Module 17) Rest Framework

1.Introduction to APIs

- **What is an API (Application Programming Interface)?**
  An **API (Application Programming Interface)** is a **set of rules and tools** that allows different software applications to **communicate with each other**.

  Ex: Imagine you're using a **weather app**.
  That app doesn't calculate the weather itself—it **uses an API** to get data from a weather service like OpenWeatherMap.

  Your app sends a request like:

  GET https://api.weather.com/current?city=Delhi

  The API sends back data like:

  ```
  {

    "city": "Delhi",
    "temperature": "36°C",
    "condition": "Sunny"

  }
  ```

  Common Types of APIs:

  - **Web APIs** (e.g., Google Maps, Razorpay, Facebook Login)
  - **Operating System APIs** (e.g., Windows API)
  - **Library APIs** (e.g., Python math module)

  APIs Are Important because :

  - Allow **integration** between different systems
  - Save time by **reusing existing functionality**
  - Improve **scalability and modularity** in software

- **Types of APIs: REST, SOAP.**
  1. REST (Representational State Transfer)
     - **REST** is a lightweight, flexible, and widely-used API type.
     - Key Features:
       - Uses **HTTP methods**: GET, POST, PUT, DELET
       - Data is usually in **JSON** (sometimes XML or plain text)
       - **Stateless**: Every request is independent
       - Easy to use and understand
       - Faster, ideal for web and mobile apps

Ex :
GET https://api.example.com/users/123

Rresponse (Json)

```
{
  "id": 123,
  "name": "Nisha",
  "email": "nisha@example.com"
}
```

2. **SOAP (Simple Object Access Protocol)**
   - **SOAP** is a protocol that defines strict standards for communication.
   - Key Features:
     - Uses **XML** for all messages
     - Relies on **WSDL** (Web Services Description Language)
     - More **secure**, supports **ACID transactions**
     - Typically used in **enterprise** or **banking systems**
     - Slower and more complex than REST

Ex : SOAP Request (XML):

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <GetUserDetails>
      <UserId>123</UserId>
    </GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

- **Why are APIs important in web development?**
  - APIs (Application Programming Interfaces) are **extremely important** in web development because they allow different software systems to **communicate and work together** efficiently.

1.Enable Communication Between Systems

APIs allow your frontend (HTML/CSS/JS) to talk to your backend (Python/Django/Node.js, etc.).

Example: When a user submits a login form, your JavaScript sends a request to a login API, which verifies credentials in the backend.

2.Separate Frontend & Backend (Decoupling)

With APIs, the frontend and backend are loosely connected. This allows:

Easier development and testing

Multiple frontends (web, mobile) using the same backend

3.Integration with Third-Party Services

You can use APIs to integrate services like:

- Google Maps
- Razorpay or PayPal (payments)
- Facebook/Google login
- Weather, News, Location APIs

4.Scalability and Reusability

Once you write an API, it can be reused across your project or even across multiple projects.

Easily scalable—new features can be added without breaking existing functionality.

5.Faster Development

APIs allow teams to work **in parallel**:

Backend team builds APIs

Frontend team consumes them without waiting

6.Mobile App Support

Most mobile apps rely on APIs to communicate with servers. Your Django REST API can power both your web app and your mobile app.

7.Security and Control

APIs can be protected with:

- Authentication (e.g., login/token
- Authorization (who can do what)
- Rate limiting

2.Requirements for Web Development Projects

- **Understanding project requirements.**
- Understanding project requirements is the first and most critical step in building any software project — including Django applications. It ensures you know exactly what to build, why, and how to prioritize it.
- It means carefully identifying :
  - What the project must do (functional needs)
  - Who will use it (target users)
  - How it should behave (non-functional needs)
  - What constraints and resources exist (limitations)

- **Types of Requirements**

1. **Functional Requirements (What it does)**
   - These are the features of your project.

2. **Non-Functional Requirements (How it performs)**
   - These are qualities of the system.
     Ex :
   - Should load pages in under 2 seconds.
   - Should load pages in under 2 seconds.
   - Use secure login (HTTPS, encryption).

3. **User Requirements (Who uses it)**
   - Different types of users have different needs.
     Ex :
     Patient: Book appointments
     Doctor: View schedule
     Admin: Manage doctors & users

4. **Technical Requirements**
   - Technology choices and limitations.
     Ex :
     Use Django + Django REST framework
     PostgreSQL as the database
     Integrate Razorpay for payments
     Host on GitHub and deploy to Render

5. **Business Requirements (Why it's built)**
   - The goal of the project.
     Ex :
     Help users find qualified doctors easily and allow online appointment booking.

Steps to Understand Requirements :

- Talk to stakeholders (users, clients, mentors)
- Write down use cases
- Break features into modules/pages
- Make wireframes or flow diagrams List out technologies needed
- List out technologies needed

- **Setting up the environment and installing necessary packages.**

step 1: Create a Project Folder
mkdir doctor_finder
cd doctor_finder

step 2: Create a Virtual Environment
A virtual environment keeps your dependencies isolated
Windows
python -m venv env
env\Scripts\activate

macOS/Linux
python3 -m venv env

source env/bin/activate

step 3 : Install Django and REST Framework
pip install django djangorestframework

step 4 : Create a Django Project
django-admin startproject config .

step 5 : Create a Django App
python manage.py startapp doctor

step 6 : Add Installed Apps in settings.py
Open config/settings.py, and add the following to INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'doctor',
]
```

step 7 : Create a requirements.txt File
Save your installed packages:
pip freeze > requirements.txt

step 8 : Run Initial Migrations and Server
python manage.py migrate
python manage.py runserver

3.Serialization in Django REST Framework

- **What is Serialization?**
- Serialization is the process of converting complex data types (like Django models or Python objects) into a format that can be easily sent over the internet or saved to a file, such as JSON or XML.
- In Django REST Framework (DRF) :
  In DRF, **serializers** are used to:
  - Convert **model instances** (like a Doctor object) into **JSON** (for API response).
  - Convert **incoming JSON data** into **Python objects** (for saving into the database).

Ex :
Model

