

## ★ Module 17: Rest Framework

### 1. Introduction to APIs

#### **Q1:What is an API (Application Programming Interface)?**

**Ans:** APIs define how requests are made and responses are delivered, often using standard formats like JSON or XML, allowing developers to integrate services without rebuilding them from scratch. For instance, a weather app uses a weather service's API to fetch real-time data by sending a request and receiving structured results.

#### **How It Works**

A client application sends a request to an API endpoint, which processes it through a server or gateway, retrieves the necessary data or executes an action, and returns a response. This intermediary role simplifies interactions, much like a waiter relaying orders in a restaurant.

#### **Key Benefits**

APIs promote efficiency by enabling modular development, foster innovation through reusable services, and support scalability in modern systems like web and mobile apps. In backend development, such as with Django, APIs power RESTful services for handling enquiries or payments in your projects.

#### **Q2:Types Of Apis(Rest , Soap)?**

**Ans:** REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) represent two prominent API architectural styles, each suited to different use cases in backend development like your Django projects.

- **REST Overview**

REST is an architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD operations on resources identified by URLs. It supports flexible data formats like JSON or XML, remains stateless for better scalability, and excels in web and mobile applications due to lightweight messaging and caching.

- **SOAP Overview**

SOAP is a strict protocol relying on XML for structured messaging, often over protocols like HTTP or SMTP, with built-in standards for security (WS-Security) and reliability. It exposes operations via functions defined in WSDL, maintaining state across requests, which suits enterprise environments needing robust error handling.

### **Q3:Why are APIs important in web development?**

**Ans:**

APIs are crucial in web development because they enable seamless communication between different software components, services, and applications, allowing developers to build more dynamic and integrated systems.

#### **Faster Development**

APIs provide pre-built functionalities like payment gateways or maps, reducing the need to code everything from scratch and accelerating project timelines for Django apps with features like enquiries or maintenance tracking.

#### **Scalability and Modularity**

They support modular architectures where components scale independently, handling increased traffic or adding features without overhauling the entire backend.

#### **Enhanced User Experience**

APIs integrate third-party services for real-time data, personalization, and interactivity, such as social logins or geolocation, making web apps more engaging and efficient.

## **2. Requirements for Web Development Projects**

### **Q1:Understanding project requirements.**

**Ans:**

#### **Gathering Requirements**

Understanding project requirements involves clarifying client or stakeholder needs through direct discussions, documentation reviews, and identifying functional (e.g., user authentication) and non-functional (e.g., performance) aspects.

#### **Key Techniques**

- Conduct stakeholder interviews to capture explicit and implicit needs.
- Use user stories or use cases to outline scenarios, like "As a tenant, I want to submit maintenance requests via the app."
- Create prototypes or wireframes for visual feedback.

## Tools and Documentation

Employ tools like Jira or Trello for tracking, and produce requirement documents such as SRS (Software Requirements Specification) with ERDs for database design in Django projects. Prioritize requirements using MoSCoW method (Must, Should, Could, Won't).

## Q2:Setting up the environment and installing necessary packages.

**Ans:**Setting up a development environment for Django projects starts with creating an isolated virtual environment to manage dependencies cleanly, followed by installing essential packages like Django itself.

### Create Virtual Environment

Navigate to your project directory and run `python -m venv .venv` (or `py -m venv .venv` on Windows) to generate a `.venv` folder with an isolated Python setup. Activate it using `source .venv/bin/activate` on Unix/macOS or `.venv\Scripts\activate` on Windows; your prompt changes to indicate activation. Exclude `.venv` from Git via `.gitignore` to avoid committing environment files.

### Install Packages

Upgrade pip first: `python -m pip install --upgrade pip`. Install Django and core packages: `pip install django` or use a `requirements.txt` file with `pip install -r requirements.txt` listing pinned versions like `Django==5.1` for reproducibility. Common additions for your projects include `djangorestframework` for APIs, `psycopg2` for PostgreSQL, and `python-decouple` for environment variables.

### Initialize Django Project

Run `django-admin startproject myproject .` to scaffold the project structure, then `python manage.py startapp home` for apps like `home` or `amenities`. Test with `python manage.py runserver` and verify `pip freeze > requirements.txt` to lock dependencies. Use PyCharm or VS Code for IDE support, aligning with your preferred tools.

### **3. Serialization in Django REST Framework**

#### **Q1:What is Serialization?**

**Ans:**Serialization converts complex data types, such as Django model instances or querysets, into formats like JSON that can be easily rendered into HTTP responses or sent over networks.

#### **Role in Django REST Framework**

Serializers act as a bridge between Django models and API endpoints, handling both data output (serialization) and input validation/deserialization for requests. For your projects, a serializer like `CommentSerializer` transforms a model's fields (e.g., email, content) into JSON for API responses in apps like enquiries or maintenance.

#### **Key Types**

- `ModelSerializer`: Automatically maps model fields, generates validators, and provides create/update methods.
- `HyperlinkedModelSerializer`: Uses hyperlinks for relationships instead of IDs, enhancing RESTful navigation.

#### **Q2:Converting Django QuerySets to JSON.**

**Ans:**

##### **Using Django Core Serializers**

Import `django.core.serializers` and use `serializers.serialize('json', queryset)` to convert a `QuerySet` of model instances into JSON, including model metadata and primary keys. Example in a view:

```
text

from django.core import serializers
from django.http import JsonResponse

def book_list(request):
    queryset = Book.objects.all()
    data = serializers.serialize('json', queryset)
    return JsonResponse(data, content_type='application/json')
```

This outputs full object details but does not work directly on `.values()` querysets.

