Django Rest Beginners Notes

Django REST Framework (DRF)

Django REST Framework (DRF) is a powerful toolkit for building Web APIs in Django. It simplifies the process of creating RESTful APIs by offering a flexible and modular architecture.

Key Features:

- **Serialization:** `DRF helps in converting Django models to JSON or other content types and vice versa`.
- **Authentication**: DRF supports multiple authentication methods, including token-based authentication (like JWT).
- **View sets and Routers**: DRF provides View Sets to reduce repetitive code for CRUD operations and Routers to automatically generate URL routes.
- **Throttling, Permissions, and Filtering**: DRF offers tools to manage rate-limiting, access control, and filtering of query results.

Serialization

Serialization in Django REST Framework (DRF) refers to the process of **converting complex data types,** like Django models or querysets, into native *Python data types that can then be easily rendered into JSON, XML, or other content types*.

Key Components of Serialization:

1. **Serializers**: Define how the model data is translated to and from a format like JSON.
2. **Deserialization**: Converts JSON or input data into Python objects, allowing you to validate the data and save it back to the database.

DRF provides multiple ways to create serializers:

1. $ModelSerializer$

This type of serializer is used when you want to automatically generate fields for a Django model. It provides an easy way to create serializers without manually defining each field.

Basic Syntax:

```
from rest_framework import serializers
from .models import MyModel
class MyModelSerializer(serializers.ModelSerializer):
class Meta:
   model = MyModel
   fields = '__all__'  # or specify the fields you need as ['field1', 'field2']
```

- **fields = '__all__' includes all fields in the model.**
- You can also specify the fields you want, e.g., fields = ['name', 'description'].

2. Serializer (Manual Serializer)

The Serializer class gives you more control over how the data is validated, represented, and serialized. You define each field explicitly, making it more flexible but also requiring more work.

```
from rest_framework import serializers

class MyManualSerializer(serializers.Serializer):
      name = serializers.CharField(max_length=100)
      description = serializers.CharField()
      price = serializers.DecimalField(max_digits=10, decimal_places=2)

     def create(self, validated_data):# Create a new instance of the model

         return MyModel.objects.create(**validated_data)

     def update(self, instance, validated_data):   # Update an existing instance

         instance.name = validated_data.get('name', instance.name)
         instance.description = validated_data.get('description', instance.descr
iption)
         instance.price = validated_data.get('price', instance.price)
         instance.save()

         return instance
```

This is useful when you have more complex validation or custom data transformations that aren't easily handled by ModelSerializer.

3. HyperlinkedModelSerializer

The HyperlinkedModelSerializer is similar to the ModelSerializer, but instead of using a primary key to represent relationships, it uses hyperlinks (URLs). This is helpful when you are working with RESTful APIs where resources are often identified via URLs.

Basic Syntax:

```
from rest_framework import serializers
from .models import MyModel
```

```
class MyModelHyperlinkedSerializer(serializers.HyperlinkedModelSerializer):

    class Meta:
      model = MyModel
      fields = ['url', 'name', 'description']
```

- The url field is automatically included by DRF to provide hyperlinks for the object’s details.

4. ListSerializer

The ListSerializer is useful for serializing multiple objects at once. By default, DRF handles lists of objects using ListSerializer, but you can customize it when you want to perform bulk updates or other batch operations.

Basic Syntax:

```
from rest_framework import serializers

   class MyChildSerializer(serializers.Serializer):
        name = serializers.CharField()

   class MyListSerializer(serializers.ListSerializer):

        child = MyChildSerializer()

   def update(self, instance, validated_data):  # Handle updating multiple objec
ts in a list
```

5. Custom Fields in Serializers

You can also define custom fields or methods in your serializers to customize the output or input.

Example of a custom field:

```
class MyModelSerializer(serializers.ModelSerializer):

    custom_field = serializers.SerializerMethodField()

 class Meta:
   model = MyModel
   fields = ['name', 'description', 'custom_field']

  def get_custom_field(self, obj):
      return f"Custom data for {obj.name}"
```

Key Fields in Serializers:

- **CharField, IntegerField, DateField, BooleanField, FloatField, DecimalField are some of the common fields used.**
- **SerializerMethodField: A read-only field that gets its value by calling a method on the serializer class.**

Common Serializer Fields

```
1. CharField
   # Description: A field for character input.

2. **IntegerField**
    - **Description**: A field for integer input.

3. **BooleanField**
    - **Description**: A field that accepts a boolean value (True or False).

4. **EmailField**
    - **Description**: A field for email addresses.

5. **DateField**
    - **Description**: A field for date input.

6. **DecimalField**
    - **Description**: A field for decimal numbers.

7. **FloatField**
    - **Description**: A field for floating-point numbers.

8. **ListField**
    - **Description**: A field that accepts a list of items.

9. **URLField**
    - **Description**: A field for URLs.
```

Core Arguments for Serializer Fields

1. **required**: Boolean value indicating whether this field is required. If set to False, the field can be omitted.
2. **allow_blank**: Applicable for fields like CharField. When set to True, the field can be blank even if it is required.
3. **default**: Sets a default value for the field if no value is provided.
4. **validators**: A list of custom validation functions that can be applied to the field data.
5. **help_text**: A string that provides additional information about the field. This can be used for API documentation.
6. **error_messages**: A dictionary that allows customization of error messages for various validation errors.

7. **source**: Specifies the attribute from the object that should be serialized. This is useful when the field name in the serializer differs from the model field name.

Deserialization

Deserialization in Django REST Framework (DRF) is the process of converting incoming data (usually in JSON format) into complex types such as Django model instances or Python objects. This process includes validating the data to ensure it meets the required criteria before it can be saved or processed.

Key Steps in Deserialization

1. **Receive Input Data:** Typically, this data is received in a request body (for example, from a POST or PUT request).
2. **Create a Serializer Instance**: A serializer instance is created using the incoming data.
3. **Validate the Data**: The serializer validates the data against the defined rules (such as required fields, field types, etc.).
4. **Save the Data:** If the data is valid, you can save it to the database or perform any other necessary operations.

