**PART A**

**Experiment No. 09**

**A.1 Aim:** Employee Management System using Django REST Framework (**DRF )**

**Objective:** To learn and understand how use CRUD operations in DRF

**A.2 Prerequisite:** HTML, CSS, Javascript, CURD in Django, Postman or Thunder Client

**A.3 Outcome:**

After successful completion of these applications students will be able to understand Django DRF Framework

**A.4 Theory: CRUD Operations in Django using DRF**

CRUD stands for **Create**, **Read**, **Update**, and **Delete**, which are the four basic operations used to manage data in a database cane be handled by the function in views.py using DRF

**Steps for Setting up Django REST Framework (DRF)**

1. **Install DRF**: pip install django**restframework**
2. **Configure Django Settings**:
   1. Add 'rest\_framework' to your INSTALLED\_APPS list in settings.py:

INSTALLED\_APPS = [

...,

'rest\_framework',

]

1. **Create Serializer for Already created Employee Model:** In myapp/**serializers**.py:

**from rest\_framework import serializers**

from .models import Employee

class EmployeeSerializer(serializers.ModelSerializer):

    class Meta:

        model = Employee

        fields = ['id', 'name', 'salary']

1. **Update Views with API Endpoints:** In myapp/views.py:

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework import status

from .models import **Employee**

from .serializers import **EmployeeSerializer**

# List Employees or Create new Employee

@api\_view(['GET', 'POST'])

def employee\_list(request):

if request.method == 'GET':

employees = Employee.objects.all()

serializer = EmployeeSerializer(employees, many=True)

return Response(serializer.data)…..

….

**Scenario Problem statement 1: Employee Management System Using Django DRF**

"Design and implement a Django-based REST API for managing employee records. This API should allow clients to perform CRUD operations on employee data.

Hints:

myproject/ # Root directory of your Django project

│

├── myproject/ # Django settings and configuration files

│ ├── \_\_init\_\_.py

│ ├── settings.py # Project settings

│ ├── urls.py # Root URL configuration

│ ├── asgi.py

│ └── wsgi.py

│

├── myapp3/ # Application directory (the app for employee management)

│ ├── \_\_init\_\_.py

│ ├── admin.py

│ ├── apps.py # Application configuration

│ ├── forms.py # Django forms for Employee model

│ ├── migrations/ # Database migrations for the app

│ │ └── \_\_init\_\_.py

│ ├── models.py # Employee model definition

│ ├── views.py # Application views (CRUD and employee-related operations)

│ ├── urls.py # URL routing for the app (employee CRUD operations)

│ └── templates/

│ ├── employee\_form.html # Template for employee creation and update

│ ├── employee\_list.html # Template to display employee list

│ └── employee\_confirm\_delete.html # Template for employee deletion confirmation

│

├── manage.py

└── db.sqlite3

**Sampe 2. URL Configuration (myapp/urls.py)**

from django.urls import path

from . import views

urlpatterns = [

    path('api/employees/', views.employee\_list, name='employee\_list\_api'),  # List or Create

    path('api/employees/<int:id>/', views.employee\_detail, name='employee\_detail\_api'),  # **Get, Update or Delete**

]

Note: Just by using the 2 uri you will be able to perform the basic crud operations

**Problem statement 2: Testing with Thunder Client or Postman**

Test the API using Thunder Client or Postman for verifying GET, POST, PUT, and DELETE requests.

**Create a new employee:**

Method: **POST**

URL: http://127.0.0.1:8000/myapp/api/employees/

Request Body:

{

"name": "Tom",

"salary": 50000.00

}

**List all employees:**

Method: GET

URL: http://127.0.0.1:8000/myapp/api/employees/

**Retrieve an employee by ID:**

Method: GET

URL: http://127.0.0.1:8000/myapp/api/employees/1/

**Update an employee by ID:**

Method: PUT

URL: http://127.0.0.1:8000/myapp/api/employees/1/

Request Body:

{

"name": "Jerry",

"salary": 60000.00

}

**Delete an employee by ID:**

Method: DELETE

URL: <http://127.0.0.1:8000/myapp/api/employees/1/>

Directory Structure of the Complete Django App

myproject/

│

├── manage.py

├── myproject/ # Project settings directory

│ ├── \_\_init\_\_.py

│ ├── settings.py # Django settings

│ ├── urls.py # Project-level URL routing

│ └── wsgi.py

│

├── myapp/ # Your Django app (e.g., Employee management)

│ ├── \_\_init\_\_.py

│ ├── admin.py

│ ├── apps.py

│ ├── models.py # Define Employee model here

**│ ├── serializers.py # Define Employee serializer here**

│ ├── urls.py # App-level URL routing

│ └── views.py # Define API views for CRUD operations

└── db.sqlite3 # SQLite database file

**ART B**

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

|  |  |
| --- | --- |
| Roll No. : S020 | Name: Husain Chhil |
| Class : MBA Tech Data Science | Batch :J1 |
| Date of Experiment : | Date/Time of Submission : |
| Grade : |  |

**B.1 Code:**

**Settings.py**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework', # Django REST Framework

'employees', # Your employee app

]

**Employees/models.py**

from django.db import models

class Employee(models.Model):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

email = models.EmailField(unique=True)

department = models.CharField(max\_length=100)

date\_joined = models.DateField()

def \_\_str\_\_(self):

return f'{self.first\_name} {self.last\_name}'

**employees/serializers.py**

from rest\_framework import serializers

from .models import Employee

class EmployeeSerializer(serializers.ModelSerializer):

class Meta:

model = Employee

fields = '\_\_all\_\_'

**employees/views.py**

from rest\_framework import viewsets

from .models import Employee

from .serializers import EmployeeSerializer

class EmployeeViewSet(viewsets.ModelViewSet):

queryset = Employee.objects.all()

serializer\_class = EmployeeSerializer

**employees/urls.py**

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from .views import EmployeeViewSet

router = DefaultRouter()

router.register(r'employees', EmployeeViewSet)

urlpatterns = [

path('', include(router.urls)),

]

**employee\_management/urls.py**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

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

]