```
class Doctor(models.Model):
    name = models.CharField(max_length=100)
    specialization = models.CharField(max_length=100)
```

Serializer

```
from rest_framework import serializers
from .models import Doctor
```

```
class DoctorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Doctor
        fields = ['id', 'name', 'specialization']
```

Output JSON (serialized):

```
{
  "id": 1,
  "name": "Dr. Nisha",
  "specialization": "Cardiologist"
}
```

- Serialization matter for :
  - Allows **frontend & backend** to communicate
  - Converts data to a **standard format (like JSON)**
  - Automatically **validates input** (e.g. email format, required fields)
  - Essential for building **RESTful APIs**

- **Converting Django QuerySets to JSON.**
- In Django, you often need to convert **QuerySets** (like `Doctor.objects.all()`) into **JSON** to:
- Send data to the frontend
- Build REST APIs
- Return responses from AJAX or API endpoints

Method 1 : Using Django REST Framework

If you're using Django REST Framework, use a Serializer**:**

Ex :

Model

```
from django.db import models

class Doctor(models.Model):
    name = models.CharField(max_length=100)
    specialization = models.CharField(max_length=100)
```

Serializer :

```
from rest_framework import serializers

class DoctorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Doctor
        fields = ['id', 'name', 'specialization']
```

View

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

```
from .serializers import DoctorSerializer

@api_view(['GET'])
def get_doctors(request):
    doctors = Doctor.objects.all()
    serializer = DoctorSerializer(doctors, many=True)
    return Response(serializer.data)
```

Output JSON :

```
[
  {
    "id": 1,
    "name": "Dr. Smith",
    "specialization": "Cardiologist"
  }
]
```

Method 2 : Using Django's Built-in serializers Module

For small tasks or without DRF:

```
from django.core import serializers
from django.http import JsonResponse
from .models import Doctor

def get_doctors(request):
    doctors = Doctor.objects.all()
    data = serializers.serialize('json', doctors)
    return JsonResponse(data, safe=False)
```

Output :

```
[
  {
    "model": "app_name.doctor",
    "pk": 1,
    "fields": {
      "name": "Dr. Smith",
      "specialization": "Cardiologist"
    }
  }
]
```

Method 3 : Manual Conversion Using .values() or .values_list()

Simpler and readable JSON:

```
from django.http import JsonResponse
from .models import Doctor

def get_doctors(request):
    data = list(Doctor.objects.values('id', 'name', 'specialization'))
    return JsonResponse(data, safe=False)
```

Output :

```
[
  {
    "id": 1,
    "name": "Dr. Smith",
    "specialization": "Cardiologist"
  }
]
```

- **Using serializers in Django REST Framework (DRF).**
- Using **serializers** in Django REST Framework (DRF) is a **core concept** for building APIs. Serializers help you convert complex Django models into **JSON**, and also validate incoming JSON to create or update models.
- A **serializer** in DRF is similar to a Django form. It:
  - Converts model/queryset data → JSON (Serialization)
  - Converts JSON input → model instances (Deserialization + Validation)

Ex : Create a model (models.py)

```
from django.db import models

class Doctor(models.Model):
    name = models.CharField(max_length=100)
    specialization = models.CharField(max_length=100)
    email = models.EmailField()
```

Create a Serializer (serializers.py)

```
from rest_framework import serializers
from .models import Doctor

class DoctorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Doctor
        fields = ['id', 'name', 'specialization', 'email']
```

Create a View (views.py)

Using DRF's function-based API view:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Doctor
from .serializers import DoctorSerializer

@api_view(['GET'])
def doctor_list(request):
    doctors = Doctor.objects.all()
    serializer = DoctorSerializer(doctors, many=True)
    return Response(serializer.data)
```

Add URL (urls.py)

```
from django.urls import path
```

```
from . import views

urlpatterns = [
    path('api/doctors/', views.doctor_list, name='doctor-list'),
]
```

Test in Browser or Postman
http://127.0.0.1:8000/api/doctors/

You'll get a JSON like:


```
[
  {
    "id": 1,
    "name": "Dr. Nisha",
    "specialization": "Dermatologist",
    "email": "nisha@example.com"
  }
]
```


other way : You define each field manually — useful when not using Django models.

```
class DoctorManualSerializer(serializers.Serializer):
 name = serializers.CharField(max_length=100)
specialization = serializers.CharField(max_length=100)
```

4.Requests and Responses in Django REST Framework

- **HTTP request methods (GET, POST, PUT, DELETE).**
- **HTTP request methods** is crucial when building APIs using Django or Django REST Framework (DRF).
- Common HTTP Request Methods : GET, POST, PUT, PATCH, DELETE

1. GET - Fetch data
   Ex :

```
[
  {
    "id": 1,
    "name": "Dr. Nisha",
    "specialization": "Cardiologist"
  }
]
```


2. POST — Create data/send data
   Ex **:**
   POST /api/doctors/

   Request body (JSON):
   {

```
  "name": "Dr. Aryan",
  "specialization": "Dermatologist"
}
```

Server creates a new doctor and returns the created data.

3. PUT - update data
   Ex :
   PUT /api/doctors/1/

   Request body:
   ```
   {
     "name": "Dr. Aryan Roy",
     "specialization": "Skin Specialist"
   }
   ```

   Updates all fields of doctor with ID 1.