Basic Syntax for Deserialization

Here's a step-by-step example of how to deserialize data using DRF:

Step 1: Define Your Model

```
# models.py

from django.db import models

class User(models.Model):
    username = models.CharField(max_length=150)
    email = models.EmailField()
    age = models.IntegerField(null=True, blank=True)
```

Step 2: Create a Serializer

```
# serializers.py
from rest_framework import serializers
from .models import User
class UserSerializer(serializers.ModelSerializer):
   class Meta:
      model = User
      fields = ['username', 'email', 'age']
```

Step 3: Handle Incoming Data in Your View

Here's how to deserialize data in a view:

```
# views.py

from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView
from .models import User
from .serializers import UserSerializer

   class UserCreateView(APIView):

      def post(self, request):
         serializer = UserSerializer(data=request.data)    # Deserialize incomin
g data
         if serializer.is_valid():                         # Validate the data
             user = serializer.save()                      # Save the validated
data
             return Response(UserSerializer(user).data, status=status.HTTP_201_C
REATED)

   return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Key Methods for Deserialization

- **is_valid()**: This method is called on the serializer instance to validate the data. It returns True if the data is valid and False otherwise. You can access validation errors using serializer.errors.
- **save()**: If the data is valid, calling this method will create or update the model instance based on the validated data. It can also accept additional keyword arguments to customize the save operation.
- **create() and update()**: If you override these methods in your serializer, you can customize how instances are created or updated during the deserialization process.

Handling Validation Errors

When deserializing data, it's important to handle validation errors properly. DRF provides structured error messages which you can return in your response. For example:

```
if serializer.is_valid():
   # Save the data
else:

return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Validation

Validation in Django Rest Framework (DRF) is an essential feature that ensures the integrity and correctness of data being processed in your API. It can occur during serialization (when converting data to and from complex types) and helps prevent invalid data from being saved to the database.

Types of Validation in DRF

1. **Field-Level Validation:** This is validation that is specific to individual fields in your serializer. You can define custom validation for fields using methods that follow the pattern validate_<field_name>.

2. **Object-Level Validation:** This involves validating the entire object or the data as a whole. You can implement this by overriding the validate method in your serializer.
3. **Built-in Validators**: DRF provides several built-in validators for common scenarios, like checking uniqueness, required fields, or value ranges.

Examples of Validation

1. Field-Level Validation

You can create custom validation for specific fields in your serializer.

Example:

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):

     class Meta:
         model = Book
         fields = ['title', 'author', 'published_date']


     def validate_title(self, value):
           if len(value) < 5:
              raise serializers.ValidationError("Title must be at least 5 charac
ters long.")
     return value


    def validate_author(self, value):
         if 'invalid' in value:
            raise serializers.ValidationError("Author name cannot contain 'inval
id'.")
    return value
```

2. Object-Level Validation

You can validate the entire object by overriding the validate method. This is useful for validating relationships between fields or cross-field dependencies.

Example:

```
class BookSerializer(serializers.ModelSerializer):

  class Meta:
    model = Book
    fields = ['title', 'author', 'published_date']


  def validate(self, attrs):
     if attrs['published_date'] > datetime.date.today():
           raise serializers.ValidationError("Published date cannot be in the fu
ture.")
  return attrs
```

3. Using Built-in Validators

DRF provides several built-in validators that you can use directly in your serializer fields.

Example:

```
from rest_framework import serializers
from django.core.validators import MinLengthValidator

class BookSerializer(serializers.ModelSerializer):
    title = serializers.CharField(validators=[MinLengthValidator(5)])  # Min len
gth of 5

class Meta:
  model = Book
  fields = ['title', 'author', 'published_date']
```

Example of Full Serializer with Validation

Here's how you might structure a full serializer that includes both field-level and object-level validation:

```
from rest_framework import serializers
from .models import Book
import datetime

class BookSerializer(serializers.ModelSerializer):

   class Meta:
    model = Book
    fields = ['title', 'author', 'published_date']


def validate_title(self, value):
     if len(value) < 5:
        raise serializers.ValidationError("Title must be at least 5 characters l
ong.")
return value


def validate(self, attrs):
   if attrs['published_date'] > datetime.date.today():
      raise serializers.ValidationError("Published date cannot be in the futur
e.")
return attrs
```

Accessing Validation Errors

If the data does not pass validation, you can access the errors through the errors attribute of the serializer.

Example:

```
serializer = BookSerializer(data=request.data)
if serializer.is_valid():
   serializer.save()
else:
  return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Views In DRF

In Django REST Framework (DRF), views are responsible for handling HTTP requests and returning responses. DRF provides two main types of views for handling HTTP requests: APIView and ViewSet.

API view is the core mechanism used to **handle HTTP requests and return HTTP responses, typically in the form of JSON data.** There are two primary ways to create API views in DRF:

1. **Function-Based Views (FBVs):** These are simple Python functions that handle HTTP requests.
2. **Class-Based Views (CBVs):** These are Python classes that handle HTTP requests. DRF provides several types of class-based views, such as APIView and ViewSet.

1. Function-Based Views (FBV)

FBVs are simple Python functions that receive an HTTP request and return an HTTP response. They are straightforward and easy to understand, especially for simple views.

Basic Syntax:

Here's a basic example of using FBVs in DRF to create an API for listing and creating providers:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Provider
from .serializers import ProviderSerializer

@api_view(['GET', 'POST'])

def provider_list(request):
   if request.method == 'GET':
       providers = Provider.objects.all()
       serializer = ProviderSerializer(providers, many=True)
       return Response(serializer.data)

  elif request.method == 'POST':
      serializer = ProviderSerializer(data=request.data)
      if serializer.is_valid():
         serializer.save()
         return Response(serializer.data, status=status.HTTP_201_CREATED)
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Explanation:

- **@api_view(['GET', 'POST']): This decorator tells DRF that this view can handle GET and POST requests.**
- **The function checks the request method and processes accordingly.**
- **If it's a GET request, it retrieves all providers and serializes them.**
- **If it's a POST request, it validates the incoming data and saves a new provider.**

2. Class-Based Views (CBV)

CBVs provide a more organized and reusable way to create views. You can define a class that inherits from DRF's built-in view classes like APIView, and override methods for different HTTP actions.

