**Blog Application with Advanced Features**

**Objective**

Create a blog application with advanced features to test your skills in Django, database design, user authentication, API development, and deployment.

**Disclaimer**

This task is part of the interview process and is intended to assess your technical skills. Completion of this task does not guarantee employment. Your performance will be evaluated in conjunction with other factors throughout the interview process. Additionally, the code you submit will not be used in any production environment or for any commercial purposes.

**Requirements**

**Models**

Create models for Post, Category, Tag, Comment, and Profile.

**Post**

* title: CharField
* content: TextField
* author: ForeignKey to Profile
* categories: ManyToManyField with Category
* tags: ManyToManyField with Tag
* created\_at: DateTimeField
* updated\_at: DateTimeField

**Category**

* name: CharField
* slug: SlugField

**Tag**

* name: CharField
* slug: SlugField

**Comment**

* post: ForeignKey to Post
* author: ForeignKey to Profile
* content: TextField
* created\_at: DateTimeField

**Profile**

* user: OneToOneField with Django's User
* bio: TextField
* profile\_picture: ImageField

**User Authentication**

Implement user registration, login, logout, and profile management using Django's built-in authentication system. Ensure secure password handling and validation.

**CRUD Operations**

Implement CRUD operations for Posts and Comments with proper access control:

* Only authenticated users can create, update, or delete their own posts and comments.

**API Development**

Develop a RESTful API using Django REST Framework (DRF) with the following endpoints:

* /api/posts/ (GET, POST)
* /api/posts/<id>/ (GET, PUT, DELETE)
* /api/categories/ (GET)
* /api/tags/ (GET)
* /api/comments/ (GET, POST)
* /api/comments/<id>/ (DELETE)

**Search and Filtering**

* Implement search functionality to search posts by title and content.
* Implement filtering by categories and tags.

**Pagination**

Add pagination to the list of posts in the API.

**Admin Interface**

Customize the Django admin interface to manage posts, categories, tags, comments, and profiles efficiently.

**Implementation Details**

**Setting Up the Project**

1. **pip packages**
2. asgiref==3.8.1
3. charset-normalizer==3.3.2
4. Django==5.0.6
5. django-extensions==3.2.3
6. django-filter==24.2
7. django-jazzmin==3.0.0
8. djangorestframework==3.15.2
9. Markdown==3.6
10. pillow==10.3.0
11. sqlparse==0.5.0
12. typing\_extensions==4.12.2
13. tzdata==2024.1
14. **Create a new Django project and app:**

bash

Copy code

“./venv/Scripts/Activate.ps1” (on windows)

on linux “source /venv/Scipts/activate”

django-admin startproject blog\_project

cd blog\_project

django-admin startapp blog\_app

django-admin startapp userauths

1. **Add blog\_app and required packages to INSTALLED\_APPS in settings.py:**

python

Copy code

INSTALLED\_APPS = [

    'jazzmin',

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'rest\_framework',

    'userauths',

    'blogApp'

]

**Models**

Create models in blog\_app/models.py:

python

Copy code

*from* django.db *import* models

*# Create your models here.*

*from* userauths.models *import* Profile

class Category(models.Model):

    name = models.CharField(*max\_length*=100)

    slug = models.SlugField(*unique*=True)

    def \_\_str\_\_(*self*):

*return* self.name

class Tag(models.Model):

    name = models.CharField(*max\_length*=100)

    slug = models.SlugField(*unique*=True)

    def \_\_str\_\_(*self*):

*return* self.name

class Post(models.Model):

    title = models.CharField(*max\_length*=200)

    content = models.TextField()

    author = models.ForeignKey(Profile, *on\_delete*=models.CASCADE)

    categories = models.ManyToManyField(Category,*related\_name*="categ")

    tags = models.ManyToManyField(Tag,*related\_name*="tags")

    created\_at = models.DateTimeField(*auto\_now\_add*=True)

    updated\_at = models.DateTimeField(*auto\_now*=True)

    def \_\_str\_\_(*self*):

*return* self.title

class Comment(models.Model):

    post = models.ForeignKey(Post, *on\_delete*=models.CASCADE, *related\_name*='comments')

    author = models.ForeignKey(Profile, *on\_delete*=models.CASCADE)

    content = models.TextField()

    created\_at = models.DateTimeField(*auto\_now\_add*=True)

    def \_\_str\_\_(*self*):