##### **Using JsonResponse with `.values()`**

For lightweight JSON without metadata, apply `.values()` to select fields, convert to list, and return via `JsonResponse(data, safe=False)`. Example:

```
text

from django.http import JsonResponse

def book_list(request):
    data = list(Book.objects.values('title', 'author'))
    return JsonResponse(data, safe=False)
```

This produces a simple list of dictionaries, ideal for REST APIs in your Django projects.

## DRF Serializers (Recommended)

Define a `ModelSerializer` in Django REST Framework for validation, nested relations, and custom output; instantiate and call `.data` on the serialized QuerySet. This method handles complex scenarios like your multi-app projects (e.g., amenities, enquiry)

## Q3: Using `serializers` in Django REST Framework (DRF).

**Ans:**Serializers in Django REST Framework (DRF) convert complex data types like QuerySets and model instances to JSON/XML and validate incoming data for API endpoints.

### Basic Serializer Usage

Define a serializer class inheriting from `serializers.Serializer`, specify fields, then instantiate with data: `serializer = CommentSerializer(data=request.data)`. Call `serializer.is_valid()` for validation and `serializer.data` or `serializer.save()` to process output/input.

### ModelSerializer Example

For Django models, use `ModelSerializer` which auto-generates fields and CRUD methods:

```
from rest_framework import serializers
from .models import Enquiry
```

```
class EnquirySerializer(serializers.ModelSerializer):
    class Meta:
        model = Enquiry
        fields = ['id', 'email', 'message', 'created_at']
```

In views: `serializer = EnquirySerializer(queryset, many=True)` serializes QuerySets; `serializer.data` yields JSON-ready dicts.

## View Integration

Use in `APIView` or generics:

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

@api_view(['GET'])
def enquiry_list(request):
    queryset = Enquiry.objects.all()
    serializer = EnquirySerializer(queryset, many=True)
    return Response(serializer.data)
```

Handles serialization, validation, and deserialization automatically for your multi-app Django projects.

## 4. Requests and Responses in Django REST Framework

### q1:HTTP request methods (GET, POST, PUT, DELETE).

**Ans:**

HTTP methods like GET, POST, PUT, and DELETE form the foundation of RESTful APIs in Django REST Framework, mapping directly to CRUD operations (Create, Read, Update, Delete).

#### **GET Method**

GET retrieves data without side effects, used for listing or fetching single resources (e.g., `Enquiry.objects.all()` for enquiry lists). Idempotent and cacheable, ideal for read-only API endpoints in your Django apps.

#### **POST Method**

POST creates new resources by submitting data via `request.data`, validated through serializers before saving (returns 201 Created on success). Use for submitting new maintenance requests or enquiries.

#### **PUT Method**

PUT updates or creates entire resources with complete data replacement, requiring full object representation (idempotent). Example: `serializer =`

`EnquirySerializer(enquiry, data=request.data, partial=False)` for full enquiry updates.

## **DELETE Method**

`DELETE` removes resources by primary key, returning `204 No Content` on success. Safe for idempotent deletion like removing amenities: `enquiry.delete()`.

## **DRF View Example**

```
python

@api_view(['GET', 'POST', 'PUT', 'DELETE'])
def enquiry_detail(request, pk):
    try:
        enquiry = Enquiry.objects.get(pk=pk)
    except Enquiry.DoesNotExist:
        return Response(status=404)

    if request.method == 'GET':
        serializer = EnquirySerializer(enquiry)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = EnquirySerializer(enquiry, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=400)
    elif request.method == 'DELETE':
        enquiry.delete()
        return Response(status=204)
```

## **Q2: Sending and receiving responses in DRF.**

**Ans:**

### **Sending Responses**

Use `Response(serializer.data)` to return serialized data with automatic content negotiation (JSON/XML based on client `Accept` header). Include status codes like `Response(data, status=status.HTTP_201_CREATED)` for POST success or `Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)` for validation failures.

## Receiving Requests

Access parsed input via `request.data` (handles JSON form-data), `request.query_params` for GET parameters, and `request.user` for authentication. DRF automatically deserializes and validates:

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

## Complete View Example

```
python

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

@api_view(['POST'])
def create_enquiry(request):
    serializer = EnquirySerializer(data=request.data)
    if serializer.is_valid():
        enquiry = serializer.save()
        return Response({'id': enquiry.id, 'message': 'Created'},
                        status=status.HTTP_201_CREATED)
    Return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

## 5. Views in Django REST Framework

**Q1: Understanding views in DRF: Function-based views vs Class-based views.**

**Ans :** Function-based views (FBVs) in DRF use `@api_view` decorators for simple, explicit HTTP method handling, while class-based views (CBVs) like `APIView`, Generic Views, and ViewSets inherit built-in functionality for more complex APIs.

### Function-Based Views

FBVs offer direct control with `@api_view(['GET', 'POST'])` decorators, manually handling `request.method` logic. Simple for basic CRUD like listing enquiries:

```
python
```

```
@api_view(['GET'])

def enquiry_list(request):

    queryset = Enquiry.objects.all()

    serializer = EnquirySerializer(queryset, many=True)

    return Response(serializer.data)
```

Best for straightforward endpoints or when learning DRF basics.