Basic Syntax:

Here's a basic example of using CBVs with APIView:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Provider
from .serializers import ProviderSerializer


class ProviderList(APIView):

   def get(self, request):
       providers = Provider.objects.all()
       serializer = ProviderSerializer(providers, many=True)
   return Response(serializer.data)

def post(self, request):
    serializer = ProviderSerializer(data=request.data)
    if serializer.is_valid():
       serializer.save()
       return Response(serializer.data, status=status.HTTP_201_CREATED)
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Explanation:

- **The class ProviderList inherits from APIView.**
- **The get method handles GET requests, retrieving and serializing the providers.**
- **The post method handles POST requests, validating the data and saving a new provider if valid.**

Choosing Between FBV and CBV

- **FBVs are simpler and better for small applications or views with minimal logic. They are also easier to understand at a glance.**

- **CBVs are more suitable for larger applications or views that require more complex logic, as they promote code reuse and organization through inheritance.**

Example URL Configuration

Regardless of whether you choose FBVs or CBVs, you need to include the view in your URL configuration.

```
from django.urls import path
from .views import provider_list  # For FBV

# from .views import ProviderList  # For CBV

urlpatterns = [

# For FBV

path('providers/', provider_list, name='provider-list'),

# For CBV

 path('providers/', ProviderList.as_view(), name='provider-list'),

]
```

Generic API Views

Django REST Framework (DRF) provides generic views to streamline the creation of common API endpoints by handling repetitive tasks like serialization, querying, and response formatting automatically. Generic views can significantly reduce the amount of code you need to write for CRUD operations.

Generic API Views Overview

DRF offers several built-in generic views, including:

- **GenericAPIView**: The base class for all generic views, which provides basic functionality.
- **ListAPIView**: For read-only endpoints that return a list of objects.
- **RetrieveAPIView**: For read-only endpoints that return a single object.
- **CreateAPIView**: For creating a new object.
- **UpdateAPIView**: For updating an existing object.
- **DestroyAPIView**: For deleting an object.
- **ModelViewSet**: Combines all of the above to provide a full set of CRUD operations.

Basic Syntax

Using a generic view typically involves the following steps:

1. **Define a class that inherits from a DRF generic view (e.g., ListAPIView, CreateAPIView, etc.).**
2. **Specify the queryset and serializer_class attributes.**
3. **Optionally override methods for custom behavior.**

Example of Using Generic API Views

Step 1: Define the Model and Serializer

Assuming you have a Provider model, here's how to set up the corresponding serializer:

```
# models.py

from django.db import models

class Provider(models.Model):
   name = models.CharField(max_length=100)
   age = models.IntegerField()
   email = models.EmailField()

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

```


# serializers.py

from rest_framework import serializers
from .models import Provider
class ProviderSerializer(serializers.ModelSerializer):

class Meta:
  model = Provider
  fields = '__all__'  # Or specify fields: ['id', 'name', 'age', 'email']
```

Step 2: Create Generic API Views

You can create API views using DRF's generic views:

```
# views.py

from rest_framework.generics import ListCreateAPIView, RetrieveUpdateDestroyAPIV
iew
from .models import Provider
from .serializers import ProviderSerializer

# View for listing and creating providers

class ProviderListCreateView(ListCreateAPIView):
   queryset = Provider.objects.all()
   serializer_class = ProviderSerializer

# View for retrieving, updating, and deleting a specific provider

class ProviderRetrieveUpdateDestroyView(RetrieveUpdateDestroyAPIView):
    queryset = Provider.objects.all()
    serializer_class = ProviderSerializer
```

Step 3: Define URL Patterns

Next, set up the URL patterns to route requests to these views:

```
# urls.py


from django.urls import path
from .views import ProviderListCreateView, ProviderRetrieveUpdateDestroyView

urlpatterns = [
path('providers/', ProviderListCreateView.as_view(), name='provider-list-creat
e'),
path('providers/<int:pk>/', ProviderRetrieveUpdateDestroyView.as_view(), name='p
rovider-detail'),

]
```

Explanation

1. ListCreateAPIView: **This view handles both GET (to list all providers) and POST (to create a new provider) requests. You only need to define the queryset and serializer_class, and DRF will take care of the rest.**
2. RetrieveUpdateDestroyAPIView: **This view handles GET, PUT, PATCH, and DELETE requests for a single provider specified by its primary key (pk). It also requires just the queryset and serializer_class.**

Advantages of Using Generic API Views

- Reduced Code: You avoid writing boilerplate code for standard operations, making your codebase cleaner and easier to maintain.
- Consistency: Generic views ensure consistent behavior across your API endpoints.
- Flexibility: You can still override methods for custom behavior while benefiting from the built-in functionality.

Generic API views are a powerful way to quickly implement common patterns for API endpoints such as listing, creating, updating, and deleting objects. These views come with several attributes that you can configure or override to customize their behavior.

Here's a brief explanation of the key attributes of **GenericAPIView**, which is the base class for all generic views in DRF.

Key Attributes of GenericAPIView

queryset

- Purpose: The queryset attribute defines the set of objects that the view will operate on. **This is used to retrieve data from the database.**

```
queryset = Provider.objects.all()
```

serializer_class

- **Purpose**: This defines the serializer that will be used to convert the data to and from Python types and JSON, as well as to validate incoming data.

```
serializer_class = ProviderSerializer
```

lookup_field

- **Purpose**: Defines the field that will be used to look up an object by its identifier. By default, this is pk (primary key), but you can customize it if needed (e.g., using a slug or another field).

```
lookup_field = 'id'
```

lookup_url_kwarg

- **Purpose**: **Defines the URL keyword argument that will be used to look up the object. If lookup_url_kwarg is not set, it defaults to the value of lookup_field.**

```
lookup_url_kwarg = 'provider_id'
```

1. **filter_backends**
    - **Purpose**: This is a list of filter backends used to filter the queryset. For example, you can use it to apply ordering, search, or custom filtering.

    ```
    from rest_framework import filters
       filter_backends = [filters.OrderingFilter, filters.SearchFilter]
       ordering_fields = ['name', 'age'] # Fields that can be used for ordering
       search_fields = ['name', 'email'] # Fields that can be searched
    ```

pagination_class

- **Purpose: Defines the pagination class used for paginating the queryset. You can set this to a custom pagination class or use DRF's built-in paginators.**

```

from rest_framework.pagination import PageNumberPagination
pagination_class = PageNumberPagination
```

-

permission_classes

- **Purpose**: **Defines the permission classes that control who can access the view. This is a list of permission classes (e.g., IsAuthenticated, AllowAny, etc.).**

```
from rest_framework.permissions import IsAuthenticated
permission_classes = [IsAuthenticated]
```

-

authentication_classes

- **Purpose**: **This attribute defines the type of authentication applied to the view. By default, DRF includes session and token authentication, but you can customize it as needed.**

```
from rest_framework.authentication import TokenAuthentication
authentication_classes = [TokenAuthentication]
```

-

throttle_classes

- **Purpose**: **This is a list of throttle classes that apply throttling to limit the rate of requests to your API.**

```
from rest_framework.throttling import UserRateThrottle
throttle_classes = [UserRateThrottle]
```

-

get_queryset()

- **Purpose**: While the queryset attribute defines a static queryset, **you can override get_queryset() to dynamically generate a queryset,** based on request parameters or custom logic.

