Django REST Framework (DRF) Complete Tutorial

1. Project & App Creation

1.1 Understanding Django Architecture

Before diving into commands, let's understand what we're building:

Think of Django like a city:

- **Project** = The entire city (your web application)
- **Apps** = Different neighborhoods in the city (user management, blog, e-commerce, etc.)

1.2 Creating a Django Project

```bash
# Install Django and DRF first
pip install django djangorestframework

# Create a new Django project
django-admin startproject taskmanager
cd taskmanager
```

What happens internally:

- Django creates a **project directory** with configuration files
- This is your application's **central hub** that coordinates everything
- Contains settings, URL routing, and deployment configurations

1.3 Project Structure Deep Dive

```
taskmanager/
├── taskmanager/          # Project configuration package
│   ├── __init__.py       # Makes it a Python package
│   ├── settings.py       # All project settings (database, apps, etc.)
│   ├── urls.py           # Main URL dispatcher (traffic controller)
│   ├── wsgi.py           # Web server gateway interface
│   └── asgi.py           # Async server gateway interface
└── manage.py             # Command-line utility (your toolkit)
```

Key Files Explained:

settings.py - Your project's control center:

```python
# This is where you configure everything
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',  # We'll add this for DRF
    'tasks',           # Our custom app (we'll create this)
]

# Database configuration
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

1.4 Creating Your First App

```bash
# Inside your project directory
python manage.py startapp tasks
```

What's an App? An app is a **self-contained module** that does one specific thing well. Examples:

- `users` app → handles user registration, login, profiles
- `tasks` app → manages todo items
- `payments` app → handles billing and subscriptions

1.5 App Structure Deep Dive

```
tasks/
├── __init__.py        # Python package marker
├── admin.py           # Admin interface configuration
├── apps.py            # App configuration
├── models.py          # Database models (your data structure)
├── serializers.py     # DRF serializers (we'll create this)
├── views.py           # View logic (request handlers)
├── urls.py            # App-specific URLs (we'll create this)
├── tests.py           # Unit tests
└── migrations/        # Database migration files
    └── __init__.py
```

1.6 Connecting App to Project

Step 1: Add your app to `settings.py`

python

```python
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',      # Add DRF
    'tasks',               # Add your app
]
```

Step 2: Configure DRF in `settings.py`

python

```python
# Add at the bottom of settings.py
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
    ],
    'DEFAULT_PARSER_CLASSES': [
        'rest_framework.parsers.JSONParser',
    ],
}
```

1.7 Best Practices for Project Organization

Multiple Apps Strategy:

```
myproject/
├── myproject/          # Project settings
├── users/              # User management app
├── tasks/              # Task management app
├── notifications/      # Notification app
└── api/                # API configuration app
```

Why separate apps?

- **Maintainability**: Each app has a single responsibility
- **Reusability**: You can use the `users` app in other projects
- **Team Development**: Different developers can work on different apps
- **Testing**: Easier to test isolated functionality

---

2. Models

2.1 What Are Models?

Simple Analogy: Models are like **blueprints for database tables**. Just like an architect's blueprint defines how a house should be built, a Django model defines how your data should be stored.

Technical Definition: A model is a Python class that inherits from `django.db.models.Model` and represents a database table.

2.2 Creating Your First Model

Create a `Task` model in `tasks/models.py`:

python

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

class Task(models.Model):
    # Text fields
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True, null=True)

    # Boolean field
    completed = models.BooleanField(default=False)

    # DateTime fields
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    due_date = models.DateTimeField(blank=True, null=True)

    # Foreign Key relationship
    owner = models.ForeignKey(User, on_delete=models.CASCADE, related_name='tasks')

    # Choices field
    PRIORITY_CHOICES = [
        ('low', 'Low'),
        ('medium', 'Medium'),
        ('high', 'High'),
    ]
    priority = models.CharField(max_length=10, choices=PRIORITY_CHOICES, default='medium')

    class Meta:
        ordering = ['-created_at']  # Order by newest first
        verbose_name = 'Task'
        verbose_name_plural = 'Tasks'

    def __str__(self):
        return f"{self.title} ({'✓' if self.completed else 'X'})"

    def is_overdue(self):
        """Custom method to check if task is overdue"""
        if self.due_date and not self.completed:
            from django.utils import timezone
            return timezone.now() > self.due_date
        return False
```

2.3 Field Types Deep Dive

Text Fields:

```python
# CharField - Limited text (like a single line)
title = models.CharField(max_length=200)  # Required: max_length

# TextField - Unlimited text (like a paragraph)
description = models.TextField()  # No max_length needed

# EmailField - Validates email format
email = models.EmailField()

# URLField - Validates URL format
website = models.URLField()
```

Numeric Fields:

```python
# IntegerField - Whole numbers
priority_level = models.IntegerField()

# FloatField - Decimal numbers
rating = models.FloatField()

# DecimalField - Precise decimal (for money)
price = models.DecimalField(max_digits=10, decimal_places=2)
```

Date/Time Fields:

```python
# DateTimeField - Date and time
created_at = models.DateTimeField(auto_now_add=True)  # Set once when created
updated_at = models.DateTimeField(auto_now=True)      # Update every save

# DateField - Just date
birth_date = models.DateField()

# TimeField - Just time
meeting_time = models.TimeField()
```

Boolean Fields:

```python
# BooleanField - True/False
is_completed = models.BooleanField(default=False)

# NullBooleanField - True/False/None (deprecated, use BooleanField with null=True)
is_verified = models.BooleanField(null=True, blank=True)
```

2.4 Field Options (Arguments)

Common Field Options:

```python
class Task(models.Model):
    # null=True: Database can store NULL values
    # blank=True: Django forms can submit empty values
    description = models.TextField(null=True, blank=True)

    # default: Default value when creating new instances
    completed = models.BooleanField(default=False)

    # choices: Dropdown options
    STATUS_CHOICES = [
        ('pending', 'Pending'),
        ('in_progress', 'In Progress'),
        ('completed', 'Completed'),
    ]
    status = models.CharField(max_length=20, choices=STATUS_CHOICES, default='pending')

    # unique: No duplicate values allowed
    slug = models.SlugField(unique=True)

    # db_index: Creates database index for faster queries
    priority = models.IntegerField(db_index=True)
```

When to use null=True vs blank=True:

- `null=True`: Database-level (can store NULL in database)
- `blank=True`: Validation-level (Django forms allow empty submission)
- Usually use both together: `null=True, blank=True`

2.5 Relationships Between Models

One-to-Many (ForeignKey):

```python
class Category(models.Model):
    name = models.CharField(max_length=100)

class Task(models.Model):
    title = models.CharField(max_length=200)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)

    # One category can have many tasks
    # Each task belongs to one category
```

Many-to-Many:

```python
class Tag(models.Model):
    name = models.CharField(max_length=50)

class Task(models.Model):
    title = models.CharField(max_length=200)
    tags = models.ManyToManyField(Tag, blank=True)

    # One task can have many tags
    # One tag can be on many tasks
```

One-to-One:

```python
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField()
    avatar = models.ImageField()

    # Each user has exactly one profile
    # Each profile belongs to exactly one user
```

on_delete Options:

```python
# CASCADE: Delete related objects when parent is deleted
owner = models.ForeignKey(User, on_delete=models.CASCADE)

# PROTECT: Prevent deletion if related objects exist
category = models.ForeignKey(Category, on_delete=models.PROTECT)

# SET_NULL: Set to NULL when parent is deleted
assigned_to = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)

# SET_DEFAULT: Set to default value
status = models.ForeignKey(Status, on_delete=models.SET_DEFAULT, default=1)
```

2.6 Database Migrations

What are migrations? Migrations are Django's way of **tracking and applying database changes**. Think of them as **version control for your database**.