*return* f'Comment by {self.author.user.username} on {self.post.title}'

Create models in userauths/models.py:

*from* django.contrib.auth.models *import* AbstractUser

*from* django.db *import* models

class User(AbstractUser):

    email = models.EmailField(*unique*=True)

    username = models.CharField(*max\_length*=100)

    USERNAME\_FIELD = 'email'

    REQUIRED\_FIELDS = ['username']

    def \_\_str\_\_(*self*):

*return* self.username

class Profile(models.Model):

    user = models.OneToOneField(User, *on\_delete*=models.CASCADE, *related\_name*='profile')

    bio = models.CharField(*max\_length*=100)

    profile\_picture = models.ImageField(*upload\_to*='profile\_pictures/', *blank*=True, *null*=True)

    def \_\_str\_\_(*self*):

*return* f'Profile of {self.user.username}'

**User Authentication**

Use Django's built-in authentication system for user registration, login, logout, and profile management. Add allauth and rest\_auth for handling authentication via API.

**CRUD Operations**

Implement views and serializers for handling CRUD operations in blog\_app/views.py ,blog\_app/serializers.py , blog\_app/permissions.py, blog\_app/forms.py, blogApp/custome-pagination.py , blog\_app/admin.py respectively.

views.py

*from* django.shortcuts *import* render, redirect, get\_object\_or\_404

*from* django.contrib *import* messages

*from* django.contrib.auth.decorators *import* login\_required

*from* .models *import* Post, Comment ,Tag ,Category

*from* .forms *import* PostForm, CommentForm

*from* .serializers *import* PostSerializer, CategorySerializer, TagSerializer, CommentSerializer

*# views.py*

*from* rest\_framework *import* generics, status

*from* rest\_framework.response *import* Response

*from* rest\_framework.permissions *import* IsAuthenticatedOrReadOnly

*from* .permissions *import* IsOwnerOrReadOnly  *# Import custom permission*

*from* rest\_framework.pagination *import* PageNumberPagination

*from* rest\_framework.generics *import* ListCreateAPIView

*from* .custom\_pagination *import* CustomPagination

*from* rest\_framework.generics *import* ListCreateAPIView

*from* django.db.models *import* Prefetch

*from* userauths.models *import* Profile

*# Create your views here.*

def home(*request*):

*# Fetch all posts and prefetch related categories and tags*

    posts = Post.objects.prefetch\_related(

        Prefetch('categories', *queryset*=Category.objects.all(), *to\_attr*='post\_categories'),

        Prefetch('tags', *queryset*=Tag.objects.all(), *to\_attr*='post\_tags')

    ).all()

*# Fetch other required data*

    comments = Comment.objects.all()

    categories = Category.objects.all()

    tags = Tag.objects.all()

    context = {

        'posts': posts,

        'comments': comments,

        'categories': categories,

        'tags': tags,

    }

    print(posts.values('categories', 'tags'))

*return* render(request, 'index.html', context)

@login\_required

def create\_post(*request*):

*if* request.method == 'POST':

        form = PostForm(request.POST)

*if* form.is\_valid():

            post = form.save(*commit*=False)

            post.author = request.user.profile  *# Assign the Profile instance of the logged-in user*

            post.save()

            form.save\_m2m()  *# Save many-to-many fields after saving the instance*

            messages.success(request, 'Post created successfully!')

*return* redirect('blogApp:home')

*else*:

*# If form is not valid, handle the form data manually*

            form\_data = {

                'author': request.user.profile,  *# Assign the Profile instance directly*

                'title': request.POST.get('title', ''),

                'content': request.POST.get('content', '')

            }

            post = Post(*author*=form\_data['author'], *title*=form\_data['title'], *content*=form\_data['content'])

            post.save()

            messages.success(request, 'Post created successfully!')

*return* redirect('blogApp:home')

*else*:

        form = PostForm()

*return* render(request, 'create\_post.html', {'form': form})

@login\_required

def update\_post(*request*, *post\_id*):

    post = get\_object\_or\_404(Post, *id*=post\_id)

*if* request.user.profile != post.author:

        messages.error(request, 'You are not authorized to update this post.')

*return* redirect('blogApp:post\_detail', *post\_id*=post.id)

*if* request.method == 'POST':

        form = PostForm(request.POST, *instance*=post)

*if* form.is\_valid():

            form.save()

            messages.success(request, 'Post updated successfully!')