```
def get_queryset(self):
    return Provider.objects.filter(is_active=True)
```

-

get_serializer_class()

- **Purpose**: **Allows you to dynamically choose a different serializer class, based on certain conditions like the request method or user type.**

```
def get_serializer_class(self):
    if self.request.method == 'GET':
           return ProviderDetailSerializer
 return ProviderSerializer
```

-

get_object()

- **Purpose**: This method is used to **retrieve a specific object, and you can override it if you need custom lookup behavior beyond the default lookup_field.**

```
def get_object(self):
   obj = get_object_or_404(Provider, id=self.kwargs['provider_id'])
return obj
```

-

get_serializer_context()

- **Purpose**: **This method allows you to pass additional context data to the serializer, such as the current user or other request-specific information.**

```
def get_serializer_context(self):
  return {'request': self.request, 'view': self}
```

Mixins

In Django REST Framework (DRF), **mixins are reusable blocks of behavior that allow you to add specific functionality to views, such as creating, updating, deleting, or listing objects. Mixins help you build custom views without having to inherit from the more complex generic views, providing fine-grained control over your API endpoints.**

Common DRF Mixins

DRF provides several useful mixins for handling standard operations. Here are the most commonly used ones:

1. **CreateModelMixin**: **Provides the ability to create a new object (handles HTTP POST requests)**.
2. **ListModelMixin**: Provides the ability to list a queryset of objects (handles HTTP GET requests).
3. **RetrieveModelMixin**: Provides the ability to retrieve a single object (handles HTTP GET requests with an object id).
4. **UpdateModelMixin**: Provides the ability to update an existing object (handles HTTP PUT and PATCH requests).
5. **DestroyModelMixin**: Provides the ability to delete an object (handles HTTP DELETE requests).

Syntax for Using Mixins

To use mixins, you typically combine them with **GenericAPIView** (or another base class). The mixins provide specific methods (create(), list(), etc.), while GenericAPIView handles the underlying request and response mechanisms.

Example of Combining Mixins with GenericAPIView

Let's break down the implementation of mixins with a concrete example.

Step 1: Define the Model and Serializer

```
# models.py

from django.db import models
class Provider(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    email = models.EmailField()

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

```
#serializers.py

from rest_framework import serializers
from .models import Provider
```

```
class ProviderSerializer(serializers.ModelSerializer):


class Meta:
 model = Provider
 fields = '__all__'
```

Step 2: Using Mixins in the View

We'll implement a basic CRUD interface for the Provider model using mixins.

```
# views.py

from rest_framework.generics import GenericAPIView
from rest_framework.mixins import ListModelMixin, CreateModelMixin, RetrieveMode
lMixin, UpdateModelMixin, DestroyModelMixin
from .models import Provider
from .serializers import ProviderSerializer

# List and Create View (GET, POST)

class ProviderListCreateView(GenericAPIView, ListModelMixin, CreateModelMixin):
    queryset = Provider.objects.all()
    serializer_class = ProviderSerializer

# For listing all providers (GET request)

def get(self, request, *args, **kwargs):
  return self.list(request, *args, **kwargs)

# For creating a new provider (POST request)

def post(self, request, *args, **kwargs):
  return self.create(request, *args, **kwargs)

# Retrieve, Update, and Destroy View (GET, PUT/PATCH, DELETE)

class ProviderDetailView(GenericAPIView, RetrieveModelMixin, UpdateModelMixin, D
estroyModelMixin):
    queryset = Provider.objects.all()
    serializer_class = ProviderSerializer

# Retrieve a single provider (GET request)

def get(self, request, *args, **kwargs):
 return self.retrieve(request, *args, **kwargs)

# Update a provider (PUT or PATCH request)
```

```
def put(self, request, *args, **kwargs):
 return self.update(request, *args, **kwargs)

def patch(self, request, *args, **kwargs):
 return self.partial_update(request, *args, **kwargs)

# Delete a provider (DELETE request)

def delete(self, request, *args, **kwargs):
  return self.destroy(request, *args, **kwargs)
```

Step 3: URL Configuration

```
# urls.py

from django.urls import path
from .views import ProviderListCreateView, ProviderDetailView

urlpatterns = [

path('providers/', ProviderListCreateView.as_view(), name='provider-list-creat
e'),

path('providers/<int:pk>/', ProviderDetailView.as_view(), name='provider-detai
l'),

]
```

Explanation of Each Mixin

1. **ListModelMixin**
    - **Used in combination with GenericAPIView to provide the ability to list multiple objects (via a GET request).**
        - Provides a list() method, which you call inside the get() method of your view.

    ```
    return self.list(request, *args, **kwargs)
    ```

2. **CreateModelMixin**
    - **Used to add the ability to create a new object (via a POST request).**
    - **Provides a create() method, which you call inside the post() method of your view.**

    ```
    return self.create(request, *args, **kwargs)
    ```

3. **RetrieveModelMixin**
    - Provides a retrieve() method, which you call inside the get() method of your view.

    - Used to **retrieve** a single object by its primary key (via a GET request).

```
return self.retrieve(request, *args, **kwargs)
```

4. **UpdateModelMixin**
   - Used to **update** an object (via PUT or PATCH requests).
   - Provides two methods: **update() for full updates (PUT) and partial_update() for partial updates (PATCH).**

```
return self.update(request, *args, **kwargs) # Full update
return self.partial_update(request, *args, **kwargs) # Partial update
```

5. **DestroyModelMixin**
   - Used to **delete** an object (via a DELETE request).
   - Provides a **destroy() method, which you call inside the delete() method of your view.**

```
return self.destroy(request, *args, **kwargs)
```

Advantages of Using Mixins

- **Modularity**: **You can mix and match behaviors as needed by combining multiple mixins in a single view.**
- **Simplicity**: By using mixins, you avoid duplicating logic for common tasks like listing, retrieving, updating, or deleting objects.
- **Customization**: Mixins give you the flexibility to override individual methods and add custom behavior while keeping your views concise.

