Name : **Dhruvit Savaliya**

Python DB and Framework

1. Introduction to APIs

- What is an API (Application Programming Interface)?

- An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other.
- It defines the methods and data formats that applications can use to request and exchange information.

- Key Concepts:
    - Requests and Responses:
      Clients send requests to an API.
      The API processes them and returns a response (usually in JSON or XML format).
    - Endpoints:
      Specific URLs provided by an API that perform different functions (e.g., /get-users, /login, /create-order).
    - Methods (common in web APIs, especially REST):
      GET: Retrieve data.
      POST: Send new data.
      PUT/PATCH: Update existing data.
      DELETE: Remove data.
    - Authentication:
      APIs often require keys or tokens to ensure secure access.

- Types of APIs: REST, SOAP.

- REST (Representational State Transfer)

    - Characteristics:

Stateless: Each request from client to server must contain all the information needed.
Uses HTTP methods: GET, POST, PUT, DELETE.
Data format: Mostly uses JSON, sometimes XML.
URL-based: Each endpoint (URL) represents a resource.

- Pros:
  Simple and easy to use.
  Lightweight and fast.
  Works well with web and mobile apps.
  Human-readable format (JSON).

- Cons:
  Less strict; may lead to inconsistent implementations.

➢ SOAP (Simple Object Access Protocol)

- Characteristics:
  Uses XML for all requests and responses.
  Works over various protocols (HTTP, SMTP, etc.).
  Has built-in error handling and security features.
  Protocol-based: Strict standards (WSDL, XML Schema).

- Pros:
  Highly secure and reliable.
  Ideal for enterprise-level apps (like banking or telecom).
  Strict contract-based structure.

- Cons:
  More complex and heavier than REST.
  Slower due to XML parsing and overhead.

- Why are APIs important in web development?

➢ APIs are super important in web development because they're the bridge that connects different systems, services, and applications — kind of like how electricity connects your light switch to the bulb.

➢ Enable Frontend-Backend Communication

APIs let your frontend (HTML/CSS/JS) talk to your backend (Django, Node.js, etc.).
Example: A user clicks “Get Doctors” → JavaScript sends a request to the API → backend returns doctor data → displayed in the browser.

- Integrate Third-Party Services
  Want to use Google Maps? Twitter feed? Payment gateway like Paytm or Stripe?
  APIs let you plug in powerful services instead of building from scratch.

- Separate Concerns (Frontend vs Backend)
  APIs create a clear separation: frontend focuses on user experience, backend handles logic & data.
  Makes the system modular, easier to develop, test, and maintain.

- Support Multi-Platform Access
  APIs allow websites, mobile apps, and even IoT devices to use the same backend.
  Example: A hospital system can have one API powering both the website and mobile app.

- Secure Data Exchange
  APIs use authentication (like API keys or tokens) to control access to sensitive data.
  Important in login systems, payment processing, etc.

- Automate Tasks & Communication
  APIs can automatically send emails, update databases, generate reports, or fetch real-time data.
  They’re key for building smart, responsive systems.

- Reusability & Scalability
  Once you build an API, you can reuse it in multiple projects or modules.
  Makes your system easier to scale and maintain over time.

2. Requirements for Web Development Projects

- Understanding project requirements.

- Project requirements define what a project should do (functional) and how it should behave (non-functional).
- It’s all about understanding the client’s or user’s needs clearly before writing any code.

- Types of Requirements:
  - Functional Requirements – What the system should do
    Features, tasks, or functions the system must perform.
  - Non-Functional Requirements – How the system should behave
    Performance, security, scalability, UI/UX expectations.

- Steps to Understand Project Requirements:
  - Meet with Stakeholders
    Clients, users, or business owners.
    Ask what they want, why they want it, and who will use the system.
  - Gather and Document Requirements
    User Stories: "As a user, I want to search for doctors by specialty."
    Use Cases: Describes user interactions step-by-step.
    Requirement Lists or Spreadsheets.
  - Clarify and Validate
    Ask questions, remove ambiguity.
    Confirm with the client: "Do you mean that users should be able to upload documents?"
  - Prioritize Features
    Must-have vs nice-to-have.
    Helps when working with limited time or budget.
  - Create Wireframes or Mockups
    Sketch the UI to visualize features.
    Tools: Figma, Balsamiq, or even paper sketches.
  - Write a Project Scope Document
    Clearly outlines what will be done, how, and by when.
    Prevents scope creep (when extra features keep getting added unexpectedly).

- Setting up the environment and installing necessary packages.

