**Postify - Project Overview**

**This project is a Django-based web application using Django Rest Framework (DRF) to implement various functionalities for managing blog posts, user accounts, comments, and like/dislike functionality. The application uses JWT authentication for secure access. PostifyAPI appears to be a social media-like platform with features for user accounts, posts, comments, and likes/dislikes functionality.**

**Project Structure**

The project consists of the following main applications:

**1. Accounts App**

* Handles user authentication and user profile management
* Location: /accounts

**2. Posts App**

* Manages post creation and management
* Location: /posts

**3. Comments App**

* Handles comment functionality on posts
* Location: /comments

**4. Likes App**

* Manages likes and dislikes functionality
* Location: /likes

**5. Postify (Project Root)**

* Contains core project configuration
* Location: /Postify

**Accounts App**

**Summary**

The Accounts app handles user authentication, registration, login, and logout functionality using Django REST Framework and JWT (JSON Web Tokens) for authentication.

**Files Structure**

**1. Models (models.py)**

Currently empty, using Django's default User model from django.contrib.auth.models.User.

**2. Views (views.py)**

Contains the main authentication logic with the following endpoints:

**User Registration**

@api\_view(['POST'])

@permission\_classes([AllowAny])

@required\_fields

@check\_existing\_user

def user\_registration(request)

**- Endpoint: /account/register/**

* **Method**: POST
* **Required Fields**: username, email, password
* **Decorators**:
  + required\_fields: Validates all required fields are present
  + check\_existing\_user: Checks for duplicate username/email
* **Response**:
  + Success (201): User registered successfully
  + Error (400): Missing fields or duplicate user
  + Error (500): Internal server error

**User Login**

@api\_view(['POST'])

@permission\_classes([AllowAny])

@required\_fields

def user\_login(request)

**- Endpoint: /account/login/**

* **Method**: POST
* **Required Fields**: username, password
* **Response**:
  + Success (200): Returns access\_token, refresh\_token, and username
  + Error (400): Invalid credentials
  + Error (500): Login failed

**User Logout**

@api\_view(['POST'])

@permission\_classes([AllowAny])

def user\_logout(request)

**- Endpoint: /account/logout/**

* **Method**: POST
* **Required Fields**: refresh\_token, access\_token (optional)
* **Response**:
  + Success (200): Successfully logged out
  + Error (400): Logout failed

**3. Middleware (middleware/middleware.py)**

Contains TokenGenerationMiddleware for JWT token handling:

class TokenGenerationMiddleware:

* Handles token generation during login
* Validates user credentials
* Generates access and refresh tokens
* Returns tokens in JSON response

**4. URLs (urls.py)**

Defines the following URL patterns:

urlpatterns = [

    path('account/register/', user\_registration, name='register'),

    path('account/login/', user\_login, name='token\_obtain\_pair'),

    path('account/logout/', user\_logout, name='logout'),

]

**Authentication Flow**

**Registration**:

* User submits username, email, and password
* System validates input and checks for existing users
* Creates new user if validation passes

**Login**:

* User submits username and password
* Middleware validates credentials
* Returns JWT tokens upon successful authentication

**Logout**:

* User submits refresh token
* System blacklists both access and refresh tokens
* User is logged out

**Error Handling**

* Comprehensive error handling with logging
* Custom error messages for different scenarios
* HTTP status codes properly implemented
* Logging implemented for debugging and monitoring

**Security Features**

* JWT token-based authentication
* Token blacklisting on logout
* Password hashing using Django's default authentication
* Permission classes for endpoint protection
* Input validation through decorators

**Dependencies**

* Django REST Framework
* SimpleJWT for token handling
* Django's built-in authentication system

**Posts App Documentation**

**Summary**

The Posts app manages blog posts with features like CRUD operations, tagging, pagination, and integration with comments and likes.

**File Structure and Flow**

**1. Models (models/models.py)**

Defines two main models:

* Post: Main post model with fields for title, content, author, etc.
* Tag: Model for post categorization

**2. Admin Configuration (admin.py)**

class PostAdmin(admin.ModelAdmin):

    list\_display = ('title', 'author', 'category', ...)

