can do it:
    - ❖ Open your app's admin.py file.

- Import your Coin model and register it with the admin site.

```
from django.contrib import admin
from django.utils.html import format_html
from .models import Coin  # Import the Coin model
from django.conf import settings  # Import Django settings module
@admin.register(Coin)  # Register the Coin model with the admin site
class CoinAdmin(admin.ModelAdmin):  # Define the admin class for Coin model
# Specify the fields to display in the list view of the admin site
list_display = ('coin_name', 'display_coin_image', 'coin_desc',
'coin_year', 'coin_country', 'coin_material', 'coin_weight',
'starting_bid', 'coin_status')
# Define a method to display the coin image in the list view
def display_coin_image(self, obj):
# Generate HTML code to display the coin image with specified width and height
return format_html('<img src="{}" style="max-width:100px;
max-height:100px;">'.format(obj.coin_image.url))
```

Now, you can run your Django development server (python manage.py runserver) and navigate to the Django admin panel (http://localhost:8000/admin) to view and manage your "coins" data. You'll be able to add, edit, and delete coin records through the admin interface.

If admin panel is non accessible, create admin superuser by the command in root directory :

```
python manage.py createsuperuser
```

Follow the prompts to signup with email, username and password. Then you'll be able to log on to the django admin and do the CRUD

==========

2. *Django Rest Framework :*

To install Django REST Framework, you can use pip, Python's package manager. Here's how you can install it:

```
pip install djangorestframework
```

If you're using a virtual environment (which is recommended), make sure it's activated before running the above command. After installation, you can start using Django REST Framework in your Django projects.

Remember to also add 'rest_framework' to the INSTALLED_APPS list in your Django project's settings.py file:

```
INSTALLED_APPS = [
    ...
    'Rest_framework',
    ...
]
```

Your expected directory must be like this to do the api rest framework, if any file missing add them in the respective:

```
BasicApp/
├── apps/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   └── __init__.py
│   ├── models.py
│   ├── serializers.py
│   ├── tests.py
│   └── views.py
├── manage.py
```

```
└── BasicProject/
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── media/
    └── coin_images/
```

To implement CRUD APIs for managing coins in your Django project, you can follow these steps:

- Create Serializers: Create serializers to convert model instances to JSON format and vice versa. You can define serializers in the apps/serializers.py file.

```
from rest_framework import serializers  # Import serializers from Django REST Framework

from .models import Coin  # Import the Coin model

class CoinSerializer(serializers.ModelSerializer):  # Define a serializer for the Coin model

    # Define a HyperlinkedIdentityField for generating hyperlinks to individual coin details

    view_details = serializers.HyperlinkedIdentityField(view_name='coin-detail', lookup_field='pk')

    class Meta:

        model = Coin  # Specify the Coin model to serialize

        fields = ['coin_id', 'coin_image', 'coin_name', 'coin_desc', 'coin_year', 'coin_country', 'coin_material', 'coin_weight', 'starting_bid', 'coin_status', 'view_details']  # Define the fields to include in the serialised representation
```

- Define Views: Create views to handle CRUD operations for the Coin model. You can define viewsets using Django REST Framework's ModelViewSet class in the apps/views.py file.

```
from django.shortcuts import render  # Import render function from Django

from rest_framework import status  # Import status codes from Django REST Framework

from rest_framework.response import Response  # Import Response class from Django REST Framework

from rest_framework import viewsets  # Import viewsets from Django REST Framework

from .models import Coin  # Import the Coin model

from .serializers import CoinSerializer  # Import the CoinSerializer

class CoinViewSet(viewsets.ModelViewSet):  # Define a viewset for the Coin model

    queryset = Coin.objects.all()  # Define the queryset to fetch all coin objects

    serializer_class = CoinSerializer  # Specify the serializer class to use for the Coin model
```

- Configure URLs: Configure URLs to map the viewset to appropriate endpoints. You can define URL patterns in the BasicProject/urls.py file.

```
from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.conf.urls.static import static

from rest_framework import routers

from apps.views import CoinViewSet

# Create a router for registering viewsets

router = routers.DefaultRouter()

# Register CoinViewSet with the router

router.register(r'coins', CoinViewSet)

# Define URL patterns
```

```
urlpatterns = [
    # Admin site URL
    path('admin/', admin.site.urls),
    # API endpoints for coins using the router
    path('api/', include(router.urls)),
]
# Serve media files in DEBUG mode
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

- Run the Development Server: Start the Django development server to test your APIs.