- Install Python
  Make sure Python is installed (preferably version 3.8 or above).
  python –version

- Create a Virtual Environment
  A virtual environment keeps project dependencies isolated.
  python -m venv env
  env\Scripts\activate

- Install Django

  Once your virtual environment is active:

  ```
  pip install django
  ```

- Start Your Django Project

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

  To run the server:

  ```
  python manage.py runserver
  ```

- Create Your First App

  For example, for a doctor app:

  ```
  python manage.py startapp doctor
  ```

- Install Additional Packages

Package	Purpose
djangorestframework	Build REST APIs
django-crispy-forms	Beautiful form rendering
django-allauth	Social login (Google, Facebook)
python-decouple	Manage environment variables
Pillow	Handle image uploads
mysqlclient or psycopg2	Use MySQL/PostgreSQL

- Create Requirements File

  Helps you or teammates install all packages later:

  ```
  pip freeze > requirements.txt
  ```

  To install from it:

  ```
  pip install -r requirements.txt
  ```

3. Serialization in Django REST Framework

- What is Serialization?

- Serialization is the process of converting complex data types (like Django models, Python objects) into a format that can be easily shared over the internet — usually JSON or XML.
- Think of it like turning your data into a readable package to send it across systems or to a frontend app.
- REST APIs use JSON for communication.

- Your Django models are Python objects — not directly transferable.
- Serializers bridge the gap between Python objects and JSON.

Converting Django QuerySets to JSON.

- A QuerySet is a list of objects fetched from the database using Django models.
- To send data from Django (backend) to JavaScript (frontend) or to build APIs, we need to convert the QuerySet into JSON format, which browsers and other apps can understand.

- Simple Ways to Convert QuerySet to JSON:
  - Using Django's Built-in Serializer

    Converts QuerySet into JSON with all model details.

    ```
    from django.core import serializers
    data = serializers.serialize('json', YourModel.objects.all())
    ```

  - Using Django REST Framework (Best for APIs)

    Uses a Serializer class to return clean and customizable JSON.

    ```
    from rest_framework import serializers

    class YourModelSerializer(serializers.ModelSerializer):
       class Meta:
          model = YourModel
          fields = '__all__'

    serializer = YourModelSerializer(queryset, many=True)
    serializer.data  # clean JSON output
    ```

  - Using JsonResponse with .values()

    Quick and easy for simple needs.

    ```
    from django.http import JsonResponse
    data = list(YourModel.objects.values())
    return JsonResponse(data, safe=False)
    ```

Using serializers in Django REST Framework (DRF).

- A serializer in DRF helps you:

  Convert Django models (Python objects) → JSON (Serialization)

  Convert JSON data → Django models (Python objects) (Deserialization)

- This is essential for sending and receiving data via APIs.

- Steps to Use Serializers in DRF
  - Step 1: Create a Django Model

    A model is a class that defines your database structure.

    Example: You might have a Doctor model with name and specialization.

  - Step 2: Create a Serializer

    A serializer is like a translator between your model and JSON.

    You tell the serializer which model to use.

    You tell it which fields you want to convert.

    It helps in sending data to the frontend or receiving user data.

  - Step 3: Create a View

    A view is a function or class that handles the API request.

    It fetches the model data.

    It uses the serializer to convert it to JSON.

    It returns that JSON as a response.

  - Step 4: Add URL for the API

    You create a path in the urls.py file so users (or frontend) can access the API using a link like:

    http://localhost:8000/api/doctors/

  - Step 5: Test Your API

    Once it's ready, you can open that URL in a browser or test it using tools like:

    Postman

    Django's built-in API browser

4. Requests and Responses in Django REST Framework

- HTTP request methods (GET, POST, PUT, DELETE).

- GET – Retrieve Data
  - Purpose: Fetch data from the server without making any changes to it.
  - Use Case: Used to request data, such as retrieving a list of doctors, showing a product detail, or fetching user info.

- Response: The server will send the requested data in a response (usually in JSON format).

➢ POST – Create Data
- Purpose: Send data to the server to create a new resource.
- Use Case: Used to submit data like creating a new doctor profile, registering a user, or adding a new product.
- Request Body: Contains data to be saved, usually in JSON format:

```
{
  "name": "Dr. John Doe",
  "specialization": "Cardiologist"
}
```

- Response: The server will typically return the created object with a 201 Created status.

➢ PUT – Update Data
- Purpose: Update an existing resource on the server with new data.
- Use Case: Used to modify or replace the data of an existing resource (e.g., updating a doctor's profile, editing a post).
- Response: The server will return the updated object with a 200 OK status.

➢ DELETE – Delete Data
- Purpose: Remove an existing resource from the server.
- Use Case: Used to delete a specific resource, like removing a doctor profile, deleting a user, or removing a product.
- Response: The server will return a 204 No Content status, indicating the resource was successfully deleted.

• Sending and receiving responses in DRF.

➢ A response is the data your API sends back to the user (browser, frontend app, mobile app, etc.).
➢ In DRF, we use the Response class to send this data.

➢ Sending a Response

return Response(data)

DRF automatically converts the data (like Python dictionaries) into JSON format.