* Configures Django admin interface for Post and Tag models
* Custom display methods like tags\_list for better admin visualization
* Implements filtering, searching, and ordering

**3. Views (views.py)**

Implements ViewSet classes using mixins:

**PostList**

* Handles listing of posts
* Uses JWT authentication
* Implements search, filtering, and pagination

**PostUserDetail**

* Retrieves single post details
* Flow:

1. Receives post ID
2. Calls PostID.get\_post\_by\_id()
3. Returns serialized post data

**PostUserUpdate**

* Updates existing posts
* Flow:

1. Receives post ID and data
2. Calls PostID.update\_post()
3. Validates permissions
4. Returns updated post

**PostUserDestroy**

* Deletes posts
* Flow:

1. Receives post ID
2. Validates permissions
3. Calls PostID.delete\_post\_by\_id()

**PostUserCreate**

* Creates new posts
* Flow:

1. Uses @validate\_fields decorator
2. Validates data through serializer
3. Assigns authenticated user as author
4. Creates post with tags

**4. Serializers (serializers/serializers.py)**

Defines multiple serializers:

class PostSerializer(serializers.ModelSerializer):

    comments = serializers.SerializerMethodField()

    likes = LikeSerializer(many=True, read\_only=True)

    tags = serializers.ListField(...)

* PostSerializer: Main serializer for posts
* CommentSerializer: Handles comment display
* LikeSerializer: Manages like information
* TagSerializer: Handles tag data

Flow:

1. Receives data from views
2. Validates fields
3. Handles tag creation/association
4. Manages nested serialization of comments and likes

**5. Utils (utils.py)**

Contains PostID class with helper methods:

class PostID:

    @staticmethod

    def get\_post\_by\_id(post\_id)

    @staticmethod

    def delete\_post\_by\_id(post\_id)

    @staticmethod

    def update\_post(post\_id, data)

* Centralized post operations
* Called by view classes
* Handles error logging and responses

**6. Tags (tags.py)**

class TagView(APIView):

    def get(self, request, tag\_name=None)

**- Handles tag-based post filtering**

* Supports single and multiple tag queries

**7. Decorators (decorators.py)**

def validate\_fields(func):

* Custom validation decorator
* Used by PostUserCreate
* Pre-validates data before processing

**8. Pagination (paginations/paginations.py)**

class CustomPagination(PageNumberPagination):

    page\_size = 3

* Implements custom pagination
* Used by PostList viewset

**Request Flow Example**

For creating a post:

1. Request hits URL defined in urls.py
2. Routed to PostUserCreate.create()
3. @validate\_fields decorator validates input
4. PostSerializer processes data
5. Post is created with associated tags
6. Response returned to client

**Authentication and Permissions**

* JWT Authentication throughout
* Permission checks in views
* Custom user permissions for post operations

**Integration Points**

* Comments app integration through serializers
* Likes app integration through serializers
* User authentication from accounts app

API Endpoints and Payloads

**1. Create Post**

**Endpoint:** POST /api/posts/create/

**Request Payload:**

{

    "title": "Sample Post Title",

    "content": "This is the main content of the post",

    "category": "BLOG",

    "is\_published": true,

    "rating": 4.5,

    "website": "https://example.com",

    "tags": [

        {"name": "Python"},

        {"name": "Programming"}

    ]

}

**Success Response (201 Created):**

{

    "id": 1,

    "title": "Sample Post Title",

    "content": "This is the main content of the post",

    "author": "username",

    "category": "BLOG",

    "is\_published": true,

    "rating": 4.50,

    "website": "https://example.com",

    "tags\_used": ["Python", "Programming"],

    "comments": "This Post Has No Comments",

    "like\_count": 0,

    "likes": []

}

**Error Response (400 Bad Request):**

{

    "Error": {

        "title": ["Title must be at least 3 characters long."],

        "rating": ["Rating must be between 1 and 5."]

    }

}

**2. Get Post Details**

**Endpoint:** POST /api/posts/detail/

**Request Payload:**

{

    "id": 1

}

**Success Response (200 OK):**

{

    "id": 1,

    "title": "Sample Post Title",

    "content": "This is the main content of the post",

    "author": "username",

    "category": "BLOG",

    "is\_published": true,

    "rating": 4.50,

    "website": "https://example.com",

    "tags\_used": ["Python", "Programming"],

    "comments": [

        {

            "content": "Great post!",

            "username": "commenter1"

        }

    ],

    "like\_count": 2,

    "likes": [

        {"user": "user1"},

        {"user": "user2"}

    ]

}

**Error Response (404 Not Found):**

{

    "Error": "Post with ID 1 not found."

}

**3. Update Post**

**Endpoint:** POST /api/posts/update/

**Request Payload:**

{

    "id": 1,

    "title": "Updated Title",

    "content": "Updated content",

    "category": "TUTORIAL",

    "is\_published": false,

    "rating": 5.0

}

**Success Response (200 OK):**

{

    "Message": "Post updated successfully.",

    "post": {

        "id": 1,

        "title": "Updated Title",

        "content": "Updated content",

        "author": "username",

        "category": "TUTORIAL",

        "is\_published": false,

        "rating": 5.00,

        "website": "https://example.com",

        "tags\_used": ["Python", "Programming"],

        "comments": [...],

        "like\_count": 2,

        "likes": [...]