## Class-Based Views

CBVs provide inheritance and reusability through classes extending `APIView` or mixins:

`APIView`: Base class with `request/response` handling:

python

```
class EnquiryList(APIView):

    def get(self, request):

        queryset = Enquiry.objects.all()

        serializer = EnquirySerializer(queryset,
many=True)

        return Response(serializer.data)
```

`Generic Views` (`ListAPIView`, `CreateAPIView`): Pre-built for common patterns, reducing boilerplate.

`ViewSets` (`ModelViewSet`): Single class handles all CRUD actions, auto-routed by DRF routers.

## 6. URL Routing in Django REST Framework

**Q1: Defining URLs and linking them to views.**

**Ans :** Defining URLs and linking them to views.

Django REST Framework (DRF) uses URL configuration to map API endpoints to views through `path()` functions or automatic routers for ViewSets.

### **Function-Based View URLs**

For `@api_view` functions, define explicit paths in `urls.py`:

```
python
```

```
# app/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('enquiries/', views.enquiry_list, name='enquiry-list'),
    path('enquiries/<int:pk>', views.enquiry_detail,
         name='enquiry-detail'), ]
```

Include app URLs in main project: `path('api/', include('app.urls'))`.

### **Class-Based View URLs**

For `APIView` classes, use `.as_view()`:

```
python
```

```
from rest_framework.views import APIView
```

```
urlpatterns = [  
  
    path('enquiries/' , EnquiryList.as_view() , name='enquiry-list') ,  
  
    path('enquiries/<int:pk>/' , EnquiryDetail.as_view() ,  
name='enquiry-detail') ,  
  
]
```

## Router URLs (Recommended for ViewSets)

Automatically generate CRUD URLs with DefaultRouter:

```
python  
  
# app/urls.py  
  
from rest_framework.routers import DefaultRouter  
  
from . import views  
  
  
  
router = DefaultRouter()  
  
router.register(r'enquiries' , views.EnquiryViewSet) #  
Creates /enquiries/ , /enquiries/{id}/  
  
urlpatterns = [  
  
    path('api/' , include(router.urls)) , ]
```

Generated URLs: /api/enquiries/ (list/create), /api/enquiries/123/  
(retrieve/update/delete).

## Project-Level Integration

In project/urls.py:

```
python

from django.urls import path, include

urlpatterns = [
    path('api/v1/', include('home.urls')), # App-specific APIs
    path('api/v1/', include('amenities.urls')),
    path('api/v1/', include('maintenance.urls')),
]
```

## 7. Pagination in Django REST Framework

### **Q1: Adding pagination to APIs to handle large data sets.**

**Ans:** Pagination splits large QuerySet results into smaller pages to improve API performance, reduce server load, and enhance client experience when handling large datasets like enquiry lists or maintenance records.

Settings Configuration:

```
python
```

```
# settings.py - Global pagination
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20 # Items per page
}
```

### **Pagination Types:**

Type	Query Parameters	Response Structure	Use Case
PageNumberPagination	?page=2	count, next, previous, results	Standard web pagination
LimitOffsetPagination	?limit=10&offset=20	count, next, previous, results	Infinite scroll
CursorPagination	?cursor=cD0yMA==	next, previous, results	Secure large datasets

View-Level Implementation:

```
python

# ViewSet with pagination

class EnquiryViewSet(viewsets.ModelViewSet):

    queryset = Enquiry.objects.all()

    serializer_class = EnquirySerializer

    pagination_class = PageNumberPagination

    page_size = 10 # Override global setting

    page_size_query_param = 'page_size'

    max_page_size = 100
```

Generic View Example:

```
python

class EnquiryList(generics.ListAPIView):

    queryset = Enquiry.objects.all()

    serializer_class = EnquirySerializer

    pagination_class = PageNumberPagination
```

Disable Pagination:

```
python

# Per view

pagination_class = None
```

```
# Query parameter  
  
GET /enquiries/?page_size=0 # Returns all records
```

API Response Format:

json

```
{ "count": 150,  
  
  "next": "http://api.example.com/enquiries/?page=2",  
  
  "previous": null,  
  
  "results": [  
  
    {"id": 1, "email": "user@example.com", "message": "..."}  
  ]}
```

### Client Usage:

text

```
GET /api/enquiries/?page=1&page_size=5
```

```
GET /api/enquiries/?limit=10&offset=20 # LimitOffset style
```

### Custom Pagination Class:

python

```
class CustomPagination(PageNumberPagination):  
  
    page_size = 25  
  
    page_size_query_param = 'page_size'  
  
    max_page_size = 100  
  
    page_query_param = 'p' # Custom parameter name
```

## **8. Settings Configuration in Django**

### **Q1: Configuring Django settings for database, static files, and API keys.**

**Ans:** Django settings configuration covers database connections, static file handling, and secure API key management, essential for production-ready projects. Proper setup ensures security, scalability, and efficient asset serving.

#### **Database Configuration**

Configure databases in the `DATABASES` dictionary within `settings.py`. For SQLite (development default), use `ENGINE: 'django.db.backends.sqlite3'` and specify `NAME`. For PostgreSQL in production, set `ENGINE: 'django.db.backends.postgresql'`, add `NAME, USER, PASSWORD, HOST, and PORT`, often loaded from environment variables for security.

