Assignment 5

Introduction to APIs

What is an API (Application Programming Interface)?:-

An **API (Application Programming Interface)** is a defined set of protocols, routines, and tools that enables different software applications to communicate with each other. It specifies how software components should interact, allowing developers to access specific features or data of an application, service, or operating system without needing to understand its internal implementation.

APIs are commonly used to integrate third-party services, enable modular software development, and facilitate interoperability between systems.

Types of APIs: REST, SOAP.:-

1. REST (Representational State Transfer)

- **Architecture style**, not a protocol.
- Uses standard HTTP methods: GET, POST, PUT, DELETE.
- **Data format**: Typically JSON or XML (JSON is more common).
- **Stateless**: Each request from the client to the server must contain all necessary information.
- **Lightweight and scalable** — ideal for web and mobile applications.

Example:

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

2. SOAP (Simple Object Access Protocol)

- **Protocol** — with strict standards.
- Uses **XML** exclusively for messaging.
- **More rigid** structure compared to REST.
- Includes **built-in error handling** and **security (WS-Security)** features.
- Common in enterprise-level applications (e.g., banking, telecom).

Example (SOAP Envelope):

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

```
  <soap:Body>
    <GetUserDetails xmlns="http://example.com/">
      <UserId>123</UserId>
    </GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

Why are APIs important in web development?:-

APIs are essential in web development because they enable seamless integration, modularity, and efficient communication between different systems, services, and applications. Here's why they are important:

1. Integration with Third-Party Services

APIs allow web applications to connect with external services such as:

- Payment gateways (e.g., Stripe, PayPal)
- Social logins (e.g., Google, Facebook)
- Maps and geolocation (e.g., Google Maps)
- Email and SMS services (e.g., SendGrid, Twilio)

2. Separation of Frontend and Backend

APIs decouple the frontend from the backend. This enables:

- Independent development and deployment
- Use of different technologies (e.g., React frontend with Django REST backend)
- Scalability and easier maintenance

3. Data Access and Automation

APIs provide structured, secure access to data and functionality, allowing:

- Dynamic content loading
- Real-time updates via APIs (e.g., live scores, stock prices)
- Automation of repetitive tasks

4. Reusability and Modularity

APIs promote code reuse. A well-designed API can be:

- Used across multiple platforms (web, mobile, desktop)
- Shared with partners or external developers
- Scaled or updated without affecting the entire system

5. Enhanced User Experience

By enabling faster, more interactive user interfaces through asynchronous API calls (e.g., using AJAX or fetch), APIs improve responsiveness and reduce page reloads.

2. Requirements for Web Development Projects

Understanding project requirements.:-

Understanding project requirements is a critical first step in successful software development. It involves gathering, analysing, and documenting the needs and expectations of stakeholders to ensure the final product meets its intended purpose.

Key Aspects:

1. Requirement Gathering

- Engage with stakeholders (clients, users, managers).
- Use methods like interviews, surveys, or brainstorming.
- Understand the problem the project aims to solve.

2. Functional Requirements

- Define what the system should do.
- Example: "User can register and log in."

3. Non-Functional Requirements

- Define how the system performs.
- Includes scalability, performance, security, and usability.
- Example: "System must respond within 2 seconds."

4. Business Requirements

- Reflect the goals and needs of the business.
- Example: "Allow users to purchase subscriptions online."

5. Technical Requirements

- Define the technical environment: platforms, APIs, databases, integrations, etc.
- Example: "Use Django REST API with PostgreSQL backend."

6. Documenting Requirements

- Create clear and concise documentation (e.g., Software Requirement Specification - SRS).
- Helps avoid misunderstandings and scope creep.

Setting Up the Environment and Installing Necessary Packages – Key Points

- **Select Python Version:**
  - Use version managers like pyenv to install and manage the correct Python version.
- **Create a Virtual Environment:**
  - Run python -m venv env to create an isolated environment for dependencies.
  - Activate it using source env/bin/activate (Linux/macOS) or env\Scripts\activate (Windows).
- **Install Required Packages:**
  - Use pip install -r requirements.txt to install dependencies listed in the file.
  - Alternatively, use poetry install if using Poetry.
- **Install Development Tools:**
  - Include tools like black (code formatter), flake8 (linter), and python-dotenv (for managing environment variables).
- **Use a Dependency File:**
  - Keep a requirements.txt or pyproject.toml to track and reproduce dependencies.
- **Document Setup Instructions:**
  - Add setup steps to a README.md or script (setup.sh) for easy onboarding and consistency.

3. Serialization in Django REST Framework

What is Serialization?

Serialization is the process of converting complex data structures—like objects, querysets, or models—into a format that can be easily stored, transmitted, or rendered. In web development, this typically means converting data into **JSON** or **XML** so it can be sent over a network or used in an API response.

Why Serialization Is Important:

- **APIs**: In RESTful APIs, serialization is used to convert backend data (like Django models) into JSON for the frontend.
- **Storage**: Serialized data can be stored in files or databases.
- **Communication**: Enables systems written in different languages to exchange data.

In Django REST Framework (DRF):

- DRF provides powerful tools to serialize and deserialize data.
- A **serializer** converts model instances to JSON, and also validates and converts JSON back into model instances.

Example:

```
from rest_framework import serializers

from .models import Product


class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ['id', 'name', 'price']
```

Converting Django QuerySets to JSON.:-

In Django, a **QuerySet** is a collection of database records. To expose this data to the frontend or through an API, it often needs to be converted into **JSON** format. There are multiple ways to do this depending on the context.

1. Using Django's Built-in serializers Module

This is useful for simple use cases.

```
from django.core import serializers

from .models import Product


queryset = Product.objects.all()

json_data = serializers.serialize('json', queryset)
```

- Returns a JSON string representing the queryset.
- Includes metadata like model name and primary key.

2. Using Django REST Framework (DRF) Serializers

This is the recommended approach for building APIs.

Serializer:

```
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'price']
```

View:

```
from rest_framework.response import Response
from .models import Product
from .serializers import ProductSerializer

def product_list(request):
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

- Converts queryset into JSON-ready Python data structures (dicts/lists).
- Automatically handles validation, formatting, and nested relationships.

3. Manually Using values() or values_list()

For lightweight or custom JSON responses.

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

def product_list(request):
    products = Product.objects.values('id', 'name', 'price')
    return JsonResponse(list(products), safe=False)
```

- Returns a list of dictionaries.
- safe=False allows returning a list instead of a dictionary.

Using serializers in Django REST Framework (DRF).:-

Serializers in Django REST Framework (DRF) are used to convert complex Django model instances or querysets into **native Python data types**, which can then be easily rendered into **JSON** or other content types. They also handle **deserialization**, validating incoming data and converting it back into model instances.

1. Create a Serializer

For a Django model like this:

```
from django.db import models


class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=8, decimal_places=2)
```

You create a serializer using DRF:

```
from rest_framework import serializers
from .models import Product


class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'price']
```

2. Use the Serializer in a View

You can use the serializer in a function-based or class-based view.

Function-Based View Example:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Product
from .serializers import ProductSerializer


@api_view(['GET'])
def product_list(request):
```

```
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

Class-Based View Example (using APIView):

```
from rest_framework.views import APIView

class ProductListView(APIView):
    def get(self, request):
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)
```

---

3. Validating and Saving Data (POST request)

POST View Example:

```
@api_view(['POST'])
def create_product(request):
    serializer = ProductSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=201)
    return Response(serializer.errors, status=400)
```

---

4. Key Features of DRF Serializers

- Automatically maps model fields to serializer fields.
- Validates input data.
- Easily extendable with custom validation methods.
- Supports nested serializers and relationships.

4. Requests and Responses in Django REST Framework:-

HTTP Request Methods: GET, POST, PUT, DELETE

HTTP request methods define the actions a client can perform on a server's resource. In web development and RESTful APIs, the four most commonly used methods are:

1. GET

- **Purpose:** Retrieve data from the server.
- **Safe & Idempotent:** Does not modify server state.
- **Use Case:** Fetching a list of products or details of a single item.

Example:

GET /api/products/1/

2. POST

- **Purpose:** Submit new data to the server to create a resource.
- **Not Idempotent:** Repeating a POST creates duplicate entries.
- **Use Case:** Creating a new user, submitting a form.

Example:

POST /api/products/

{

"name": "T-Shirt",

"price": "19.99"

}

3. PUT

- **Purpose:** Update an existing resource **completely**.
- **Idempotent:** Repeating the same request results in the same resource.
- **Use Case:** Replacing all fields of an existing product.

Example:

PUT /api/products/1/

{

"name": "Updated T-Shirt",

"price": "24.99"

}

4. DELETE

- **Purpose:** Remove a resource from the server.
- **Idempotent:** Repeating it has no additional effect after the first success.
- **Use Case:** Deleting a user account or product.

Example:

DELETE /api/products/1/

Sending and Receiving Responses in Django REST Framework (DRF):-

In Django REST Framework, **responses** are managed using the Response object, which is an enhanced version of Django's standard HttpResponse. It handles content negotiation, formatting (like JSON), and appropriate status codes.

1. Importing the Response Class