    }

}

**Error Response (403 Forbidden):**

{

    "Message": "Editing post is restricted to the User only"

}

**4. Delete Post**

**Endpoint:** POST /api/posts/delete/

**Request Payload:**

{

    "id": 1

}

**Success Response (200 OK):**

{

    "message": "Post successfully deleted."

}

**Error Response (403 Forbidden):**

{

    "Message": "Delete post is restricted to the User only"

}

**5. List Posts**

**Endpoint:** GET /api/posts/list/

**Query Parameters:**

* page: Page number (default: 1)
* page\_size: Number of posts per page (default: 3)
* search: Search term for title/content
* ordering: Field to order by (e.g., -is\_published)

**Success Response (200 OK):**

{

    "count": 10,

    "next": "http://api/posts/list/?page=2",

    "previous": null,

    "results": [

        {

            "id": 1,

            "title": "First Post",

            "content": "Content here",

            "author": "username",

            "category": "BLOG",

            "is\_published": true,

            "rating": 4.50,

            "website": "https://example.com",

            "tags\_used": ["Python"],

            "comments": [...],

            "like\_count": 2,

            "likes": [...]

        },

        // ... more posts

    ]

}

**6. Get Posts by Tags**

**Endpoint:** GET /api/posts/tag/<tag\_name>/ or GET /api/posts/tags/?tags=Python&tags=Programming

**Success Response (200 OK):**

[

    {

        "id": 1,

        "title": "Python Post",

        "content": "Python content",

        "author": "username",

        "category": "TUTORIAL",

        "is\_published": true,

        "rating": 4.50,

        "website": "https://example.com",

        "tags\_used": ["Python", "Programming"],

        "comments": [...],

        "like\_count": 0,

        "likes": []

    },

    // ... more posts with matching tags

]

**Error Response (404 Not Found):**

{

    "Message": "No posts found with the tag 'Python'."

}

**Custom Permission**

**Summary**

The UserPermission class is a custom permission class that ensures only the author of a post can edit or delete it, while allowing read access to all authenticated users.

**Implementation Details**

permission.py

Apply

class UserPermission(permissions.BasePermission):

    message = "Editing/Delete post is restricted to the author only"

Permission Logic Flow

1. Basic Authentication Check:

if not request.user or not request.user.is\_authenticated:

    return False

* Immediately denies access if user is not authenticated
* Returns False for anonymous users

1. Safe Methods Check:

if request.method in permissions.SAFE\_METHODS:

    return True

* Allows read-only operations (GET, HEAD, OPTIONS)
* SAFE\_METHODS = ('GET', 'HEAD', 'OPTIONS')

1. Author Verification:

return obj.author.id == request.user.id

- Compares the requesting user's ID with the post author's ID

 Returns True only if they match

Usage in Views

from core.permission import UserPermission

class PostViewSet(viewsets.ModelViewSet):

    permission\_classes = [permissions.IsAuthenticated, UserPermission]

**Error Message**

When permission is denied, returns:

{"detail": "Editing/Delete post is restricted to the author only"}

**Integration with Other Apps**

This permission class can be used with any model that has an author field referencing the user model. Currently used in:

* Posts app
* Comments app
* Any other app requiring author-based permissions

**Likes App**

**Overview**

The Likes app manages the like functionality for posts, allowing users to like and unlike posts with proper validation and tracking.

**Components and Flow**

**1. Models (models/models.py)**

class Like(models.Model):

    post = models.ForeignKey('posts.Post', ...)

    user = models.ForeignKey(User, ...)

    created\_at = models.DateTimeField(...)

**Key Features:**

* Links likes to posts and users
* Ensures unique likes through unique\_together
* Uses dynamic import for Post model to avoid circular dependencies
* Custom string representation for admin interface

**2. Serializers (serializers/serializers.py)**

class LikeSerializer(serializers.ModelSerializer):

    post\_id = serializers.IntegerField(required=True)

    title = serializers.CharField(source='post.title', read\_only=True)

**Features:**

* Handles post\_id validation
* Returns post title in responses
* Read-only fields for user and creation time

**3. Views (views.py)**

class LikeViewSet(viewsets.ViewSet):

    authentication\_classes = [JWTAuthentication]

    permission\_classes = [IsAuthenticated]

**Like Creation Flow:**

1. User sends POST request with post\_id
2. Validates request data using LikeSerializer
3. Checks if post exists
4. Verifies user hasn't already liked the post
5. Creates new like record

**Unlike Flow:**

1. User sends POST request to unlike endpoint
2. Validates post\_id
3. Verifies post exists
4. Checks if like exists
5. Deletes like record

**4. URLs (urls.py)**

urlpatterns = [

    path('like/', LikeViewSet.as\_view({'post': 'create'})),

    path('unlike/', LikeViewSet.as\_view({'post': 'destroy'})),

]

**5. Admin Configuration (admin.py)**

@admin.register(Like)

class LikeAdmin(admin.ModelAdmin):

    list\_display = ('post', 'user', 'created\_at')

**API Documentation**

**1. Like a Post**

**Endpoint:** POST /api/like/

**Request:**

{

    "post\_id": 1

}

**Success Response (201 Created):**

{

    "message": "Post liked successfully."