Creating migrations:

```bash
# After creating/modifying models
python manage.py makemigrations

# Apply migrations to database
python manage.py migrate
```

What happens during makemigrations:

1. Django compares your current models with the last migration
2. Creates a new migration file with the differences
3. Stores it in `app/migrations/0001_initial.py`

Migration file example:

python

```python
# tasks/migrations/0001_initial.py
from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True

    dependencies = []

    operations = [
        migrations.CreateModel(
            name='Task',
            fields=[
                ('id', models.AutoField(primary_key=True)),
                ('title', models.CharField(max_length=200)),
                ('completed', models.BooleanField(default=False)),
                ('created_at', models.DateTimeField(auto_now_add=True)),
            ],
        ),
    ]
```

2.7 Model Methods and Properties

Custom methods in models:

python

```python
class Task(models.Model):
    title = models.CharField(max_length=200)
    completed = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField(null=True, blank=True)

    def is_overdue(self):
        """Check if task is overdue"""
        if self.due_date and not self.completed:
            from django.utils import timezone
            return timezone.now() > self.due_date
        return False

    @property
    def status_display(self):
        """Human-readable status"""
        if self.completed:
            return "✅ Completed"
        elif self.is_overdue():
            return "⚠️ Overdue"
        else:
            return "📝 Pending"

    def mark_complete(self):
        """Mark task as completed"""
        self.completed = True
        self.save()
```

Using custom methods:

python

```python
# In views or anywhere in your code
task = Task.objects.get(id=1)
print(task.status_display)  # "✅ Completed"

if task.is_overdue():
    print("This task is overdue!")

task.mark_complete()  # Marks task as done and saves to DB
```

2.8 Meta Class Options

Common Meta options:

```python
class Task(models.Model):
    title = models.CharField(max_length=200)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        # Default ordering for queries
        ordering = ['-created_at']  # Newest first

        # Database table name
        db_table = 'custom_tasks'

        # Human-readable names
        verbose_name = 'Task'
        verbose_name_plural = 'Tasks'

        # Unique constraints
        unique_together = ['title', 'owner']  # No duplicate titles per owner

        # Database indexes for faster queries
        indexes = [
            models.Index(fields=['completed', 'created_at']),
        ]

        # Permissions
        permissions = [
            ('can_mark_complete', 'Can mark tasks as complete'),
        ]
```

2.9 Common Model Mistakes and Solutions

Mistake 1: Forgetting migrations

```python
# After changing models, always run:
python manage.py makemigrations
python manage.py migrate
```

Mistake 2: Not handling NULL values

```python
# Bad
description = models.TextField()  # Will cause errors if empty

# Good
description = models.TextField(null=True, blank=True)
```

Mistake 3: Circular imports

```python
# Bad - importing models in models.py
from .views import some_function

# Good - import inside methods when needed
def some_model_method(self):
    from .views import some_function
    return some_function()
```

Mistake 4: Not using str method

```python
# Bad - objects show as <Task: Task object (1)>
class Task(models.Model):
    title = models.CharField(max_length=200)

# Good - shows meaningful representation
class Task(models.Model):
    title = models.CharField(max_length=200)

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

2.10 Testing Your Model

Create and test your model in Django shell:

```bash
python manage.py shell
```

```python
# Import your model
from tasks.models import Task
from django.contrib.auth.models import User

# Create a user first
user = User.objects.create_user('john', 'john@example.com', 'password123')

# Create a task
task = Task.objects.create(
    title="Learn Django REST Framework",
    description="Master DRF concepts and build APIs",
    owner=user,
    priority='high'
)

# Query tasks
all_tasks = Task.objects.all()
completed_tasks = Task.objects.filter(completed=True)
user_tasks = Task.objects.filter(owner=user)

# Use custom methods
print(task.status_display)
print(task.is_overdue())
```

---

3. Serializers

3.1 What Are Serializers?

Simple Analogy: Think of serializers as **translators** at the United Nations. They convert between different languages:

- **Python objects** → **JSON** (when sending API responses)
- **JSON** → **Python objects** (when receiving API requests)

Technical Definition: Serializers convert complex data types (like Django model instances) into native Python data types that can be easily rendered into JSON, XML, or other content types.

3.2 Why Do We Need Serializers?

The Problem:

```python
# Your Django model (Python object)
task = Task.objects.get(id=1)
print(task)  # <Task: Learn Django>

# You can't send this to a frontend or API client!
# APIs need JSON format:
# {
#   "id": 1,
#   "title": "Learn Django",
#   "completed": false
# }
```

The Solution:

```python
# Serializer converts model to dictionary
serializer = TaskSerializer(task)
print(serializer.data)
# Output: {'id': 1, 'title': 'Learn Django', 'completed': False}

# Convert to JSON for API response
import json
json_data = json.dumps(serializer.data)
# Output: '{"id": 1, "title": "Learn Django", "completed": false}'
```

3.3 Creating Your First Serializer

Create `tasks/serializers.py`:

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

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'  # Include all model fields
```

What this does:

- **Automatically creates fields** based on your Task model
- **Handles serialization** (Model → JSON)
- **Handles deserialization** (JSON → Model)
- **Includes validation** based on model field definitions

3.4 ModelSerializer vs Serializer

ModelSerializer (Recommended for most cases):

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'

    # Automatically gets:
    # - All model fields
    # - Field validation from model
    # - create() and update() methods
```

Plain Serializer (More control, more work):

```python
class TaskSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(max_length=200)
    completed = serializers.BooleanField(default=False)
    created_at = serializers.DateTimeField(read_only=True)

    def create(self, validated_data):
        """Create and return a new Task instance"""
        return Task.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """Update and return an existing Task instance"""
        instance.title = validated_data.get('title', instance.title)
        instance.completed = validated_data.get('completed', instance.completed)
        instance.save()
        return instance
```

When to use which:

- **ModelSerializer**: 90% of cases (DRY principle)
- **Plain Serializer**: When you need complex custom logic or non-model data

3.5 Field Selection and Control

Include specific fields:

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'created_at']
        # Only these fields will be included in serialization
```

Exclude specific fields:

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        exclude = ['owner']  # All fields except 'owner'
```

Read-only and write-only fields:

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'
        read_only_fields = ['id', 'created_at', 'updated_at']  # Can't be modified via API
        extra_kwargs = {
            'password': {'write_only': True},  # Won't appear in responses
            'email': {'read_only': True},      # Can't be modified
        }
```

3.6 Custom Fields and Methods

Adding computed/custom fields:

```python
class TaskSerializer(serializers.ModelSerializer):
    # Custom field that doesn't exist in model
    status_display = serializers.CharField(source='status_display', read_only=True)

    # Custom field with method
    days_since_created = serializers.SerializerMethodField()

    # Custom field with different source
    task_owner = serializers.CharField(source='owner.username', read_only=True)

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'status_display',
                  'days_since_created', 'task_owner']

    def get_days_since_created(self, obj):
        """Custom method for SerializerMethodField"""
        from django.utils import timezone
        delta = timezone.now() - obj.created_at
        return delta.days
```

Output example:

```json
{
    "id": 1,
    "title": "Learn Django",
    "completed": false,
    "status_display": "📝 Pending",
    "days_since_created": 3,
    "task_owner": "john_doe"
}
```

3.7 Validation in Serializers

Field-level validation:

python

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'

    def validate_title(self, value):
        """Validate individual field"""
        if len(value) < 3:
            raise serializers.ValidationError("Title must be at least 3 characters long")

        if 'spam' in value.lower():
            raise serializers.ValidationError("Title cannot contain spam")

        return value

    def validate_due_date(self, value):
        """Validate due date is in future"""
        from django.utils import timezone
        if value and value < timezone.now():
            raise serializers.ValidationError("Due date cannot be in the past")
        return value
```

Object-level validation:

```python
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'

    def validate(self, data):
        """Validate across multiple fields"""
        # Check if high priority tasks have due dates
        if data.get('priority') == 'high' and not data.get('due_date'):
            raise serializers.ValidationError(
                "High priority tasks must have a due date"
            )

        # Check if completed tasks have completion dates
        if data.get('completed') and not data.get('completed_at'):
            from django.utils import timezone
            data['completed_at'] = timezone.now()

        return data
```

Using validation:

```python
# In your views or tests
serializer = TaskSerializer(data=request_data)
if serializer.is_valid():
    task = serializer.save()
else:
    print(serializer.errors)
    # Output: {'title': ['Title must be at least 3 characters long']}
```

3.8 Nested Serializers

Related object serialization:

```python
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email']

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = ['id', 'name']

class TaskSerializer(serializers.ModelSerializer):
    owner = UserSerializer(read_only=True)  # Nested user data
    category = CategorySerializer(read_only=True)  # Nested category data

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'owner', 'category']
```

Output with nested data:

```json
{
    "id": 1,
    "title": "Learn Django",
    "completed": false,
    "owner": {
        "id": 1,
        "username": "john_doe",
        "email": "john@example.com"
    },
    "category": {
        "id": 1,
        "name": "Education"
    }
}
```

Many-to-Many relationships:

```python
class TaskSerializer(serializers.ModelSerializer):
    tags = TagSerializer(many=True, read_only=True)  # Many tags per task

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'tags']
```

3.9 Writable Nested Serializers

Creating related objects:

python

```python
class TaskSerializer(serializers.ModelSerializer):
    category = CategorySerializer()  # Remove read_only=True

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'category']

    def create(self, validated_data):
        # Extract category data
        category_data = validated_data.pop('category')

        # Create or get category
        category, created = Category.objects.get_or_create(**category_data)

        # Create task with category
        task = Task.objects.create(category=category, **validated_data)
        return task

    def update(self, instance, validated_data):
        # Handle category updates
        category_data = validated_data.pop('category', None)
        if category_data:
            category, created = Category.objects.get_or_create(**category_data)
            instance.category = category