```
from rest_framework.response import Response
```

2. Sending Responses

✅ **Basic JSON Response:**

```
from rest_framework.decorators import api_view

@api_view(['GET'])
def example_view(request):
    data = {'message': 'Hello from DRF'}
    return Response(data)
```

- Automatically returns JSON.
- Content-Type is set to application/json.

✅ **With Serialized Data:**

```
from .models import Product
from .serializers import ProductSerializer
```

```
@api_view(['GET'])
def product_list(request):
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

- Converts queryset to JSON using the serializer.
- many=True is used for lists of objects.

✅ **Sending Status Codes:**

```
from rest_framework import status

@api_view(['POST'])
def create_product(request):
    serializer = ProductSerializer(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)
```

- Use status module for clarity and best practices.

3. Receiving Data in DRF

When a client sends data (typically via POST, PUT, or PATCH), DRF automatically parses the request body and makes it available in:

```
request.data
```

Example:

```
@api_view(['POST'])
def create_user(request):
    name = request.data.get('name')
    email = request.data.get('email')
    return Response({'status': 'User received', 'name': name})
```

Views in Django REST Framework

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

Function-Based Views (FBVs) vs. Class-Based Views (CBVs)

Django REST Framework supports both **Function-Based Views** and **Class-Based Views**, allowing developers to choose the approach that best fits their needs.

- **Function-Based Views (FBVs)**

FBVs use standard Python functions and are decorated with @api_view to define which HTTP methods they support.

Example:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Product
from .serializers import ProductSerializer


@api_view(['GET'])
def product_list(request):
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

- **Class-Based Views (CBVs)**

CBVs use Python classes to handle different HTTP methods by defining methods like get(), post(), etc. DRF provides APIView and a range of generic views for common patterns.

Example using APIView:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .models import Product
from .serializers import ProductSerializer


class ProductListView(APIView):
    def get(self, request):
        products = Product.objects.all()
```

```
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

6. URL Routing in Django REST Framework

Defining URLs and linking them to views:-

In Django and DRF, URLs define how HTTP requests are routed to specific view functions or classes. Each URL pattern is mapped to a corresponding view that handles the request and returns a response.

- **1. Define Views**

Example Function-Based View:

```
@api_view(['GET'])
def product_list(request):
    ...
```

Example Class-Based View:

```
class ProductListView(APIView):
    ...
```

- **2. Create URL Patterns**

In your app's urls.py file:

For Function-Based View:

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

urlpatterns = [
    path('products/', product_list, name='product-list'),
]
```

For Class-Based View (use .as_view()):

```
from .views import ProductListView
urlpatterns = [
    path('products/', ProductListView.as_view(), name='product-list'),
```

]

- **3. Include App URLs in Project URLConf**

In your project's main urls.py (typically at the root level):

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('your_app_name.urls')),  # link to app's urls
]
```

- **4. Using Routers (For ViewSets)**

If you're using ViewSet:

```
from rest_framework.routers import DefaultRouter
from .views import ProductViewSet

router = DefaultRouter()
router.register(r'products', ProductViewSet)
urlpatterns = router.urls
```

7. Pagination in Django REST Framework:-

Pagination is essential when dealing with large datasets, as it improves performance and usability by dividing the data into manageable chunks (pages) instead of returning everything at once.