**CRUD OPERATIONS**

{

"first\_name": "Om",

"last\_name": "Lande",

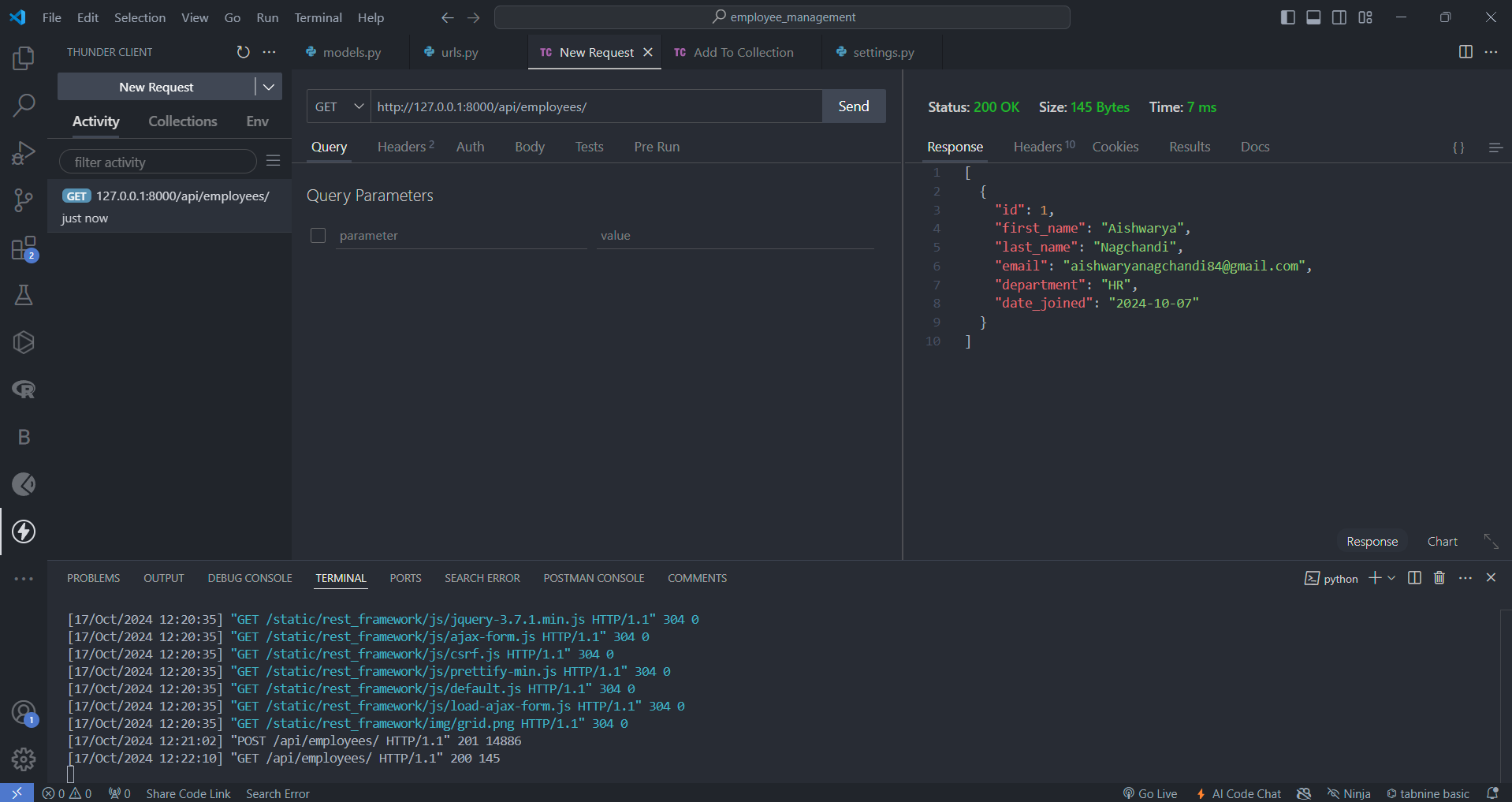
"email": "om.lande@company.com",