        # Update other fields
        for attr, value in validated_data.items():
            setattr(instance, attr, value)

        instance.save()
        return instance
```

3.10 Different Serializers for Different Actions

Create different serializers for different use cases:

python

```python
# Detailed serializer for retrieving tasks
class TaskDetailSerializer(serializers.ModelSerializer):
    owner = UserSerializer(read_only=True)
    category = CategorySerializer(read_only=True)
    tags = TagSerializer(many=True, read_only=True)
    status_display = serializers.CharField(source='status_display', read_only=True)

    class Meta:
        model = Task
        fields = '__all__'

# Simple serializer for listing tasks
class TaskListSerializer(serializers.ModelSerializer):
    owner_name = serializers.CharField(source='owner.username', read_only=True)

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'priority', 'owner_name']

# Create/Update serializer (no read-only fields for input)
class TaskCreateSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = ['title', 'description', 'priority', 'due_date', 'category']

    def create(self, validated_data):
        # Add current user as owner
        request = self.context.get('request')
        validated_data['owner'] = request.user
        return super().create(validated_data)
```

3.11 Serializer Context

Passing extra data to serializers:

```python
# In your view
class TaskViewSet(viewsets.ModelViewSet):
    def get_serializer_context(self):
        context = super().get_serializer_context()
        context['request'] = self.request
        context['current_user'] = self.request.user
        context['view_name'] = 'task_api'
        return context

# In your serializer
class TaskSerializer(serializers.ModelSerializer):
    is_owner = serializers.SerializerMethodField()

    class Meta:
        model = Task
        fields = ['id', 'title', 'completed', 'is_owner']

    def get_is_owner(self, obj):
        request = self.context.get('request')
        if request and request.user:
            return obj.owner == request.user
        return False
```

3.12 Error Handling and Common Mistakes

Common Mistake 1: Forgetting validation

```python
# Bad - No validation
serializer = TaskSerializer(data=request.data)
task = serializer.save()  # This might fail!

# Good - Always validate first
serializer = TaskSerializer(data=request.data)
if serializer.is_valid():
    task = serializer.save()
    return Response(serializer.data, status=201)
else:
    return Response(serializer.errors, status=400)
```

Common Mistake 2: Circular imports

```python
# Bad - Circular import
# serializers.py
from .views import SomeView

# Good - Import inside method
class TaskSerializer(serializers.ModelSerializer):
    def validate_something(self, value):
        from .utils import validate_helper  # Import here
        return validate_helper(value)
```

Common Mistake 3: Not handling None values

```python
# Bad - Will crash if owner is None
class TaskSerializer(serializers.ModelSerializer):
    owner_name = serializers.CharField(source='owner.username')

# Good - Handle None values
class TaskSerializer(serializers.ModelSerializer):
    owner_name = serializers.SerializerMethodField()

    def get_owner_name(self, obj):
        return obj.owner.username if obj.owner else 'No Owner'
```

3.13 Testing Serializers

Test serialization (Model → JSON):

```python
# In tasks/tests.py
from django.test import TestCase
from .models import Task
from .serializers import TaskSerializer

class TaskSerializerTest(TestCase):
    def test_task_serialization(self):
        task = Task.objects.create(
            title="Test Task",
            completed=False
        )

        serializer = TaskSerializer(task)
        data = serializer.data

        self.assertEqual(data['title'], 'Test Task')
        self.assertEqual(data['completed'], False)
        self.assertIn('id', data)
```

Test deserialization (JSON → Model):

```python
def test_task_deserialization(self):
    data = {
        'title': 'New Task',
        'completed': True,
        'priority': 'high'
    }

    serializer = TaskSerializer(data=data)
    self.assertTrue(serializer.is_valid())

    task = serializer.save()
    self.assertEqual(task.title, 'New Task')
    self.assertTrue(task.completed)
```

Test validation:

```python
def test_validation_errors(self):
    data = {
        'title': 'x',  # Too short
        'priority': 'invalid'  # Invalid choice
    }

    serializer = TaskSerializer(data=data)
    self.assertFalse(serializer.is_valid())
    self.assertIn('title', serializer.errors)
    self.assertIn('priority', serializer.errors)
```

---

4. ViewSets

4.1 What Are ViewSets?

Simple Analogy: ViewSets are like **restaurant servers**. Just as a server handles different customer requests (taking orders, serving food, processing payments), ViewSets handle different HTTP requests (GET, POST, PUT, DELETE) for your API endpoints.

Technical Definition: ViewSets are classes that group together the logic for handling multiple related views. Instead of writing separate view functions for each HTTP method, you get all CRUD operations in one place.

4.2 Why Use ViewSets?

Without ViewSets (Traditional Django Views):

```python
# You'd need separate view functions
def task_list(request):      # GET /tasks/
def task_create(request):    # POST /tasks/
def task_detail(request, pk): # GET /tasks/1/
def task_update(request, pk): # PUT /tasks/1/
def task_delete(request, pk): # DELETE /tasks/1/
```

With ViewSets (DRF Way):

```python
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    # This automatically provides:
    # list()    - GET /tasks/
    # create()  - POST /tasks/
    # retrieve() - GET /tasks/1/
    # update()  - PUT /tasks/1/
    # destroy() - DELETE /tasks/1/
```

4.3 Types of ViewSets

ModelViewSet (Most Common):

```python
from rest_framework import viewsets
from .models import Task
from .serializers import TaskSerializer

class TaskViewSet(viewsets.ModelViewSet):
    """
    A viewset that provides default `create()`, `retrieve()`, `update()`,
    `partial_update()`, `destroy()` and `list()` actions.
    """
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

ReadOnlyModelViewSet (Read-only operations):

```python
class TaskViewSet(viewsets.ReadOnlyModelViewSet):
    """
    A viewset that provides default `list()` and `retrieve()` actions only.
    No create, update, or delete operations.
    """
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

ViewSet (Custom implementation):

```python
from rest_framework.decorators import action
from rest_framework.response import Response

class TaskViewSet(viewsets.ViewSet):
    """
    A simple ViewSet for listing or retrieving tasks.
    You must implement all methods yourself.
    """
    def list(self, request):
        tasks = Task.objects.all()
        serializer = TaskSerializer(tasks, many=True)
        return Response(serializer.data)

    def create(self, request):
        serializer = TaskSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=201)
        return Response(serializer.errors, status=400)

    def retrieve(self, request, pk=None):
        task = Task.objects.get(pk=pk)
        serializer = TaskSerializer(task)
        return Response(serializer.data)
```

4.4 Creating Your First ViewSet

Step 1: Create the ViewSet Create/update `tasks/views.py`:

```python
from rest_framework import viewsets
from rest_framework.decorators import action
from rest_framework.response import Response
from rest_framework import status
from django.shortcuts import get_object_or_404
from .models import Task
from .serializers import TaskSerializer

class TaskViewSet(viewsets.ModelViewSet):
    """
    ViewSet for managing tasks.
    Provides CRUD operations for Task model.
    """
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    def get_queryset(self):
        """
        Override to filter tasks by current user.
        """
        user = self.request.user
        if user.is_authenticated:
            return Task.objects.filter(owner=user).order_by('-created_at')
        return Task.objects.none()  # Return empty queryset for anonymous users

    def perform_create(self, serializer):
        """
        Override to set the owner to the current user when creating a task.
        """
        serializer.save(owner=self.request.user)
```

Step 2: Create URLs Create `tasks/urls.py`:

```python
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet

router = DefaultRouter()
router.register(r'tasks', TaskViewSet, basename='task')

urlpatterns = router.urls
```

Step 3: Include in Main URLs Update main `taskmanager/urls.py`:

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('tasks.urls')),
]
```

4.5 Understanding ViewSet Methods

Automatic Methods in ModelViewSet:

python

```python
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    # These methods are automatically provided:

    def list(self, request):
        """
        GET /api/v1/tasks/
        Returns a list of all tasks
        """
        # Implementation is automatic
        pass

    def create(self, request):
        """
        POST /api/v1/tasks/
        Creates a new task
        """
        # Implementation is automatic
        pass

    def retrieve(self, request, pk=None):
        """
        GET /api/v1/tasks/1/
        Returns a specific task
        """
        # Implementation is automatic
        pass

    def update(self, request, pk=None):
        """
        PUT /api/v1/tasks/1/
        Updates a task (full update)
        """
        # Implementation is automatic
        pass

    def partial_update(self, request, pk=None):
        """
        PATCH /api/v1/tasks/1/
        Partially updates a task
        """
        # Implementation is automatic
```

```
        pass

    def destroy(self, request, pk=None):
        """
        DELETE /api/v1/tasks/1/
        Deletes a task
        """
        # Implementation is automatic
        pass
```