Example: Sending a message

```
return Response({"message": "Hello, World!"})
```

- Receiving Data from the User

  request.data

  This is how your API gets the data sent by the user, usually from a form or a frontend app.

- Validating and Saving the Data

  When you get data from the user, you use a serializer to:

  Check if the data is valid.

  Save it to the database.

```
serializer = DoctorSerializer(data=request.data)
if serializer.is_valid():
   serializer.save()
```

5. Views in Django REST Framework

- Understanding views in DRF: Function-based views vs Class-based views.

- Function-Based Views (FBV)

  These are simple functions.

  You manually check the request method (GET, POST, etc.).

  Best for small and simple APIs.

- Class-Based Views (CBV)

  These are Python classes that handle different actions using class methods.

  DRF provides ready-to-use classes like:

  APIView

  ListAPIView, CreateAPIView, RetrieveUpdateDestroyAPIView, etc.

  Best for larger projects or when reusing logic.

Feature	Function-Based View (FBV)	Class-Based View (CBV)
Style	Uses regular Python functions	Uses Python classes
Easy to understand	Simple and readable	Slightly more complex
Best for	Small projects & quick tasks	Large projects with reusable logic
Reusability	Not easily reusable	Easily reusable and extendable

6. URL Routing in Django REST Framework

- Defining URLs and linking them to views.

- In Django and Django REST Framework (DRF), URLs (Uniform Resource Locators) are essential for connecting web addresses to the corresponding views.
- When a user visits a URL, Django looks up the matching view to handle the request and send a response.
- This process is known as URL routing, and it ensures that each API endpoint or page is handled correctly.
- Helps Django know which function or class (view) to call.
- Organizes the API structure with meaningful paths (like /api/doctors/).
- Makes your web application or API easily accessible and scalable.

- Components Involved
    - View – Handles the logic and returns a response.
    - urls.py in app – Maps a URL path to a view.
    - urls.py in project – Includes app-level URLs into the main project routing.

- Step 1: Create a View

  Function-Based View (FBV) Example:

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

@api_view(['GET'])
def doctor_list(request):
    return Response({"message": "List of doctors"})
```

  Class-Based View (CBV) Example:

```
from rest_framework.views import APIView

from rest_framework.response import Response

class DoctorList(APIView):

def get(self, request):

    return Response({"message": "List of doctors (CBV)"})
```

- Step 2: Define URLs in App's urls.py

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

```
from .views import doctor_list  # Or DoctorList if using CBV

urlpatterns = [
    path('doctors/', doctor_list, name='doctor-list'),  # For FBV
    # path('doctors/', DoctorList.as_view(), name='doctor-list'),  # For CBV
]
```

- Step 3: Include App URLs in Project's Main urls.py

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

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

7. Pagination in Django REST Framework

- Adding pagination to APIs to handle large data sets.

- Pagination is the process of dividing a large dataset into smaller chunks (pages) so that the client does not get overwhelmed with too much data at once.
- Improves performance (less data to process and transmit).
- Enhances user experience (faster load times).
- Reduces memory usage on the client side.
- Makes APIs scalable.

- Pagination in Django REST Framework

  DRF provides built-in pagination classes you can use:

  PageNumberPagination – Simple pagination with page numbers.

  LimitOffsetPagination – Allows limit and offset for more control.

  CursorPagination – For large data, maintains consistency with cursors.

- Step 1: Set Pagination in settings.py

  Choose the type of pagination you want.

  Example using PageNumberPagination:

```
REST_FRAMEWORK = {
```

```
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10  # Number of items per page
}
```

- Step 2: Use a List View (like ListAPIView)

  You can use ListAPIView to return paginated data.

  Example:

```
from rest_framework.generics import ListAPIView
from .models import Doctor
from .serializers import DoctorSerializer

class DoctorList(ListAPIView):
    queryset = Doctor.objects.all()
    serializer_class = DoctorSerializer
```

- Step 3: (Optional) Custom Pagination Class

  You can create your own pagination class.

  Example:

```
from rest_framework.pagination import PageNumberPagination

class CustomDoctorPagination(PageNumberPagination):
    page_size = 5
    page_size_query_param = 'page_size'
    max_page_size = 100
```

  Then use it in your view:

```
from .pagination import CustomDoctorPagination

class DoctorList(ListAPIView):

    queryset = Doctor.objects.all()

    serializer_class = DoctorSerializer

    pagination_class = CustomDoctorPagination
```

8. Settings Configuration in Django

- Configuring Django settings for database, static files, and API keys.

- Database Configuration

  Django needs to connect to a database to store and manage your data (like users, posts, doctors, etc.).

  By default, Django uses SQLite.

  You can also use MySQL or PostgreSQL.

  Example (SQLite):

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

- Static Files Configuration

  Static files are your CSS, JavaScript, and images.

  You tell Django:

  Where your static files are

  How to serve them

  Settings:

```
STATIC_URL = '/static/'  # URL to access static files
STATICFILES_DIRS = [BASE_DIR / 'static']  # Your custom static folder
STATIC_ROOT = BASE_DIR / 'staticfiles'   # Folder where static files are collected
```

- Media Files Configuration (for file uploads)

  Media files are user-uploaded files (like profile photos).

  Settings:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

- API Keys Configuration

  If you're using external services (like Google Maps or Paytm), you need to keep your API keys safe.

  Use environment variables instead of writing the key directly in settings.py.

  Example using os.environ:

```
import os
GOOGLE_API_KEY = os.environ.get('GOOGLE_API_KEY')
```

9. Project Setup

- Setting up a Django REST Framework project.

➢ Step 1: Install Django and DRF

Open your terminal and install Django and Django REST Framework:

```
pip install django djangorestframework
```

➢ Step 2: Create a New Django Project

Create a project folder using:

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

➢ Step 3: Create a Django App

Now create an app where you will build your APIs:

```
python manage.py startapp myapi
```

➢ Step 4: Add Apps to settings.py

Open myproject/settings.py and add these to INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    'rest_framework',  # Django REST Framework
    'myapi',         # Your app
]
```

➢ Step 5: Create Models (Optional)

Define your data model in myapi/models.py:

```
from django.db import models

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

➢ Step 6: Create a Serializer

In myapi/serializers.py:

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

class DoctorSerializer(serializers.ModelSerializer):
   class Meta:
      model = Doctor
      fields = '__all__'
```

- Step 7: Create a View

  In myapi/views.py:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .models import Doctor
from .serializers import DoctorSerializer

class DoctorList(APIView):
    def get(self, request):
        doctors = Doctor.objects.all()
        serializer = DoctorSerializer(doctors, many=True)
        return Response(serializer.data)
```

- Step 8: Define URLs

  In myapi/urls.py:

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

urlpatterns = [
    path('doctors/', DoctorList.as_view()),
]
```

- Step 9: Run the Server

  python manage.py runserver

10. Social Authentication, Email, and OTP Sending API

- Implementing social authentication (e.g., Google, Facebook) in Django.

- Social Authentication allows users to log in using their social media accounts like Google, Facebook, etc.
- Benefits:

  Easy login for users

  No need to remember another password

  Secure and widely used

- Step 1: Install Required Packages

  pip install social-auth-app-django

- Step 2: Add to settings.py

```
INSTALLED_APPS = [
    ...
    'social_django',
]
AUTHENTICATION_BACKENDS = (
    'social_core.backends.google.GoogleOAuth2',   # for Google
    'social_core.backends.facebook.FacebookOAuth2',  # for Facebook
    'django.contrib.auth.backends.ModelBackend',
)
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
        },
    },
]
```

- Step 3: Set up Social App Keys

  Then in settings.py, add:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'your-google-client-id'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'your-google-client-secret'

SOCIAL_AUTH_FACEBOOK_KEY = 'your-facebook-app-id'
SOCIAL_AUTH_FACEBOOK_SECRET = 'your-facebook-app-secret'
```

- Step 4: Set Login Redirects

```
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

- Step 5: Add URLs

  In urls.py:

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

```
urlpatterns = [
    ...
    path('auth/', include('social_django.urls', namespace='social')),  # social
auth URLs
]
```

- Step 6: Add Links in Template

```
<a href="{% url 'social:begin' 'google-oauth2' %}">Login with Google</a>
<a href="{% url 'social:begin' 'facebook' %}">Login with Facebook</a>
```

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

- Third-party APIs like SendGrid (for emails) and Twilio (for SMS) help you:
  - Send OTPs securely
  - Send confirmation emails
  - Ensure email/SMS delivery
  - Track email status (delivered, opened, etc.)

- Sending Emails with SendGrid
  - Step 1: Install SendGrid

    ```
    pip install sendgrid
    ```
  - Step 2: Get API Key

    Sign up at https://sendgrid.com

    Go to Settings > API Keys and create a key
  - Step 3: Use in Django View

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

    def send_email():
        sg = sendgrid.SendGridAPIClient(api_key='your_sendgrid_api_key')
        message = Mail(
            from_email='you@example.com',
            to_emails='user@example.com',
            subject='Your OTP Code',
            html_content='<strong>Your OTP is 123456</strong>'
        )
        response = sg.send(message)
        print(response.status_code)
    ```

- Sending OTP via Twilio SMS
    - Step 1: Install Twilio

      pip install twilio
    - Step 2: Get Twilio Account SID & Auth Token

      Sign up at https://twilio.com

      Get Account SID, Auth Token, and Phone Number
    - Step 3: Use in Django View

```
from twilio.rest import Client

def send_otp_sms(phone_number, otp):
    account_sid = 'your_account_sid'
    auth_token = 'your_auth_token'
    client = Client(account_sid, auth_token)

    message = client.messages.create(
        body=f'Your OTP code is {otp}',
        from_='+1234567890',  # your Twilio number
        to=phone_number
    )

    print(message.sid)
```

11. RESTful API Design

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

- REST (Representational State Transfer) is a set of rules for building web APIs.
- It helps create easy-to-use and scalable APIs.