*else*:

            print("method reached update")

            post.title = request.POST.get('title', '')

            post.content = request.POST.get('content', '')

            post.save()

            messages.error(request, 'Form is not valid. Please correct the errors.')

*return* redirect('blogApp:home')

*else*:

        form = PostForm(*instance*=post)

*return* render(request, 'update\_post.html', {'form': form, 'post': post})

@login\_required

def delete\_post(*request*, *post\_id*):

    post = get\_object\_or\_404(Post, *id*=post\_id)

*if* request.user.profile != post.author:

        messages.error(request, 'You are not authorized to delete this post.')

*return* redirect('blogApp:home')

*if* request.method == 'POST':

        print("method reached delete")

        post.delete()

        messages.success(request, 'Post deleted successfully!')

*return* redirect('blogApp:home')  *# Redirect to home or any other appropriate page after deletion*

*return* render(request, 'delete\_post.html', {'post': post})

@login\_required

def post\_detail(*request*, *post\_id*):

    post = get\_object\_or\_404(Post, *id*=post\_id)

    comments = post.comments.all()

    comment\_form = CommentForm()

*return* render(request, 'post\_detail.html', {'post': post, 'comments': comments, 'comment\_form': comment\_form})

@login\_required

def create\_comment(*request*, *post\_id*):

    post = get\_object\_or\_404(Post, *id*=post\_id)

*if* request.method == 'POST':

        form = CommentForm(request.POST)

*if* form.is\_valid():

            comment = form.save(*commit*=False)

            comment.author = request.user.profile

            comment.post = post

            comment.save()

            messages.success(request, 'Comment added successfully!')

*return* redirect('blogApp:home')

*else*:

        form = CommentForm()

*return* render(request, 'create\_comment.html', {'form': form, 'post': post})

@login\_required

def update\_comment(*request*, *comment\_id*):

    comment = get\_object\_or\_404(Comment, *id*=comment\_id)

*if* request.user.profile != comment.author:

        messages.error(request, 'You are not authorized to update this comment.')

*return* redirect('blogApp:home', *post\_id*=comment.post.id)

*if* request.method == 'POST':

        form = CommentForm(request.POST, *instance*=comment)

*if* form.is\_valid():

            form.save()

            messages.success(request, 'Comment updated successfully!')

*return* redirect('blogApp:post\_detail', *post\_id*=comment.post.id)

*else*:

        form = CommentForm(*instance*=comment)

*return* render(request, 'update\_comment.html', {'form': form, 'comment': comment})

@login\_required

def delete\_comment(*request*, *comment\_id*):

    comment = get\_object\_or\_404(Comment, *id*=comment\_id)

*if* request.user.profile != comment.author:

        messages.error(request, 'You are not authorized to delete this comment.')

*return* redirect('blogApp:home')

*if* request.method == 'POST':

        comment.delete()

        messages.success(request, 'Comment deleted successfully!')

*return* redirect('blogApp:home')

*return* render(request, 'delete\_comment.html', {'comment': comment})

serializers.py

*from* rest\_framework *import* serializers

*from* .models *import* Post, Category, Tag, Comment

*from* userauths.models *import* Profile  *# Import Profile from userauths app*

*from* .permissions *import* IsOwnerOrReadOnly  *# Import custom permission*

class ProfileSerializer(serializers.ModelSerializer):

    class Meta:

        model = Profile

        fields = '\_\_all\_\_'  *# Include all fields for Profile*

class CategorySerializer(serializers.ModelSerializer):

    class Meta:

        model = Category

        fields = '\_\_all\_\_'

class TagSerializer(serializers.ModelSerializer):

    class Meta:

        model = Tag

        fields = '\_\_all\_\_'

*from* rest\_framework *import* serializers

*from* .models *import* Post, Category, Tag

*from* .serializers *import* CategorySerializer, TagSerializer

*from* rest\_framework.exceptions *import* ValidationError

class PostSerializer(serializers.ModelSerializer):

    categories = serializers.PrimaryKeyRelatedField(

*queryset*=Category.objects.all(),

*many*=True,

*required*=False

    )

    tags = serializers.PrimaryKeyRelatedField(

*queryset*=Tag.objects.all(),

*many*=True,

*required*=False

    )

    class Meta:

        model = Post

        fields = '\_\_all\_\_'

        read\_only\_fields = ['author']  *# Make author read-only*

*# Custom create method to handle categories and tags*

    def create(*self*, *validated\_data*):

        categories\_data = validated\_data.pop('categories', [])

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

        request = self.context.get('request')

*if* request and request.user.is\_authenticated:

*try*:

                author\_profile = request.user.profile

*except* Profile.DoesNotExist:

*raise* ValidationError("Profile does not exist for this user.")

            validated\_data['author'] = author\_profile

*else*:

*raise* ValidationError("User must be logged in.")