#### **Static Files Setup**

Define `STATIC_URL = '/static/'` and `STATICFILES_DIRS` for collecting CSS, JS, and images from app directories. Run `python manage.py collectstatic` to gather files into `STATIC_ROOT` for web servers like Nginx. Enable `ManifestStaticFilesStorage` in `STORAGES` for cache-busting via content hashes.

#### **API Keys Management**

Store API keys in environment variables accessed via

`os.environ.get('API_KEY_NAME')` in `settings.py`, avoiding hardcoding. Use a `.env` file with `python-dotenv` for local development, and add `.env` to `.gitignore`. For Django REST Framework, integrate packages like `djangorestframework-apikey` with custom permissions like `HasAPIKey`.

## 9. Project Setup

### **Q1:Setting up a Django REST Framework project.**

**Ans:**Django REST Framework (DRF) simplifies building robust APIs on top of Django projects. Setting it up involves installing packages, configuring settings, creating serializers and views, and wiring up URLs for endpoints.

#### **Project Setup Steps**

Create a virtual environment with `python -m venv env`, activate it (`source env/bin/activate`), then install Django and DRF using `pip install django djangorestframework`. Start a Django project (`django-admin startproject myproject .`) and app (`python manage.py startapp myapi`), then run initial migrations (`python manage.py migrate`).

#### **Settings Configuration**

Add '`rest_framework`' to `INSTALLED_APPS` in `settings.py`. Include basic DRF settings like `REST_FRAMEWORK = {'DEFAULT_PERMISSION_CLASSES': ['rest_framework.permissions.IsAuthenticated']}` for authentication control. For browsable API access, add login URLs later.

#### **Serializers and Models**

Define models in `models.py` (e.g., a simple `Todo` model with fields like `title` and `completed`). Create `serializers.py` with `ModelSerializer` classes inheriting from `serializers.ModelSerializer`, specifying `model = Todo` and `fields = '__all__'`.

#### **Views and URL Routing**

Use `ViewSets` in `views.py` (e.g., `ModelViewSet` with `queryset = Todo.objects.all()` and `serializer_class = TodoSerializer`). Set up

```
routers in urls.py: router = routers.DefaultRouter();
router.register(r'todos', views.TodoViewSet) and include path ('',
include(router.urls)). Test endpoints at http://127.0.0.1:8000/todos/
after python manage.py runserver.
```

## **10. Social Authentication, Email, and OTP Sending API**

### **Q1: Implementing social authentication (e.g., Google, Facebook) in Django.**

**Ans:** Django social authentication enables users to log in via providers like Google or Facebook using packages such as django-allauth. This approach streamlines registration and leverages OAuth2 for secure token exchange.

#### **Package Installation**

Install django-allauth with pip install django-allauth. Add 'allauth', 'allauth.account', 'allauth.socialaccount', and provider-specific apps (e.g., 'allauth.socialaccount.providers.google') to INSTALLED\_APPS in settings.py. Set SITE\_ID = 1, AUTHENTICATION\_BACKENDS = ['allauth.account.auth\_backends.AuthenticationBackend'], and LOGIN\_REDIRECT\_URL = '/' for post-login routing.

#### **Provider Configuration**

Register social applications in Django admin: log in at /admin/, navigate to "Social applications," add entries for Google/Facebook with Client ID, Secret Key from their developer consoles, and select your site (e.g., example.com). For Google, enable OAuth consent screen and add authorized redirect URLs like

`http://localhost:8000/accounts/google/login/callback/`.

#### **URL and Migration Setup**

Include path('accounts/', include('allauth.urls')) in main urls.py. Run python manage.py migrate to create necessary tables for user associations. Add login links in templates: {%- load socialaccount %}{% providers\_media\_js %}{% socialaccount\_login 'google' %} for buttons.

## Custom Handling

Override pipelines in `settings.py` for user creation logic, like `SOCIALACCOUNT_AUTO_SIGNUP = True` or custom `SOCIAL_AUTH_PIPELINE` for email association. Test by running `python manage.py runserver` and accessing `/accounts/google/login/`.

## Q2: Sending emails and OTPs using third-party APIs like Twilio, SendGrid.

**Ans:** Django integrates seamlessly with third-party services like Twilio for OTPs via SMS and SendGrid for emails, enhancing user verification and notifications. These services require API keys stored securely in environment variables, with Django's email backend or client libraries handling the transmission.

### SendGrid Email Setup

Install `pip install sendgrid` or `sendgrid-django`, then configure `settings.py` with `EMAIL_HOST = 'smtp.sendgrid.net'`, `EMAIL_PORT = 587`, `EMAIL_USE_TLS = True`, `EMAIL_HOST_USER = 'apikey'`, and `EMAIL_HOST_PASSWORD = os.environ.get('SENDGRID_API_KEY')`. Send emails using Django's `send_mail()`: `send_mail('Subject', 'Message', settings.DEFAULT_FROM_EMAIL, ['recipient@example.com'])`, or `EmailMessage` for HTML templates with attachments.

### Twilio OTP Implementation

Sign up for Twilio, get Account SID, Auth Token, and a phone number, then install `pip install twilio`. In views, use `from twilio.rest import Client; client = Client(account_sid, auth_token)` to send OTP:  
`client.messages.create(body='Your OTP is 123456', from_='+1234567890', to='+919876543210')` [prior Django context]. Verify OTPs by storing a random 6-digit code in session or cache (e.g., `cache.set('otp_ + phone, code, 300)`), checking against user input.