}

**Error Responses:**

{

    "message": "You have already liked this post."

}

// or

{

    "message": "Post not found"

}

**2. Unlike a Post**

**Endpoint:** POST /api/unlike/

**Request:**

{

    "post\_id": 1

}

**Success Response (200 OK):**

{

    "message": "Post unliked successfully."

}

**Error Responses:**

{

    "message": "You have not liked this post."

}

// or

{

    "message": "Post not found"

}

**Function Call Flow**

1. **Like Creation:**

graph TD

    A[Client Request] --> B[URLs]

    B --> C[LikeViewSet.create]

    C --> D[LikeSerializer Validation]

    D --> E[Check Post Exists]

    E --> F[Check Existing Like]

    F --> G[Create Like]

    G --> H[Return Response]

1. **Unlike Flow:**

graph TD

    A[Client Request] --> B[URLs]

    B --> C[LikeViewSet.destroy]

    C --> D[LikeSerializer Validation]

    D --> E[Check Post Exists]

    E --> F[Find Existing Like]

    F --> G[Delete Like]

    G --> H[Return Response]

**Integration with Posts App**

* Likes are associated with posts through ForeignKey relationship
* Post model can access likes through reverse relationship: post.likes.all()
* Like counts are available in post serialization

Endpoints

1. List Comments

URL: /comments/

Method: GET

Description: Retrieves a list of all comments or a specific comment by ID.

Request Payload (Optional):

{

"id": 1

}

Response:

* Success (200):
* [
* {
* "id": 1,
* "post": 1,
* "user": "john\_doe",
* "content": "Nice Posts",
* "created\_at": "2024-12-25T10:00:00Z"
* }
* ]
* Error (404):
* {
* "detail": "No comment found with the given ID."
* }

Authentication Required: Yes

2. Create Comment

URL: /comments/create/

Method: POST

Description: Creates a new comment on a post.

Request Payload:

{

"content": "Nice Posts",

"post\_id": 1

}

Response:

* Success (201):
* {
* "id": 2,
* "post": 1,
* "user": "john\_doe",
* "content": "Nice Posts",
* "created\_at": "2024-12-25T10:05:00Z"
* }
* Error (404):
* {
* "detail": "Post not found."
* }

Authentication Required: Yes

3. Update Comment

URL: /comments/update/

Method: POST

Description: Updates an existing comment.

Request Payload:

{

"id": 1,

"content": "Updated content"

}

Response:

* Success (200):
* {
* "Message": "Comment updated successfully.",
* "comment": {
* "id": 1,
* "post": 1,
* "user": "john\_doe",
* "content": "Updated content",
* "created\_at": "2024-12-25T10:00:00Z"
* }
* }
* Error (404):
* {
* "Error": "Comment not found."
* }

Authentication Required: Yes

4. Delete Comment

URL: /comments/delete/

Method: POST

Description: Deletes a comment by ID.

Request Payload:

{

"id": 1

}

Response:

* Success (200):
* {
* "message": "Comment successfully deleted."
* }
* Error (404):
* {
* "Error": "Comment not found."
* }

Authentication Required: Yes

Models

Comment Model

| Field | Type | Description |
| --- | --- | --- |
| id | Integer (Primary Key) | Unique identifier for the comment |
| post | ForeignKey | References the Post model |
| user | ForeignKey | References the User model |
| content | TextField | Content of the comment |
| created\_at | DateTimeField | Timestamp when the comment was created |

Serializers

CommentSerializer

* Fields:
  + id
  + post
  + user
  + content
  + created\_at
* Read-only Fields:
  + id
  + user
  + created\_at

Validation:

* content must not be empty or too short.

Permissions

* IsAuthenticated: Required for all endpoints.

Error Handling

* 404 Not Found: Returned when the requested resource does not exist.
* 400 Bad Request: Returned for invalid payloads.
* 401 Unauthorized: Returned when the user is not authenticated.

Logging

* Logs are maintained for debugging and monitoring API usage.
* Levels include INFO, ERROR, and DEBUG.

Authentication

* JWT Authentication is implemented using rest\_framework\_simplejwt.