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

*for* category *in* categories\_data:

            post.categories.add(category)

*for* tag *in* tags\_data:

            post.tags.add(tag)

*return* post

*# Apply custom permission*

    def get\_permissions(*self*):

*if* self.context['request'].method == 'DELETE':

*return* [IsOwnerOrReadOnly()]

*return* []

*from* rest\_framework.exceptions *import* ValidationError

*from* rest\_framework *import* serializers

*from* .models *import* Comment

*from* .models *import* Profile  *# Import Profile model if not already imported*

*from* rest\_framework.exceptions *import* ValidationError

class CommentSerializer(serializers.ModelSerializer):

    class Meta:

        model = Comment

        fields = '\_\_all\_\_'

        read\_only\_fields = ['author']  *# Make author read-only for updates*

    def create(*self*, *validated\_data*):

        request = self.context.get('request')

*if* request and request.user.is\_authenticated:

*try*:

                author\_profile = request.user.profile

*except* Profile.DoesNotExist:

*raise* ValidationError("Profile does not exist for this user.")

            validated\_data['author'] = author\_profile

*else*:

*raise* ValidationError("User must be logged in.")

        comment = Comment.objects.create(\*\*validated\_data)

*return* comment

permissions.py

*# permissions.py*

*from* rest\_framework *import* permissions

*from* django.shortcuts *import* redirect

class IsOwnerOrReadOnly(permissions.BasePermission):

    """

    Custom permission to only allow owners of an object to delete it.

    """

    def has\_object\_permission(*self*, *request*, *view*, *obj*):

*# Read permissions are allowed to any request,*

*# so we'll always allow GET, HEAD, or OPTIONS requests.*

*if* request.method in permissions.SAFE\_METHODS:

*return* True

*# If the request method is DELETE, check ownership*

*elif* request.method == 'DELETE':

*# If the user is authenticated, check if they are the owner*

*if* request.user.is\_authenticated:

*return* obj.author.user == request.user

*# If the user is not authenticated, raise a permission denied error*

*else*:

*return* False  *# Return False instead of redirecting*

*# For other request methods (PUT, PATCH, etc.), we allow read-only access*

*return* True

forms.py

*from* django *import* forms

*from* .models *import* Post,Comment

class PostForm(forms.ModelForm):

    class Meta:

        model = Post

        fields = ['title', 'content', 'categories', 'tags']  *# Assuming categories and tags are many-to-many fields*

    def \_\_init\_\_(*self*, \**args*, \*\**kwargs*):

        super().\_\_init\_\_(\*args, \*\*kwargs)

*# Add custom initialization if needed*

        self.fields['categories'].widget.attrs.update({'class': 'form-control', 'multiple': True})

        self.fields['tags'].widget.attrs.update({'class': 'form-control', 'multiple': True})

class CommentForm(forms.ModelForm):

    class Meta:

        model = Comment

        fields = ['content']

    def \_\_init\_\_(*self*, \**args*, \*\**kwargs*):

        super().\_\_init\_\_(\*args, \*\*kwargs)

*# Add custom initialization if needed*

        self.fields['content'].widget.attrs.update({'class': 'form-control', 'rows': 3})

**Pagination**

Add pagination to the list views in blogApp/custome-pagination.py:

*from* rest\_framework.pagination *import* PageNumberPagination

*from* rest\_framework.response *import* Response

class CustomPagination(PageNumberPagination):

    page\_size\_query\_param = 'items\_per\_page'

    page\_query\_param = 'page'

    max\_page\_size = 100  *# Default maximum page size*

    def get\_page\_size(*self*, *request*):

        """

        Get the number of items per page from the request query parameters.