### Best Practices

Load credentials from `.env` using `python-decouple` or `django-environ` to avoid hardcoding. Use Celery for async tasks on production to prevent blocking: `@shared_task def send_otp(phone, otp): ....` Test in development with Twilio's sandbox and SendGrid's single sender verification.

## 11. RESTful API Design

### **Q1: REST principles: statelessness, resource-based URLs, and using HTTP methods for CRUD operations.**

**Ans:** REST principles form the foundation of scalable web APIs, emphasizing stateless communication, resource-oriented design, and standard HTTP methods for operations. These ensure predictability, cacheability, and ease of maintenance in Django REST Framework implementations.

#### **Statelessness**

Statelessness requires each HTTP request to contain all necessary information for processing, without server reliance on prior session data. Servers treat every request independently, improving scalability by avoiding state storage and enabling horizontal scaling across multiple instances. Clients manage state via tokens (e.g., JWTs) included in headers, aligning with HTTP's inherent design.

#### **Resource-Based URLs**

URLs represent resources as nouns (e.g., `/api/users/123/` for a specific user), using hierarchical paths to denote relationships like `/api/users/123/orders/`. Avoid verbs in paths; actions derive from HTTP methods instead. This uniform interface promotes discoverability through hypermedia links (HATEOAS).

#### **HTTP Methods for CRUD**

Map CRUD operations to HTTP verbs: GET retrieves resources (`/users/` lists all), POST creates new ones, PUT/PATCH updates (idempotent full/partial), and DELETE removes them. Return appropriate status codes like 200 OK, 201 Created, 204 No Content, or 404 Not Found for clear feedback.

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

### **Q1: What is CRUD, and why is it fundamental to backend development?**

**Ans:** CRUD stands for Create, Read, Update, and Delete—the four fundamental operations for managing data in databases and applications. These operations form the core of backend development by enabling persistent data manipulation across web APIs, mobile apps, and storage systems.

#### **CRUD Operations Defined**

Create adds new records (mapped to SQL INSERT or HTTP POST), Read retrieves data (SQL SELECT or GET), Update modifies existing entries (SQL UPDATE or PUT/PATCH), and Delete removes them (SQL DELETE or HTTP DELETE). In Django REST Framework, these align directly with ViewSets handling list/create (GET/POST), retrieve/update/partial\_update/destroy (GET/PUT/PATCH/DELETE) [prior conversation context].

#### **Why Fundamental to Backend**

CRUD provides a universal pattern for data persistence, ensuring applications can handle user interactions scalably without custom logic for every action. Backend developers rely on it for RESTful APIs, ORM queries (like Django's Model methods), and database transactions, promoting modularity, security through permissions, and easy testing. Without CRUD mastery, building maintainable projects like your Django apps becomes inefficient.

## **13. Authentication and Authorization API**

### **Q1: Difference between authentication and authorization.**

**Ans:** Authentication verifies a user's identity (who they are), while authorization determines their permissions (what they can do). Authentication precedes authorization in security workflows, ensuring only verified users gain controlled access.

#### **Key Differences**

Authentication involves credentials like passwords, tokens, or biometrics to confirm identity, often via login forms or OAuth flows in Django (e.g., `django-allauth` for social

login). Authorization checks roles or policies post-authentication, using Django's permission classes like `IsAuthenticated` or custom DRF permissions

Aspect	Authentication	Authorization
Purpose	Verifies identity	Grants/denies access
Timing	First step	After authentication
Examples	Username/password, JWT tokens	Role-based access (RBAC), scopes
Failure	Access denied entirely	Partial access restricted <a href="#">auth0</a>

## Django Context

In DRF projects, set `DEFAULT_AUTHENTICATION_CLASSES` (e.g., `TokenAuthentication`) for identity checks and `DEFAULT_PERMISSION_CLASSES` (e.g., `IsAdminUser`) for action control. This aligns with your social auth and API setups, preventing unauthorized CRUD operations.

## Q1: Implementing authentication using Django REST Framework's token-based system.

**Ans:**

Django REST Framework's token-based authentication provides a simple, secure way to verify API requests using unique tokens per user. Tokens are generated upon login and sent in the `Authorization: Token <key>` header for protected endpoints.

## Setup Configuration

Add '`rest_framework.authtoken`' to `INSTALLED_APPS` and configure `REST_FRAMEWORK = {'DEFAULT_AUTHENTICATION_CLASSES': ['rest_framework.authentication.TokenAuthentication']}` in `settings.py`. Run `python manage.py migrate` to create the `authtoken_token` table storing user tokens.

## Token Generation

Include `path('api-token-auth/', obtain_auth_token)` in `urls.py` for the login endpoint. Clients POST username/password to `/api-token-auth/` to receive a

`token: {"token": "9054f7aa9305e012b3c2300408c3cddf390fcddde"}`. Use `python manage.py drf_create_token <username>` for manual testing.

## Protected Views

Apply `@permission_classes([IsAuthenticated])` or `permission_classes = [IsAuthenticated]` to views/ViewSets. Access protected data with headers: `curl -H "Authorization: Token 9054f7aa9305e012b3c2300408c3cddf390fcddde"` `http://127.0.0.1:8000/api/users/`. In Python requests: `headers={'Authorization': 'Token your_token_key'}`.