Notes

* Ensure the payload format matches the examples provided.
* Use proper authentication headers while accessing secured endpoints.

not like this i will provide my code to u

import logging

from rest\_framework import mixins, viewsets, status, filters, permissions

from rest\_framework.exceptions import ValidationError, PermissionDenied

from rest\_framework.permissions import IsAuthenticated

from rest\_framework.response import Response

from rest\_framework\_simplejwt.authentication import JWTAuthentication

from core import UserPermission

from .models.models import Post

from .pagniations.pagniations import CustomPagination

from .serializers.serializers import PostSerializer

from .utlis import PostID

from .decorators import validate\_fields

logger = logging.getLogger(\_\_name\_\_)

"""

Using mixins allows us to add specific functionality (e.g., list, delete,retrieve) to views.

viewsets.GenericViewSet provides the base functionality for creating custom views with mixins.

"""

class PostList(mixins.ListModelMixin, viewsets.GenericViewSet):

queryset = Post.objects.all() # Fetch all Post objects

serializer\_class = PostSerializer # Serialize Post model data

pagination\_class = CustomPagination # Paginate the results

filter\_backends = [filters.SearchFilter, filters.OrderingFilter] # Enable search and ordering

search\_fields = ['title', 'content'] # Searchable fields

ordering\_fields = ['is\_published'] # Fields to order by

ordering = ['-is\_published'] # Default ordering (newest first)

authentication\_classes = [JWTAuthentication] # Use JWT for authentication

permission\_classes = [IsAuthenticated] # Authenticated users can write; others can only read

class PostUserDetail(mixins.RetrieveModelMixin, viewsets.GenericViewSet):

serializer\_class = PostSerializer

authentication\_classes = [JWTAuthentication]

permission\_classes = [IsAuthenticated]

def create(self, request):

post\_id = request.data.get('id') # Get the ID from the request payload

post = PostID.get\_post\_by\_id(post\_id) # Directly call the static method of PostH

if isinstance(post, Response): # If the helper(PostID) function returns a Response (error), return it directly

return post

serializer = self.get\_serializer(post) # Otherwise, process the post instance and return the response

return Response(serializer.data, status=status.HTTP\_200\_OK)

class PostUserUpdate(mixins.UpdateModelMixin, viewsets.GenericViewSet):

serializer\_class = PostSerializer

authentication\_classes = [JWTAuthentication]

permission\_classes = [permissions.IsAuthenticated, UserPermission]

def update(self, request, \*args, \*\*kwargs):

logger.info("Received update request.")

post\_id = request.data.get('id') # Extract post ID from payload

data = request.data # Full payload

try:

updated = PostID.update\_post(post\_id, data) # Call the `update\_post` method

if isinstance(updated, Response): # If the helper method returns a Response, return it directly

logger.error(f"Post update failed for ID {post\_id}: {updated.data}")

return updated

logger.info(f"Post with ID {post\_id} updated successfully.")

self.check\_object\_permissions(request, updated)

serializer = self.get\_serializer(updated) # Serialize the updated post

# print(serializer)

logger.info(f"{serializer}")

response = Response({"Message": "Post updated successfully.", "post": serializer.data},

status=status.HTTP\_200\_OK)

logger.info(f"Response for update request: {response.data}")

return response

except PermissionDenied as e:

logger.warning(f"Permission denied for user {request.user.username} to update post {post\_id}")

return Response({"Message": "Editing post is restricted to the User only"},

status=status.HTTP\_403\_FORBIDDEN)

except Exception as e:

# Log unexpected errors

logger.error(f"Unexpected error during post update for Post ID {post\_id}: {str(e)}", exc\_info=True)

return Response({"Error": "An unexpected error occurred while updating the post."},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

class PostUserDestroy(mixins.DestroyModelMixin, viewsets.GenericViewSet):

serializer\_class = PostSerializer

authentication\_classes = [JWTAuthentication]

permission\_classes = [IsAuthenticated]

permission\_classes = [permissions.IsAuthenticated, UserPermission]

def delete(self, request, \*args, \*\*kwargs):

post\_id = request.data.get('id') # Get the ID from the request payload

try:

try:

delete\_post = Post.objects.get(id=post\_id)

except Post.DoesNotExist:

logger.error(f"Post with ID {post\_id} not found.")

return Response({"Error": "Post not found."}, status=status.HTTP\_404\_NOT\_FOUND)

self.check\_object\_permissions(request, delete\_post)

response = PostID.delete\_post\_by\_id(post\_id) # Call the helper method to delete the post

logger.info(f"Post with ID {post\_id} successfully deleted.")

return response # Return the response from the helper method

except PermissionDenied as e:

logger.warning(f"Permission denied for user {request.user.username} to update post {post\_id}")

return Response({"Message": "Delete post is restricted to the User only"}, status=status.HTTP\_403\_FORBIDDEN)

class PostUserCreate(viewsets.GenericViewSet, mixins.CreateModelMixin):

serializer\_class = PostSerializer

authentication\_classes = [JWTAuthentication]

permission\_classes = [IsAuthenticated]