4.6 Customizing ViewSet Behavior

Override methods for custom behavior:

python

```python
from rest_framework import viewsets, status
from rest_framework.response import Response
from django.utils import timezone

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    def create(self, request, *args, **kwargs):
        """
        Override create to add custom logic
        """
        # Add created timestamp
        if 'created_at' not in request.data:
            request.data['created_at'] = timezone.now()

        # Call the parent create method
        response = super().create(request, *args, **kwargs)

        # Add custom response message
        if response.status_code == status.HTTP_201_CREATED:
            response.data['message'] = 'Task created successfully!'

        return response

    def update(self, request, *args, **kwargs):
        """
        Override update to add validation
        """
        partial = kwargs.pop('partial', False)
        instance = self.get_object()

        # Custom validation
        if instance.completed and not request.user.is_staff:
            return Response(
                {'error': 'Cannot modify completed tasks'},
                status=status.HTTP_400_BAD_REQUEST
            )

        return super().update(request, *args, **kwargs)

    def destroy(self, request, *args, **kwargs):
        """
        Override destroy to add soft delete
        """
```

```
        """
        instance = self.get_object()

        # Soft delete instead of hard delete
        instance.is_deleted = True
        instance.deleted_at = timezone.now()
        instance.save()

        return Response(
            {'message': 'Task deleted successfully'},
            status=status.HTTP_204_NO_CONTENT
        )
```

4.7 Custom Querysets and Filtering

Dynamic querysets based on request:

python

```python
class TaskViewSet(viewsets.ModelViewSet):
    serializer_class = TaskSerializer

    def get_queryset(self):
        """
        Dynamic queryset based on user and query parameters
        """
        user = self.request.user
        queryset = Task.objects.filter(owner=user)

        # Filter by completion status
        completed = self.request.query_params.get('completed', None)
        if completed is not None:
            completed_bool = completed.lower() == 'true'
            queryset = queryset.filter(completed=completed_bool)

        # Filter by priority
        priority = self.request.query_params.get('priority', None)
        if priority:
            queryset = queryset.filter(priority=priority)

        # Filter by date range
        start_date = self.request.query_params.get('start_date', None)
        end_date = self.request.query_params.get('end_date', None)

        if start_date:
            queryset = queryset.filter(created_at__gte=start_date)
        if end_date:
            queryset = queryset.filter(created_at__lte=end_date)

        # Search in title and description
        search = self.request.query_params.get('search', None)
        if search:
            from django.db.models import Q
            queryset = queryset.filter(
                Q(title__icontains=search) |
                Q(description__icontains=search)
            )

        return queryset.order_by('-created_at')
```

Usage examples:

```bash
# Get completed tasks
GET /api/v1/tasks/?completed=true

# Get high priority tasks
GET /api/v1/tasks/?priority=high

# Search tasks
GET /api/v1/tasks/?search=django

# Combine filters
GET /api/v1/tasks/?completed=false&priority=high&search=urgent
```

4.8 Different Serializers for Different Actions

Use different serializers based on action:

```python
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()

    def get_serializer_class(self):
        """
        Return different serializers for different actions
        """
        if self.action == 'list':
            return TaskListSerializer      # Minimal fields for listing
        elif self.action == 'create':
            return TaskCreateSerializer    # Fields needed for creation
        elif self.action in ['retrieve', 'update', 'partial_update']:
            return TaskDetailSerializer    # Full fields for detail view
        return TaskSerializer  # Default serializer

    def get_serializer_context(self):
        """
        Add extra context to serializer
        """
        context = super().get_serializer_context()
        context['request'] = self.request
        context['action'] = self.action
        return context
```

4.9 Permissions and Authentication

Add permissions to ViewSets:

```python
from rest_framework.permissions import IsAuthenticated, IsAuthenticatedOrReadOnly
from rest_framework.authentication import TokenAuthentication, SessionAuthentication

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    authentication_classes = [TokenAuthentication, SessionAuthentication]
    permission_classes = [IsAuthenticated]

    def get_permissions(self):
        """
        Instantiates and returns the list of permissions for this view.
        """
        if self.action == 'list':
            permission_classes = [IsAuthenticatedOrReadOnly]
        elif self.action == 'create':
            permission_classes = [IsAuthenticated]
        elif self.action in ['retrieve', 'update', 'partial_update', 'destroy']:
            permission_classes = [IsAuthenticated]  # + custom permission for ownership
        else:
            permission_classes = [IsAuthenticated]

        return [permission() for permission in permission_classes]
```

Custom permission for task ownership:

```python
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Custom permission to only allow owners of a task to edit it.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions for any request (GET, HEAD, OPTIONS)
        if request.method in permissions.SAFE_METHODS:
            return True

        # Write permissions only for task owner
        return obj.owner == request.user

# Use in ViewSet
class TaskViewSet(viewsets.ModelViewSet):
    permission_classes = [IsAuthenticated, IsOwnerOrReadOnly]
```

4.10 Error Handling in ViewSets

Custom error handling:

python

```python
from rest_framework import status
from rest_framework.response import Response
from django.db import IntegrityError

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    def create(self, request, *args, **kwargs):
        try:
            return super().create(request, *args, **kwargs)
        except IntegrityError as e:
            return Response(
                {'error': 'A task with this title already exists'},
                status=status.HTTP_400_BAD_REQUEST
            )
        except Exception as e:
            return Response(
                {'error': 'An unexpected error occurred'},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )

    def update(self, request, *args, **kwargs):
        try:
            return super().update(request, *args, **kwargs)
        except Task.DoesNotExist:
            return Response(
                {'error': 'Task not found'},
                status=status.HTTP_404_NOT_FOUND
            )

    def destroy(self, request, *args, **kwargs):
        try:
            instance = self.get_object()
            if instance.completed:
                return Response(
                    {'error': 'Cannot delete completed tasks'},
                    status=status.HTTP_400_BAD_REQUEST
                )
            return super().destroy(request, *args, **kwargs)
        except Task.DoesNotExist:
            return Response(
                {'error': 'Task not found'},
                status=status.HTTP_404_NOT_FOUND
```

```
)
```

4.11 Pagination in ViewSets

Add pagination to handle large datasets:

In `settings.py`:

```python
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20
}
```

Custom pagination:

```python
from rest_framework.pagination import PageNumberPagination

class TaskPagination(PageNumberPagination):
    page_size = 10
    page_size_query_param = 'page_size'
    max_page_size = 100

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    pagination_class = TaskPagination
```

Paginated response example:

```json
{
    "count": 100,
    "next": "http://api.example.org/tasks/?page=3",
    "previous": "http://api.example.org/tasks/?page=1",
    "results": [
        {
            "id": 1,
            "title": "Task 1",
            "completed": false
        }
    ]
}
```

4.12 Testing ViewSets

Test ViewSet endpoints:

python

```python
from django.test import TestCase
from rest_framework.test import APIClient
from rest_framework import status
from django.contrib.auth.models import User
from .models import Task

class TaskViewSetTest(TestCase):
    def setUp(self):
        self.client = APIClient()
        self.user = User.objects.create_user(
            username='testuser',
            password='testpass123'
        )
        self.client.force_authenticate(user=self.user)

        self.task = Task.objects.create(
            title='Test Task',
            description='Test Description',
            owner=self.user
        )

    def test_list_tasks(self):
        """Test retrieving task list"""
        response = self.client.get('/api/v1/tasks/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data['results']), 1)
        self.assertEqual(response.data['results'][0]['title'], 'Test Task')

    def test_create_task(self):
        """Test creating a new task"""
        data = {
            'title': 'New Task',
            'description': 'New Description',
            'priority': 'high'
        }
        response = self.client.post('/api/v1/tasks/', data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Task.objects.count(), 2)

    def test_retrieve_task(self):
        """Test retrieving a specific task"""
        response = self.client.get(f'/api/v1/tasks/{self.task.id}/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['title'], 'Test Task')
```

```
    def test_update_task(self):
        """Test updating a task"""
        data = {'title': 'Updated Task'}
        response = self.client.patch(f'/api/v1/tasks/{self.task.id}/', data)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

        self.task.refresh_from_db()
        self.assertEqual(self.task.title, 'Updated Task')

    def test_delete_task(self):
        """Test deleting a task"""
        response = self.client.delete(f'/api/v1/tasks/{self.task.id}/')
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
        self.assertEqual(Task.objects.count(), 0)

    def test_unauthorized_access(self):
        """Test that unauthenticated users cannot access tasks"""
        self.client.force_authenticate(user=None)
        response = self.client.get('/api/v1/tasks/')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

---

5. Routers

5.1 What Are Routers?

Simple Analogy: Routers are like **traffic directors** at a busy intersection. Just as traffic directors guide cars to the right lanes based on where they want to go, routers guide HTTP requests to the right ViewSet methods based on the URL pattern and HTTP method.

Technical Definition: Routers in DRF automatically generate URL patterns for your ViewSets. Instead of manually writing URL patterns for each endpoint, routers create them based on ViewSet conventions.

5.2 Why Use Routers?

Without Routers (Manual URL patterns):