## 14. OpenWeatherMap API Integration

### Q1: Introduction to OpenWeatherMap API and how to retrieve weather data.

**Ans:**

#### API Overview

Sign up at [openweathermap.org](https://openweathermap.org) to generate a free API key (under API Keys tab), valid for basic current weather (60 calls/minute) and 5-day forecasts. Core endpoints include Current Weather (`/data/2.5/weather`), One Call API 3.0 for comprehensive data (hourly/daily forecasts, alerts), and geocoding for lat/lon by city name.

#### Retrieving Data

Make GET requests with your key:

```
https://api.openweathermap.org/data/2.5/weather?q=London&appid={YOUR_API_KEY}&units=metric returns JSON like {"main": {"temp": 15.5}, "weather": [{"description": "clear sky"}]}. Use lat/lon for precision: /weather?lat=51.5074&lon=-0.1278&appid={key}. In Django, use requests.get() within views: response = requests.get(url).json(); weather = response['weather'][0]['main'].
```

#### Django Integration Tips

Store API key in `.env` via `os.environ.get('OWM_API_KEY')` for security. Cache responses with Django's cache framework (`cache.set('london_weather', data, 1800)`) to respect rate limits. Handle errors (e.g., 401 for invalid key) and add to DRF serializers for authenticated weather endpoints in your projects.

## 15. GoogleMaps GeoCoding API

### **Q1: Using Google Maps Geocoding API to convert addresses into coordinates.**

**Ans:**Google Maps Geocoding API converts human-readable addresses into precise latitude/longitude coordinates and vice versa, essential for location-based Django features like mapping user addresses or weather lookups. It requires a Google Cloud API key with billing enabled, complementing your OpenWeatherMap integration for full geolocation workflows.

#### **API Setup**

Create a Google Cloud project, enable the Geocoding API under APIs & Services, and generate an API key. Restrict the key to Geocoding API only for security, storing it in `.env` as `GOOGLE_MAPS_API_KEY`. Free tier offers ~\$200 monthly credit; monitor usage to avoid overages.

#### **Python Implementation**

```
Install googlemaps via pip install googlemaps, then initialize: gmaps = googlemaps.Client(key=os.environ.get('GOOGLE_MAPS_API_KEY')). Geocode addresses with geocode_result = gmmaps.geocode('1600 Amphitheatre Parkway, Mountain View, CA'), extracting lat = result[0]['geometry']['location']['lat']; lng = result[0]['geometry']['location']['lng'].
```

#### **Django Integration**

```
Create a DRF view: @api_view(['GET']) def geocode_address(request): address = request.GET.get('address'); result = gmmaps.geocode(address); return Response({'lat': result[0]['geometry']['location']['lat'], 'lng': result[0]['geometry']['location']['lng']}). Cache results (cache.set(f"geo_{address}", coords, 3600)) and handle ZERO_RESULTS status for invalid addresses.
```

#### **Error Handling & Best Practices**

Check `status` field: `OK`, `ZERO_RESULTS`, or `OVER_QUERY_LIMIT`. Use `components=country:IN` parameter for India-specific accuracy (Ahmedabad addresses). Combine with OpenWeatherMap: geocode → fetch weather by lat/lon for authenticated user endpoints.

## 16. GitHub API Integration

### **Q1: Introduction to GitHub API and how to interact with repositories, pull requests, and issues.**

**Ans:** GitHub API enables programmatic interaction with repositories, pull requests, issues, and user data through REST endpoints and GraphQL. It requires authentication via Personal Access Tokens (PAT) or OAuth for private resources, making it perfect for Django apps managing code workflows or integrating with your projects.

#### **Authentication Setup**

Generate a PAT at [github.com/settings/tokens](https://github.com/settings/tokens) with scopes like `repo, issues, pull_requests`. Use PyGithub library: `pip install PyGithub`, then `from github import Github; g = Github("ghp_your_token")`. Store tokens securely in Django `settings.py` via environment variables.

#### **Repository Operations**

List repos: `user = g.get_user(); repos = user.get_repos()` iterates through repositories. Access specific repo: `repo = g.get_repo("owner/repo_name")`. Create repo: `user.create_repo("new-repo")`. Clone programmatically or fetch contents: `contents = repo.get_contents("path/to/file.py")`.

#### **Pull Requests and Issues**

Create PR: `pr = repo.create_pull(title="Fix bug", body="Details", head="branch", base="main")`. List open PRs: `prs = repo.get_pulls(state='open')`. For issues: `issues = repo.get_issues(state='open')`; create new: `repo.create_issue(title="Bug report", body="Steps to reproduce")`. Comment: `issue = repo.get_issue(1); issue.create_comment("Thanks for reporting!")`.

#### **Django Integration**

Build DRF endpoints: `@api_view(['GET']) def github_repos(request): g = Github(settings.GITHUB_TOKEN); repos = g.get_user().get_repos(); return Response([{'name': r.name, 'url': r.html_url} for r in repos])`. Use for authenticated user dashboards tracking your Django project PRs and issues.

## 17. Twitter API Integration

### **Q1: Using Twitter API to fetch and post tweets, and retrieve user data.**

**Ans:** Twitter API (now X API) enables fetching tweets, posting content, and retrieving user data through REST endpoints, requiring OAuth authentication for most operations. Python libraries like Tweepy simplify integration into Django apps for social features.

