****

[**1. Serializing, Listing, Filtering, and Paginating Models**](#_8xkynvoc2nfb) **2**

[Creating a Django Rest Framework Serializer to Serialize a Model](#_r6cgwu6bon91) 2

[Creating a ListAPIView Subclass](#_f0p3kv2atycr) 2

[Connecting an APIView to a Route](#_dw0c94bw1sbf) 2

[Filter Back Ends with URL Query Parameters](#_19mf5sktmjqh) 3

[Enabling Full-Text Search Filter Backend](#_x9gylbsiy6hp) 3

[Enabling Pagination of QuerySets in API Responses](#_ucwb0vefhf6k) 3

[**2. Create, Retrieve, Update, and Delete (CRUD) Operations for Models**](#_h0k9e947q60q) **4**

[Creating a CreateAPIView Subclass](#_6lgrkoqo2vb) 4

[Connecting a CreateAPIView to the Router](#_ji71kux2zqrc) 4

[Creating a DestroyAPIView Subclass](#_i5oy3a63pub) 5

[Connecting a DestroyAPIView to the Router](#_r3pil5qrd7ub) 5

[Creating an UpdateAPIView Subclass](#_yhozrwht0dkq) 5

[Connecting an UpdateAPIView to the Router](#_99dkp9y3bnty) 5

[**3. Managing Serializer Fields, Relations, and Validation**](#_qa31xavpt743) **6**

[Serializer with Only Selected Fields](#_j6ie6ppiob7v) 6

[Serializer That Shows Model Relationships](#_lzz3drkhynnb) 6

[Number Fields with Serializers](#_peco2q8n02s0) 6

[Date and Time Fields with Serializers](#_8l31cmhlt1z1) 7

[Lists, Dicts, and JSON Objects](#_sm9zwhave220) 7

[Serializer with ImageField and FileField](#_lmy5n47bu0om) 7

[**4. Testing API Views**](#_kksry4g4svs5) **7**

[Test Case for a CreateAPIView Subclass](#_6t9r06xrrx3q) 7

[Test Case for a DestroyAPIView Subclass](#_lb12p7mdj1u9) 8

[Test Case for a ListAPIView Subclass](#_ywu73xj8il72) 8

[Unit Test for an UpdateAPIView Subclass](#_60nk5781cv38) 8

[Handling Image Uploads in a Unit Test](#_zh45k9t0mnj0) 9

### **1. Serializing, Listing, Filtering, and Paginating Models**

#### **Creating a Django Rest Framework Serializer to Serialize a Model**

To create a serializer for a model, you need to define a class inheriting from serializers.ModelSerializer. This class will automatically generate fields based on the model's attributes. Here's an example:

| from rest\_framework import serializers from.models import YourModel  class YourModelSerializer(serializers.ModelSerializer):  class Meta:  model = YourModel  fields = '\_\_all\_\_' # Or specify individual fields |
| --- |

#### **Creating a ListAPIView Subclass**

ListAPIView is a generic view that displays a list of objects. You can extend it to customize its behavior:

| from rest\_framework.generics import ListAPIView from.models import YourModel from.serializers import YourModelSerializer  class YourModelListView(ListAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer |
| --- |

#### **Connecting an APIView to a Route**

Routes are defined in URLs.py using path converters and the as\_view() method of your view class. Example:

| from django.urls import path from.views import YourModelListView  urlpatterns = [  path('yourmodel/', YourModelListView.as\_view(), name='yourmodel-list'), ] |
| --- |

#### **Filter Back Ends with URL Query Parameters**

Django REST Framework provides several ways to filter querysets, including through filterset classes or direct filtering in views. Example using direct filtering:

| from rest\_framework.filters import SearchFilter  class YourModelListView(ListAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer  filter\_backends = [SearchFilter]  search\_fields = ['field1', 'field2'] |
| --- |

#### **Enabling Full-Text Search Filter Backend**

For enabling full-text search, ensure you have django.contrib.postgres.search installed and configure the search\_fields in your view:

| from rest\_framework.filters import SearchFilter  class YourModelListView(ListAPIView):  queryset = YourModel.objects.annotate(search=SearchExpression("title", "description"))  serializer\_class = YourModelSerializer  filter\_backends = [SearchFilter]  search\_fields = ['search'] |
| --- |

#### **Enabling Pagination of QuerySets in API Responses**

Pagination can be enabled by setting the pagination\_class attribute in your view:

| from rest\_framework.pagination import PageNumberPagination  class StandardResultsSetPagination(PageNumberPagination):  page\_size = 10  page\_size\_query\_param = 'page\_size'  max\_page\_size = 100  class YourModelListView(ListAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer  pagination\_class = StandardResultsSetPagination |
| --- |

### **2. Create, Retrieve, Update, and Delete (CRUD) Operations for Models**

#### **Creating a CreateAPIView Subclass**

CreateAPIView handles POST requests to create new instances of a model:

| from rest\_framework.generics import CreateAPIView from.models import YourModel from.serializers import YourModelSerializer  class YourModelCreateView(CreateAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer |
| --- |

#### **Connecting a CreateAPIView to the Router**

Use Django REST Framework's routers to connect your view to a URL pattern:

| from django.urls import path, include from rest\_framework.routers import DefaultRouter from.views import YourModelCreateView  router = DefaultRouter() router.register(r'yourmodel/create', YourModelCreateView)  urlpatterns = [  path('', include(router.urls)), ] |
| --- |