4. PATCH – update partial data
   Ex :
   PATCH /api/doctors/1/

   Request body:
   ```
   {
     "specialization": "Neurologist"
   }
   ```

   Only updates the specialization field.

5. DELETE – remove data
   Ex:
   DELETE /api/doctors/1/

   Deletes the doctor with ID 1. No content is returned.

- **Sending and receiving responses in DRF.**
- In DRF, **sending** and **receiving** responses is handled efficiently using built-in tools like:
  - Response object for sending data
  - request.data for receiving data

Ex : building a simple Doctor API with the ability to **receive data** (POST) and **send data** (GET).

1. Import Required DRF Tools
   ```
   from rest_framework.decorators import api_view
   from rest_framework.response import Response
   from rest_framework import status
   ```

2. Sending a Response (GET)

```
@api_view(['GET'])
def get_doctors(request):
    doctors = Doctor.objects.all()
    serializer = DoctorSerializer(doctors, many=True)
    return Response(serializer.data)  #  sending response
```

This sends a JSON response like:

```
[
  {
    "id": 1,
    "name": "Dr. Aryan",
    "specialization": "Cardiologist"
  }
]
```

3. Receiving Data (POST)

```
@api_view(['POST'])
def create_doctor(request):
    serializer = DoctorSerializer(data=request.data)  #  receiving input
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

- **request.data :**
  - It contains the incoming request body (e.g., JSON).
  - Automatically parsed into Python data types (dict, list, etc.)

- **Response():**
  - Used to return data in JSON format.
  - Handles proper content-type and status code.

5.Views in Django REST Framework

- **Understanding views in DRF: Function-based views vs Class-based views.**
- In Django REST Framework (DRF), **views** are the entry points for handling HTTP requests and returning responses. DRF supports both **Function-Based Views (FBVs)** and **Class-Based Views (CBVs)** to define how your application handles API requests.

1. Function-Based Views (FBVs) :

These are simple Python functions that take a request and return a response. They are easier to understand and quicker to write, especially for beginners or very simple use cases.

- Pros:
  - Simple and explicit.
  - Easy to understand for small or simple APIs.
  - Great for learning and debugging

- Cons:
  - Can become messy or repetitive for complex logic.

- Not reusable without additional effort.

Ex :

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer

@api_view(['GET', 'POST'])
def student_list(request):
    if request.method == 'GET':
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

2. Class-Based Views (CBVs)

These use Python classes to encapsulate request handling logic. DRF provides powerful generic and mixin-based views that reduce boilerplate code.

- Pros:
  - More reusable and extendable
  - Cleaner code structure for larger applications.
  - DRF provides many **Generic Views** and **ViewSets** to save time.

- Cons:
  - Slightly steeper learning curve.
  - Can be overkill for simple endpoints.

Ex :

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer

class StudentList(APIView):
    def get(self, request):
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)

    def post(self, request):
```

```
            serializer = StudentSerializer(data=request.data)
            if serializer.is_valid():
                serializer.save()
                return Response(serializer.data, status=status.HTTP_201_CREATED)
            return Response(serializer.errors,
        status=status.HTTP_400_BAD_REQUEST)
```

6. URL Routing in Django REST Framework

- **Defining URLs and linking them to views.**
- In Django (and Django REST Framework), defining URLs and linking them to views is an essential part of routing incoming requests to the correct functionality of your app. Here's a step-by-step guide with examples for both **Function-Based Views (FBVs)** and **Class-Based Views (CBVs)**.
- define a view :
  1. Function-Based View (FBV) :

```
# views.py
from django.http import HttpResponse

def home_view(request):
    return HttpResponse("Welcome to the homepage!")
```

  2. Class-Based View (CBV) :

```
# views.py
from django.views import View
from django.http import HttpResponse

class HomeView(View):
    def get(self, request):
        return HttpResponse("Welcome to the homepage!")
```

- Define the URL pattern :
- In your app folder, locate the file called urls.py. If it doesn't exist, create it.
- urls.py inside your Django app:

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home_view, name='home'),  # For FBV
    # OR
    path('', views.HomeView.as_view(), name='home'),  # For CBV
]
```

- Include app URLs in the project-level URL configuration
- In your project folder (`projectname/urls.py`):

```
# projectname/urls.py
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('yourappname.urls')),  # Link your app's URLs
]
```

7. Pagination in Django REST Framework

- **Adding pagination to APIs to handle large data sets.**
- In Django REST Framework (DRF), **pagination** helps you split large datasets into smaller chunks that are easier to load and navigate through. Here's how you can add pagination to your APIs:

1. Add Pagination in settings.py
   DRF provides built-in pagination classes. The most common are:
   PageNumberPagination (default pagination)
   LimitOffsetPagination
   CursorPagination (for large, frequently changing datasets)
   Example using PageNumberPagination:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10  # Number of results per page
}
```

2. Your View Should Return a List (e.g., ListAPIView)
   Ex :

```
# models.py
class Student(models.Model):
    name = models.CharField(max_length=100)
    roll = models.CharField(max_length=10)
```

   Ex : serializer

```
# serializers.py
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = '__all__'
```