- **1. Enable Pagination in settings.py**

Add or update the following in your **project's settings.py**:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10,  # Adjust the number of results per page
}
```

🔹 2. Use a Paginated View

No changes are needed in your views if you're using generic views, APIView, or ViewSet. Pagination is automatically applied when you return a queryset.

Example:

```
class ProductListAPIView(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

The API response will now look like:

```
{
    "count": 100,
    "next": "http://example.com/api/products/?page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "name": "Product A",
            "price": "19.99"
        },
        ...
    ]
}
```

🔹 3. Custom Pagination (Optional)

To define a custom pagination class:

```
from rest_framework.pagination import PageNumberPagination


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

Then use it in your view:

```
class ProductListAPIView(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    pagination_class = CustomPagination
```

8. Settings Configuration in Django

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

Proper configuration in settings.py ensures your Django app runs securely and smoothly across environments (development, staging, production).

📦 1. Database Configuration

In settings.py, configure the database settings:

```
import os
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',  # or 'django.db.backends.sqlite3'
        'NAME': os.getenv('DB_NAME', 'mydb'),
        'USER': os.getenv('DB_USER', 'myuser'),
        'PASSWORD': os.getenv('DB_PASSWORD', 'mypassword'),
        'HOST': os.getenv('DB_HOST', 'localhost'),
        'PORT': os.getenv('DB_PORT', '5432'),
    }
}
```

✅ **Tip**: Use environment variables to avoid hardcoding credentials.

🎨 2. Static Files Configuration

Static files are CSS, JS, and images used in your frontend.

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / 'static']        # Dev files
STATIC_ROOT = BASE_DIR / 'staticfiles'          # Production output (used with collectstatic)
```

- Run python manage.py collectstatic before deployment.

- For production, serve using WhiteNoise, Nginx, or a CDN.

🖼 3. Media Files Configuration

Media files are user-uploaded content like images, PDFs, etc.

MEDIA_URL = '/media/'

MEDIA_ROOT = BASE_DIR / 'media'

- Use this in urls.py (only in development):

```
from django.conf import settings
from django.conf.urls.static import static

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

🔐 4. API Keys Configuration

To store API keys (e.g., Stripe, SendGrid, etc.), use a .env file or environment variables.

Example:

```
import os
from decouple import config  # if using python-decouple

STRIPE_SECRET_KEY = config('STRIPE_SECRET_KEY')  # or os.getenv('STRIPE_SECRET_KEY')
```

Create a .env file (not committed to Git):

```
STRIPE_SECRET_KEY=your_stripe_key_here
```

✅ **Tip**: Use libraries like python-decouple or django-environ for better env var management.

9. Project Setup

Setting up a Django REST Framework project:-

Here's a step-by-step guide to setting up a **Django project with Django REST Framework** to build APIs:

✅ 1. Create a Virtual Environment

```
python -m venv env
```

```
source env/bin/activate     # On Windows: env\Scripts\activate
```

✅ **2. Install Django and DRF**

```
pip install django djangorestframework
```

✅ **3. Start a New Django Project**

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

✅ **4. Create a Django App**

```
python manage.py startapp api
```

✅ **5. Add Apps to settings.py**

```
# myproject/settings.py

INSTALLED_APPS = [
    ...
    'rest_framework',
    'api',  # your app
]
```

✅ **6. Define a Model (Example)**

```
# api/models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.name
```

✅ 7. Create and Apply Migrations

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

✅ 8. Create a Serializer

```
# api/serializers.py
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
```

✅ 9. Create a View (Function-Based or Class-Based)

```
# api/views.py
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Product
from .serializers import ProductSerializer

@api_view(['GET'])
def product_list(request):
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

✅ 10. Configure URLs

In api/urls.py:

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

urlpatterns = [
    path('products/', product_list),
]
```

In myproject/urls.py:

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

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

✅ **11. Run the Server**

```
python manage.py runserver
```

Visit:
📍 http://127.0.0.1:8000/api/products/

10. Social Authentication, Email, and OTP Sending API

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

To integrate **Google or Facebook login** in Django, the best approach is using the **social-auth-app-django** package.

✅ **1. Install Required Packages**

bash

CopyEdit

```
pip install social-auth-app-django
```

✅ **2. Add to INSTALLED_APPS in settings.py**

python

CopyEdit

```
INSTALLED_APPS = [
    ...
    'social_django',
]
```

Also add middleware:

python

CopyEdit