```python
# urls.py - You'd need to write all these manually
from django.urls import path
from .views import TaskViewSet

urlpatterns = [
    path('tasks/', TaskViewSet.as_view({'get': 'list', 'post': 'create'})),
    path('tasks/<int:pk>/', TaskViewSet.as_view({
        'get': 'retrieve',
        'put': 'update',
        'patch': 'partial_update',
        'delete': 'destroy'
    })),
    path('tasks/<int:pk>/mark_complete/', TaskViewSet.as_view({'post': 'mark_complete'})),
    # ... more manual URL patterns
]
```

With Routers (Automatic URL generation):

```python
# urls.py - Much cleaner!
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet

router = DefaultRouter()
router.register(r'tasks', TaskViewSet, basename='task')

urlpatterns = router.urls
```

5.3 Types of Routers

DefaultRouter (Most Common):

```python
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register(r'tasks', TaskViewSet, basename='task')

# Generates these URLs automatically:
# GET    /tasks/         -> TaskViewSet.list()
# POST   /tasks/         -> TaskViewSet.create()
# GET    /tasks/{id}/    -> TaskViewSet.retrieve()
# PUT    /tasks/{id}/    -> TaskViewSet.update()
# PATCH  /tasks/{id}/    -> TaskViewSet.partial_update()
# DELETE /tasks/{id}/    -> TaskViewSet.destroy()
# GET    /               -> API root view (shows all available endpoints)
```

SimpleRouter (No API root):

```python
from rest_framework.routers import SimpleRouter

router = SimpleRouter()
router.register(r'tasks', TaskViewSet, basename='task')

# Generates the same URLs as DefaultRouter, but without API root view
```

5.4 Setting Up Routers

Step 1: Create router in app's urls.py Create `tasks/urls.py`:

```python
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet

# Create router instance
router = DefaultRouter()

# Register ViewSets with the router
router.register(r'tasks', TaskViewSet, basename='task')

# The router generates urlpatterns automatically
urlpatterns = router.urls
```

Step 2: Include router URLs in main project Update `taskmanager/urls.py`:

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('tasks.urls')),  # Include app URLs with version prefix
]
```

Step 3: Test your URLs

```bash
# Start development server
python manage.py runserver

# Your API endpoints are now available at:
# http://localhost:8000/api/v1/tasks/
# http://localhost:8000/api/v1/tasks/1/
# http://localhost:8000/api/v1/       # API root (shows all endpoints)
```

5.5 Router Registration Options

Basic registration:

```python
router.register(r'tasks', TaskViewSet)
# Uses model name for basename automatically: 'task'
```

Custom basename:

```python
router.register(r'tasks', TaskViewSet, basename='my-tasks')
# Useful when ViewSet doesn't have a queryset attribute
# Or when you want custom URL names
```

Custom prefix:

```python
router.register(r'my-tasks', TaskViewSet, basename='task')
# URLs will be: /api/v1/my-tasks/ instead of /api/v1/tasks/
```

5.6 Multiple ViewSets and Organization

Register multiple ViewSets:

```python
# tasks/urls.py
from rest_framework.routers import DefaultRouter
from .views import TaskViewSet, CategoryViewSet, TagViewSet

router = DefaultRouter()

# Register multiple ViewSets
router.register(r'tasks', TaskViewSet, basename='task')
router.register(r'categories', CategoryViewSet, basename='category')
router.register(r'tags', TagViewSet, basename='tag')

urlpatterns = router.urls
```

Organize multiple apps:

```python
# Main project urls.py
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/tasks/', include('tasks.urls')),      # Task-related endpoints
    path('api/v1/users/', include('users.urls')),      # User-related endpoints
    path('api/v1/auth/', include('authentication.urls')), # Auth endpoints
]
```

5.7 Understanding Generated URLs

What router.register() creates:

```python
router.register(r'tasks', TaskViewSet, basename='task')
```

Generated URL patterns:

```python
# Equivalent manual URL patterns:
urlpatterns = [
    # List and Create
    path('tasks/', TaskViewSet.as_view({
        'get': 'list',      # GET /tasks/ -> list all tasks
        'post': 'create'    # POST /tasks/ -> create new task
    }), name='task-list'),

    # Detail operations
    path('tasks/<int:pk>/', TaskViewSet.as_view({
        'get': 'retrieve',       # GET /tasks/1/ -> get specific task
        'put': 'update',         # PUT /tasks/1/ -> full update
        'patch': 'partial_update', # PATCH /tasks/1/ -> partial update
        'delete': 'destroy'      # DELETE /tasks/1/ -> delete task
    }), name='task-detail'),
]
```

URL name patterns:

- List endpoint: `{basename}-list` → `task-list`
- Detail endpoint: `{basename}-detail` → `task-detail`
- Custom actions: `{basename}-{action-name}` → `task-mark-complete`

5.8 Custom Routes and Actions

Router integration with custom actions:

```python
# In your ViewSet
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    @action(detail=True, methods=['post'])
    def mark_complete(self, request, pk=None):
        """Mark a task as complete"""
        task = self.get_object()
        task.completed = True
        task.save()
        return Response({'status': 'task marked as complete'})

    @action(detail=False, methods=['get'])
    def completed(self, request):
        """Get all completed tasks"""
        completed_tasks = self.get_queryset().filter(completed=True)
        serializer = self.get_serializer(completed_tasks, many=True)
        return Response(serializer.data)

# Router automatically generates URLs for these actions:
# POST /tasks/1/mark_complete/  -> mark_complete action
# GET  /tasks/completed/        -> completed action
```

Understanding detail vs non-detail actions:

```python
@action(detail=True)  # Requires task ID in URL
def mark_complete(self, request, pk=None):
    # URL: /tasks/1/mark_complete/
    # Works on specific task instance
    pass

@action(detail=False)  # No task ID in URL
def completed(self, request):
    # URL: /tasks/completed/
    # Works on entire queryset
    pass
```

5.9 Router Configuration Options

Trailing slash configuration:

```python
# Allow URLs with or without trailing slash
router = DefaultRouter()
router.trailing_slash = '/?'  # Optional trailing slash

# Force trailing slash (default)
router.trailing_slash = '/'

# No trailing slash
router.trailing_slash = ''
```

Custom route names:

```python
router.register(r'tasks', TaskViewSet, basename='my-task')
# Generated names: 'my-task-list', 'my-task-detail'

# Use in templates or reverse URL lookups:
from django.urls import reverse
list_url = reverse('my-task-list')
detail_url = reverse('my-task-detail', kwargs={'pk': 1})
```

5.10 Testing Router URLs

Test router URL generation:

python

```python
# tests/test_urls.py
from django.test import TestCase
from django.urls import reverse, resolve
from rest_framework.test import APIClient
from tasks.views import TaskViewSet

class RouterURLTest(TestCase):
    def setUp(self):
        self.client = APIClient()

    def test_task_list_url(self):
        """Test task list URL is correctly generated"""
        url = reverse('task-list')
        self.assertEqual(url, '/api/v1/tasks/')

        # Test URL resolves to correct view
        resolver = resolve('/api/v1/tasks/')
        self.assertEqual(resolver.func.cls, TaskViewSet)

    def test_task_detail_url(self):
        """Test task detail URL with ID"""
        url = reverse('task-detail', args=[1])
        self.assertEqual(url, '/api/v1/tasks/1/')

    def test_custom_action_urls(self):
        """Test custom action URLs are generated"""
        # Non-detail action
        completed_url = reverse('task-completed')
        self.assertEqual(completed_url, '/api/v1/tasks/completed/')

        # Detail action
        mark_complete_url = reverse('task-mark-complete', args=[1])
        self.assertEqual(mark_complete_url, '/api/v1/tasks/1/mark_complete/')

    def test_api_root(self):
        """Test API root view shows registered endpoints"""
        response = self.client.get('/api/v1/')
        self.assertEqual(response.status_code, 200)
        self.assertIn('tasks', response.data)
        self.assertIn('http://testserver/api/v1/tasks/', response.data['tasks'])
```

---

6. Custom Actions

6.1 What Are Custom Actions?

Simple Analogy: Custom actions are like **special menu items** at a restaurant. While the restaurant has standard offerings (appetizers, main courses, desserts = CRUD operations), custom actions are like "Chef's Special" or "Today's Combo" - unique operations that don't fit the standard categories but provide specific functionality your users need.