   Example View with Pagination:

```
# views.py
from rest_framework.generics import ListAPIView
from .models import Student
from .serializers import StudentSerializer

class StudentListView(ListAPIView):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

3. Define the URL for the View
   Ex :
   # urls.py
   from django.urls import path
   from .views import StudentListView

   urlpatterns = [
      path('students/', StudentListView.as_view(), name='student-list'),
   ]

4. API Response with Pagination
   When you call /students/, the paginated response will look like this:
   Ex :
   {
      "count": 100,
      "next": "http://localhost:8000/students/?page=2",
      "previous": null,
      "results": [
         {
            "id": 1,
            "name": "Alice",
            "roll": "S101"
         },
         ...
      ]
   }

8. Settings Configuration in Django

- **Configuring Django settings for database, static files, and API keys.**
- Configuring Django settings properly is essential for smooth and secure app operation. Below is a structured guide to configuring **database**, **static files**, and **API keys** in your Django project's settings.py.
  1. DATABASE CONFIGURATION
     By default, Django uses SQLite. To configure for **PostgreSQL** :
     # settings.py

     DATABASES = {
        'default': {
           'ENGINE': 'django.db.backends.postgresql',
           'NAME': 'your_db_name',
           'USER': 'your_db_user',
           'PASSWORD': 'your_db_password',
           'HOST': 'localhost', # or your database server IP
           'PORT': '5432',   # default PostgreSQL port
        }
     }

     Install PostgreSQL dependencies:
     pip install psycopg2-binary

2. STATIC AND MEDIA FILES
   These are used to serve CSS, JS, images, etc.
   a. Development settings
      # settings.py

      STATIC_URL = '/static/'
      MEDIA_URL = '/media/'

      STATICFILES_DIRS = [BASE_DIR / 'static']
      MEDIA_ROOT = BASE_DIR / 'media'
      STATIC_ROOT = BASE_DIR / 'staticfiles' # for production (e.g. collectstatic)

      In your urls.py (development):
      from django.conf import settings
      from django.conf.urls.static import static

      urlpatterns = [
          ...
      ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

3. API KEY MANAGEMENT
   Never hardcode API keys in settings.py. Use **environment variables** for security.
   a. Load .env file using python-decouple:

      pip install python-decouple

   b. Create a .env file:
      SECRET_KEY=your-secret-key
      API_KEY=your-api-key
      DEBUG=True

   c. Update settings.py:
      from decouple import config

      SECRET_KEY = config('SECRET_KEY')
      DEBUG = config('DEBUG', default=False, cast=bool)
      MY_API_KEY = config('API_KEY')

9. Project Setup

- **Setting up a Django REST Framework project.**
- Setting up a **Django REST Framework (DRF)** project involves several steps to get your backend API up and running efficiently. Here's a clear step-by-step guide:
  1. INSTALL DJANGO AND DRF
     pip install django djangorestframework

     If you plan to use environment variables:
     pip install python-decouple

  2. CREATE YOUR DJANGO PROJECT

```
django-admin startproject myproject
cd myproject
python manage.py startapp api
```

3. UPDATE settings.py
   a. Add required apps:
   b. Optional: DRF Configuration

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
    ],
    'DEFAULT_PARSER_CLASSES': [
        'rest_framework.parsers.JSONParser',
    ],
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

4. DEFINE MODELS IN api/models.py

```
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
```

Then run:

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

5. CREATE A SERIALIZER

Create api/serializers.py:

```
from rest_framework import serializers
from .models import Item

class ItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = Item
        fields = '__all__'
```

6. CREATE API VIEWS

Create api/views.py:

```
from rest_framework import generics
from .models import Item
from .serializers import ItemSerializer

class ItemListCreateAPIView(generics.ListCreateAPIView):
```

```
queryset = Item.objects.all()
serializer_class = ItemSerializer
```

7. CONFIGURE URLS
   a. In api/urls.py (create this file):

```
from django.urls import path
from .views import ItemListCreateAPIView

urlpatterns = [
    path('items/', ItemListCreateAPIView.as_view(), name='item-list'),
]
```

   b. In your main urls.py:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),  # Your app's API routes
]
```

8. TEST THE API
   Run the development server:
   python manage.py runserver

   Visit:
   http://127.0.0.1:8000/api/items/
   You should see the DRF JSON API.

9. Enable Browsable API
   If you'd like a browsable API during development, add this to REST_FRAMEWORK:

```
'DEFAULT_RENDERER_CLASSES': [
    'rest_framework.renderers.JSONRenderer',
    'rest_framework.renderers.BrowsableAPIRenderer',  # enable browsable UI
],
```

10. Social Authentication, Email, and OTP Sending API

- **Implementing social authentication (e.g., Google, Facebook) in Django.**
- Implementing **social authentication** (Google, Facebook, etc.) in Django is straightforward using django-allauth or dj-rest-auth if you're building an API-based system. Here's a full implementation guide using Django + DRF + Google/Facebook login with dj-rest-auth + django-allauth.
  1. Install Required Packages
     pip install django-allauth dj-rest-auth djangorestframework

  2. Update INSTALLED_APPS in settings.py

```
INSTALLED_APPS = [
    'django.contrib.sites',  # required by allauth
```

```
    'rest_framework',
    'rest_framework.authtoken',

    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',

    'dj_rest_auth',
    'dj_rest_auth.registration',
]
```

3. Additional Settings

```
SITE_ID = 1

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
}

ACCOUNT_EMAIL_VERIFICATION = 'none'
ACCOUNT_AUTHENTICATION_METHOD = 'username_email'
ACCOUNT_EMAIL_REQUIRED = True
```

4. Add URLs

In your **project** urls.py:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('auth/', include('dj_rest_auth.urls')),
    path('auth/registration/', include('dj_rest_auth.registration.urls')),
    path('auth/social/', include('allauth.socialaccount.urls')),  # for Google/Facebook
OAuth
]
```

5. Setup OAuth Apps

Google OAuth

Visit Google Developer Console
Create a new project → Configure **OAuth consent screen**
Add OAuth credentials (Web app)
Set **Redirect URI**:
http://localhost:8000/auth/social/google/login/callback/

Facebook Login

Go to Facebook for Developers
Create a new App → Set up Facebook Login

Set **Redirect URI**:
http://localhost:8000/auth/social/facebook/login/callback/

6. Add Client ID & Secret to settings.py

