Django Rest Framework

Django REST Framework (DRF) is a powerful library built on top of Django that helps you create RESTful APIs easily.

DRF helps you convert your Django models and views into API endpoints that return data in JSON format — perfect for mobile apps, frontend frameworks (like React/Vue), or any external service.

Why Use DRF?

- Easy to build APIs for frontend apps (React, Vue, etc.)
- Supports authentication, permissions, filtering, pagination
- Clean, modular, and secure

Official document : https://www.django-rest-framework.org/

Step-by-Step Installation Process of DRF

Step 1: Install Django

```
pip install django
```

Step 2: Create a Django project

```
django-admin startproject myproject
cd myproject
```

Step 3: Create an app (e.g., movies)

```
python manage.py startapp movies
```

Step 4: Install Django REST Framework

```
pip install djangorestframework
```

Step 5: Add 'rest_framework' to INSTALLED_APPS in settings.py

```
# myproject/settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',   # Add this
```

```
    'movies',           # Your app
]
```

api/ folder

In a Django + Django REST Framework (DRF) project, creating an api/ folder (or package) is not required, but it is considered a good practice in larger or well-structured projects.

A typical Django app has this structure:

```
myapp/
├── admin.py
├── apps.py
├── models.py
├── views.py
├── urls.py
├── serializers.py
├── tests.py
```

But when you start separating logic for API endpoints (especially with DRF), your app can look like this:

```
myapp/
├── api/
│   ├── __init__.py
│   ├── views.py
│   ├── serializers.py
│   ├── urls.py
│   └── permissions.py
├── models.py
├── admin.py
├── apps.py
├── tests.py
```

In huge projects, you can go even further:

```
myapp/
├── api/
│   ├── v1/
│   │   ├── views.py
│   │   ├── serializers.py
│   │   ├── urls.py
│   │   └── permissions.py
│   └── v2/   # for future API versioning
├── models.py
...
```

Reason/Benifits to use api/ :

- Separate API logic : Cleaner organization
- Scalability : Easier to manage multiple views
- DRF best practices : Helps structure serializers/views
- Versioning ready : Easy to support v1, v2, etc.

What is Serializer?

Serialization is the process of converting a Python object (like a dictionary, list, or class instance) into a JSON string.

```
json.dumps() – Python object to JSON string
```

What is Deserializer?

Deserialization is the opposite, it takes the JSON string and converts it back into the original Python object.

```
json.loads() – JSON string back to Python object
```

Example

```
import json

# A Python dictionary (original object)
data = {
    "name": "Alice",
    "age": 30,
    "is_student": False
}

# Serialization: Convert Python dict to JSON string
json_string = json.dumps(data)
print("Serialized JSON string:", json_string)

# Deserialization: Convert JSON string back to Python dict
original_data = json.loads(json_string)
print("Deserialized Python object:", original_data)
```

What is serializers.py in DRF

In Django, especially in Django REST Framework (DRF), a serializer is used to convert complex data types like Django models into native Python datatypes (like dict) that can then be rendered into JSON, XML, etc.

- Serialization: Convert Django model instances or other Python objects to JSON (or other formats).
- Deserialization: Convert JSON data from API requests into Python objects.

- Validation: Ensure incoming data is valid before saving to the database.

```
myapp/
├── models.py        # Your Django models
├── views.py         # Your API views
├── serializers.py   # Serializers for converting data
├── urls.py          # Routes/URLs for API
```

There are two types of serializers :

1. serializers.Serializer
2. serializers.ModelSerializer

serializers.Serializer and serializers.ModelSerializer in DRF

serializers.Serializer

A serializer where you define each field yourself and tell it how to create or update objects.It involve manual process.

ex : Filling out a form manually — you choose what fields to include and what to do with the data.

Use it when:

- You're not using Django models (just plain data).
- You want custom control over the logic.
- You need to handle complex or non-standard data.

```
# models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_year = models.IntegerField()

----------------------------------------------------------------

# serializers.py
# Import the base Serializer class and field types from Django REST Framework
from rest_framework import serializers

# Define a serializer for a Book object
class BookSerializer(serializers.Serializer):
    # Define each field manually
    title = serializers.CharField(max_length=100)              # String field with
max 100 characters
    author = serializers.CharField(max_length=100)             # String field with
max 100 characters
    published_year = serializers.IntegerField()                # Integer field for
```

```
year

    # This method is called when creating a new Book instance from validated data
    def create(self, validated_data):
        # Assumes there's a Book model defined elsewhere
        # **validated_data unpacks the dict into keyword arguments
        return Book.objects.create(**validated_data)

    # This method is called when updating an existing Book instance
    def update(self, instance, validated_data):
        # Get each field from the input data; if it's not provided, keep the
current value
        instance.title = validated_data.get('title', instance.title)
        instance.author = validated_data.get('author', instance.author)
        instance.published_year = validated_data.get('published_year',
instance.published_year)

        # Save the updated instance to the database
        instance.save()

        # Return the updated object
        return instance
```

serializers.ModelSerializer

A serializer that automatically creates fields from your Django model and handles saving and updating.

ex : A smart form that already knows what fields to use because it looks at your model.