        """

*return* int(request.query\_params.get(self.page\_size\_query\_param, self.max\_page\_size))

    def paginate\_queryset(*self*, *queryset*, *request*, *view*=None):

        """

        Paginate a queryset according to the custom pagination parameters.

        """

        self.page\_size = self.get\_page\_size(request)

*return* super().paginate\_queryset(queryset, request, view)

    def get\_paginated\_response(*self*, *data*):

        """

        Return a paginated-style response with custom pagination metadata.

        """

        next\_url = self.get\_next\_link()

        previous\_url = self.get\_previous\_link()

*return* Response({

            'pagination': {

                'next': next\_url,

                'previous': previous\_url,

                'count': self.page.paginator.count,

                'total\_pages': self.page.paginator.num\_pages,

                'current\_page': self.page.number,

                'results': data,

            }

        })

**Admin Interface**

Customize the Django admin interface to manage posts, categories, tags, comments, and profiles in blog\_app/admin.py:

*from* django.contrib *import* admin

*from* .models *import* Category, Tag, Post, Comment

@admin.register(Category)

class CategoryAdmin(admin.ModelAdmin):

    prepopulated\_fields = {'slug': ('name',)}

@admin.register(Tag)

class TagAdmin(admin.ModelAdmin):

    prepopulated\_fields = {'slug': ('name',)}

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

    list\_display = ('title', 'author', 'created\_at', 'updated\_at')

    list\_filter = ('categories', 'tags', 'author', 'created\_at')

    search\_fields = ('title', 'content','categories\_\_name','tags\_\_name')

*# prepopulated\_fields = {'slug': ('title',)}  # If you have a slug field in Post model*

@admin.register(Comment)

class CommentAdmin(admin.ModelAdmin):

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

    list\_filter = ('author', 'post', 'created\_at')

    search\_fields = ('content',)

**Now Moving To blogApp urls**

*from* django.contrib *import* admin

*from* django.urls *import* path,include,re\_path

*from* django.conf *import* settings

*from* django.views.static *import* serve

*from* .views *import* (home,create\_post,update\_post,delete\_post,post\_detail,create\_comment,update\_comment,delete\_comment,

PostListAPIView,PostRetrieveUpdateDestroyAPIView,CategoryListAPIView,TagListAPIView,CommentListCreateAPIView,CommentDestroyAPIView)

app\_name = "blogApp"

urlpatterns = [

path("",home,*name*="home"),

   path('post/create/', create\_post, *name*='create\_post'),

    path('post/<int:post\_id>/update/', update\_post, *name*='update\_post'),

    path('post/<int:post\_id>/delete/', delete\_post, *name*='delete\_post'),

    path('post/<int:post\_id>/', post\_detail, *name*='post\_detail'),

    path('post/<int:post\_id>/comment/create/', create\_comment, *name*='create\_comment'),

    path('comment/<int:comment\_id>/update/', update\_comment, *name*='update\_comment'),

    path('comment/<int:comment\_id>/delete/', delete\_comment, *name*='delete\_comment'),

*# API Is THE Following*

    path('api/posts/<int:pk>/', PostRetrieveUpdateDestroyAPIView.as\_view(), *name*='post-detail'),

    path('api/comments/<int:pk>/', CommentDestroyAPIView.as\_view(), *name*='comment-detail'),

    path('api/posts/', PostListAPIView.as\_view(), *name*='post-list-create'),

    path('api/categories/', CategoryListAPIView.as\_view(), *name*='category-list'),

    path('api/tags/', TagListAPIView.as\_view(), *name*='tag-list'),

    path('api/comments/', CommentListCreateAPIView.as\_view(), *name*='comment-list-create'),

]

**API Development**

Adding API views using Django REST Framework in blog\_app/views.py:

class PostListAPIView(ListCreateAPIView):

    queryset = Post.objects.all()

    serializer\_class = PostSerializer

    pagination\_class = CustomPagination

class CategoryListAPIView(ListCreateAPIView):

    queryset = Category.objects.all()

    serializer\_class = CategorySerializer

    pagination\_class = CustomPagination

class TagListAPIView(ListCreateAPIView):

    queryset = Tag.objects.all()

    serializer\_class = TagSerializer

    pagination\_class = CustomPagination

class CommentListCreateAPIView(ListCreateAPIView):

    queryset = Comment.objects.all()

    serializer\_class = CommentSerializer

    pagination\_class = CustomPagination

    def get\_permissions(*self*):