```
SOCIALACCOUNT_PROVIDERS = {
    'google': {
        'APP': {
            'client_id': 'your-google-client-id',
            'secret': 'your-google-client-secret',
            'key': ''
        }
    },
    'facebook': {
        'APP': {
            'client_id': 'your-facebook-app-id',
            'secret': 'your-facebook-app-secret',
            'key': ''
        }
    }
}
```

7. Migrate & Run
   python manage.py migrate
   python manage.py runserver

- **Sending emails and OTPs using third-party APIs like Twilio, SendGrid.**

1. SENDING EMAILS WITH SENDGRID
   Install SendGrid : pip install sendgrid django-sendgrid-v5

   Add SendGrid Settings to settings.py

```
EMAIL_BACKEND = "sendgrid_backend.SendgridBackend"
SENDGRID_API_KEY = "your-sendgrid-api-key"

# Optional settings
SENDGRID_SANDBOX_MODE_IN_DEBUG = False
SENDGRID_ECHO_TO_STDOUT = True

DEFAULT_FROM_EMAIL = 'no-reply@yourdomain.com'
```

   Use python-decouple for secure key management:

```
from decouple import config

SENDGRID_API_KEY = config('SENDGRID_API_KEY')
```

   - Send Email from Django View or Utility

```
from django.core.mail import send_mail

send_mail(
    subject='Your OTP Code',
    message='Your OTP is 123456',
```

```
        from_email='no-reply@yourdomain.com',
        recipient_list=['user@example.com'],
        fail_silently=False,
    )
```

2. GENERATE AND SEND OTP

```
import random

def generate_otp():
    return str(random.randint(100000, 999999))
```

3. SENDING SMS OTP USING TWILIO

Install Twilio : pip install twilio

Add to settings.py

```
TWILIO_ACCOUNT_SID = config('TWILIO_ACCOUNT_SID')
TWILIO_AUTH_TOKEN = config('TWILIO_AUTH_TOKEN')
TWILIO_PHONE_NUMBER = config('TWILIO_PHONE_NUMBER')
```

Send SMS in Utility Function

```
from twilio.rest import Client
from django.conf import settings

def send_otp_sms(to_phone, otp):
    client = Client(settings.TWILIO_ACCOUNT_SID,
settings.TWILIO_AUTH_TOKEN)
    message = client.messages.create(
        body=f"Your OTP code is {otp}",
        from_=settings.TWILIO_PHONE_NUMBER,
        to=to_phone
    )
    return message.sid
```

Ex : full otp flow

```
def send_otp(request):
    phone = request.POST.get('phone')
    otp = generate_otp()

    send_otp_sms(phone, otp)  # Twilio
    send_mail(              # Optional: Send via email too
        'Your OTP Code',
        f'Your OTP is {otp}',
        'no-reply@yourdomain.com',
        ['user@example.com']
    )
    return JsonResponse({'status': 'OTP sent'})
```

11. RESTful API Design

- **REST principles: statelessness, resource-based URLs, and using HTTP methods for CRUD operations.**

- REST (Representational State Transfer) principles is crucial for designing clean and scalable
  APIs
  1. **statelessness**
- Each request from the client to the server must contain all the information needed to understand and process the request.
- **No session or login state** is stored on the server between requests.
- Authentication is usually done using tokens (e.g., **JWT** or API keys) included in **each** request.
  Ex :
  GET /api/user/123/
  Authorization: Bearer <token>

  2.Resource-Based URLs
  Use **nouns** (resources) in URLs, not verbs.
  Good URL design:
  Use plural nouns
  Avoid verbs in URLs
  Hierarchical and meaningful

  3.HTTP Methods for CRUD
  Each HTTP method corresponds to a CRUD action:

HTTP Method	CRUD Operation	CRUD Operation
GET	Read	Retrieve data
POST	create	Submit new data
PUT	Update	Replace existing respurce
PATCH	Partial Update	Modified part of a resource
DELETE	Delete	Remove resource

12. CRUD API (Create, Read, Update, Delete)

- **What is CRUD, and why is it fundamental to backend development**
- **CRUD** stands for the four basic operations that are typically performed on data in a database:

Operation	**Meaning**	**SQL Equivalent**	**HTTP Method(in REST)**
C	**Create**	**INSERT**	**POST**
R	**Read (Retrieve)**	**SELECT**	**GET**
U	**Update**	**UPDATE**	**PUT/PATCH**
D	**Delete**	**DELETE**	**DELETE**

CRUD forms the **core operations** behind nearly every backend system or web application.

1. Manages Data Lifecycle

Every application — whether it's an e-commerce site, blog, or banking system — needs to:

Add new data (Create)
Display or fetch data (Read)
Edit existing data (Update)

Remove outdated data (Delete)
CRUD handles this **complete data flow**.
2. Directly Maps to Database Operations
Backend development is all about interacting with the database:
CRUD maps exactly to SQL queries.
Frameworks like Django ORM or SQLAlchemy abstract these actions but are still built on
Frameworks like Django ORM or SQLAlchemy abstract these actions but are still built oncrud

3. Foundation of REST APIs
RESTful APIs rely on CRUD using HTTP methods:
Every REST endpoint typically implements one or more CRUD actions on a resource.

4. Frameworks & Tools Are CRUD-Centric
Frameworks like **Django**, **Express.js**, and **Laravel** are built around CRUD patterns:
Django: ModelViewSet, CreateView, UpdateView, etc
Rails: scaffold creates full CRUD controllers
DRF: ListCreateAPIView, RetrieveUpdateDestroyAPIView

5. Essential for Admin Panels & CMS
Backend dashboards (admin panels) are CRUD interfaces:
Admins create, view, edit, and delete data without writing raw queries.

13. Authentication and Authorization API

- **Difference between authentication and authorization.**

Feature	**Authentication**	**Authorization**
Definition	Verifying **who** the user is	Determining **what** the user is allowed to do
Purpose	Confirms identity (e.g., username & password)	Grants or denies access to resources or actions
Occurs When	Always comes **first** in a security process	Comes **after** authentication is successful
Output	Valid/invalid user identity	Allowed/denied access or permissions
Example	Login using email and password	Admin can edit users; regular users cannot
Django Example	Using login API with Token or Session	Checking @user_passes_test, IsAdminUser, etc.
Implemented By	Authentication classes (e.g., TokenAuth)	Permission classes (e.g., IsAuthenticated)

- **Implementing authentication using Django REST Framework's token-based system**
- Implementing **authentication using Django REST Framework's token-based system** involves a few clear steps. This system provides each authenticated user with a token which must be included in the headers of future requests for authentication.

1. Install Django REST Framework and Token Auth Package
   If you haven't already: pip install djangorestframework
   Token authentication is included in djangorestframework.authtoken, so add it too:
   pip install djangorestframework.authtoken

2. Add to INSTALLED_APPS in settings.py
   ```
   INSTALLED_APPS = [
       ...
       'rest_framework',
       'rest_framework.authtoken',
   ]
   ```

3. Add Token Authentication to DRF Settings
   ```
   REST_FRAMEWORK = {
       'DEFAULT_AUTHENTICATION_CLASSES': [
           'rest_framework.authentication.TokenAuthentication',
       ],
       'DEFAULT_PERMISSION_CLASSES': [
           'rest_framework.permissions.IsAuthenticated',
       ],
   }
   ```
   You can change IsAuthenticated to AllowAny if you want public access to some API.

4. Run Migrations
   python manage.py migrate

5. Create Token Automatically When User Is Created (optional)
   ```
   from django.conf import settings
   from django.dispatch import receiver
   from django.db.models.signals import post_save
   from rest_framework.authtoken.models import Token