@validate\_fields # Apply the custom validation decoratora

def create(self, request, \*args, \*\*kwargs):

try:

author = request.user if request.user.is\_authenticated else None # Check if the user is authenticated, and assign the author field

serializer = self.get\_serializer(data=request.data)

if serializer.is\_valid(): # Validate serializer and add the authenticated author

post = serializer.save(author=author) # Assign the author (authenticated user)

logger.info(f"Post created: {post}")

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

logger.error(f"Validation errors: {serializer.errors}")

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

except ValidationError as e: # Handle validation error raised in the decorator

logger.error(f"Validation error: {e.detail}")

return Response({"Error": e.detail}, status=status.HTTP\_400\_BAD\_REQUEST)

except Exception as e: # Handle unexpected errors

logger.error(f"Unexpected error during post creation: {str(e)}", exc\_info=True)

return Response({"Error": f"An unexpected error occurred: {str(e)}"},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

this is my views.py code of posts app

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

this is utlis.py code of posts end points

import logging

from rest\_framework.exceptions import ValidationError

from rest\_framework.response import Response

from rest\_framework import status, request

from .models.models import Post, Tag # Assuming you have a Tag model for tags

from .serializers.serializers import PostSerializer

logger = logging.getLogger(\_\_name\_\_)

class PostID():

@staticmethod

def get\_post\_by\_id(post\_id):

if not post\_id:

logger.error("Post ID is required but not provided.")

return Response({"Error": "Post ID is required in the payload."}, status=status.HTTP\_400\_BAD\_REQUEST)

try:

post = Post.objects.get(id=post\_id)

logger.info(f"Post retrieved successfully with ID {post\_id}")

return post # Return the post instance for further processing

except Post.DoesNotExist:

logger.warning(f"Post with ID {post\_id} not found.")

return Response({"Error": f"Post with ID {post\_id} not found."}, status=status.HTTP\_404\_NOT\_FOUND)

@staticmethod

def delete\_post\_by\_id(post\_id):

logger.info(f"Attempting to delete post with ID: {post\_id}")

# Retrieve the post using the helper method

post = PostID.get\_post\_by\_id(post\_id)

if isinstance(post, Response):

logger.error(f"Failed to delete post. Post with ID {post\_id} not found.")

return post # Return error Response directly

try:

post.delete()

logger.info(f"Post with ID {post\_id} successfully deleted.")

return Response({"message": "Post successfully deleted."}, status=status.HTTP\_200\_OK)

except Exception as e:

logger.error(f"An error occurred while deleting post with ID {post\_id}: {str(e)}")

return Response({"error": "An error occurred while deleting the post."},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

# class PostID:

@staticmethod

def update\_post(post\_id, data):

post = PostID.get\_post\_by\_id(post\_id)

if isinstance(post, Response):

logger.error(f"Failed to update post. Post with ID {post\_id} not found.")

return post

try:

serializer = PostSerializer()

title = data.get('title')

content = data.get('content')

category = data.get('category')

is\_published = data.get('is\_published')

rating = data.get('rating')

website = data.get('website')

# tags=data.get('name')

if title is not None:

title = serializer.validate\_title(title)

if rating is not None:

rating = serializer.validate\_rating(rating)

if is\_published is not None:

is\_published = serializer.validate\_is\_published(is\_published)

# if category is not None:

# category = serializer.validate\_category(category)

post.title = title

post.content = content

post.category = category

post.is\_published = is\_published

post.rating = rating

post.website = website

# post.tags=tags

post.save()

return post

except ValidationError as e: # Catch validation errors and return them in the response

logger.error(f"Validation error during post update for post ID {post\_id}: {e.detail}")

return Response({"Error": e.detail}, status=status.HTTP\_400\_BAD\_REQUEST)

except Exception as e:

logger.error(f"Unexpected error during post update for post ID {post\_id}: {str(e)}", exc\_info=True)

return Response({"Error": "An unexpected error occurred while updating the post."},

status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

this is my urls.py code

from django.urls import path

from .views import PostList, PostUserDetail, PostUserUpdate, PostUserDestroy, PostUserCreate

from rest\_framework import routers

from .tags import TagView

app\_name = 'Posts'

simple\_router = routers.SimpleRouter()

List\_Post = PostList.as\_view({'post': 'list'})

Update\_Post = PostUserUpdate.as\_view({'post': 'update'})

Delete\_Post = PostUserDestroy.as\_view({'post': 'delete'})

User\_ID = PostUserDetail.as\_view({'post': 'create'})

Create\_Post = PostUserCreate.as\_view({'post': 'create'})