Technical Definition: Custom actions are additional endpoints you can add to ViewSets beyond the standard CRUD operations. They allow you to implement business logic that doesn't fit into create, read, update, or delete patterns.

6.2 Why Use Custom Actions?

Standard CRUD is limited:

- `list()` - Get all tasks
- `retrieve()` - Get one task
- `create()` - Create task
- `update()` - Update task
- `destroy()` - Delete task

But real applications need more:

- Mark task as complete
- Get task statistics
- Archive old tasks
- Generate reports
- Send notifications
- Batch operations
- AI-powered features

6.3 The @action Decorator

Basic syntax:

```python
from rest_framework.decorators import action
from rest_framework.response import Response

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    @action(detail=True, methods=['post'])
    def mark_complete(self, request, pk=None):
        """Mark a specific task as complete"""
        task = self.get_object()  # Gets task by pk from URL
        task.completed = True
        task.save()

        return Response({
            'status': 'success',
            'message': f'Task "{task.title}" marked as complete'
        })
```

Key parameters explained:

- `detail=True/False`: Does this action work on a specific object (True) or the collection (False)?
- `methods=['post']`: Which HTTP methods are allowed
- `pk=None`: Primary key parameter (automatically provided for detail=True actions)

6.4 Detail vs Non-Detail Actions

Detail Actions (detail=True):

python

```python
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    @action(detail=True, methods=['post'])
    def mark_complete(self, request, pk=None):
        """Works on a SPECIFIC task"""
        task = self.get_object()  # Gets the task with given pk
        task.completed = True
        task.save()
        return Response({'status': 'Task completed'})

    @action(detail=True, methods=['get'])
    def subtasks(self, request, pk=None):
        """Get subtasks for a SPECIFIC task"""
        task = self.get_object()
        subtasks = task.subtasks.all()
        serializer = SubtaskSerializer(subtasks, many=True)
        return Response(serializer.data)

    # URLs generated:
    # POST /tasks/1/mark_complete/
    # GET  /tasks/1/subtasks/
```

Non-Detail Actions (detail=False):

```python
class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    @action(detail=False, methods=['get'])
    def completed(self, request):
        """Works on the ENTIRE collection"""
        completed_tasks = self.get_queryset().filter(completed=True)
        serializer = self.get_serializer(completed_tasks, many=True)
        return Response(serializer.data)

    @action(detail=False, methods=['post'])
    def bulk_complete(self, request):
        """Mark MULTIPLE tasks as complete"""
        task_ids = request.data.get('task_ids', [])
        tasks = self.get_queryset().filter(id__in=task_ids)
        tasks.update(completed=True)

        return Response({
            'status': 'success',
            'completed_count': len(task_ids)
        })

    # URLs generated:
    # GET  /tasks/completed/
    # POST /tasks/bulk_complete/
```

6.5 Different HTTP Methods for Actions

GET Actions (Retrieve data):

python

```python
@action(detail=False, methods=['get'])
def statistics(self, request):
    """Get task statistics"""
    queryset = self.get_queryset()

    stats = {
        'total_tasks': queryset.count(),
        'completed_tasks': queryset.filter(completed=True).count(),
        'pending_tasks': queryset.filter(completed=False).count(),
        'high_priority': queryset.filter(priority='high').count(),
        'overdue_tasks': queryset.filter(
            due_date__lt=timezone.now(),
            completed=False
        ).count(),
    }

    return Response(stats)

@action(detail=True, methods=['get'])
def history(self, request, pk=None):
    """Get change history for a specific task"""
    task = self.get_object()
    # Assuming you have a history tracking system
    history = task.history.all().order_by('-timestamp')
    serializer = TaskHistorySerializer(history, many=True)
    return Response(serializer.data)
```

POST Actions (Create/modify data):

python

```python
@action(detail=True, methods=['post'])
def duplicate(self, request, pk=None):
    """Create a copy of an existing task"""
    original_task = self.get_object()

    # Create new task with same data
    new_task = Task.objects.create(
        title=f"Copy of {original_task.title}",
        description=original_task.description,
        priority=original_task.priority,
        owner=request.user  # Assign to current user
    )

    serializer = self.get_serializer(new_task)
    return Response(serializer.data, status=201)

@action(detail=False, methods=['post'])
def import_from_csv(self, request):
    """Import tasks from CSV file"""
    csv_file = request.FILES.get('csv_file')
    if not csv_file:
        return Response(
            {'error': 'No CSV file provided'},
            status=400
        )

    # Process CSV file
    import csv
    import io

    csv_content = csv_file.read().decode('utf-8')
    csv_reader = csv.DictReader(io.StringIO(csv_content))

    created_tasks = []
    for row in csv_reader:
        task = Task.objects.create(
            title=row['title'],
            description=row.get('description', ''),
            priority=row.get('priority', 'medium'),
            owner=request.user
        )
        created_tasks.append(task)

    return Response({
```

```
        'status': 'success',
        'imported_count': len(created_tasks)
    })
```

PUT/PATCH Actions (Update operations):

python

```
@action(detail=True, methods=['patch'])
def update_priority(self, request, pk=None):
    """Update just the priority of a task"""
    task = self.get_object()
    new_priority = request.data.get('priority')

    if new_priority not in ['low', 'medium', 'high']:
        return Response(
            {'error': 'Invalid priority. Must be low, medium, or high'},
            status=400
        )

    task.priority = new_priority
    task.save()

    serializer = self.get_serializer(task)
    return Response(serializer.data)

@action(detail=False, methods=['patch'])
def bulk_update_priority(self, request):
    """Update priority for multiple tasks"""
    task_ids = request.data.get('task_ids', [])
    new_priority = request.data.get('priority')

    if not task_ids:
        return Response({'error': 'No task IDs provided'}, status=400)

    updated_count = self.get_queryset().filter(
        id__in=task_ids
    ).update(priority=new_priority)

    return Response({
        'status': 'success',
        'updated_count': updated_count
    })
```

DELETE Actions (Remove/archive data):

python

```python
@action(detail=True, methods=['delete'])
def archive(self, request, pk=None):
    """Archive a task instead of deleting it"""
    task = self.get_object()
    task.is_archived = True
    task.archived_at = timezone.now()
    task.save()

    return Response({
        'status': 'success',
        'message': 'Task archived successfully'
    })

@action(detail=False, methods=['delete'])
def clear_completed(self, request):
    """Delete all completed tasks"""
    deleted_count = self.get_queryset().filter(completed=True).count()
    self.get_queryset().filter(completed=True).delete()

    return Response({
        'status': 'success',
        'deleted_count': deleted_count
    })
```

6.6 Real-World Custom Action Examples

AI/Smart Features:

python

```python
@action(detail=True, methods=['get'])
def smart_summary(self, request, pk=None):
    """Generate AI summary for a task (hypothetical AI service)"""
    task = self.get_object()

    # Simulate AI processing
    summary_data = {
        'task_id': task.id,
        'title': task.title,
        'complexity_score': 7.5,
        'estimated_time': '2 hours',
        'suggested_subtasks': [
            'Research requirements',
            'Create initial draft',
            'Review and refine',
            'Final testing'
        ],
        'similar_tasks': Task.objects.filter(
            title__icontains=task.title.split()[0]
        ).exclude(id=task.id).values('id', 'title')[:3]
    }

    return Response(summary_data)

@action(detail=False, methods=['post'])
def ai_prioritize(self, request):
    """Use AI to automatically prioritize tasks"""
    tasks = self.get_queryset().filter(completed=False)

    # Simulate AI prioritization logic
    for task in tasks:
        if 'urgent' in task.title.lower() or 'asap' in task.description.lower():
            task.priority = 'high'
        elif task.due_date and (task.due_date - timezone.now()).days <= 1:
            task.priority = 'high'
        elif 'later' in task.title.lower() or 'someday' in task.description.lower():
            task.priority = 'low'
        else:
            task.priority = 'medium'
        task.save()

    return Response({
        'status': 'success',
        'message': 'Tasks prioritized using AI',
```

```
    'total_processed': tasks.count()
})
```