- Statelessness

  Each API request should not depend on previous requests.

  The server does not store session data.

  The client (browser or app) must send all required data (like token, user ID) with every request.

  Makes APIs simple and scalable.

  Example:

  GET /api/users/

Headers: Authorization: Bearer <token>

- Resource-Based URLs

  In REST, data is treated as resources, and URLs are used to access these resources.

  Each type of data (like users, posts, doctors) gets its own URL.

  Nouns, not verbs are used in URLs.

  Example:

  GET /api/doctors/ → List all doctors

  GET /api/doctors/5/ → Get doctor with ID 5

- HTTP Methods for CRUD

  Use standard HTTP methods to perform CRUD operations:

HTTP Method	Action	Example URL	Purpose
GET	Read/Retrieve	/api/doctors/	List all doctors
POST	Create	/api/doctors/	Add new doctor
PUT	Update	/api/doctors/5/	Replace doctor info
PATCH	Partial Update	/api/doctors/5/	Update part of doctor info
DELETE	Delete	/api/doctors/5/	Remove doctor

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

- What is CRUD, and why is it fundamental to backend development?

- CRUD stands for:

Operation	Description
Create	Adds new data to the database (e.g., creating a user, adding a doctor's profile)
Read	Retrieves existing data from the database (e.g., viewing a list of doctors)
Update	Modifies existing data (e.g., editing a doctor's info)
Delete	Removes data (e.g., deleting a doctor's profile)

- CRUD operations are the foundation of every application that stores and manipulates data.
- Why is CRUD Fundamental to Backend Development?

    - Core Functionality of Web Apps

        Every dynamic application must interact with data.

        Whether it's a blog, an e-commerce site, a hospital management system, or a social media app, CRUD operations are what allow users to:

        Sign up or log in
        Add and view content
        Edit profiles or posts
        Delete unwanted data

    - Database Operations

        CRUD operations directly translate to SQL queries or Django ORM operations behind the scenes:

        Create → INSERT
        Read → SELECT
        Update → UPDATE
        Delete → DELETE

    - Powering REST APIs

        RESTful APIs are designed around CRUD operations:

        GET = Read
        POST = Create
        PUT/PATCH = Update
        DELETE = Delete

    - User Interaction

        CRUD enables users to interact with the app's data:

        Create: Register, post articles, add comments
        Read: View pages, profiles, feeds
        Update: Edit posts, update settings
        Delete: Remove data or deactivate accounts

    - Essential for Admin Panels

        Admin interfaces (like Django Admin) are entirely based on CRUD.

        Admins use these to manage data without touching the code.

13. Authentication and Authorization API

- Difference between authentication and authorization.

Authentication	Authorization
In the authentication process, the identity of users are checked for providing the access to the system.	While in authorization process, a the person's or user's authorities are checked for accessing the resources.
In the authentication process, users or persons are verified.	While in this process, users or persons are validated.
It is done before the authorization process.	While this process is done after the authentication process.
It needs usually the user's login details.	While it needs the user's privilege or security levels.
Authentication determines whether the person is user or not.	While it determines What permission does the user have?
Generally, transmit information through an ID Token.	Generally, transmit information through an Access Token.
The user authentication is visible at user end.	The user authorization is not visible at the user end.

- Implementing authentication using Django REST Framework's token-based system.

➢ Token-based authentication allows users to log in and receive a token.
➢ This token is sent with future requests to prove who they are — like a digital ID card.

➢ Step-by-Step Implementation
  - Step 1: Install Required Packages

    pip install djangorestframework
    pip install djangorestframework-simplejwt
    Also, make sure rest_framework is added in INSTALLED_APPS in your settings.py:

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

- Step 2: Configure REST Framework

  In settings.py:

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

- Step 3: Create Token Table

  You need to create database tables for token authentication:

```
python manage.py migrate
```

- Step 4: Generate Tokens for Users

  In your urls.py, add the endpoint to generate a token:

```
from django.urls import path
from rest_framework.authtoken.views import
obtain_auth_token

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

- Step 5: Protect Your Views

  Add authentication_classes and permission_classes to your views:

```
from rest_framework.authentication import
TokenAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView
from rest_framework.response import Response

class ProtectedDoctorListView(APIView):
   authentication_classes = [TokenAuthentication]
   permission_classes = [IsAuthenticated]

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

- Install djangorestframework & token auth, Add config in settings.py, Migrate database, Add token URL, Authenticate user and use token, Protect views using DRF classes

14. OpenWeatherMap API Integration

- Introduction to OpenWeatherMap API and how to retrieve weather data.

