Exercise: Building a Blog API with Advanced DRF Concepts

Objective: Build a RESTful API for a blog application using Django Rest Framework, incorporating advanced concepts like customizing generic views, custom permissions, and serializer validation.

Instructions:
1. Set up a new Django project and create a new app called "blogapi".

2. Create the necessary models for the blog application, including a "Post" model with fields like "title", "content", "author", and "created_at". You can also add additional models like "Comment" or "Category" if desired.

3. Implement the necessary serializers for your models, ensuring that the relationships between models are properly serialized. Additionally, add validation logic to the serializers to enforce specific constraints, such as ensuring the title is unique or the content has a minimum length.

4. Create class-based views using DRF's generic class-based views for handling CRUD operations for the "Post" model. Customize the behavior of these views by overriding methods or using mixins. For example, override the `get_queryset()` method to filter the list of posts based on certain criteria, or override the `perform_create()` method to add additional logic when creating a new post.

Implement the following endpoints using the class-based views:
- GET /api/posts/ - Retrieve a list of all posts.
- POST /api/posts/ - Create a new post.
- GET /api/posts/{id}/ - Retrieve details of a specific post.
- PUT /api/posts/{id}/ - Update an existing post.
- DELETE /api/posts/{id}/ - Delete a post.

5. Implement custom permissions to restrict access to certain API endpoints. For example, only authenticated users should be able to update or delete their own posts. Implement a custom permission class to enforce this restriction.

6. Create additional endpoints to handle related actions, such as adding comments to posts or retrieving posts by a specific author. Customize the behavior of these endpoints using appropriate class-based views or mixins.

7. Implement JWT authentication to secure your API. Integrate JWT authentication for all API endpoints. Provide guidance on how to generate JWT tokens and include them in requests for authenticated endpoints.

8. Test your API using tools like Postman or cURL to ensure that all endpoints are working correctly and that authentication and permissions are enforced properly.

9. Bonus: Implement pagination to limit the number of posts returned in a single request and enable filtering or searching based on specific criteria, such as filtering posts by category or searching for posts containing certain keywords.

10. Bonus: Implement additional features like adding comments to posts or allowing users to like or dislike posts.

Sample Answer: Building a Blog API with Advanced DRF Concepts

1. Set up a new Django project and create a new app called "blogapi".

2. Create the necessary models for the blog application, including a "Post" model with fields like "title", "content", "author", and "created_at". You can also add additional models like "Comment" or "Category" if desired.

```
python
from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    title = models.CharField(max_length=100, unique=True)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title
```

3. Implement the necessary serializers for your models, ensuring that the relationships between models are properly serialized. Additionally, add validation logic to the serializers to enforce specific constraints.

```
python
from rest_framework import serializers
from .models import Post

class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = '__all__'
        extra_kwargs = {
            'title': {'validators': []},  # Remove default title
validators
            'content': {'min_length': 10},  # Set minimum content length
to 10 characters
        }

    def validate_title(self, value):
        # Custom validation logic for title uniqueness
        if Post.objects.filter(title=value).exists():
            raise serializers.ValidationError('Title must be unique.')
        return value
```

4. Create class-based views using DRF's generic class-based views for handling CRUD operations for the "Post" model. Customize the behavior of these views by overriding methods or using mixins.

```
python
from rest_framework import generics
from .models import Post
from .serializers import PostSerializer

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    def get_queryset(self):
        # Custom filtering logic for retrieving posts
        category = self.request.query_params.get('category')
        if category:
            return Post.objects.filter(category=category)
        return super().get_queryset()

class
PostRetrieveUpdateDestroyView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

5. Implement custom permissions to restrict access to certain API endpoints. For example, only authenticated users should be able to update or delete their own posts. Implement a custom permission class to enforce this restriction.

```
python
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.author == request.user
```

```
class
PostRetrieveUpdateDestroyView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [IsOwnerOrReadOnly]
```

6. Create additional endpoints to handle related actions, such as adding comments to posts or retrieving posts by a specific author. Customize the behavior of these endpoints using appropriate class-based views or mixins.

```
python
from rest_framework import generics, mixins
from .models import Post, Comment
from .serializers import PostSerializer, CommentSerializer

class CommentCreateView(generics.CreateAPIView):
    queryset = Comment.objects.all()
    serializer_class = CommentSerializer

class PostAuthorListView(generics.ListAPIView):
    serializer_class = PostSerializer

    def get_queryset(self):
        # Retrieve posts by a specific author
        author_id = self.kwargs['author_id']
        return Post.objects.filter(author_id=author_id)
```

7. Add authentication to your API using JWT (JSON Web Tokens). Implement token-based authentication for all API endpoints.

```
python
from rest_framework_simplejwt.views import TokenObtainPairView,
TokenRefreshView


urlpatterns = [
    # other URLs
    path('api/token/', TokenObtainPairView.as_view(), name='token-
obtain-pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token-
refresh'),
]
```

8. Test your API using tools like Postman or cURL to ensure that all endpoints are working correctly and that authentication and permissions are enforced properly.

9. Bonus: Implement pagination to limit the number of posts returned in a single request and enable filtering or searching based on specific criteria, such as filtering posts by category or searching for posts containing certain keywords.

```
python
from rest_framework.pagination import LimitOffsetPagination

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    pagination_class = LimitOffsetPagination
```

Remember, this is just a sample answer to guide you. Feel free to modify it as per your specific requirements or preferences.