Reporting and Analytics:

python

```python
@action(detail=False, methods=['get'])
def productivity_report(self, request):
    """Generate productivity report for current user"""
    from datetime import timedelta
    user_tasks = self.get_queryset()

    # Calculate date ranges
    today = timezone.now().date()
    week_ago = today - timedelta(days=7)
    month_ago = today - timedelta(days=30)

    report = {
        'user': request.user.username,
        'report_date': today,
        'summary': {
            'total_tasks': user_tasks.count(),
            'completed_tasks': user_tasks.filter(completed=True).count(),
            'completion_rate': 0,
        },
        'this_week': {
            'created': user_tasks.filter(created_at__gte=week_ago).count(),
            'completed': user_tasks.filter(
                completed=True,
                updated_at__gte=week_ago
            ).count(),
        },
        'this_month': {
            'created': user_tasks.filter(created_at__gte=month_ago).count(),
            'completed': user_tasks.filter(
                completed=True,
                updated_at__gte=month_ago
            ).count(),
        },
        'by_priority': {
            'high': user_tasks.filter(priority='high').count(),
            'medium': user_tasks.filter(priority='medium').count(),
            'low': user_tasks.filter(priority='low').count(),
        }
    }

    # Calculate completion rate
    if report['summary']['total_tasks'] > 0:
        report['summary']['completion_rate'] = round(
            (report['summary']['completed_tasks'] /
```

```
            report['summary']['total_tasks']) * 100, 2
        )

    return Response(report)

@action(detail=False, methods=['get'])
def export_csv(self, request):
    """Export tasks to CSV file"""
    import csv
    from django.http import HttpResponse

    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="tasks.csv"'

    writer = csv.writer(response)
    writer.writerow(['ID', 'Title', 'Description', 'Priority', 'Completed', 'Created'])

    for task in self.get_queryset():
        writer.writerow([
            task.id,
            task.title,
            task.description,
            task.priority,
            'Yes' if task.completed else 'No',
            task.created_at.strftime('%Y-%m-%d %H:%M:%S')
        ])

    return response
```

Batch Operations:

python

```python
@action(detail=False, methods=['post'])
def batch_operations(self, request):
    """Perform batch operations on multiple tasks"""
    operation = request.data.get('operation')
    task_ids = request.data.get('task_ids', [])

    if not task_ids:
        return Response({'error': 'No task IDs provided'}, status=400)

    tasks = self.get_queryset().filter(id__in=task_ids)

    if operation == 'complete':
        tasks.update(completed=True)
        message = f'Marked {len(task_ids)} tasks as complete'

    elif operation == 'archive':
        tasks.update(is_archived=True, archived_at=timezone.now())
        message = f'Archived {len(task_ids)} tasks'

    elif operation == 'delete':
        count = tasks.count()
        tasks.delete()
        message = f'Deleted {count} tasks'

    elif operation == 'set_priority':
        priority = request.data.get('priority')
        if priority not in ['low', 'medium', 'high']:
            return Response({'error': 'Invalid priority'}, status=400)
        tasks.update(priority=priority)
        message = f'Updated priority for {len(task_ids)} tasks'

    else:
        return Response({'error': 'Invalid operation'}, status=400)

    return Response({
        'status': 'success',
        'message': message,
        'processed_count': len(task_ids)
    })
```

6.7 Custom Action Parameters and Validation

Handle query parameters:

python

```python
@action(detail=False, methods=['get'])
def search(self, request):
    """Advanced search with multiple parameters"""
    from django.db.models import Q
    from datetime import datetime

    # Get query parameters
    query = request.query_params.get('q', '')
    priority = request.query_params.get('priority', '')
    completed = request.query_params.get('completed', '')
    date_from = request.query_params.get('date_from', '')
    date_to = request.query_params.get('date_to', '')

    # Start with base queryset
    queryset = self.get_queryset()

    # Apply filters
    if query:
        queryset = queryset.filter(
            Q(title__icontains=query) | Q(description__icontains=query)
        )

    if priority:
        queryset = queryset.filter(priority=priority)

    if completed:
        completed_bool = completed.lower() == 'true'
        queryset = queryset.filter(completed=completed_bool)

    if date_from:
        try:
            from_date = datetime.strptime(date_from, '%Y-%m-%d').date()
            queryset = queryset.filter(created_at__date__gte=from_date)
        except ValueError:
            return Response({'error': 'Invalid date_from format. Use YYYY-MM-DD'}, status=400)

    if date_to:
        try:
            to_date = datetime.strptime(date_to, '%Y-%m-%d').date()
            queryset = queryset.filter(created_at__date__lte=to_date)
        except ValueError:
            return Response({'error': 'Invalid date_to format. Use YYYY-MM-DD'}, status=400)

    # Serialize and return results
```

```
    serializer = self.get_serializer(queryset, many=True)
    return Response({
        'results': serializer.data,
        'count': queryset.count(),
        'filters_applied': {
            'query': query,
            'priority': priority,
            'completed': completed,
            'date_from': date_from,
            'date_to': date_to
        }
    })
```

Request data validation:

python

```python
@action(detail=True, methods=['post'])
def set_reminder(self, request, pk=None):
    """Set a reminder for a task"""
    from datetime import datetime
    task = self.get_object()

    # Validate required fields
    reminder_time = request.data.get('reminder_time')
    reminder_type = request.data.get('reminder_type', 'email')

    if not reminder_time:
        return Response(
            {'error': 'reminder_time is required'},
            status=400
        )

    # Validate reminder_type
    valid_types = ['email', 'sms', 'push']
    if reminder_type not in valid_types:
        return Response(
            {'error': f'reminder_type must be one of: {valid_types}'},
            status=400
        )

    # Validate datetime format
    try:
        reminder_datetime = datetime.fromisoformat(reminder_time.replace('Z', '+00:00'))
    except ValueError:
        return Response(
            {'error': 'Invalid datetime format. Use ISO format: 2023-12-25T10:30:00Z'},
            status=400
        )

    # Check that reminder is in the future
    if reminder_datetime <= timezone.now():
        return Response(
            {'error': 'Reminder time must be in the future'},
            status=400
        )

    # Create reminder (assuming you have a Reminder model)
    # reminder = Reminder.objects.create(
    #     task=task,
    #     reminder_time=reminder_datetime,
```

```
#     reminder_type=reminder_type,
#     created_by=request.user
# )

return Response({
    'status': 'success',
    'message': 'Reminder set successfully',
    'reminder_time': reminder_datetime.isoformat()
})
```

6.8 Custom Serializers for Actions

Use different serializers for actions:

python

```python
class TaskStatusSerializer(serializers.Serializer):
    """Simple serializer for status updates"""
    status = serializers.CharField()
    message = serializers.CharField()

class TaskStatsSerializer(serializers.Serializer):
    """Serializer for task statistics"""
    total_tasks = serializers.IntegerField()
    completed_tasks = serializers.IntegerField()
    completion_rate = serializers.FloatField()

class BulkCompleteSerializer(serializers.Serializer):
    """Serializer for bulk complete action"""
    task_ids = serializers.ListField(
        child=serializers.IntegerField(),
        min_length=1,
        help_text="List of task IDs to mark as complete"
    )
    notify_user = serializers.BooleanField(
        default=True,
        help_text="Send notification to user about completed tasks"
    )

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    @action(detail=True, methods=['post'])
    def mark_complete(self, request, pk=None):
        """Mark task as complete with custom response serializer"""
        task = self.get_object()
        task.completed = True
        task.completed_at = timezone.now()
        task.save()

        # Use custom serializer for response
        response_data = {
            'status': 'success',
            'message': f'Task "{task.title}" marked as complete'
        }
        serializer = TaskStatusSerializer(response_data)
        return Response(serializer.data)

    @action(detail=False, methods=['get'])
```

```python
def statistics(self, request):
    """Get statistics with custom serializer"""
    queryset = self.get_queryset()

    stats_data = {
        'total_tasks': queryset.count(),
        'completed_tasks': queryset.filter(completed=True).count(),
        'completion_rate': 0.0
    }

    if stats_data['total_tasks'] > 0:
        stats_data['completion_rate'] = (
            stats_data['completed_tasks'] / stats_data['total_tasks']
        ) * 100

    serializer = TaskStatsSerializer(stats_data)
    return Response(serializer.data)

@action(detail=False, methods=['post'])
def bulk_complete(self, request):
    """Bulk complete tasks with input validation"""
    # Use serializer for input validation
    input_serializer = BulkCompleteSerializer(data=request.data)

    if not input_serializer.is_valid():
        return Response(input_serializer.errors, status=400)

    # Extract validated data
    task_ids = input_serializer.validated_data['task_ids']
    notify_user = input_serializer.validated_data['notify_user']

    # Perform bulk operation
    tasks = self.get_queryset().filter(id__in=task_ids)
    updated_count = tasks.update(
        completed=True,
        completed_at=timezone.now()
    )

    # Send notification if requested
    if notify_user and updated_count > 0:
        # Send notification logic here
        pass

    return Response({
        'status': 'success',
        'completed_count': updated_count,
        'notification_sent': notify_user
```

```
})
```