#### **API Access Setup**

Create a developer account at developer.x.com, generate an app, and obtain API Key, Secret, Access Token, and Secret (OAuth 1.0a) or Client ID/Secret (OAuth 2.0). For v2 API (recommended), use OAuth 2.0 User Context with PKCE flow for user actions like posting. Store credentials in Django `.env: TWITTER_API_KEY, TWITTER_API_SECRET, etc.`

#### **Fetching Tweets and Users**

Install `pip install tweepy`, authenticate: `auth = tweepy.OAuth1UserHandler(api_key, api_secret, access_token, access_token_secret); api = tweepy.Client(consumer_key=api_key, ...)`. Fetch recent tweets: `tweets = api.get_users_tweets(id=user_id, max_results=10)` returns JSON with text, author\_id, created\_at. Get user: `user = api.get_user(username='username')` for profile data.

#### **Posting Tweets**

Post via `api.create_tweet(text="Hello from Django! #Python")`, handling media with `media_ids=[api.media_upload(filename).media_id]`. Rate limits: 1500 tweets/15min (v2 POST /tweets). For OAuth 2.0: generate authorization URL, handle callback to exchange code for access\_token, then `bearer_token` for read-only or user token for write.

#### **Django DRF Integration**

```
Create authenticated endpoint: @api_view(['GET'])
@permission_classes([IsAuthenticated])
def user_tweets(request):
    tweets = client.get_users_tweets(id=request.user.twitter_id)
    return
```

```
Response(tweets.data). Cache responses and respect rate limits with  
cache.set('user_tweets', data, 900) for your project dashboards.
```

## 18. REST Countries API Integration

### **Q1: Introduction to REST Countries API and how to retrieve country-specific data.**

**Ans:** REST Countries API offers free access to comprehensive country data including names, capitals, populations, currencies, languages, flags, and borders via simple REST endpoints. No authentication required, it's perfect for Django apps needing location context alongside your weather/geocoding integrations.

#### **Core Endpoints**

Access all countries: `GET https://restcountries.com/v3.1/all?fields=name,capital,population,region,flags,currencies` returns JSON array with filtered fields. Get specific country: `/v3.1/name/india` (fuzzy matching) or `/v3.1/alpha/in` (ISO code). Filter by region: `/v3.1/region/asia`, language: `/v3.1/lang/hindi`, or currency: `/v3.1/currency/inr`.

#### **Python/Django Usage**

Use

```
requests.get('https://restcountries.com/v3.1/all?fields=name,capital,cca2')  
in views: countries = response.json(); india_data = next(c for c in countries  
if c['cca2'] == 'IN') extracts {'name': {'common': 'India'}, 'capital': ['New  
Delhi']}. Cache results: cache.set('all_countries', data, 86400) for 24hr validity.
```

#### **Integration Examples**

Combine with Google Geocoding: geocode address → match to REST Countries currency/timezone → fetch OpenWeatherMap data. DRF endpoint:

```
@api_view(['GET']) def country_info(request, code): response =  
requests.get(f'https://restcountries.com/v3.1/alpha/{code.upper()}'); return  
Response(response.json()). Use cca3 codes for uniqueness in your Django models.
```

## 19. Email Sending API

### **Q1: Using email sending APIs like SendGrid and Mailchimp to send transactional emails.**

**Ans:** SendGrid and Mailchimp excel at transactional emails—individual, triggered messages like order confirmations or password resets—unlike marketing bulk sends. Both integrate cleanly with Django via API keys, supporting templates and analytics for your backend projects.

#### **SendGrid Configuration**

```
Install pip install sendgrid, configure settings.py:  
EMAIL_HOST='smtp.sendgrid.net', EMAIL_HOST_USER='apikey',  
EMAIL_HOST_PASSWORD=os.environ.get('SENDGRID_API_KEY'), EMAIL_PORT=587,  
EMAIL_USE_TLS=True. Send via Django: send_mail('Welcome!', 'Your OTP: 123456',  
settings.DEFAULT_FROM_EMAIL, ['user@example.com']) or use SendGrid client for  
dynamic templates: Mail(from_email='noreply@yourapp.com',  
to_emails='user@example.com', html_content='<h1>Verify Email</h1>').
```

#### **Mailchimp Transactional (Mandrill)**

```
Access via Mandrill (Mailchimp's transactional arm), generate API key. Install pip  
install mailchimp-transactional, create client: client =  
MailchimpTransactional.ApiClient(); response = client.messages.send({'key':  
api_key, 'message': {'html': template, 'to': [{'email': user_email}]}}).  
Django views: @api_view(['POST']) def send_welcome(request):  
MailchimpTransactional.send_template(user.email, 'welcome-template-id',  
{'name': user.name}).
```

#### **Django Best Practices**

```
Use Celery tasks: @shared_task def send_transactional_email(email,  
template_id, context): ... for async delivery. Track bounces/opens via webhook  
endpoints (path('sendgrid-webhook/', webhook_view)). Templates via  
render_to_string('emails/welcome.html', {'user': user}) for HTML emails with  
your Django models.
```

## **20.SMS Sending APIs (Twilio)**

### **Q1: Introduction to Twilio API for sending SMS and OTPs.**

**Ans:** Twilio API provides reliable SMS and voice services for sending OTPs, notifications, and 2FA in Django applications. It uses simple HTTP requests with Account SID and Auth Token authentication, perfect for your backend projects with phone verification flows.