# Tags = TagView.as\_view({'post': 'create'})

urlpatterns = simple\_router.urls

urlpatterns = urlpatterns + [

path('posts/list/', List\_Post, name='List\_Posts'),

path('posts/create/', Create\_Post, name='Create\_Post'),

path('posts/update/', Update\_Post, name='Update\_Post'),

path('posts/delete/', Delete\_Post, name='Delete\_Post'),

path('posts/id/', User\_ID, name='User\_ID'),

# endpoints for tag

path('posts/tag/<str:tag\_name>/', TagView.as\_view(), name='single-tag-filter'),

path('posts/tag/', TagView.as\_view(), name='multi-tag-filter'),

]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mode.py of posts

from django.db import models

from django.contrib.auth.models import User

from django.utils.translation import gettext\_lazy as \_

class Post(models.Model):

title = models.CharField(\_('Title'), max\_length=50, help\_text=\_(

"The title of the post (max 200 characters)."), blank=False) # A title for the post (text)

content = models.TextField(\_('Content'), max\_length=200, help\_text=\_(

"The main content of the post. Use rich text for full details."), ) # Content of the post (long text)

created\_at = models.DateTimeField(auto\_now\_add=True, help\_text=\_(

"The date and time when the post was created."), ) # Date when post was created

updated\_at = models.DateTimeField(auto\_now=True, help\_text=\_(

"The date and time when the post was last updated."), ) # Date when post was last updated

author = models.ForeignKey(User, on\_delete=models.CASCADE, related\_name='posts',

help\_text=\_(

"The author of this post (related to User model).")) # Author of the post (related to User model)

# Choice field for post category (Example: Blog, News, etc.)

CATEGORY\_CHOICES = [

('BLOG', \_('Blog')),

('NEWS', \_('News')),

('REVIEW', \_('Review')),

('TUTORIAL', \_('Tutorial')),

]

category = models.CharField(\_('Category'), max\_length=50, choices=CATEGORY\_CHOICES, default='BLOG',blank=True, null=True,

help\_text=\_("Category of the post (e.g., Blog, News)."))

is\_published = models.BooleanField(\_('Publication Date'), default=True,

help\_text=\_(

"Flag indicating whether the post is published.")) # A boolean field to mark if the post is published

rating = models.DecimalField(\_('Rating'), max\_digits=3, decimal\_places=2, default=0.00,

help\_text=\_(

"Rating of the post (e.g., a scale from 0 to 5).")) # A DecimalField for rating or price if the post has any

website = models.URLField(\_('Website'), blank=True, null=True, help\_text=\_(

"An external link related to the post.")) # A URL field for an external link or reference

# A JSON field for storing additional metadata (requires PostgreSQL)

metadata = models.JSONField(\_('Metadata'), blank=True, null=True,

help\_text=\_("Additional metadata related to the post (e.g., JSON structure)."))

# A field for tags (using Many-to-Many relationship)

tags = models.ManyToManyField(

'Tag', blank=True, help\_text=\_("Tags associated with this post (e.g., #Blog, #lifestyle)."))

def \_\_str\_\_(self):

return self.title

class Meta:

ordering = ['-created\_at']

db\_table = "Postify\_post" # Custom table name for the database

verbose\_name = \_("Post") # Singular name for the model

verbose\_name\_plural = \_("Posts") # Plural name for the model

class Tag(models.Model):

name = models.CharField(max\_length=50, unique=True, blank=True, null=True,

help\_text=\_("Tag name to categorize posts."))

def \_\_str\_\_(self):

return self.name

class Meta:

db\_table = "Postify\_tag" # Custom table name for the database

verbose\_name = \_("Tag") # Singular name for the model

verbose\_name\_plural = \_("Tags")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

serializer.py

import logging

from rest\_framework import serializers

from ..models.models import Post, Tag

from django.apps import apps

# logger = logging.getLogger(\_\_name\_\_)

class CommentSerializer(serializers.ModelSerializer):

"""Serializer for displaying comments on a post."""

user = serializers.StringRelatedField() # Display the username of the commenter

class Meta:

Comment = apps.get\_model('comments',

'Comment') # Dynamic import. Used this because directly import is not working

model = Comment

fields = ['user', 'content']

class LikeSerializer(serializers.ModelSerializer):

"""Serializer for displaying likes on a post."""

user = serializers.StringRelatedField() # Display the username of the liker

class Meta:

Like = apps.get\_model('likes', 'Like') # Dynamic import. Used this because directly import is not working

model = Like

fields = ['user']

class TagSerializer(serializers.ModelSerializer):

"""Serializer for the Tag model."""

class Meta:

model = Tag

fields = ['name']

class PostSerializer(serializers.ModelSerializer):

"""Serializer for the Post model."""

author = serializers.CharField(source='author.username', read\_only=True) # Fetch the username of the author