#### **Creating a DestroyAPIView Subclass**

DestroyAPIView handles DELETE requests to remove instances:

| from rest\_framework.generics import DestroyAPIView from.models import YourModel from.serializers import YourModelSerializer  class YourModelDeleteView(DestroyAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer |
| --- |

#### **Connecting a DestroyAPIView to the Router**

Similar to connecting other views, use the router:

| router.register(r'yourmodel/delete/(?P<pk>\d+)', YourModelDeleteView) |
| --- |

#### **Creating an UpdateAPIView Subclass**

UpdateAPIView handles PUT/PATCH requests to update existing instances:

| from rest\_framework.generics import UpdateAPIView from.models import YourModel from.serializers import YourModelSerializer  class YourModelUpdateView(UpdateAPIView):  queryset = YourModel.objects.all()  serializer\_class = YourModelSerializer |
| --- |

#### **Connecting an UpdateAPIView to the Router**

Connect via the router:

| router.register(r'yourmodel/update/(?P<pk>\d+)', YourModelUpdateView) |
| --- |

### **3. Managing Serializer Fields, Relations, and Validation**

#### **Serializer with Only Selected Fields**

Specify the fields you want to include in the serialized output:

| class YourModelSerializer(serializers.ModelSerializer):  class Meta:  model = YourModel  fields = ('id', 'name') # Specify the fields you want |
| --- |

#### **Serializer That Shows Model Relationships**

Use nested serializers to represent related models:

| class RelatedModelSerializer(serializers.ModelSerializer):  class Meta:  model = RelatedModel  fields = '\_\_all\_\_'  class YourModelSerializer(serializers.ModelSerializer):  related\_model = RelatedModelSerializer()   class Meta:  model = YourModel  fields = '\_\_all\_\_' |
| --- |

#### **Number Fields with Serializers**

Define custom methods or use built-in field types like IntegerField, DecimalField, etc.

| class YourModelSerializer(serializers.ModelSerializer):  age = serializers.IntegerField()   class Meta:  model = YourModel  fields = '\_\_all\_\_' |
| --- |

#### **Date and Time Fields with Serializers**

Use DateTimeField for datetime serialization:

| class YourModelSerializer(serializers.ModelSerializer):  timestamp = serializers.DateTimeField()   class Meta:  model = YourModel  fields = '\_\_all\_\_' |
| --- |

#### **Lists, Dicts, and JSON Objects**

Serialize complex Python objects by defining custom methods:

| class YourModelSerializer(serializers.ModelSerializer):  def to\_representation(self, instance):  representation = super().to\_representation(instance)  representation['custom\_field'] = {'key': 'value'}  return representation |
| --- |

#### **Serializer with ImageField and FileField**

Handle file uploads by specifying ImageField or FileField in your serializer:

| class YourModelSerializer(serializers.ModelSerializer):  image = serializers.ImageField(max\_length=None, use\_url=True)   class Meta:  model = YourModel  fields = '\_\_all\_\_' |
| --- |

### **4. Testing API Views**

#### **Test Case for a CreateAPIView Subclass**

Use Django's testing tools to write tests for your views:

| from django.test import TestCase from rest\_framework.test import APIClient from.models import YourModel from.serializers import YourModelSerializer  class YourModelCreateViewTests(TestCase):  def setUp(self):  self.client = APIClient()  self.data = {'name': 'Test Name'}   def test\_create\_view(self):  response = self.client.post('/yourmodel/create/', self.data, format='json')  self.assertEqual(response.status\_code, 201)  self.assertTrue(YourModel.objects.exists()) |
| --- |

#### **Test Case for a DestroyAPIView Subclass**

Similarly, test the deletion functionality:

| def test\_delete\_view(self):  obj = YourModel.objects.create(name='Test Name')  response = self.client.delete(f'/yourmodel/delete/{obj.id}/', format='json')  self.assertEqual(response.status\_code, 204)  self.assertFalse(YourModel.objects.filter(id=obj.id).exists()) |
| --- |

#### **Test Case for a ListAPIView Subclass**

Test listing functionality:

| def test\_list\_view(self):  YourModel.objects.create(name='Test Name')  response = self.client.get('/yourmodel/')  self.assertEqual(response.status\_code, 200)  self.assertContains(response, 'Test Name') |
| --- |

#### **Unit Test for an UpdateAPIView Subclass**

Testing update operations:

| def test\_update\_view(self):  obj = YourModel.objects.create(name='Old Name')  response = self.client.put(f'/yourmodel/update/{obj.id}/', {'name': 'New Name'}, format='json')  self.assertEqual(response.status\_code, 200)  obj.refresh\_from\_db()  self.assertEqual(obj.name, 'New Name') |
| --- |

#### **Handling Image Uploads in a Unit Test**

Testing file uploads requires mocking the request.FILES dictionary:

| def test\_upload\_image(self):  obj = YourModel.objects.create(name='Test Name')  with open('path/to/test/image.jpg', 'rb') as img:  response = self.client.post(  '/yourmodel/upload/',  {'image': img},  format='multipart',  )  self.assertEqual(response.status\_code, 201)  obj.refresh\_from\_db()  self.assertIsNotNone(obj.image) |
| --- |