Concrete View Classes

In Django REST Framework (DRF), **concrete view classes are predefined views that handle common actions like listing, creating, retrieving, updating, or deleting objects. These classes combine generic views with the necessary mixins, offering a simple, high-level way to implement API endpoints without writing the same logic repeatedly.**

DRF provides the following concrete view classes, which cover basic CRUD operations:

List of Concrete View Classes

1. **ListAPIView**: **For listing objects (GET request).**
2. **CreateAPIView:** For creating new objects (POST request).
3. **RetrieveAPIView**: For retrieving a single object (GET request with an object id).
4. **UpdateAPIView**: **For updating an existing object (PUT or PATCH request).**
5. **DestroyAPIView: For deleting an object (DELETE request).**
6. **ListCreateAPIView: For listing objects and creating new ones (GET and POST requests).**
7. **RetrieveUpdateAPIView: For retrieving and updating a single object (GET, PUT, PATCH requests).**
8. **RetrieveDestroyAPIView: For retrieving and deleting a single object (GET, DELETE requests).**

9. **RetrieveUpdateDestroyAPIView: For retrieving, updating, and deleting a single object (GET, PUT, PATCH, DELETE requests).**

These concrete classes are built on top of GenericAPIView and the appropriate mixins, meaning they simplify CRUD operations significantly.

Basic Syntax and Usage of Concrete View Classes

Let's take a look at how you can use these concrete view classes with an example.

Step 1: Define a Model and Serializer

```
# models.py

from django.db import models
class Task(models.Model):
  title = models.CharField(max_length=100)
  description = models.TextField()
  completed = models.BooleanField(default=False)

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

```
# serializers.py

rom rest_framework import serializers
from .models import Task

class TaskSerializer(serializers.ModelSerializer):

  class Meta:
   model = Task
   fields = '__all__'
```

f

Step 2: Using Concrete View Classes in Views

1. **ListAPIView: To list all tasks.**

   -
     ```
     # views.py

     from rest_framework.generics import ListAPIView
     from .models import Task
     from .serializers import TaskSerializer

       class TaskListView(ListAPIView):
         queryset = Task.objects.all()
         serializer_class = TaskSerializer
     ```

2. **CreateAPIView: To create a new task.**

-

-
  ```
  # views.py

  from rest_framework.generics import CreateAPIView
  from .models import Task
  from .serializers import TaskSerializer

   class TaskCreateView(CreateAPIView):
      queryset = Task.objects.all()
      serializer_class = TaskSerializer
  ```

  - **3.** RetrieveAPIView: To retrieve a specific task by its ID.

  ```
  # views.py

  from rest_framework.generics import RetrieveAPIView
  from .models import Task
  from .serializers import TaskSerializer
  class TaskDetailView(RetrieveAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
  ```

  -
  - **4.UpdateAPIView: To update an existing task.**

  ```
  # views.py

   from rest_framework.generics import UpdateAPIView
   from .models import Task
   from .serializers import TaskSerializer
   class TaskUpdateView(UpdateAPIView):
     queryset = Task.objects.all()
     serializer_class = TaskSerializer
  ```

  - **5.DestroyAPIView:** To delete a task.

  ```
  # views.py

  from rest_framework.generics import DestroyAPIView
  from .models import Task
  from .serializers import TaskSerializer
  class TaskDeleteView(DestroyAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
  ```

  - **6.RetrieveUpdateDestroyAPIView: For a single view that handles retrieving, updating, and deleting a task.**

  ```
  # views.py

  from rest_framework.generics import RetrieveUpdateDestroyAPIView
  ```

```
from .models import Task
from .serializers import TaskSerializer
class TaskRetrieveUpdateDestroyView(RetrieveUpdateDestroyAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

Step 3: URL Configuration

```
# urls.py

from django.urls import path
from .views import TaskListView, TaskCreateView, TaskDetailView, TaskUpdateView,
TaskDeleteView, TaskRetrieveUpdateDestroyView

urlpatterns = [

path('tasks/', TaskListView.as_view(), name='task-list'),

path('tasks/create/', TaskCreateView.as_view(), name='task-create'),

path('tasks/<int:pk>/', TaskDetailView.as_view(), name='task-detail'),

path('tasks/<int:pk>/update/', TaskUpdateView.as_view(), name='task-update'),

path('tasks/<int:pk>/delete/', TaskDeleteView.as_view(), name='task-delete'),

path('tasks/<int:pk>/rud/', TaskRetrieveUpdateDestroyView.as_view(), name='task-
rud'),

]
```

ViewSets

In Django REST Framework (DRF), ViewSets are a higher-level abstraction over views that allow you to combine the logic for handling multiple types of requests (like list, create, retrieve, update, and delete) in a single class. Instead of defining individual views for each action (as with GenericAPIView), you can define a single ViewSet that handles the common CRUD actions in one place.

Types of ViewSets in DRF

1. **ViewSet: A basic viewset that you can use to define your own actions.**
2. **ModelViewSet: A viewset that provides default implementations for the standard actions (list, create, retrieve, update, partial_update, and destroy).**
3. **ReadOnlyModelViewSet: A viewset that provides only read-only actions (list, retrieve).**

Basic Example of Using ViewSets

Let's go through an example where we use ModelViewSet to handle CRUD operations for a Task model.

Step 1: Define the Model and Serializer

```
# models.py

from django.db import models
class Task(models.Model):
  title = models.CharField(max_length=100)
  description = models.TextField()
  completed = models.BooleanField(default=False)

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

```
# serializers.py

from rest_framework import serializers
from .models import Task
class TaskSerializer(serializers.ModelSerializer):

class Meta:
  model = Task
  fields = '__all__'
```

Step 2: Define the ViewSet

In this case, we will use ModelViewSet, which automatically provides the standard actions.

```
# views.py

from rest_framework import viewsets
from .models import Task
from .serializers import TaskSerializer
class TaskViewSet(viewsets.ModelViewSet):
   queryset = Task.objects.all()
   serializer_class = TaskSerializer
```

Step 3: URL Configuration with Routers

To automatically generate the routes for the ViewSet, we use DRF's routers. This will automatically create routes for all the actions (list, retrieve, create, update, delete).

```
# urls.py

from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet

router = DefaultRouter()

router.register(r'tasks', TaskViewSet, basename='task')
urlpatterns = [
path('', include(router.urls)),

]
```

How the URL Routing Works

Using the DefaultRouter, the following routes will automatic**ally be generated:**

- GET /tasks/ → **Lists all tasks (list)**
- POST /tasks/ → **Creates a new task (create)**
- GET /tasks/<id>/ → **Retrieves a specific task (retrieve)**
- PUT /tasks/<id>/ → **Updates a specific task (update)**
- PATCH /tasks/<id>/ → **Partially updates a specific task (partial_update)**
- DELETE /tasks/<id>/ → **Deletes a specific task (destroy)**

You don't need to manually specify each action in the urls.py file. The router handles this for you.