#### **Account Setup**

Sign up at [twilio.com](https://www.twilio.com), verify your phone, and get Account SID/Auth Token from the dashboard. Purchase a phone number (India +91 supported) via Console > Phone Numbers > Buy a number. Store credentials in `.env: TWILIO_ACCOUNT_SID, TWILIO_AUTH_TOKEN, TWILIO_PHONE_NUMBER`.

#### **Sending SMS/OTP**

Install `pip install twilio`, then:

```
python

from twilio.rest import Client
client = Client(account_sid, auth_token)
message = client.messages.create(
    body='Your OTP is 123456',
    from_='+12345678901', # Your Twilio number
    to='+919876543210' # User phone
)
```

Returns `message.sid` for tracking delivery status.

#### **OTP Implementation in Django**

Generate OTP: `otp = str(random.randint(100000, 999999))`, cache it

`cache.set(f'otp_{phone}', otp, 300)`, send via Twilio. Verify endpoint:

```
python
```

```
@api_view(['POST'])
def verify_otp(request):
    phone = request.data['phone']
    user_otp = request.data['otp']
    stored_otp = cache.get(f'otp_{phone}')
```

```
if user_otp == stored_otp:  
    login(request, user) # Django auth  
    return Response({'success': True})
```

## **21. Payment Integration (PayPal, Stripe)**

### **Q1: Introduction to integrating payment gateways like PayPal and Stripe.**

**Ans:** Payment gateways like PayPal and Stripe enable secure online transactions in Django applications through API keys and webhooks. Both support one-time payments, subscriptions, and refunds, integrating seamlessly with DRF for authenticated checkouts in your projects.

#### **PayPal Integration**

Create a PayPal developer account, generate sandbox Client ID/Secret from developer.paypal.com. Install `pip install python-paypal-rest-sdk`, configure in `settings.py`: `PAYPAL_CLIENT_ID`, `PAYPAL_SECRET`. Create payment in views:

```
python
```

```
payment = paypalrestsdk.Payment({  
    "intent": "sale",  
    "payer": {"payment_method": "paypal"},  
    "transactions": [{"amount": {"total": "99.00", "currency": "USD"}}]  
})  
payment.create()  
return redirect(payment.links[1].href) # Approval URL
```

Handle callback in `execute_payment` view to capture funds.

#### **Stripe Integration**

Install `pip install stripe`, set `stripe.api_key = os.environ.get('STRIPE_SECRET_KEY')`. Create Checkout Session:

```
python
```

```
session = stripe.checkout.Session.create()
```

```

payment_method_types=['card'],
line_items=[{
    'price_data': {'currency': 'inr', 'product_data': {'name': 'Service'}, 'unit_amount': 9900},
    'quantity': 1,
}],
mode='payment',
success_url='yourdomain.com/success',
cancel_url='yourdomain.com/cancel'
)
return redirect(session.url, code=303)

```

Use Stripe Elements for custom forms or Checkout for hosted UI.

## Django Best Practices

Create Order model tracking payment status, use `@csrf_exempt` webhook endpoints (`path('stripe-webhook/', stripe_webhook)`), verify signatures: `stripe.Webhook.construct_event(payload, sig_header, webhook_secret)`. Use Celery for async invoice emails post-payment. Test with Stripe CLI tunneling and PayPal sandbox.

## 22.Google Maps API Integration

### Q1:Using Google Maps API to display maps and calculate distances between locations.

**Ans:**Google Maps API enables interactive maps, geocoding, and distance calculations in Django projects through JavaScript frontend integration and Python backend API calls. It builds on your existing Geocoding API setup, adding Maps JavaScript API for rendering and Distance Matrix API for route optimization.

#### API Setup

Enable Maps JavaScript API, Places API, and Distance Matrix API in Google Cloud Console alongside your existing Geocoding API. Add to `settings.py`:  
`GOOGLE_MAPS_API_KEY`. Frontend templates load: `<script async`

```
src="https://maps.googleapis.com/maps/api/js?key={ { GOOGLE_MAPS_API_KEY } }&callback=initMap"></script>
```

## Displaying Maps

Create Django view passing API key:

```
python
```

```
def map_view(request):  
    context = {'google_api_key': settings.GOOGLE_MAPS_API_KEY}  
    return render(request, 'maps/map.html', context)
```

Template with basic map:

```
xml
```

```
<div id="map" style="height: 500px;"></div>  
<script>  
function initMap() {  
  const map = new google.maps.Map(document.getElementById("map"), {  
    zoom: 8, center: {lat: 23.0225, lng: 72.5714} // Ahmedabad  
  }); }  
</script>
```

## Distance Calculations

Backend Distance Matrix API call:

```
python
```

```
import requests  
def calculate_distance(origin, destination):  
    url = f"https://maps.googleapis.com/maps/api/distancematrix/json"  
    params = {  
        'origins': origin,  
        'destinations': destination,  
        'key': settings.GOOGLE_MAPS_API_KEY
```

```
}

response = requests.get(url, params=params).json()
distance = response['rows'][0]['elements'][0]['distance']['text']
duration = response['rows'][0]['elements'][0]['duration']['text']
return distance, duration
```

**DRF endpoint:** @api\_view(['GET']) def distance\_calc(request): distance, duration = calculate\_distance(request.GET['from'], request.GET['to']); return Response({'distance': distance, 'duration': duration}).