# comments = CommentSerializer(many=True, read\_only=True) # Nested comments

comments = serializers.SerializerMethodField() # Custom field for comments

likes = LikeSerializer(many=True, read\_only=True) # Nested likes

like\_count = serializers.SerializerMethodField() # Display the total number of likes

tags = serializers.ListField(

child=serializers.DictField(), write\_only=True)

tags\_used = serializers.SerializerMethodField(read\_only=True) # Display tags as names

class Meta:

model = Post

fields = ['id', 'title', 'content', 'author', 'category', 'is\_published', 'rating', 'website', 'tags',

'comments', 'like\_count', 'likes', "tags\_used"]

def get\_comments(self, obj):

"""Get all comments for the post."""

comments\_qs = obj.comments.all()

if comments\_qs.exists():

return [

{"content": comment.content, "username": comment.user.username}

for comment in comments\_qs

]

# logger.info(f"No comments found for Post ID {obj.id}")

return "This Post Has No Comments"

def get\_like\_count(self, obj):

"""Get the total number of likes for the post."""

like\_count = obj.likes.count()

# logger.info(f"Post ID {obj.id} has {like\_count} likes")

return like\_count

def get\_tags\_used(self, obj):

"""Display tags as a list of names."""

return [tag.name for tag in obj.tags.all()]

def create(self, validated\_data):

print("create\_tag")

tags\_data = validated\_data.pop('tags', []) # Extract tags data

post = Post.objects.create(\*\*validated\_data) # Create the post

for tag\_data in tags\_data: # joining tags with the post

print("helllllooooooooooo")

tag\_name = tag\_data.get("name") # Extract the tag name

tag, created = Tag.objects.get\_or\_create(name=tag\_name) # Create or retrieve the tag

print(tag, "taaagggggg")

post.tags.add(tag) # Associate the tag with the post

return post

# Associate tags with the post

def validate\_title(self, value):

# print("you are in the serializer function")

if len(value) < 0:

raise serializers.ValidationError("Title cannot be Null")

elif len(value) < 3:

raise serializers.ValidationError("Title must be at least 3 characters long.")

return value

def validate\_rating(self, value):

"""Validate the rating field."""

try:

value = float(value)

except ValueError:

raise serializers.ValidationError("Rating must be a number between 1 and 5.")

if value < 1 or value > 5:

raise serializers.ValidationError("Rating must be between 1 and 5.")

return value

def validate\_is\_published(self, value):

"""Validate the is\_published field."""

if not isinstance(value, bool):

raise serializers.ValidationError("is\_published must be a boolean value. i.e true and false")

return value

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

decoratory.py

from rest\_framework.exceptions import ValidationError

import logging

from .serializers.serializers import PostSerializer

logger = logging.getLogger(\_\_name\_\_)

def validate\_fields(func):

"""Custom decorator to validate post data before creating or updating."""

def wrapper(self, request, \*args, \*\*kwargs):

# Extract data from the request

data = request.data

serializer = PostSerializer(data=data)

if not serializer.is\_valid():

raise ValidationError(serializer.errors) # Raise validation error if data is invalid

# If validation passes, continue to the original function

return func(self, request, \*args, \*\*kwargs)

return wrapper

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tags.py

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import status

from .models.models import Post, Tag

from .serializers.serializers import PostSerializer

class TagView(APIView):

"""

Supports:

- Single tag in the URL path: /api/posts/tag/<tag\_name>/

- Multiple tags as query parameters: /api/posts/tags/?tags=Python&tags=Programming

"""

def get(self, request, tag\_name=None, \*args, \*\*kwargs):

# Check if a single tag is provided in the URL

if tag\_name:

posts = Post.objects.filter(tags\_\_name=tag\_name).distinct()

if not posts.exists():

return Response({"Message": f"No posts found with the tag '{tag\_name}'."},

status=status.HTTP\_404\_NOT\_FOUND)

serializer = PostSerializer(posts, many=True)

return Response(serializer.data, status=status.HTTP\_200\_OK)

# Check if multiple tags are provided as query parameters

tag\_names = request.query\_params.getlist('tags')

if tag\_names:

posts = Post.objects.filter(tags\_\_name\_\_in=tag\_names).distinct()

if not posts.exists():

return Response(

{"Message": f"No posts found matching the tags: {', '.join(tag\_names)}."},

status=status.HTTP\_404\_NOT\_FOUND)

serializer = PostSerializer(posts, many=True)

return Response(serializer.data, status=status.HTTP\_200\_OK)

# No tags provided in either path or query parameters

return Response(

{"Message": "No tags provided. Use a single tag in the URL path or multiple tags as query parameters."},

status=status.HTTP\_400\_BAD\_REQUEST

)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

now make