*if* self.request.method == 'DELETE':

*return* [IsOwnerOrReadOnly()]  *# Assuming IsOwnerOrReadOnly checks permissions*

*return* []

**Search and Filtering**

search and filtering functionality in Frontend By Using Django Admin Page in blogApp/admin.py and as for client Javascript Lib (Jquery) is used

blogApp/admin.py

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

    list\_display = ('title', 'author', 'created\_at', 'updated\_at')

    list\_filter = ('categories', 'tags', 'author', 'created\_at')

    search\_fields = ('title', 'content','categories\_\_name','tags\_\_name')

*# prepopulated\_fields = {'slug': ('title',)}  # If you have a slug field in Post model*

@admin.register(Comment)

class CommentAdmin(admin.ModelAdmin):

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

    list\_filter = ('author', 'post', 'created\_at')

    search\_fields = ('content',)

(as for Client)

$(document).ready(function() {

    let my\_Categ\_ids = [];

    let my\_tags\_ids = [];

*// Function to handle filtering based on current selections*

    function filterPosts() {

        var searchText = $('#Search\_input').val().toLowerCase();

        $('.post-container').each(function() {

            var postTitle = $(*this*).data('post\_title').toLowerCase();

            var postContent = $(*this*).data('post\_contain').toLowerCase();

            var postTags = $(*this*).data('post\_tags'); *// Array of tags*

            var postCategories = $(*this*).data('post\_categories'); *// Array of categories*

*// Check if either post title or content contains the search text*

            var titleMatch = postTitle.includes(searchText);

            var contentMatch = postContent.includes(searchText);

*// Check if any of the post's tags match the selected ones*

            var tagsMatch = my\_tags\_ids.length === 0 || postTags.some(*tag* => my\_tags\_ids.includes(tag.toString()));

*// Check if any of the post's categories match the selected ones*

            var categoriesMatch = my\_Categ\_ids.length === 0 || postCategories.some(*category* => my\_Categ\_ids.includes(category.toString()));

            if ((searchText === '' || titleMatch || contentMatch) && tagsMatch && categoriesMatch) {

                $(*this*).show(); *// Display the post if it matches*

            } else {

                $(*this*).hide(); *// Hide the post if it doesn't match*

            }

        });

    }

*// Handle input in the search field*

    $('#Search\_input').on('input', function() {

        filterPosts();

    });

*// Handle changes in category checkboxes*

    $(document).on('change', 'input[name="categories"]', function() {

        let categoryId = $(*this*).data("categories").toString();

        if ($(*this*).prop('checked')) {

*// Checkbox is checked*

            if (!my\_Categ\_ids.includes(categoryId)) {

                my\_Categ\_ids.push(categoryId);

            }

        } else {

*// Checkbox is unchecked*

            let index = my\_Categ\_ids.indexOf(categoryId);

            if (index !== -1) {

                my\_Categ\_ids.splice(index, 1);

            }

        }

        filterPosts();

    });

*// Handle changes in tag checkboxes*

    $(document).on('change', 'input[name="tags"]', function() {

        let tagId = $(*this*).data("tags").toString();

        if ($(*this*).prop('checked')) {

*// Checkbox is checked*

            if (!my\_tags\_ids.includes(tagId)) {

                my\_tags\_ids.push(tagId);

            }

        } else {

*// Checkbox is unchecked*

            let index = my\_tags\_ids.indexOf(tagId);

            if (index !== -1) {

                my\_tags\_ids.splice(index, 1);

            }

        }

        filterPosts();

    });

*// Initial filter application*

    filterPosts();

});

As the html code that apply on filter in client html page snippet is

    {% for post in posts %}

    <div class="post-container"  data-post\_tags='[{% for tag in post.post\_tags %}"{{ tag.id }}"{% if not forloop.last %},{% endif %}{% endfor %}]'  data-post\_categories='[{% for category in post.post\_categories %}"{{ category.id }}"{% if not forloop.last %},{% endif %}{% endfor %}]' data-post\_title="{{ post.title }}" data-post\_contain="{{ post.content }}" id="{{ post.id }}">