  ```
  python manage.py runserver
  ```

  Your CRUD APIs for managing coins should now be accessible at /api/coins/ endpoint. You can perform CRUD operations (Create, Read, Update, Delete) using tools like Postman or by sending HTTP requests directly.

If you have implemented the CRUD APIs for managing coins as described earlier and have navigated to http://127.0.0.1:8000/api/coins/, you should see the following:

- **List of Coins**: If you've implemented the list view for your CoinViewSet, you should see a list of all coins available in your database. Each coin object will be represented in JSON format.
- **Create New Coin Form**: If you've implemented the create view for your CoinViewSet, you should see a form or a way to submit new coin data to create a new coin object.
- **Individual Coin Detail**: If you've implemented the retrieve view for your CoinViewSet, you can append the ID of a specific coin to the URL to view the details of that coin. For example, http://127.0.0.1:8000/api/coins/1/ would show the details of the coin with ID 1.

- **Update Coin Form**: If you've implemented the update view for your CoinViewSet, you should be able to edit the details of an existing coin by appending the ID of the coin to the URL and making a PUT or PATCH request with the updated data.
- **Delete Coin Endpoint**: If you've implemented the destroy view for your CoinViewSet, you should be able to delete a coin by appending the ID of the coin to the URL and making a DELETE request.

- To add a "view_details" link for showing individual details of each coin when displaying the list of all coins, you can modify the CoinSerializer to include a hyperlink field for the individual coin detail endpoint. Here's how you can do it:

```
        # apps/serializers.py
from rest_framework import serializers
from .models import Coin
class CoinSerializer(serializers.ModelSerializer):
    view_details =
serializers.HyperlinkedIdentityField(view_name='coin-detail',
lookup_field='pk')
    class Meta:
        model = Coin
        fields = ['id', 'coin_name', 'coin_desc', 'coin_year', 'coin_country',
'view_details']
```

In this serializer:

- We've added a new field called view_details, which is a HyperlinkedIdentityField.
- We've specified the view_name parameter as 'coin-detail', which corresponds to the URL pattern for the individual coin detail endpoint.

- ❖ We've specified the lookup_field parameter as 'pk', which indicates that the primary key of the coin should be used to construct the URL.

Now, you'll be able to view the list in rest framework with individual links for CRUD.

==========

3. *Testing the built API's :*

You can use pytest as an alternative to Django's built-in test runnerInstall it in the main root directory:

```
pip install pytest
```

And also modify this in setting.py in INSTALLED APPS :

```
INSTALLED_APPS = [
    'django_filters',
]
```

After installing these packages, you should have everything you need to write and run test cases for your Django APIs.

Your directory should be like this :

```
BasicApp/
├── apps/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   └── __init__.py
│   ├── models.py
│   ├── serializers.py
│   ├── tests.py                    <<test file should be placed here
│   └── views.py
├── manage.py
└── BasicProject/
```

```
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── media/
    └── coin_images/
```

Once you've placed the test file in the correct directory, you can run the tests again to verify their functionality.

Your test file can be like this to run test case on the functions like list, retrieve, create, update, delete : (tests.py)