ViewSet Types Explained

1. **ViewSet:**
    - **A base class that provides the building blocks for creating custom actions. It doesn't provide any pre-built functionality for CRUD operations.**

```
class CustomViewSet(viewsets.ViewSet):
   def list(self, request):
   # Custom logic for listing objects
   pass

   def retrieve(self, request, pk=None):
    # Custom logic for retrieving an object
    pass
```

ModelViewSet:

- Extends ViewSet and automatically provides CRUD operations (list, create, retrieve, update, destroy).
- Ideal for quick CRUD endpoints based on a model.

ReadOnlyModelViewSet:

- **A read-only version of ModelViewSet that provides only the list and retrieve actions.**
- Useful for situations where you want to expose data for reading but not allow modifications.

```
class TaskReadOnlyViewSet(viewsets.ReadOnlyModelViewSet):
   queryset = Task.objects.all()
   serializer_class = TaskSerializer
```

Authentication

Django REST Framework (DRF) provides several authentication methods to authenticate users for API access. Authentication is the process of determining which user is making the request. In DRF, authentication is used to ensure that only authorized users can access certain endpoints or resources.

Common Authentication Methods in DRF

Session Authentication (default in Django)

Basic Authentication

Token Authentication

JWT (JSON Web Token) Authentication

Custom Authentication

Each method has its own use case depending on the type of API and how you want to manage security.

1. Session Authentication

This method uses Django's session framework to authenticate users. It is typically used for browser-based, HTML-rendered views and is the default authentication method in Django. This is most suitable for APIs consumed by web clients that are also using Django sessions.

Example:

A user logs into your Django website.

The API checks the session ID in the browser cookies for the user.

To enable session authentication in DRF:

```
# settings.py


REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework.authentication.SessionAuthentication',

]

}
```

This method is not ideal for APIs consumed by non-browser clients (e.g., mobile apps) because sessions rely on cookies.

2. Basic Authentication

This method uses a simple HTTP authentication scheme where the username and password are sent in the HTTP header (Base64 encoded). It is easy to set up but not secure, as credentials are sent in each request and need to be transmitted over HTTPS.

To enable Basic Authentication in DRF:

```
# settings.py

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework.authentication.BasicAuthentication',

]

}
```

This method is mainly useful for testing but not recommended for production since credentials are exposed unless you use HTTPS.

3. Token Authentication

In this method, each user is assigned a token that is used to authenticate API requests. The token is sent in the headers with each request instead of a username and password.

Example:

A user logs in and receives a token.

The user sends the token with every API request in the Authorization header.

curl -X GET "http://example.com/api/tasks/" -H "Authorization: Token <your-token>"

Steps to Implement Token Authentication:

```
Install the rest_framework.authtoken package:
pip install djangorestframework
pip install djangorestframework-authtoken
```

```
Add 'rest_framework.authtoken' to your INSTALLED_APPS.
```

Migrate the database to create the token table:

```
python manage.py migrate
```

Enable token authentication in the settings:

```
# settings.py

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework.authentication.TokenAuthentication',
]

}
```

Create a token for a user via Django shell:

```
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User

user = User.objects.get(username='your_username')

token = Token.objects.create(user=user)

print(token.key)
```

Now you can use this token in the Authorization header for all requests.

4. JWT (JSON Web Token) Authentication

JWT is a more modern authentication method that uses JSON Web Tokens to verify users. It is stateless (does not store session information on the server), making it ideal for scalable, distributed systems.

How JWT Works:

A user logs in and receives a JWT token.

The token is sent with each request in the Authorization header.

The server verifies the token for each request.

To implement JWT in DRF, we typically use the djangorestframework-simplejwt package.

Steps to Implement JWT Authentication:

Install the package:

```
pip install djangorestframework-simplejwt
```

Add JWT authentication to your settings:

```
# settings.py

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework_simplejwt.authentication.JWTAuthentication',

]
}
```

Define URLs for obtaining and refreshing tokens:

```
# urls.py

from rest_framework_simplejwt.views import
TokenObtainPairView,

TokenRefreshView,

)

urlpatterns = [

path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),

path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),

]
```

To obtain a token, send a POST request with your username and password:

**curl -X POST "http://example.com/api/token/" **

- **d '{"username": "myuser", "password": "mypassword"}' **
- **H "Content-Type: application/json"**

The response will contain an access token and a refresh token.

Use the token in the Authorization header for future requests:

curl -X GET "http://example.com/api/tasks/" \

- **H "Authorization: Bearer <your-access-token>"**

5. Custom Authentication

If you have specific authentication needs (e.g., custom token mechanisms), you can implement your own custom authentication by subclassing BaseAuthentication.

Example of a Custom Authentication Class:

```
# custom_auth.py

from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed
from django.contrib.auth.models import User

class CustomTokenAuthentication(BaseAuthentication):

  def authenticate(self, request):
    token = request.META.get('HTTP_AUTHORIZATION')
      if not token:
        return None

try:
    user = User.objects.get(auth_token=token)

except User.DoesNotExist:
   raise AuthenticationFailed('Invalid token')

return (user, None)
```

Add this custom authentication class to the settings:

```
# settings.py

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'path.to.custom_auth.CustomTokenAuthentication',

]

}
```

Now, your custom authentication method will be used to verify requests.

Combining Multiple Authentication Methods

You can enable multiple authentication methods by adding them to the DEFAULT_AUTHENTICATION_CLASSES setting. DRF will attempt each method in the order they are listed, and authentication will pass as soon as one of them succeeds.

```
# settings.py

REST_FRAMEWORK = {

'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework.authentication.SessionAuthentication',

'rest_framework.authentication.BasicAuthentication',

'rest_framework.authentication.TokenAuthentication',

]

}
```

Permissions

In Django REST Framework (DRF), permissions are used to grant or deny access to different parts of the API based on various conditions, such as user roles, authentication status, or custom logic. Permissions come into play after authentication has been successfully performed, ensuring that only certain users can access particular views or actions.