   @receiver(post_save, sender=settings.AUTH_USER_MODEL)
   def create_auth_token(sender, instance=None, created=False, **kwargs):
       if created:
           Token.objects.create(user=instance)
   ```

   Connect the signal in your apps.py or ready():
   ```
   def ready(self):
       import yourapp.signals
   ```

6. Create a View for Getting Token (Login API)
   DRF provides a built-in view: IN urls.py
   ```
   from django.urls import path
   from rest_framework.authtoken.views import obtain_auth_token

   urlpatterns = [
       path('api-token-auth/', obtain_auth_token, name='api_token_auth'),
   ]
   ```

To login :
URL : POST /api-token-auth/
Payload:
```
{
  "username": "yourusername",
  "password": "yourpassword"
}
```

Response :
```
{
  "token": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}
```

7. Use the Token in API Requests
   Include this token in your headers for protected APIs:
   Authorization: Token your_token_here

8. Create a Protected API View
```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated

class ProtectedView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({'message': 'You are authenticated'})
```

14. OpenWeatherMap API Integration

- **Introduction to OpenWeatherMap API and how to retrieve weather data.**
- The **OpenWeatherMap API** is a popular and free-to-use weather service that provides **current weather**, **forecast**, and **historical weather data** for any location in the world.
- Features:
  - Current Weather
  - 5 Day / 3 Hour Forecast
  - Air Pollution Data
  - Historical Weather
  - Weather Alerts

- Retrieve Weather Data Using OpenWeatherMap API
  1. Sign Up and Get an API Key
     Go to: https://openweathermap.org/api
     Create a free account
     Navigate to **API keys** in your profile
     Copy the default API key (you'll use this in requests)

  2. Choose an API Endpoint
     Ex : Current Weather Data API

https://api.openweathermap.org/data/2.5/weather

3. Make a Request
   You can retrieve weather data using various parameters like city name, geographic coordinates, ZIP code, etc.
   By city name
   https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY
4. Optional Parameters
   q=London – city name
   appid=YOUR_API_KEY – your personal API key
   units=metric or units=imperial – for °C or °F
   lang=en – language of the response

5. Sample Python Code Using requests

```
import requests

API_KEY = 'your_api_key_here'
city = 'Delhi'
url = 
f'https://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&appid={API_KEY}'

response = requests.get(url)
data = response.json()

if response.status_code == 200:
    print(f"City: {data['name']}")
    print(f"Temperature: {data['main']['temp']} °C")
    print(f"Weather: {data['weather'][0]['description']}")
else:
    print("Error:", data['message'])
```

sample json

```
{
  "name": "London",
  "main": {
    "temp": 18.32
  },
  "weather": [
    {
      "description": "light rain"
    }
  ]
}
```

15. Google Maps Geocoding API

- **Using Google Maps Geocoding API to convert addresses into coordinates.**
- The **Google Maps Geocoding API** allows you to convert **addresses into geographic coordinates** and vice versa. This is essential for mapping, location search, and navigation features.

1. Create a Google Cloud Project and Get an API Key
   Go to: https://console.cloud.google.com/
   Create a new project.
   Enable **"Geocoding API"** under APIs & Services > Library.
   Navigate to **Credentials** and generate an **API key**
   Restrict the key to avoid abuse (recommended).

2. API Endpoint
   https://maps.googleapis.com/maps/api/geocode/json

3. Making a Request
   Required parameters:
   address: The physical address you want to geocode.
   key: Your API key.
   Ex : url
   https://maps.googleapis.com/maps/api/geocode/json?address=Taj+Mahal,+Agra&key=YOUR_API_KEY

4. Sample Python Code