      <img src="{{ post.author.profile\_picture.url }}" class="profile-picture">

      <div class="post-content">

        <h4>{{ post.title }}</h4>

        <p>{{ post.content }}</p>

          <li style="display: inline-block;">

            {% if request.user.is\_authenticated %}

            <a href="{% url 'blogApp:post\_detail' post\_id=post.id %}">View Post</a></li>

            {% endif %}

        {% if request.user.email == post.author.user.email  %}

      <li style=" margin-left: 10px;  display: inline-block;">

        <form method="POST" action="{% url 'blogApp:update\_post' post\_id=post.id %}">

          {% csrf\_token %}

          <button type="submit" style=" cursor: pointer;  text-decoration: underline; color: blue; background: none; border: none; padding: 0; font: inherit; cursor: pointer;">

            Update Post

        </button>

          <input type="hidden" name="post\_id" value="{{ post.id }}">

        </form>

      </li>

      <li style=" margin-left: 10px; display: inline-block;">

        <form method="POST" action="{% url 'blogApp:delete\_post' post\_id=post.id %}">

          {% csrf\_token %}

          <button type="submit" style=" cursor: pointer;  text-decoration: underline; color: blue;  background: none; border: none; padding: 0; font: inherit; cursor: pointer;">

            Delete Post

          </button>

          <input type="hidden" name="post\_id" value="{{ post.id }}">

        </form>

      </li>

      {% endif %}

      <h5>Created at: {{ post.created\_at }} </h5>

      </div>

    </div>

    {% endfor %}

**Now Moving To userauth App**

Models.py

*from* django.contrib.auth.models *import* AbstractUser

*from* django.db *import* models

class User(AbstractUser):

    email = models.EmailField(*unique*=True)

    username = models.CharField(*max\_length*=100)

    USERNAME\_FIELD = 'email'

    REQUIRED\_FIELDS = ['username']

    def \_\_str\_\_(*self*):

*return* self.username

class Profile(models.Model):

    user = models.OneToOneField(User, *on\_delete*=models.CASCADE, *related\_name*='profile')

    bio = models.CharField(*max\_length*=100)

    profile\_picture = models.ImageField(*upload\_to*='profile\_pictures/', *blank*=True, *null*=True)

    def \_\_str\_\_(*self*):

*return* f'Profile of {self.user.username}'

admin.py

*from* django.contrib *import* admin

*from* userauths.models *import* User,Profile

*from* django.utils.html *import* format\_html

*# Register your models here.*

class UserAdmin(admin.ModelAdmin):

    list\_display = ["username","email"]

admin.site.register(User,UserAdmin)

class UserProfileAdmin(admin.ModelAdmin):

    list\_display = ["display\_profile\_picture","user", "bio"]

    def display\_profile\_picture(*self*, *obj*):

*if* obj.profile\_picture:

*return* format\_html('<img src="{}" style="max-height: 80px; max-width: 80px; border-radius:50%; "/>', obj.profile\_picture.url)

*else*:

*return* '(No image)'

    display\_profile\_picture.short\_description = 'Profile Picture'

admin.site.register(Profile,UserProfileAdmin)

forms.py

*from* django *import* forms

*from* django.contrib.auth.forms *import* UserCreationForm

*from* .models *import* User, Profile

*from* django.contrib.auth *import* authenticate

class UserRegistrationForm(UserCreationForm):

    email = forms.EmailField()

    class Meta:

        model = User

        fields = ['email', 'username', 'password1', 'password2', ]

    def save(*self*, *commit*=True):

        user = super().save(*commit*=False)

        user.email = self.cleaned\_data['email']

        user.username = self.cleaned\_data['username']

*if* commit:

            user.save()

*return* user

    def clean\_email(*self*):

        email = self.cleaned\_data.get('email')

*if* User.objects.filter(*email*=email).exists():

*raise* forms.ValidationError('This email address is already in use.')