- The OpenWeatherMap API is a free (with paid tiers) web service that provides current weather, forecast, and historical weather data for any location in the world.
- You can access weather data using HTTP requests in your app or website.

- Step-by-Step Guide to Use OpenWeatherMap API
    - Step 1: Sign Up and Get an API Key

      Go to https://openweathermap.org/api

      Create a free account

      After login, go to your API keys page and copy your key (e.g., abc123xyz)

    - Step 2: Choose an API Endpoint

      The most commonly used endpoint is:

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

    - Step 3: Send a Request

      You can make a request using:

      City name

      Latitude & longitude

      City ID

      ZIP code

      Example: Get Weather by City Name

      GET

      https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY&units=metric

    - Step 4: Example Response

```
{
 "coord": { "lon": -0.1257, "lat": 51.5085 },
 "weather": [{ "main": "Clouds", "description": "broken clouds" }],
 "main": {
  "temp": 15.52,
  "feels_like": 14.87,
  "humidity": 72
 },
 "name": "London"
```

```
}
```

- Step 5: Python Example (Using requests)

```
import requests

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

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

print(f"City: {data['name']}")
print(f"Temperature: {data['main']['temp']} °C")
print(f"Condition: {data['weather'][0]['description']}")
```

15. Google Maps Geocoding API

- Using Google Maps Geocoding API to convert addresses into coordinates.

➢ The Geocoding API is a service by Google that lets you convert an address into geographic coordinates (latitude and longitude), and vice versa (reverse geocoding).

➢ Step-by-Step Guide: Converting Address to Coordinates

- Step 1: Get a Google API Key

  Go to the Google Cloud Console
  Create a new project (or select one)
  Enable the **Geocoding API** from the API Library
  Go to APIs & Services > Credentials
  Generate a new API Key

- Generate a new API Key

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

GET
https://maps.googleapis.com/maps/api/geocode/json?address=New+York,+NY&key=YOUR_API_KEY

- Step 3: Sample JSON Response

```
{
  "results": [
    {
      "geometry": {
        "location": {
          "lat": 40.7127753,
          "lng": -74.0059728
        }
      },
      "formatted_address": "New York, NY, USA"
    }
  ],
  "status": "OK"
}
```

- Step 4: Python Example Using requests

```
import requests

address = "New York, NY"
api_key = "YOUR_API_KEY"
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']
    print(f"Latitude: {location['lat']}")
    print(f"Longitude: {location['lng']}")
else:
    print("Error:", data['status'])
```

16. GitHub API Integration

- Introduction to GitHub API and how to interact with repositories, pull requests, and issues.

➢ The GitHub API allows developers to interact with GitHub programmatically.

➢ You can:

List and manage repositories
Create and update issues
Manage pull requests
Access user data
Monitor activity and contributions

➢ Authentication

To use the GitHub API securely:

Go to https://github.com/settings/tokens
Generate a Personal Access Token (PAT) with scopes like:
repo (for accessing repos, PRs, issues)
user (for user info)

Then use it in headers:

Authorization: token YOUR_PERSONAL_ACCESS_TOKEN

➢ Get Public Repositories of a User

Request:

GET https://api.github.com/users/octocat/repos

Response:

```
[
 {
  "name": "Hello-World",
  "html_url": "https://github.com/octocat/Hello-World",
  "language": "Python"
 },
 ...
]
```

➢ Create an Issue in a Repo

Request:

POST https://api.github.com/repos/username/repo-name/issues

Headers:

Authorization: token YOUR_TOKEN
Content-Type: application/json

- Create a Pull Request

  Request:

  POST https://api.github.com/repos/username/repo-name/pulls

  Body:

```
{
 "title": "Fix login bug",
 "head": "bugfix/login-issue",
 "base": "main",
 "body": "This PR fixes issue #15"
}
```

- Example with Python and requests

```
import requests

TOKEN = "your_token"
REPO = "username/repo-name"
headers = {"Authorization": f"token {TOKEN}"}

# Get issues
response = requests.get(f"https://api.github.com/repos/{REPO}/issues",
headers=headers)
issues = response.json()
for issue in issues:
   print(f"Issue: {issue['title']} - {issue['html_url']}")
```

17. Twitter API Integration

- Using Twitter API to fetch and post tweets, and retrieve user data.

- The Twitter API lets developers:

  Search for tweets

  Post tweets

  Get user profile data

  Monitor trends and mentions

  You need to apply for a developer account and get API keys to use it.