```
import requests

API_KEY = 'YOUR_API_KEY'
address = 'Taj Mahal, Agra'

url = f'https://maps.googleapis.com/maps/api/geocode/json?address={address}&key={API_KEY}'

response = requests.get(url)
data = response.json()

if data['status'] == 'OK':
    location = data['results'][0]['geometry']['location']
    lat = location['lat']
    lng = location['lng']
    print(f"Latitude: {lat}, Longitude: {lng}")
else:
    print("Geocoding failed:", data['status'])
```

   sample json response


```
{
  "results": [
    {
      "geometry": {
        "location": {
          "lat": 27.1751448,
          "lng": 78.0421422
        }
      },
```

"formatted_address": "Dharmapuri, Forest Colony, Tajganj, Agra, Uttar Pradesh 282001, India"
}
],
"status": "OK"
}

16. GitHub API Integration

- **Introduction to GitHub API and how to interact with repositories, pull requests, and issues.**
- The **GitHub API** allows developers to **programmatically interact** with GitHub — enabling you to **manage repositories, issues, pull requests, commits**, and more

1. Overview of GitHub API
   Base url:
   https://api.github.com/
   Authentication:
   For public data: No auth needed (rate limited).
   For private repos or higher rate limit: Use a **Personal Access Token (PAT)**.
   Auth Header:
   Authorization: token YOUR_TOKEN_HERE

2. Working with Repositories
   Get Public Repository Info
   GET /repos/{owner}/{repo}
   Ex : in python
   import requests

   ```
   url = 'https://api.github.com/repos/octocat/Hello-World'
   response = requests.get(url)
   data = response.json()

   print(f"Repo Name: {data['name']}")
   print(f"Description: {data['description']}")
   print(f"Stars: {data['stargazers_count']}")
   ```

3. Working with Issues
   List Issues in a Repo
   GET /repos/{owner}/{repo}/issues

   Create a New Issue (auth required)
   POST /repos/{owner}/{repo}/issues
   Headers:
   Authorization: token YOUR_TOKEN

   Body:
   ```
   {
    "title": "Bug in login",
    "body": "The login button doesn't work.",
    "assignees": ["username"]
   }
   ```

4. Working with Pull Requests
   List Pull Requests
   GET /repos/{owner}/{repo}/pulls
   Create a Pull Request
   POST /repos/{owner}/{repo}/pulls
   Headers:
   Authorization: token YOUR_TOKEN

   Body:
   ```
   {
    "title": "Fix typo",
    "head": "feature-branch",
    "base": "main",
    "body": "Fixed typo in README"
   }
   ```

   head is the name of the branch with your changes, base is where you want to merge it.

   Authentication: Generate a Personal Access Token (PAT)
   Go to GitHub → Settings → Developer Settings → Personal Access Tokens.
   Generate token with scopes like repo, workflow, read:org.
   Use in headers:
   Authorization: token YOUR_TOKEN

   Use GitHub's official **API Explorer**: https://docs.github.com/en/rest

17.Twitter API Integration

- **Using Twitter API to fetch and post tweets, and retrieve user data.**
- Prerequisites :
  1. Twitter Developer Account
     Go to https://developer.twitter.com/
     Apply for a developer account and create a **Project + App**
     Generate your **API keys and tokens** from the **Keys and Tokens** tab:
     API Key
     API Key Secret
     Access Token
     Access Token Secret

  2. **Install Tweepy** (Python wrapper for the Twitter API)
     pip install tweepy

     setup Ex :
     ```
     import tweepy

     API_KEY = 'your_api_key'
     API_SECRET = 'your_api_secret'
     ACCESS_TOKEN = 'your_access_token'
     ACCESS_SECRET = 'your_access_secret'
     ```

```
auth = tweepy.OAuth1UserHandler(API_KEY, API_SECRET,
ACCESS_TOKEN, ACCESS_SECRET)
api = tweepy.API(auth)
```

1. Retrieve User Profile Data

```
user = api.get_user(screen_name='elonmusk')
print(f"Username: {user.name}")
print(f"Followers: {user.followers_count}")
print(f"Bio: {user.description}")
```

2. Fetch Recent Tweets from a User

```
tweets = api.user_timeline(screen_name='nasa', count=5)

for tweet in tweets:
    print(f"{tweet.created_at} - {tweet.text}\n")
```

3. Post a Tweet

```
tweet = api.update_status("Hello from Tweepy and the Twitter API!")
print("Tweet posted successfully!")
```

Ex :

```
def lab_twitter_activity(username, message):
    # 1. Fetch user info
    user = api.get_user(screen_name=username)
    print(f"User: {user.name}, Followers: {user.followers_count}, Bio:
{user.description}")

    # 2. Fetch recent tweets
    print("\nRecent Tweets:")
    tweets = api.user_timeline(screen_name=username, count=5)
    for tweet in tweets:
        print(f"- {tweet.text}")

    # 3. Post a tweet
    new_tweet = api.update_status(message)
    print(f"\n New Tweet Posted: {new_tweet.text}")

# Run it
lab_twitter_activity('nasa', 'Hello, Earth! This is a test tweet via Tweepy. ')
```

notes :

Twitter API v1.1 is easier for basic tasks using Tweepy.
v2 API has more advanced features (followers lookup, tweet engagement), but requires Bearer Token and is more strict.
Free access is limited under Twitter's **new API tiers** (X/Twitter). Consider **Twitter Blue for Devs** or apply for elevated access.

18. REST Countries API Integration

- **Introduction to REST Countries API and how to retrieve country-specific data.**
- The **REST Countries API** is a free and open API that provides detailed information about countries around the world, such as:
- Country name, capital, population, currency, Language,Borders,Flags,etc.
- Base URL : https://restcountries.com/v3.1/
  Ex : Get info for India
  Request url : https://restcountries.com/v3.1/name/india

  Sample json :