*return* email

class UserLoginForm(forms.Form):

    email = forms.EmailField()

    password = forms.CharField(*widget*=forms.PasswordInput)

    def clean(*self*):

        cleaned\_data = super().clean()

        email = cleaned\_data.get('email')

        password = cleaned\_data.get('password')

*if* email and password:

            user = authenticate(*email*=email, *password*=password)

*if* not user:

*raise* forms.ValidationError('Invalid email or password.')

class UserUpdateForm(forms.ModelForm):

    email = forms.EmailField()

    class Meta:

        model = User

        fields = ['username', 'email']

class ProfileUpdateForm(forms.ModelForm):

    class Meta:

        model = Profile

        fields = ['bio', 'profile\_picture']

        widgets = {

            'bio': forms.TextInput(*attrs*={'placeholder': 'Enter your bio'}),

        }

urls.py

*from* django.contrib *import* admin

*from* django.urls *import* path,include,re\_path

*from* django.conf *import* settings

*from* django.views.static *import* serve

*from* .views *import* register\_user,login\_user,logout\_user

app\_name = "userauths"

urlpatterns = [

path("register/",register\_user,*name*="register"),

path("login/",login\_user,*name*="login"),

path("logout\_user",logout\_user,*name*="logout\_user")

]

Views.py

*from* django.shortcuts *import* render, redirect

*from* django.contrib.auth *import* authenticate, login, logout

*from* django.contrib *import* messages

*from* django.contrib.auth.decorators *import* login\_required

*from* .forms *import* UserRegistrationForm, UserLoginForm, UserUpdateForm, ProfileUpdateForm

*from* userauths.models *import* Profile

*from* django.contrib.auth *import* login

*from* django.shortcuts *import* render, redirect

*from* django.contrib *import* messages

*from* .forms *import* UserRegistrationForm, ProfileUpdateForm

*from* .models *import* Profile

def register\_user(*request*):

*if* request.method == 'POST':

        user\_form = UserRegistrationForm(request.POST)

        profile\_form = ProfileUpdateForm(request.POST, request.FILES)

*if* user\_form.is\_valid() and profile\_form.is\_valid():

*# Save User object*

            user = user\_form.save()

*# Extract profile form data*

            bio = profile\_form.cleaned\_data.get('bio', None)

            profile\_picture = profile\_form.cleaned\_data.get('profile\_picture', None)

*# Create Profile object linked to the user*

            Profile.objects.create(*user*=user, *bio*=bio, *profile\_picture*=profile\_picture)

*# Log in the user after successful registration*

            login(request, user)

*# Redirect to home or profile page*

            messages.success(request, f'Account created for {user.username}! You are now logged in.')

*return* redirect('blogApp:home')  *# Adjust this to your profile view name or URL*

*else*:

*# Handle form validation errors*

            messages.error(request, 'Form validation error. Please check the form fields.')

*else*:

        user\_form = UserRegistrationForm()

        profile\_form = ProfileUpdateForm()

    context = {

        'user\_form': user\_form,

        'profile\_form': profile\_form,

    }

*return* render(request, 'register.html', context)

def login\_user(*request*):

*if* request.method == 'POST':

        form = UserLoginForm(request.POST)

*if* form.is\_valid():

            email = form.cleaned\_data.get('email')

            password = form.cleaned\_data.get('password')

            user = authenticate(request, *username*=email, *password*=password)

*if* user is not None:

                login(request, user)

                messages.success(request, f'Welcome back, {user.username}!')

*return* redirect('blogApp:home')  *# Replace 'home' with your desired redirect URL*

*else*:

                messages.error(request, 'Invalid email or password. Please try again.')

*else*:

        form = UserLoginForm()

*return* render(request, 'login.html', {'form': form})

@login\_required

def logout\_user(*request*):

    logout(request)

    messages.info(request, 'You have been logged out.')

*return* redirect('userauths:login')

@login\_required

def profile(*request*):

*if* request.method == 'POST':

        user\_form = UserUpdateForm(request.POST, *instance*=request.user)

        profile\_form = ProfileUpdateForm(request.POST, request.FILES, *instance*=request.user.profile)

*if* user\_form.is\_valid() and profile\_form.is\_valid():

            user\_form.save()

            profile\_form.save()

            messages.success(request, 'Your profile has been updated!')

*return* redirect('profile')

*else*:

        user\_form = UserUpdateForm(*instance*=request.user)

        profile\_form = ProfileUpdateForm(*instance*=request.user.profile)

    context = {

        'user\_form': user\_form,

        'profile\_form': profile\_form

    }

*return* render(request, 'profile.html', context)

**Conclusion**

This documentation provides a comprehensive guide for creating a blog application with advanced features using Django. The implementation covers models, user authentication, CRUD operations, API development, search and filtering, pagination, and admin interface customization. Follow the steps and code snippets to build and deploy your blog application successfully.