- Get Access

  Go to https://developer.twitter.com/

  Create a Twitter Developer Account

Create a Project & App

Get your credentials:

Bearer Token (for read access)

API Key, API Secret, Access Token, and Access Token Secret (for write access)

- Fetch Recent Tweets Using Keywords

  Endpoint:

  GET https://api.twitter.com/2/tweets/search/recent?query=django

  Headers:

  Authorization: Bearer YOUR_BEARER_TOKEN

  Python Example:

```
import requests

BEARER_TOKEN = 'YOUR_BEARER_TOKEN'
query = 'django'

url = f"https://api.twitter.com/2/tweets/search/recent?query={query}"
headers = {"Authorization": f"Bearer {BEARER_TOKEN}"}

response = requests.get(url, headers=headers)
print(response.json())
```

- Get User Data by Username

  Endpoint:

  GET https://api.twitter.com/2/users/by/username/{username}

  Example:

```
username = "elonmusk"
url = f"https://api.twitter.com/2/users/by/username/{username}"
response = requests.get(url, headers=headers)
print(response.json())
```

- Post a Tweet

  For this, you need OAuth 1.0a user context using:

  API Key

  API Secret Key

  Access Token

  Access Token Secret

  You can use tweepy (a Python library):

```
pip install tweepy
```

  Example:

```
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)

# Post a tweet
api.update_status("Hello, Twitter from Python!")
```

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 public web service that provides detailed information about countries around the world in JSON format.

- You can use it to get data such as:
  - Country name and code
  - Capital
  - Region and subregion
  - Languages
  - Currencies
  - Population
  - Area, borders, timezones, and flags

- API Endpoint
  - The base URL is:
    - https://restcountries.com/v3.1/

- Common API Endpoints

Purpose	Endpoint Example
Get all countries	https://restcountries.com/v3.1/all

Search by country name	https://restcountries.com/v3.1/name/{name}
Search by country code	https://restcountries.com/v3.1/alpha/{code}
Search multiple codes	https://restcountries.com/v3.1/alpha?codes=col,no,ee

- Example: Get Data About India
  - Request:

    GET https://restcountries.com/v3.1/name/india
  - Sample JSON Response:

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

- Python Example Using requests

```
import requests

country = "india"
url = f"https://restcountries.com/v3.1/name/{country}"
```

```
response = requests.get(url)
data = response.json()[0]  # First result

print(f"Country: {data['name']['common']}")
print(f"Capital: {data['capital'][0]}")
print(f"Population: {data['population']}")
print(f"Currency: {list(data['currencies'].keys())[0]}")
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 sent automatically based on user actions or triggers, such as:
  - Registration confirmation
  - Purchase receipts
  - Password reset instructions
  - Notifications or alerts
- These emails are essential for communication and often need to be delivered quickly and reliably.

- SendGrid API

  SendGrid is a popular email delivery service that allows you to send transactional emails via their API.

  Step-by-Step Guide: SendGrid

  - Sign Up: Create an account at SendGrid.
  - Generate API Key: After logging in, go to the Settings > API Keys section and create a new key.
  - Install the SendGrid Library: In Python, use sendgrid library:

    ```
    pip install sendgrid
    ```
  - Send an Email:

    ```
    import sendgrid
    from sendgrid.helpers.mail import Mail, Email, To, Content

    # Your SendGrid API Key
    ```

```
sg =
sendgrid.SendGridAPIClient(api_key='YOUR_SENDGRID_API_KEY')

# Email components
from_email = Email("your-email@example.com")
to_email = To("recipient@example.com")
subject = "Welcome to Our Service!"
content = Content("text/plain", "Thank you for signing up!")

# Create email
mail = Mail(from_email, to_email, subject, content)

# Send email
response = sg.send(mail)
print(response.status_code)
print(response.body)
print(response.headers)
```

- Mailchimp Transactional Email API (Mandrill)

  Mailchimp offers a service for sending emails via their Mandrill API, which is designed for transactional email delivery.

  Step-by-Step Guide: Mailchimp Mandrill API

  - Sign Up: Create an account at Mailchimp.
  - Get API Key: Go to Account > Extras > API Keys to generate an API key.
  - Install Mailchimp's Mandrill Library:

    ```
    pip install mandrill
    ```
  - Send an Email:

    ```
    import mandrill

    mandrill_client = mandrill.Mandrill('YOUR_MANDRILL_API_KEY')

    message = {
       'from_email': 'your-email@example.com',
       'to': [{'email': 'recipient@example.com', 'name': 'Recipient Name'}],
       'subject': 'Welcome!',
       'text': 'Thank you for joining our service!',
    }

    result = mandrill_client.messages.send(message=message)
    print(result)
    ```

20. SMS Sending APIs (Twilio)

➢ Introduction to Twilio API for sending SMS and OTPs.

➢ Twilio is a cloud communications platform that lets you programmatically send and receive SMS, make and receive calls, and perform other communication functions using its APIs

➢ Global SMS Delivery: Send messages worldwide.
➢ Secure OTP Generation: Easily generate and validate one-time passwords (OTPs).
➢ Reliable and Scalable: High delivery rate and scalable infrastructure.

➢ Getting Started with Twilio SMS API

- Create a Twilio Account

  Sign up at https://www.twilio.com/

  Get your:

  Account SID

  Auth Token

  Twilio phone number

- Install Twilio Python SDK

  pip install twilio

- Send an SMS with Twilio (Basic Example)