```
from django.test import TestCase
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APIClient
from django.core.files.uploadedfile import SimpleUploadedFile
from .models import Coin
class CoinAPITestCase(TestCase):
    def setUp(self):
        self.client = APIClient()
        self.coin1 = Coin.objects.create(
            coin_name='Test Coin 1',
            coin_desc='Description of Test Coin 1',
            coin_year=2022,
            coin_country='Test Country',
            coin_material='Test Material',
            coin_weight=10.5,
            starting_bid=100.0,
            coin_status='available'
        )
```

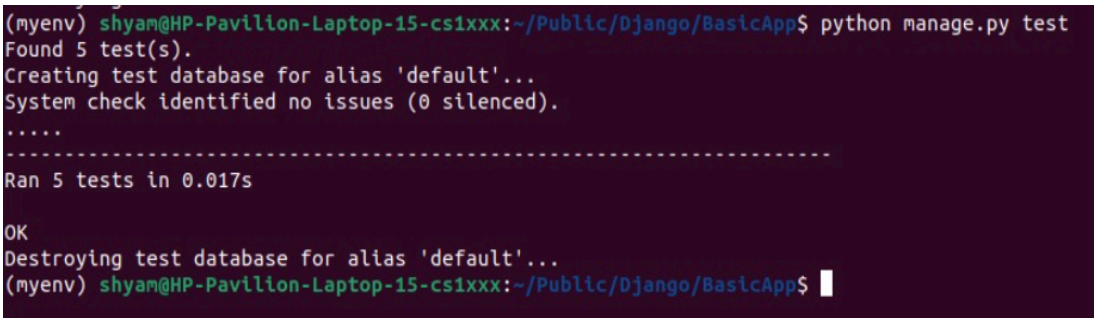
```
    def test_list_coins(self):
        url = reverse('coin-list')
        response = self.client.get(url)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
    def test_retrieve_coin(self):
        url = reverse('coin-detail', kwargs={'pk': self.coin1.pk})
        response = self.client.get(url)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
    def test_delete_coin(self):
        url = reverse('coin-detail', kwargs={'pk': self.coin1.pk})
        response = self.client.delete(url)
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
    def test_create_coin(self):
        url = reverse('coin-list')
        data = {
            'coin_name': 'New Test Coin',
            'coin_desc': 'Description of New Test Coin',
            'coin_year': 2023,
            'coin_country': 'New Test Country',
            'coin_material': 'New Test Material',
            'coin_weight': 15.0,
            'starting_bid': 150.0,
            'coin_status': 'available'
        }
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)


    def test_update_coin(self):
```

```
        url = reverse('coin-detail', kwargs={'pk': self.coin1.pk})
        data = {
            'coin_name': 'Updated Test Coin',
            'coin_desc': 'Updated Description of Test Coin 1',
            'coin_year': 2023,
            'coin_country': 'Updated Test Country',
            'coin_material': 'Updated Test Material',
            'coin_weight': 15.0,
            'starting_bid': 150.0,
            'coin_status': 'available'
        }
        response = self.client.put(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

After configuring the test file, after activating your project, you can test your cases by running the command :

```
python manage.py test
```

Your expected output in the terminal after testing should be like :



You can see the 5 test cases have been passed with ‘OK’.

==========

4. *Search Implementation through URL :*

First, you need to install the Django Filter package if you haven't already:

```
pip install django-filter
```

Update your urls.py to include a new URL pattern that accepts the search parameters as path parameters:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
from rest_framework import routers
from apps.views import CoinViewSet, CoinSearchView
# Create a router for registering viewsets
router = routers.DefaultRouter()
# Register CoinViewSet with the router
router.register(r'coins', CoinViewSet)
# Define URL patterns
urlpatterns = [
    # Admin site URL
    path('admin/', admin.site.urls),
    # API endpoints for coins using the router
    path('api/', include(router.urls)),
    path('coins/search/<path:path_params>/', CoinSearchView.as_view(), name='coin-search'),
]
# Serve media files in DEBUG mode
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

In this updated URL pattern, <str:field> represents the field name (e.g., coin_name, coin_desc, coin_year), and <str:value> represents the corresponding search value.

To handle a variable number of fields and values in the URL, you can modify the view to parse the URL dynamically. You can use regular expressions to capture the field-value pairs from the URL and then filter the queryset accordingly :

```
from django.shortcuts import render  # Import render function from Django

from rest_framework import status  # Import status codes from Django REST Framework

from rest_framework.response import Response  # Import Response class from Django REST Framework

from rest_framework import viewsets  # Import viewsets from Django REST Framework

from .models import Coin  # Import the Coin model

from .serializers import CoinSerializer  # Import the CoinSerializer

from rest_framework.views import APIView

import re

class CoinViewSet(viewsets.ModelViewSet):  # Define a viewset for the Coin model

    queryset = Coin.objects.all()  # Define the queryset to fetch all coin objects

    serializer_class = CoinSerializer  # Specify the serializer class to use for the Coin model

class CoinSearchView(APIView):

    def get(self, request, *args, **kwargs):

        # Extract search parameters from the URL path

        path_params = kwargs.get('path_params')

        # Parse the path_params string into field-value pairs

        search_params = {}

        if path_params:

            # Split the path_params string by '/'

            path_params_list = path_params.split('/')

            # Ensure there are an even number of elements (field-value pairs)
```

```
        if len(path_params_list) % 2 == 0:
            for i in range(0, len(path_params_list), 2):
                search_params[path_params_list[i]] = path_params_list[i+1]
    # Filter Coin objects based on the provided search parameters
    coins = Coin.objects.all()
    for field, value in search_params.items():
        coins = coins.filter(**{field: value})
    if not coins:
        return Response({"message": "No coins found for the provided search parameters"}, status=404)
    # Serialize the filtered queryset
    serializer = CoinSerializer(coins, many=True, context={'request': request})
    return Response(serializer.data)
```