```
[
  {
    "name": {
      "common": "India",
      "official": "Republic of India"
    },
    "capital": ["New Delhi"],
    "region": "Asia",
    "population": 1380004385,
    "area": 3287590,
    "languages": {
      "eng": "English",
      "hin": "Hindi"
    },
    "flags": {
      "png": "https://flagcdn.com/w320/in.png"
    }
  }
]
```

  Ex : python fetch country info

```
import requests

country_name = "India"
url = f"https://restcountries.com/v3.1/name/{country_name}"

response = requests.get(url)
data = response.json()[0]

print(f"Country: {data['name']['common']}")
print(f"Capital: {data['capital'][0]}")
print(f"Region: {data['region']}")
print(f"Population: {data['population']}")
print(f"Languages: {', '.join(data['languages'].values())}")
print(f"Flag URL: {data['flags']['png']}")
```

19. Email Sending APIs (SendGrid, Mailchimp)

- Using email sending APIs like SendGrid and Mailchimp to send transactional emails.

- Transactional emails are **automated, real-time messages** triggered by user actions like:
- Sign-up confirmations, Password reset links, Order receipts, OTPs / Alerts,

1, **SendGrid API** (by Twilio)

Ex : Send Transactional Email with SendGrid

1. Create a SendGrid Account
   Go to: https://sendgrid.com
   Create an account
   Go to **Settings > API Keys** → create a new key with "Full Access"

2. Install SendGrid Python Library : pip install sendgrid
3. Sample Python Code Using SendGrid

```
import os
from sendgrid import SendGridAPIClient
from sendgrid.helpers.mail import Mail

message = Mail(
    from_email='your_email@example.com',
    to_emails='recipient@example.com',
    subject='Welcome to Our Service!',
    html_content='<strong>Thank you for signing up!</strong>'
)

try:
    sg = SendGridAPIClient('YOUR_SENDGRID_API_KEY')
    response = sg.send(message)
    print(response.status_code)
except Exception as e:
    print(e)
```

   - notes :
   - You can personalize messages with names, OTPs, etc
   - You can also send attachments or use dynamic templates.

2.Mailchimp Transactional Email (formerly Mandrill)

Mailchimp: Key Facts

Mailchimp's main product is for **marketing** emails.

Use **Mailchimp Transactional** (https://mandrillapp.com) for transactional use.

Requires a **Mailchimp account** with paid tier to access Mandrill.

Send Email with Mailchimp Transactional

1. Create Account & Enable Transactional Email
   Create a Mailchimp account
   Navigate to the **Transactional Email** section
   Generate **API key** from Mandrill dashboard

2. Install requests (or use mandrill SDK)
   pip install requests
3. Send Email via Mailchimp Transactional API
   import requests

   url = "https://mandrillapp.com/api/1.0/messages/send.json"

   payload = {
     "key": "YOUR_MANDRILL_API_KEY",
     "message": {
       "from_email": "your_email@example.com",
       "to": [
         {
           "email": "recipient@example.com",
           "type": "to"
         }
       ],
       "subject": "OTP for Login",
       "html": " Your OTP is 123456 "
     }
   }

   response = requests.post(url, json=payload)
   print(response.json())

21. Payment Integration (PayPal, Stripe)

- **Introduction to integrating payment gateways like PayPal and Stripe.**
- Payment gateways like **PayPal** and **Stripe** enable applications and websites to accept secure **online payments** via credit/debit cards, wallets, or bank transfers.
- Use a Payment Gateway for :
- Secure handling of card details (PCI compliance)
- Support for multiple payment methods
- Webhooks for real-time payment status
- Easy refunds, subscriptions, invoicing

1. Stripe Payment Integration
   1. Stripe Payment Integration (Quick Start)
   Go to https://stripe.com
   Create an account → Get your **Publishable Key** and **Secret Key**

   2. Install Stripe Python SDK
   pip install stripe

   3. Create a Checkout Session (Python Backend)
   import stripe

   stripe.api_key = "your_secret_key"

```
session = stripe.checkout.Session.create(
    payment_method_types=["card"],
    line_items=[{
        'price_data': {
            'currency': 'usd',
            'product_data': {
                'name': 'T-shirt',
            },
            'unit_amount': 2000,  # $20.00 in cents
        },
        'quantity': 1,
    }],
    mode='payment',
    success_url='https://yourdomain.com/success',
    cancel_url='https://yourdomain.com/cancel',
)

print(session.url)  # Redirect user to this URL
```

2. PayPal Payment Integration

1. Create a PayPal Developer Account
   Go to: https://developer.paypal.com
   Create **sandbox** and **live** credentials (Client ID & Secret)

2. Use the PayPal Checkout SDK (or REST API)
   pip install paypalrestsdk

3. Create a Payment

```
import paypalrestsdk

paypalrestsdk.configure({
    "mode": "sandbox",  # or "live"
    "client_id": "YOUR_CLIENT_ID",
    "client_secret": "YOUR_CLIENT_SECRET"
})

payment = paypalrestsdk.Payment({
    "intent": "sale",
    "payer": {
        "payment_method": "paypal"},
    "redirect_urls": {
        "return_url": "http://localhost:8000/payment/success",
        "cancel_url": "http://localhost:8000/payment/cancel"},
    "transactions": [{
        "item_list": {
            "items": [{
                "name": "Book",
                "sku": "123",
```

```
            "price": "10.00",
            "currency": "USD",
            "quantity": 1}]},
        "amount": {
            "total": "10.00",
            "currency": "USD"},
        "description": "Book purchase"}]})

if payment.create():
    print("Payment created successfully")
    for link in payment.links:
        if link.method == "REDIRECT":
            print("Redirect for approval: %s"      % (link.href))
else:
print(‘error’)
```

22. Google Maps API Integration

- **Using Google Maps API to display maps and calculate distances between locations.**
  refer module 16 que .20 Answer.