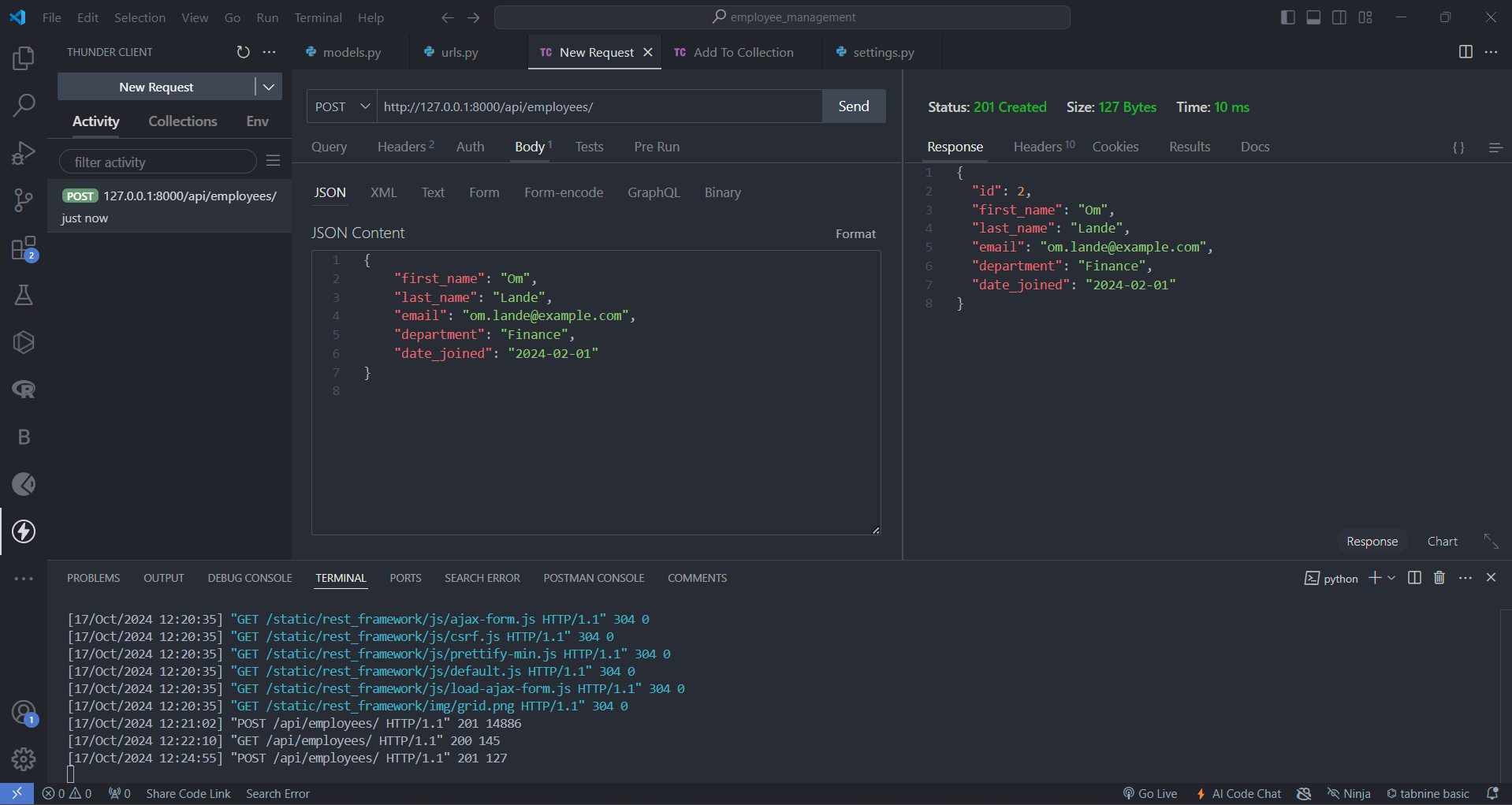
"department": "Finance",

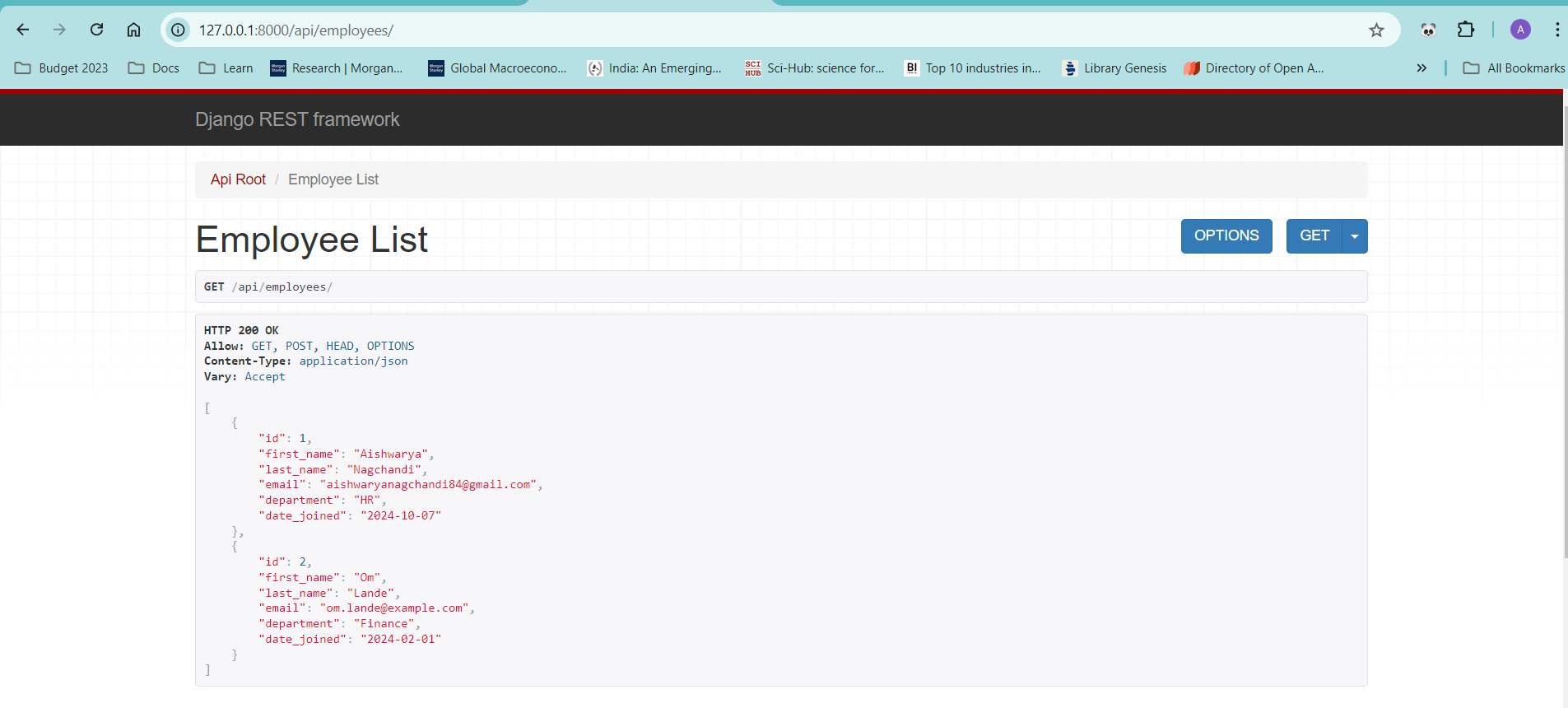
"date\_joined": "2024-01-01"

