1. Django Rest Framework

Introduction to Django Rest Framework

Django REST framework (DRF) is a powerful and elegant toolkit built on top of the Django web framework. It simplifies the process of creating robust and well-structured RESTful APIs.

What is an API?

An API (Application Programming Interface) is a method for applications to communicate with each other. It defines how data can be requested, manipulated, and exchanged.

Why Use DRF?

DRF offers several advantages for building APIs:

Rapid Development: DRF provides pre-built components and conventions, saving you time and effort.

Flexibility: It supports various data formats, authentication mechanisms, and view patterns.

Scalability: DRF handles complex APIs efficiently, making them maintainable for long-term use.

Django rest framework setup

Prerequisites:

Python: Ensure you have Python installed on your system. You can check by running python --version in your terminal.

Django: You should have a basic understanding of Django web development.

Steps To Create a Project

1. Create a Virtual Environment :

Using a virtual environment helps isolate project dependencies and avoid conflicts with other Python projects on your system.

Tools like venv (built-in) or virtualenv can be used to create a virtual environment. Refer to

documentation for specific commands.

2. Install Django and DRF:

Open your terminal or command prompt and activate your virtual environment (if created).

Install Django and DRF using the pip package manager:

pip install django djangorestframework

3. Create a New Django Project:

Use the django-admin command to create a new project:

django-admin startproject myproject

4. Create a Django App:

Within your project directory, create a new app for your API using:

cd myproject

django-admin startapp myapi

5. Configure Django Settings:

Open your project's settings.py file and add the following:

INSTALLED_APPS: Include 'rest_framework' in the INSTALLED_APPS list.

6. Add Your App to Project URLs:

In your project's urls.py, import the include function from django.urls and add a pattern to include your app's URLs:

```
from django.urls import path, include

    urlpatterns = [

    path('api/', include('myapi.urls')),

     ]
```

7. Create Your API Endpoints:

Within your app's urls.py, import the api_view decorator from rest_framework.decorators and define URLs for your API endpoints.

```
    from rest_framework.decorators import api_view
```

```
@api_view(['GET'])
def hello_world(request):return
Response({'message': 'Hello, World!'})
urlpatterns = [
path('hello/', hello_world),
]
```

8. Run Your Development Server:

Navigate to your project directory and run the Django development server:

Run These Command

```
python manage.pu migrate
python manage.py runserver
```

API view decorators

In Django REST framework (DRF), API view decorators are a convenient way to define views that handle API requests.

They simplify the process of associating an HTTP method (GET, POST, PUT, etc.) with a specific function or class-based view.

Common Decorators:

@api_view(['GET']): This decorator marks a function or class as an API view and specifies that it handles GET requests. You can include a list of allowed HTTP methods within square brackets (e.g., ['GET', 'POST']).

@api_view(): This decorator allows the view to handle any HTTP method (GET, POST, PUT, DELETE, etc.) by default. However, it's generally recommended to be more explicit about allowed methods.

Here's an example demonstrating **@api_view**:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
```

```
@api_view(['GET'])
def hello_world(request):
    return Response({'message': 'Hello, World!'})
```

GET, POST, PUT method in API view decorators

Common Decorators and Methods:

1.@api_view(['GET'])

This decorator marks a view as handling GET requests. GET requests are typically used to retrieve data from a server.

2.@api_view(['POST'])

This decorator marks a view as handling POST requests. POST requests are commonly used to create new resources on the server.

3.@api_view(['PUT'])

This decorator marks a view as handling PUT requests. PUT requests are used to completely replace a resource on the server with the provided data.

4.@api_view(['DELETE'])

This decorator marks a view as handling DELETE requests. DELETE requests are used to remove a resource from the server.

Explanation:

get_products handles GET requests to retrieve a list of products.

create_product handles POST requests to create a new product.

update_product handles PUT requests to update an existing product by its ID.

delete_product handles DELETE requests to remove a product by its ID.

Creating model

Create a Models File:

Within your Django app directory (e.g., myapp), create a new Python file named models.py. This file will house your model definitions.

Define Your Model Class:

In models.py, import the models module from django.db.

Define a model class inheriting from models.Model.

Within the class, use fields from django.db.models to define the data structure for your model.

Here are some common field types:

CharField, TextField, IntegerField, BooleanField, DateField, ForeignKey. and more

Example:

```
from django.db import models

class Product(models.Model):
    name = CharField(max_length=255)
    description = TextField()
    price = DecimalField(max_digits=10, decimal_places=2)
    stock = PositiveIntegerField()
```

Migrate Your Database:

Run the following command in your terminal to create the database tables based on your model definitions:

```
python manage.py makemigrations
python manage.py migrate
```

Writing serialisers

Serializers in Django REST framework (DRF) play a crucial role in data exchange between your API and other applications. They act as intermediaries, converting data between Python objects (like model instances) and various formats like JSON, XML, or YAML.

ModelSerializer: If your serializer directly maps to a Django model, inherit from rest_framework.serializers.ModelSerializer. This class automatically generates serializer fields based on your model fields.

Serializer: If you're dealing with custom data structures that don't correspond to a model, inherit from rest_framework.serializers.Serializer. You'll need to define each field manually in this case.

Example:

Writing a ModelSerializer:

```
from rest_framework.serializers import ModelSerializer

from .models import Product

class ProductSerializer(ModelSerializer):

    class Meta:

        model = Product

        fields = '__all__'
```

Explanation:

We inherit from ModelSerializer and specify the model (Product) we're serializing.

The Meta class defines configuration for the serializer:

model: The model the serializer interacts with.

fields: A list of model fields to include in the serialized representation (you can also specify specific fields instead of '__all__').

Writing a Custom Serializer:

```
from rest_framework.serializers import Serializer
```

```
class UserRatingSerializer(Serializer):
    user_id = IntegerField()
    rating = DecimalField(max_digits=2, decimal_places=1)
    comment = CharField(max_length=255, required=False)
```

Explanation:

We inherit from Serializer and define each field manually.

We specify the field type, maximum length and whether the field is required.

CRUD method using serializers

1. Create Operation (POST):

A client sends a POST request with data in the desired format (e.g., JSON) to the appropriate API endpoint.

The view receives the request and deserializes the data using the appropriate serializer.

The serializer validates the incoming data based on its defined rules.

If validation passes, the serializer uses the validated data to create a new object in the database (assuming it maps to a model).

The view serializes the newly created object again using the serializer and returns it as a response.

2. Read Operation (GET):

A client sends a GET request to an API endpoint representing a specific resource (e.g., /products/123) or a collection of resources (e.g., /products/).

The view retrieves the requested data from the database based on the endpoint URL (e.g., by ID).

The view serializes the retrieved data using the appropriate serializer.

The serialized data is returned as a response.

3. Update Operation (PUT):

A client sends a PUT request with updated data in the desired format to the API endpoint representing a specific resource (e.g., /products/123).

The view deserializes the request data using the serializer.

The serializer validates the data and updates the corresponding object in the database with the validated data.

The view serializes the updated object again and returns it as a response.

4. Delete Operation (DELETE):

A client sends a DELETE request to the API endpoint representing a specific resource (e.g., /products/123).

The view retrieves the object based on the ID (or other identifier) from the URL.

The object is deleted from the database.

The view returns a success response (e.g., status code 204 No Content).

Validation in serializers

Built-in validators: Enforce rules like required fields, maximum lengths, and valid data types.

Custom validation: Create your own validation logic for specific needs (e.g., checking for duplicate emails).

Serializing foreign key in Django rest framework

Foreign Keys: When models have relationships (e.g., a Product model might have a foreign key to a Category model), you need to handle these relationships during serialization.

Common Approaches:

PrimaryKeyRelatedField:

Use this if you only need the ID of the related object.

It's simple and efficient for retrieving basic information.

SlugRelatedField:

If your related model has a unique slug field, use this to serialize by slug instead of ID.

This can be useful for URLs or user-friendly representations.

APIView class in DRF

APIView :

APIView is a class-based view specifically designed for building RESTful APIs in DRF.

It inherits from Django's View class and provides functionalities tailored for API development.

ModelViewSet in DRF

ModelViewSet is a powerful class that simplifies building API views for models. It inherits from GenericViewSet and provides a set of default actions that handle common CRUD (Create, Read, Update, Delete) operations on your model data.

Status code in DRF

Status codes play a crucial role in communication between your Django REST framework (DRF) API and clients. They provide informative messages indicating the outcome of API requests.

Understanding HTTP Status Codes:

Informational: These codes indicate informational responses, typically used for progress updates (e.g., 100 Continue).

Success: These codes signify successful requests (e.g., 200 OK, 201 Created).

Redirection: These codes indicate redirection (e.g., 301 Moved Permanently).

Client Error: These codes indicate errors caused by the client (e.g., 400 Bad Request, 404 Not Found).

Server Error: These codes indicate errors on the server side (e.g., 500 Internal Server Error).

Token Authentication in DRF

Token authentication is a widely used approach for securing APIs in Django REST framework (DRF). It involves exchanging a username and password for a unique token that clients use to authenticate subsequent requests

Permissions in DRF

Common Permission Classes in DRF:

AllowAny: Grants access to anyone, regardless of authentication status (use with caution in production).

IsAuthenticated: Requires users to be authenticated to access the endpoint.

IsAdminUser: Restricts access to authenticated users with admin privileges.

Example :

```
from rest_framework.permissions import IsAuthenticatedOrReadOnly

class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]
```

Summary of the course

API Concepts: The role of APIs in data exchange and how DRF simplifies API creation in Django.

DRF Setup: Installation and configuration steps for using DRF in your project.

API Views: Using API views (like APIView) to handle requests and responses.

Decorators: Specifying allowed HTTP methods (GET, POST, PUT, etc.) for your views.

CRUD Operations: Performing Create, Read, Update, Delete operations on data using serializers.

Serializers: Converting data between formats (e.g., models to JSON) and validating data during CRUD.

Serializer Techniques: Advanced features like method fields, serializer permissions, and handling foreign keys.

ModelViewSet: Simplifying API view creation for models with default CRUD actions.

Status Codes: Understanding and using HTTP status codes for clear API communication.

Token Authentication: Implementing token-based authentication for securing your API.

Permissions: Controlling user access to API endpoints using different permission classes.

Thank you!