Use it when:

- You are working with a Django model (e.g., Book, User).
- You want to quickly build an API.
- You don't need custom serialization logic.

```
# models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_year = models.IntegerField()

----------------------------------------------------------------

# serializers.py
from rest_framework import serializers
from .models import Book  # Assume this is the Django model you're serializing

# This serializer automatically maps model fields to serializer fields
class BookSerializer(serializers.ModelSerializer):
    # Meta class tells DRF how to build the serializer
```

```
    class Meta:
        model = Book  # Connect this serializer to the Book model
        fields = ['id', 'title', 'author', 'published_year']
        # 'fields' lists all the fields you want to include in the API
response/input
```

Note :

Feature	`serializers.Serializer` **(Manual)**	`serializers.ModelSerializer` **(Automatic)**
Field definitions	You write each field manually	Auto-generated from model
`create()` / `update()`	You must write them manually	Automatically handled unless overridden
Boilerplate code	More code to write	Much less code
Flexibility	Full control over everything	Can be customized, but meant for quick model-to-API use
Use case	Use for custom data or when not using a model	Use when working directly with a Django model

What is views.py?

It's a Python file where you write views that handle HTTP methods like GET, POST, PUT, DELETE. Just like Django's regular views handle HTML requests, DRF views handle API (JSON) requests.

DRF provides several types of views to choose from, depending on your needs.

View Type	Description
`APIView`	Basic class-based view (full control)
`GenericAPIView`	Adds model + serializer support
Mixins (`CreateModelMixin`, etc.)	Add reusable logic for CRUD
`ViewSet` / `ModelViewSet`	Group multiple actions (`list`, `create`, etc.) in one class

In Django REST Framework (DRF), you can write views in two main styles:

1. Function-Based Views (FBV) — using @api_view()
2. Class-Based Views (CBV) — using APIView

Function-Based Views (FBV) — using @api_view()

This is the simplest way to write an API. It's just a Python function.

When to use FBV:

- You're building small or simple APIs.
- You want quick and readable logic.

- Great for beginners.

```
# Import the decorator to make a function-based API view that supports specific
HTTP methods
from rest_framework.decorators import api_view

# Import Response class to return HTTP responses in JSON format
from rest_framework.response import Response

# Import your Book model (assumed to be defined elsewhere)
from .models import Book

# Import the serializer that converts Book instances to/from JSON
from .serializers import BookSerializer

# Use the @api_view decorator to specify this view accepts GET and POST requests
only
@api_view(['GET', 'POST'])
def book_list(request):
    # Handle GET request to list all books
    if request.method == 'GET':
        # Retrieve all Book objects from the database
        books = Book.objects.all()

        # Serialize the queryset to JSON format
        # many=True means we expect multiple Book objects
        serializer = BookSerializer(books, many=True)

        # Return serialized data in an HTTP response
        return Response(serializer.data)

    # Handle POST request to create a new book
    elif request.method == 'POST':
        # Deserialize JSON data from request to BookSerializer for validation
        serializer = BookSerializer(data=request.data)

        # Check if data is valid according to serializer rules
        if serializer.is_valid():
            # Save the new Book instance to the database
            serializer.save()

            # Return the saved data with HTTP 201 Created status
            return Response(serializer.data, status=201)

        # If data invalid, return errors with HTTP 400 Bad Request status
        return Response(serializer.errors, status=400)
```

Class-Based Views (CBV) — APIView

This is a more structured and reusable way using classes.

When to use CBV:

- You want to organize logic better (especially for large projects).
- You want to reuse methods by inheritance.
- Better for adding permissions, throttling, etc., in a clean way.

```
# Import the decorator to define allowed HTTP methods on a function-based view
from rest_framework.decorators import api_view

# Import the Response object to send back JSON responses
from rest_framework.response import Response

# Import your Django model (Book) — represents data from the database
from .models import Book

# Import the serializer to convert between model instances and JSON
from .serializers import BookSerializer

# Define a function-based API view that accepts GET and POST requests
@api_view(['GET', 'POST'])
def book_list(request):
    # ------------------- HANDLE GET REQUEST -------------------
    if request.method == 'GET':
        # Fetch all Book records from the database
        books = Book.objects.all()

        # Serialize all book objects into Python data (list of dicts)
        # many=True tells the serializer to handle multiple objects
        serializer = BookSerializer(books, many=True)

        # Return serialized data as a JSON response
        return Response(serializer.data)

    # ------------------- HANDLE POST REQUEST -------------------
    elif request.method == 'POST':
        # Deserialize incoming JSON request data into a Book object
        serializer = BookSerializer(data=request.data)

        # Check if the incoming data is valid according to serializer rules
        if serializer.is_valid():
            # Save the new book to the database
            serializer.save()

            # Return the saved book data with HTTP status 201 (Created)
            return Response(serializer.data, status=201)

        # If the data is invalid, return validation error messages with 400 (Bad
Request)
        return Response(serializer.errors, status=400)
```

What Are HTTP Status Codes?

Status codes are standardized numbers sent in HTTP responses to tell the client what happened.

Code Range	Category	Meaning
1xx	Informational	Request received, continuing process
2xx	Success	The request was successful
3xx	Redirection	Further action needs to be taken
4xx	Client Error	The request was incorrect (your fault)
5xx	Server Error	The server failed (server's fault)

Common status sodes in DRF

Code	Name	When to Use
200	`HTTP_200_OK`	Standard response for success (`GET`, `PUT`)
201	`HTTP_201_CREATED`	New resource was created (`POST`)
204	`HTTP_204_NO_CONTENT`	Successful, but no content to return (e.g., `DELETE`)
400	`HTTP_400_BAD_REQUEST`	Client sent invalid data
401	`HTTP_401_UNAUTHORIZED`	User is not authenticated
403	`HTTP_403_FORBIDDEN`	Authenticated but not allowed to do the action
404	`HTTP_404_NOT_FOUND`	Resource not found
500	`HTTP_500_INTERNAL_SERVER_ERROR`	Server crashed or misconfigured