}

**B.2 Output**

**

**

**

**

**B.3 Conclusion:**

Made request to the API for adding, updating and deleting data from database

**B.3 Observations and Learning:**

Learned how to use and implement DRF projects for creating Endpoint for a Web Service

**B.4 Question of Curiosity**

1. How does Django REST Framework (DRF) handle serialization for database models?

Ans:

Django REST Framework (DRF) handles serialization for database models through serializers, which translate complex data types (like QuerySets) into JSON, XML, or other formats. For models, DRF provides ModelSerializer, a shortcut for creating serializers that automatically handle field definitions and data validation based on the model's schema. It uses fields to specify which model fields should be included.

1. What is the role of api\_view in Django REST Framework?

Ans:

In Django REST Framework, the @api\_view decorator is used to transform standard Django views into RESTful API views, which can handle HTTP methods like GET, POST, PUT, and DELETE. It's applied to function-based views to specify which HTTP methods the view will respond to.

1. How do you implement authentication and permissions in Django REST Framework?

Ans:

In Django REST Framework, authentication verifies the identity of a user, while permissions determine what an authenticated user can access. DRF provides multiple authentication classes like BasicAuthentication, TokenAuthentication, and SessionAuthentication. Permissions are managed through classes like IsAuthenticated, AllowAny, and custom permissions.

1. What are the advantages of using class-based views over function-based views in DRF?

Ans:

Class-based views (CBVs) in Django REST Framework offer greater flexibility and reusability compared to function-based views (FBVs). They allow you to structure code in a modular way using generic views like ListAPIView and CreateAPIView. This means you can break down logic into multiple methods (get, post, etc.), enabling easier extensions and customizations. Additionally, you can leverage mixins and inheritance for DRY code.

1. What is the difference between Serializer and ModelSerializer in Django REST Framework?

Ans:

In DRF, a Serializer allows full control over the data you handle, defining all fields, validation, and serialization logic manually. In contrast, a ModelSerializer is a subclass of Serializer that automatically generates fields and validations based on a specific Django model, saving time and reducing boilerplate code.

1. How does Django REST Framework handle versioning of APIs, and why is it important?

Ans:

Django REST Framework supports API versioning to handle changes in APIs over time without breaking existing clients. Versioning is important to maintain backward compatibility, especially as APIs evolve. DRF provides several versioning schemes, such as URLPathVersioning, QueryParameterVersioning, and HeaderVersioning, which can be set globally or per view. To implement versioning, you configure it in settings.py and can access the version in your views.