This setup allows you to construct URLs with a flexible number of field-value pairs, enabling dynamic searches based on your preferences.

==========

5. *Fake data generation :*

To implement fake data generation in your Django app, you can use libraries like Faker, which allows you to generate realistic fake data for various fields.

Here's how you can do it:

- *Install Faker: First, install the Faker library using pip.*

  ```
  pip install Faker
  ```

- *Create a Management Command:*

  In Django, management commands are used for various administrative tasks. You can create a custom management command to generate fake data.Create a new Python module within one of your Django apps, such as

management/commands/ directory, and create a Python file, e.g., generate_fake_data.py.

- *You need to install the requests module and beautifulSoup. You can do this using pip, the Python package manager :*

  pip install requests

  pip install beautifulsoup4

- *Write the Management Command:*

  In the generate_fake_data.py file, write the code to generate fake data using Faker and populate your database.

```
from django.core.management.base import BaseCommand
from faker import Faker
from apps.models import Coin
from django.core.files.base import ContentFile
import requests
import os
from bs4 import BeautifulSoup
import re
class Command(BaseCommand):
    help = 'Generate fake data for Coin model'
    def handle(self, *args, **options):
        fake = Faker()
        # Create the folder if it doesn't exist
        if not os.path.exists('media/coin_images'):
            os.makedirs('media/coin_images')
        for _ in range(10):
            # Generate a random search query
            search_query = "coin"
```

```
        # Generate a Google image search URL
        search_url = f"https://www.google.com/search?q={search_query}&tbm=isch"
        # Send a GET request to Google Images
        response = requests.get(search_url)
        soup = BeautifulSoup(response.content, 'html.parser')
        # Find all image elements
        images = soup.find_all('img')
        # Extract image URLs
        image_urls = [img['src'] for img in images if img.get('src')]
        # Filter image URLs to remove non-image links
        image_urls = [url for url in image_urls if re.match(r'^https?://', url)]
        # Choose a random image URL
        if image_urls:
            image_url = fake.random_element(elements=image_urls)
        else:
            image_url = "https://via.placeholder.com/400x400"  # Placeholder URL
        # Download the image from the URL
        image_response = requests.get(image_url)
        # Create a Coin object with fake data
        coin = Coin(
            coin_name=fake.word(),
            coin_desc=fake.sentence(),
            coin_year=fake.random_int(min=1800, max=2023),
            coin_country=fake.country(),
            coin_material=fake.word(),
            coin_weight=fake.random_number(digits=2),
            starting_bid=fake.random_number(digits=3),
```

```
            coin_status=fake.random_element(elements=('available', 'sold', 'pending'))

        )

            # Save the image to the 'coin_images' folder

            image_path = f"media/coin_images/coin_{fake.random_number(digits=4)}.png"

            with open(image_path, 'wb') as image_file:

                image_file.write(image_response.content)

            # Assign the image path to the coin_image field

            coin.coin_image.save(os.path.basename(image_path), ContentFile(image_response.content), save=False)

            coin.save()

        self.stdout.write(self.style.SUCCESS('Successfully generated fake data'))
```

- *Run the Management Command:*

  ```
  python manage.py generate_fake_data
  ```

  This command will populate your Coin model with 100 fake records. You can adjust the number of records and the fields to generate as needed. Additionally, you can customise the fake data generation logic based on your specific requirements and the structure of your model.

==========