```
from twilio.rest import Client

# Twilio credentials from the console
account_sid = 'your_account_sid'
auth_token = 'your_auth_token'
client = Client(account_sid, auth_token)

message = client.messages.create(
   body='Hello, this is a test SMS from Twilio!',
   from_='+1234567890',  # Your Twilio number
   to='+919999999999'    # Recipient's number
)

print(f"Message sent! SID: {message.sid}")
```

➢ Sending OTP with Twilio (Manual Method)

- Generate an OTP

```
import random
otp = random.randint(100000, 999999)
```

- Send OTP via SMS

```
message = client.messages.create(
  body=f'Your OTP is {otp}',
  from_='+1234567890',
  to='+919999999999'
)
```

- Verify OTP

  You can store the OTP in session or database and verify it during user input.

21. Payment Integration (PayPal, Stripe)

- Introduction to integrating payment gateways like PayPal and Stripe.

➢ A payment gateway is a service that processes online payments securely.
➢ It authorizes credit card or direct payments for online transactions.

➢ PayPal vs Stripe – Quick Comparison

Feature	PayPal	Stripe
Popularity	Widely used for online checkout	Preferred by developers & SaaS
Setup	Quick and easy	Requires some coding knowledge
User Experience	Redirects to PayPal sometimes	Seamless, stays on your site
Ideal For	Quick integrations, donations	Subscriptions, full checkout

➢ PayPal Integration (Basic)

- Create a PayPal Developer Account

  Go to: https://developer.paypal.com/

  Create a sandbox app to get your Client ID and Secret
- Install PayPal SDK (for REST API)

  pip install paypalrestsdk
- Sample Python Code (PayPal Payment)

```
import paypalrestsdk

paypalrestsdk.configure({
```

```
    "mode": "sandbox",
    "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": [{
        "amount": {"total": "10.00", "currency": "USD"},
        "description": "Test payment"
    }]
})

if payment.create():
    print("Payment created successfully")
    for link in payment.links:
        if link.rel == "approval_url":
            print("Redirect user to:", link.href)
else:
    print(payment.error)
```

- Stripe Integration (Basic)
  - Create a Stripe Account

    Go to: https://dashboard.stripe.com/register

    Get your Publishable Key and Secret Key
  - Install Stripe SDK

    pip install stripe
  - Sample Python Code (Stripe Checkout Session)

```
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': 'Test Product'},
            'unit_amount': 1000,  # in cents
        },
        'quantity': 1,
    }],
    mode='payment',
    success_url='http://localhost:8000/payment-success',
    cancel_url='http://localhost:8000/payment-cancel',
)

print("Redirect to:", session.url)
```

22. Google Maps API Integration

- Using Google Maps API to display maps and calculate distances between locations.

➢ The **Google Maps Platform** provides powerful APIs to integrate:
Interactive maps
Geolocation
Place autocomplete
Distance calculation
Directions and more

➢ Here's a friendly and practical introduction to using the Google Maps API to:
Display maps
Calculate distances between locations

➢ Step-by-Step Setup
- Create a Google Cloud Project
  Visit: https://console.cloud.google.com/
  Create a project and enable:
    Maps JavaScript API
    Distance Matrix API
    Geocoding API
- Get Your API Key
  Go to APIs & Services > Credentials and generate an API key.

- Displaying a Map (JavaScript)

```
<!DOCTYPE html>
<html>
<head>
 <title>Simple Map</title>
 <style>
  #map { height: 400px; width: 100%; }
 </style>
</head>
<body>
 <h3>My Google Map</h3>
 <div id="map"></div>

 <script>
  function initMap() {
   const location = { lat: 28.6139, lng: 77.2090 }; // Example: New Delhi
   const map = new google.maps.Map(document.getElementById("map"), {
    zoom: 10,
    center: location,
   });
   const marker = new google.maps.Marker({ position: location, map: map });
  }
 </script>

 <script async

src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initM
ap">
 </script>
</body>
</html>
```

- Calculate Distance Between Two Locations

```
<script>
 function calculateDistance() {
  const service = new google.maps.DistanceMatrixService();
  service.getDistanceMatrix(
   {
    origins: ['New Delhi, India'],
```

```
        destinations: ['Mumbai, India'],
        travelMode: 'DRIVING',
      },
      (response, status) => {
        if (status === 'OK') {
          const distance = response.rows[0].elements[0].distance.text;
          const duration = response.rows[0].elements[0].duration.text;
          alert(`Distance: ${distance}, Duration: ${duration}`);
        }
      }
    );
  }
</script>
```