6.9 Permissions for Custom Actions

Action-specific permissions:

python

```python
from rest_framework.permissions import IsAuthenticated

class TaskViewSet(viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer

    def get_permissions(self):
        """Set permissions based on action"""
        if self.action in ['mark_complete', 'set_reminder']:
            # Only task owner can mark complete or set reminders
            permission_classes = [IsAuthenticated]  # Add custom IsTaskOwner here
        elif self.action in ['bulk_delete', 'admin_report']:
            # Only admin users can perform bulk operations
            permission_classes = [IsAuthenticated]  # Add IsAdminUser here
        elif self.action == 'statistics':
            # Anyone can view statistics
            permission_classes = [IsAuthenticated]
        else:
            # Default permissions for CRUD operations
            permission_classes = [IsAuthenticated]

        return [permission() for permission in permission_classes]

    @action(detail=True, methods=['post'])
    def mark_complete(self, request, pk=None):
        """Only task owner can mark as complete"""
        task = self.get_object()

        # Check if user is task owner
        if task.owner != request.user:
            return Response(
                {'error': 'You can only mark your own tasks as complete'},
                status=403
            )

        task.completed = True
        task.save()
        return Response({'status': 'Task marked as complete'})
```

6.10 Error Handling in Custom Actions

Comprehensive error handling:

python

```python
@action(detail=True, methods=['post'])
def assign_to_user(self, request, pk=None):
    """Assign task to another user with proper error handling"""
    try:
        task = self.get_object()
    except Task.DoesNotExist:
        return Response(
            {'error': 'Task not found'},
            status=status.HTTP_404_NOT_FOUND
        )

    # Check if user has permission to assign tasks
    if task.owner != request.user and not request.user.is_staff:
        return Response(
            {'error': 'You can only assign your own tasks'},
            status=status.HTTP_403_FORBIDDEN
        )

    # Validate input
    assignee_id = request.data.get('assignee_id')
    if not assignee_id:
        return Response(
            {'error': 'assignee_id is required'},
            status=status.HTTP_400_BAD_REQUEST
        )

    try:
        from django.contrib.auth.models import User
        assignee = User.objects.get(id=assignee_id)
    except User.DoesNotExist:
        return Response(
            {'error': 'Assigned user not found'},
            status=status.HTTP_400_BAD_REQUEST
        )

    # Check business rules
    if task.completed:
        return Response(
            {'error': 'Cannot assign completed tasks'},
            status=status.HTTP_400_BAD_REQUEST
        )

    if assignee == task.owner:
        return Response(
```

```
            {'error': 'Task is already assigned to this user'},
            status=status.HTTP_400_BAD_REQUEST
        )

    try:
        # Perform the assignment
        old_owner = task.owner
        task.owner = assignee
        task.assigned_at = timezone.now()
        task.assigned_by = request.user
        task.save()

        return Response({
            'status': 'success',
            'message': f'Task assigned to {assignee.username}',
            'task_id': task.id,
            'new_owner': assignee.username
        })

    except Exception as e:
        # Log the error for debugging
        import logging
        logger = logging.getLogger(__name__)
        logger.error(f"Error assigning task {task.id}: {str(e)}")

        return Response(
            {'error': 'An unexpected error occurred'},
            status=status.HTTP_500_INTERNAL_SERVER_ERROR
        )
```

6.11 Testing Custom Actions

Test custom actions thoroughly:

python

```python
from django.test import TestCase
from rest_framework.test import APIClient
from rest_framework import status
from django.contrib.auth.models import User
from .models import Task

class TaskCustomActionsTest(TestCase):
    def setUp(self):
        self.client = APIClient()
        self.user = User.objects.create_user('test', 'test@example.com', 'pass')
        self.other_user = User.objects.create_user('other', 'other@example.com', 'pass')
        self.client.force_authenticate(user=self.user)

        self.task = Task.objects.create(
            title='Test Task',
            description='Test Description',
            owner=self.user,
            completed=False
        )

    def test_mark_complete_success(self):
        """Test successful task completion"""
        url = f'/api/v1/tasks/{self.task.id}/mark_complete/'
        response = self.client.post(url)

        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['status'], 'success')

        # Verify task is actually completed
        self.task.refresh_from_db()
        self.assertTrue(self.task.completed)

    def test_mark_complete_not_owner(self):
        """Test that only owner can mark task complete"""
        # Switch to other user
        self.client.force_authenticate(user=self.other_user)

        url = f'/api/v1/tasks/{self.task.id}/mark_complete/'
        response = self.client.post(url)

        self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)

        # Verify task is still not completed
        self.task.refresh_from_db()
```

```
        self.assertFalse(self.task.completed)

    def test_statistics_action(self):
        """Test statistics endpoint"""
        # Create additional tasks
        Task.objects.create(title='Task 2', owner=self.user, completed=True)
        Task.objects.create(title='Task 3', owner=self.user, completed=False)

        url = '/api/v1/tasks/statistics/'
        response = self.client.get(url)

        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['total_tasks'], 3)
        self.assertEqual(response.data['completed_tasks'], 1)
        self.assertAlmostEqual(response.data['completion_rate'], 33.33, places=2)

    def test_bulk_complete_action(self):
        """Test bulk completion with validation"""
        # Create more tasks
        task2 = Task.objects.create(title='Task 2', owner=self.user)
        task3 = Task.objects.create(title='Task 3', owner=self.user)

        url = '/api/v1/tasks/bulk_complete/'
        data = {
            'task_ids': [self.task.id, task2.id, task3.id],
            'notify_user': True
        }
        response = self.client.post(url, data, format='json')

        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['completed_count'], 3)

        # Verify all tasks are completed
        for task in [self.task, task2, task3]:
            task.refresh_from_db()
            self.assertTrue(task.completed)

    def test_bulk_complete_invalid_data(self):
        """Test bulk complete with invalid input"""
        url = '/api/v1/tasks/bulk_complete/'
        data = {
            'task_ids': [],  # Empty list should fail validation
            'notify_user': 'invalid'  # Invalid boolean
        }
        response = self.client.post(url, data, format='json')

        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

```
        self.assertIn('task_ids', response.data)
```

6.12 Best Practices for Custom Actions

1. Use clear, descriptive action names:

```python
# Good - clear what the action does
@action(detail=True, methods=['post'])
def mark_complete(self, request, pk=None):
    pass

@action(detail=False, methods=['get'])
def export_csv(self, request):
    pass

# Bad - unclear action names
@action(detail=True, methods=['post'])
def do_something(self, request, pk=None):
    pass
```

2. Choose appropriate HTTP methods:

```python
# GET for retrieving data
@action(detail=False, methods=['get'])
def statistics(self, request):
    pass

# POST for creating/triggering actions
@action(detail=True, methods=['post'])
def mark_complete(self, request, pk=None):
    pass

# DELETE for removal/cleanup actions
@action(detail=False, methods=['delete'])
def clear_completed(self, request):
    pass
```

3. Proper error handling and validation:

python

```python
@action(detail=True, methods=['post'])
def well_designed_action(self, request, pk=None):
    # 1. Validate permissions first
    if not request.user.is_authenticated:
        return Response({'error': 'Authentication required'}, status=401)

    # 2. Validate input data
    required_field = request.data.get('required_field')
    if not required_field:
        return Response({'error': 'required_field is missing'}, status=400)

    # 3. Handle object not found
    try:
        obj = self.get_object()
    except Task.DoesNotExist:
        return Response({'error': 'Task not found'}, status=404)

    # 4. Business logic validation
    if obj.completed:
        return Response({'error': 'Cannot modify completed tasks'}, status=400)

    # 5. Perform the action with error handling
    try:
        # Your business logic here
        result = "operation_completed"
        return Response({'status': 'success', 'result': result})
    except Exception as e:
        return Response({'error': 'Operation failed'}, status=500)
```

4. Consistent response format:

python

```python
# Maintain consistent response structure
def success_response(message, data=None):
    response = {'status': 'success', 'message': message}
    if data:
        response['data'] = data
    return Response(response)

def error_response(message, status_code=400):
    return Response({'status': 'error', 'message': message}, status=status_code)
```

Summary

This comprehensive Django REST Framework tutorial covers:

1. **Project & App Creation** - Setting up Django projects and organizing apps
2. **Models** - Defining data structures with fields, relationships, and methods
3. **Serializers** - Converting between Python objects and JSON with validation
4. **ViewSets** - Handling HTTP requests with automatic CRUD operations
5. **Routers** - Automatically generating URL patterns for your ViewSets
6. **Custom Actions** - Adding specialized endpoints beyond standard CRUD

Each component works together to create powerful, well-structured REST APIs. The patterns and practices covered here provide a solid foundation for building professional Django REST Framework applications.

Key Takeaways:

- Start with proper project structure and app organization
- Use ModelSerializer for most cases, plain Serializer for complex logic
- ModelViewSet handles 90% of API needs automatically
- DefaultRouter generates clean, RESTful URLs
- Custom actions extend functionality beyond basic CRUD
- Always validate input and handle errors gracefully
- Test your API endpoints