DRF provides several built-in permission classes and also allows for custom permission classes to fit specific requirements.

Types of Permissions in DRF

1. **AllowAny**
2. **IsAuthenticated**
3. **IsAdminUser**
4. **IsAuthenticatedOrReadOnly**
5. **DjangoModelPermissions**
6. **DjangoModelPermissionsOrAnonReadOnly**
7. **Custom Permissions**

1. AllowAny

This permission allows unrestricted access to the view or endpoint. Any user, whether authenticated or not, can access the resource.

Example:

```
from rest_framework.permissions import AllowAny
from rest_framework.views import APIView

class MyPublicView(APIView):
  permission_classes = [AllowAny]

def get(self, request):
  return Response({"message": "This is accessible by anyone!"})
```

Use AllowAny when you want to make certain views or endpoints publicly accessible.

2. IsAuthenticated

This permission restricts access to authenticated users only. If a user is not authenticated (i.e., they are anonymous), access is denied.

Example:

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView
```

```
class MyPrivateView(APIView):
  permission_classes = [IsAuthenticated]

def get(self, request):
  return Response({"message": "Hello, authenticated user!"})
```

If a user tries to access this view without being authenticated, they will receive a 401 Unauthorized response.

3. IsAdminUser

This permission restricts access to users who are staff members (i.e., users with is_staff=True). Only admin users can access the view.

Example:

```
from rest_framework.permissions import IsAdminUser
from rest_framework.views import APIView

class AdminOnlyView(APIView):
     permission_classes = [IsAdminUser]

def get(self, request):
 return Response({"message": "Welcome, admin!"})
```

If a non-admin user tries to access this view, they will receive a 403 Forbidden response.

4. IsAuthenticatedOrReadOnly

This **permission allows full access to authenticated users, but only read-only access (GET, HEAD, OPTIONS) for unauthenticated users.** This is useful for public APIs where users can read data without logging in, but must authenticate to make changes.

Example:

```
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from rest_framework.views import APIView

class ReadOrEditView(APIView):
  permission_classes = [IsAuthenticatedOrReadOnly]

def get(self, request):
  return Response({"message": "Anyone can read this!"})

def post(self, request):
 return Response({"message": "Only authenticated users can post this!"})
```

Unauthenticated users will be restricted from performing write operations (POST, PUT, DELETE, etc.).

5. DjangoModelPermissions

This permission ties into Django's model-level permissions. It requires the user to have the appropriate add, change, or delete permissions on the model to perform those actions. Users must be authenticated, and permissions are assigned based on their roles in the admin interface.

Example:

```
from rest_framework.permissions import DjangoModelPermissions
from rest_framework.viewsets import ModelViewSet
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelViewSet(ModelViewSet):
  queryset = MyModel.objects.all()
  serializer_class = MyModelSerializer
  permission_classes = [DjangoModelPermissions]
```

In this case, users need to have the corresponding model-level permissions (add_mymodel, change_mymodel, delete_mymodel) to perform each action.

6. DjangoModelPermissionsOrAnonReadOnly

This is a variation of DjangoModelPermissions. It grants read-only access to anonymous users while applying model-level permissions for authenticated users.

Example:

```
from rest_framework.permissions import DjangoModelPermissionsOrAnonReadOnly
from rest_framework.viewsets import ModelViewSet
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelViewSet(ModelViewSet):
  queryset = MyModel.objects.all()
  serializer_class = MyModelSerializer
  permission_classes = [DjangoModelPermissionsOrAnonReadOnly]
```

Unauthenticated users can view the data, but authenticated users need specific model permissions to modify it.

7. **Custom Permissions**

DRF allows for custom permissions when you need more complex or specific logic for your views. A custom permission class is a subclass of BasePermission and implements a has_permission or has_object_permission method.

Example of Custom Permission:

```
from rest_framework.permissions import BasePermission
class IsOwner(BasePermission):

"""

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

```
"""

def has_object_permission(self, request, view, obj):

# Only allow access if the user is the owner of the object

    return obj.owner == request.user

Use this custom permission in a view:

from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView
from .permissions import IsOwner

class OwnerOnlyView(APIView):
   permission_classes = [IsAuthenticated, IsOwner]

def get(self, request, pk):

  obj = MyModel.objects.get(pk=pk)
  self.check_object_permissions(request, obj)  # Important for object-level perm
ission check
return Response({"message": "You are the owner of this object!"})
```

This custom permission ensures that only the owner of the object can perform certain actions on it.

Applying Permissions to Views

1.Globally: You can set default permissions for the entire API in settings.py:

```
REST_FRAMEWORK = {
'DEFAULT_PERMISSION_CLASSES': [

'rest_framework.permissions.IsAuthenticated',

]

}
```

2.On a View Level: You can apply permissions to specific views by setting the permission_classes attribute:

```
from rest_framework.permissions import IsAuthenticatedOrReadOnly

class MyView(APIView):
  permission_classes = [IsAuthenticatedOrReadOnly]

def get(self, request):
    return Response({"message": "This is a read-only view for unauthenticated use
rs!"})
```

3.On a ViewSet Level: Similarly, you can apply permissions to viewsets:

```
from rest_framework.permissions import IsAdminUser
from rest_framework.viewsets import ModelViewSet
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelViewSet(ModelViewSet):
  queryset = MyModel.objects.all()
  serializer_class = MyModelSerializer
  permission_classes = [IsAdminUser]
```

Combining Permissions

You can combine multiple permission classes. DRF will evaluate each one, and the user must satisfy all of them to be granted access.

Example:

```
from rest_framework.permissions import IsAuthenticated, IsAdminUser

class AdminOnlyAuthenticatedView(APIView):
  permission_classes = [IsAuthenticated, IsAdminUser]

def get(self, request):
  return Response({"message": "You are an authenticated admin!"})

In this example, the user must both be authenticated and an admin to access the
view.
```

Pagination

Pagination in Django REST Framework (DRF) is a way to divide large sets of data into smaller, manageable parts, typically by displaying only a few records per page. Pagination allows APIs to return data in chunks rather than all at once, which improves performance and usability, especially for APIs that deal with large datasets.

Types of Pagination in DRF
DRF provides several built-in pagination styles:

PageNumberPagination: The default pagination style, which divides data into pages using a page number.

CursorPagination: This style is suitable for large datasets and provides an opaque cursor to navigate through the results.

LimitOffsetPagination: This style uses query parameters limit and offset to control pagination.

PageNumberPagination

This is the most common type of pagination. It allows you to specify the number of items per page and navigate using a page number.

Configuration:
In
settings.py, define the pagination class globally

```
settings.py
REST_FRAMEWORK = {
'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 10,  # Number of records per page
}
Example with View:
python
Copy code
from rest_framework.pagination import PageNumberPagination
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
pagination_class = PageNumberPagination  # Optional, use if overriding global pa
gination
```

To access paginated results, the client can request different pages by using the page query parameter:

```
GET /api/mymodels/?page=2
```

2. LimitOffsetPagination

This pagination style allows the client to specify the number of results to return (limit) and an offset to define where to start returning results.

Configuration:
In
settings.py, define the pagination class

```
settings.py
REST_FRAMEWORK = {
'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
'PAGE_SIZE': 10,  # Default number of records if no limit is provided
}
Example with View:
from rest_framework.pagination import LimitOffsetPagination
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer
```

```
class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
pagination_class = LimitOffsetPagination  # Optional, use if overriding global p
agination
```

To access paginated results, the client can request a specific limit

```
GET /api/mymodels/?limit=10&offset=20
```

This will return 10 items starting from the 21st item (0-indexed).

CursorPagination

Cursor pagination is ideal for large datasets where ordering by an index is required. Instead of page numbers or offsets, it uses a unique cursor (an opaque token) to mark the position of the next page.

Configuration:
In
settings.py, define the pagination class:

```
#settings.py
REST_FRAMEWORK = {
'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.CursorPagination',
'PAGE_SIZE': 10,
}
Example with View
from rest_framework.pagination import CursorPagination
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
pagination_class = CursorPagination  # Optional, use if overriding global pagina
tion
```

To access paginated results, the client uses a cursor parameter:

```
GET /api/mymodels/?cursor=abc123
```

The API provides a cursor in the response that can be used to retrieve the next set of results.

Custom Pagination
You can also create custom pagination classes by subclassing DRF's pagination classes.

Example of Custom Pagination:

```
from rest_framework.pagination import PageNumberPagination
class CustomPageNumberPagination(PageNumberPagination):
page_size = 5
page_size_query_param = 'page_size'  # Allow the client to specify the page size
max_page_size = 100  # Limit the maximum number of records per page
Use the custom pagination class in a view:
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer
from .pagination import CustomPageNumberPagination

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
pagination_class = CustomPageNumberPagination
```

How to Get Paginated Data
A typical response with pagination might look like this:

```
json

{
"count": 100,  # Total number of records
"next": "http://api.example.com/mymodels/?page=3",  # URL for the next page
"previous": "http://api.example.com/mymodels/?page=1",  # URL for the previous p
age
"results": [
{
"id": 21,
"name": "Item 21",
"description": "Description for item 21"
},
...
]
}
The next and previous fields provide links to the next and previous pages, and t
he results field contains the paginated data.
```

Filter Classes

In Django REST Framework (DRF), filtering allows you to restrict the data returned by your API based on certain conditions. DRF supports filtering using query parameters, and you can achieve this with the django-filter package or DRF's built-in FilterBackend.

Key Filtering Approaches

Types of Filter Classes

FilterBackends: DRF uses FilterBackends to apply filtering logic.

Django Filter: A powerful and flexible third-party package used to filter queryset data.

1. **Simple Filtering Using Query Parameters**
   You can directly filter querysets in your views using query_params passed in the URL. This is a basic form of filtering.

Example:

```
python

from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer
```

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer

```
def get_queryset(self):
    queryset = super().get_queryset()
    param = self.request.query_params.get('param', None)  # Get 'param' from que
ry params
    if param is not None:
        queryset = queryset.filter(field=param)  # Filter the queryset
    return queryset
```

Here, you can filter by adding a query parameter in the URL like:

```
http
Copy code
GET /api/mymodels/?param=value
2. Using django-filter for More Advanced Filtering
django-filter is a robust package used for more complex filtering in DRF. It all
ows you to define custom filter logic using a FilterSet class.
```

```
Installation:
bash
Copy code
pip install django-filter
Add to settings.py:
python
Copy code
```

```
#settings.py
```

```
REST_FRAMEWORK = {
'DEFAULT_FILTER_BACKENDS': ['django_filters.rest_framework.DjangoFilterBacken
d'],
}
```

Define a FilterSet:

You define filter logic using a FilterSet class from django_filters.

Example:

```
### import django_filters
from myapp.models import MyModel

class MyModelFilter(django_filters.FilterSet):
name = django_filters.CharFilter(field_name='name', lookup_expr='icontains')  #
Filter by name (case insensitive)
min_age = django_filters.NumberFilter(field_name='age', lookup_expr='gte')    #
Filter by minimum age
max_age = django_filters.NumberFilter(field_name='age', lookup_expr='lte')    #
Filter by maximum age

class Meta:
    model = MyModel
    fields = ['name', 'min_age', 'max_age']
```

Use the FilterSet in a View:

```
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer
from myapp.filters import MyModelFilter


class MyModelListView(ListAPIView):
  queryset = MyModel.objects.all()
  serializer_class = MyModelSerializer
  filter_backends = [DjangoFilterBackend]
  filterset_class = MyModelFilter  # Apply the custom filterset
  To filter results, the client can pass query parameters like:
```

```
GET /api/mymodels/?name=john&min_age=20&max_age=50
```

3. SearchFilter

SearchFilter allows you to search for specific terms in fields of the model.

Example:

4. OrderingFilter

OrderingFilter allows you to specify the order of the results based on certain fields.

```
### from rest_framework.filters import SearchFilter
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
filter_backends = [SearchFilter]
search_fields = ['name', 'description']  # Fields to search in
To perform a search, the client uses the search query parameter:

http

GET /api/mymodels/?search=john
```

Example:

```
## from rest_framework.filters import OrderingFilter
from rest_framework.generics import ListAPIView
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelListView(ListAPIView):
queryset = MyModel.objects.all()
serializer_class = MyModelSerializer
filter_backends = [OrderingFilter]
ordering_fields = ['name', 'age']  # Fields to order by
To order the results, the client can use the ordering query parameter:

http

GET /api/mymodels/?ordering=name  # Order by name (ascending)
GET /api/mymodels/?ordering=-age  # Order by age (descending)
```