```
MIDDLEWARE = [
    ...
    'social_django.middleware.SocialAuthExceptionMiddleware',
]
```

✅ 3. Add Social Auth Settings in settings.py

```
AUTHENTICATION_BACKENDS = (
    'social_core.backends.google.GoogleOAuth2',     # for Google
    'social_core.backends.facebook.FacebookOAuth2', # for Facebook
    'django.contrib.auth.backends.ModelBackend',
)
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

✅ 4. Add Keys and Secrets

Use your .env file or settings.py for now (but never commit secrets to Git):

- **For Google:**

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

🔹 **For Facebook:**

SOCIAL_AUTH_FACEBOOK_KEY = 'your-facebook-app-id'

SOCIAL_AUTH_FACEBOOK_SECRET = 'your-facebook-app-secret'

✅ **5. Update urls.py**

In project/urls.py:

```
from django.urls import path, include


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

✅ **6. Add Login Buttons in Template**

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

✅ **7. Create OAuth App on Google / Facebook**

- **Google:** https://console.cloud.google.com/
- **Facebook:** https://developers.facebook.com/

Set **redirect URI** to:
http://localhost:8000/auth/complete/google-oauth2/
http://localhost:8000/auth/complete/facebook/

✅ **8. Migrate and Run**

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

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

To send **emails** (e.g., OTPs or notifications) and **SMS OTPs** in Django, we commonly use:

- **SendGrid** – for sending email
- **Twilio** – for sending SMS

✅ 1. Sending Email with SendGrid

🔹 Step 1: Install SendGrid Python SDK

```
pip install sendgrid
```

🔹 Step 2: Get SendGrid API Key

- Go to https://app.sendgrid.com/
- Create an account → API Keys → Create API Key

🔹 Step 3: Send Email from Django

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

def send_otp_email(to_email, otp):
    message = Mail(
        from_email='your_verified_sender@example.com',
        to_emails=to_email,
        subject='Your OTP Code',
        html_content=f'<strong>Your OTP is: {otp}</strong>'
    )
    try:
        sg = SendGridAPIClient(os.getenv('SENDGRID_API_KEY'))
        response = sg.send(message)
        print(response.status_code)
```

```
    except Exception as e:
        print(e)
```

Environment variable setup (.env):

```
SENDGRID_API_KEY=your_api_key_here
```

✅ 2. Sending OTP via SMS with Twilio

🔹 Step 1: Install Twilio SDK

```
pip install twilio
```

🔹 Step 2: Get Twilio Credentials

- Sign up at https://www.twilio.com/
- Get your:
  - **Account SID**
  - **Auth Token**
  - **Twilio phone number**

🔹 Step 3: Send OTP via SMS

```
from twilio.rest import Client
import os

def send_otp_sms(to_number, otp):
    account_sid = os.getenv('TWILIO_ACCOUNT_SID')
    auth_token = os.getenv('TWILIO_AUTH_TOKEN')
    from_number = os.getenv('TWILIO_PHONE')

    client = Client(account_sid, auth_token)

    message = client.messages.create(
        body=f'Your OTP is: {otp}',
```

```
    from_=from_number,
    to=to_number
  )

  print(message.sid)
```

.env Example:

```
TWILIO_ACCOUNT_SID=your_sid_here
TWILIO_AUTH_TOKEN=your_token_here
TWILIO_PHONE=+1234567890
```

✅ **3. Generating OTP**

```
import random

def generate_otp(length=6):
  return ''.join([str(random.randint(0, 9)) for _ in range(length)])
```

11. RESTful API Design

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

1. Statelessness

Each request from the client to the server must contain all the information needed to understand and process it. The server doesn't store any context between requests. So, if you're sending multiple requests, you need to include things like authentication (e.g., a token) every time.

2. Resource-Based URLs

REST treats everything as a resource — like users, posts, files — and each resource has its own URL. You describe **what** you're working with, not **what** you're trying to do.

Examples:

- GET /books → fetch all books
- GET /books/123 → fetch book with ID 123
- POST /books → create a new book
- DELETE /books/123 → delete book 123

3. Using HTTP Methods for CRUD

REST uses standard HTTP methods to perform operations:

- GET → Read something (e.g., data)
- POST → Create something new
- PUT or PATCH → Update something
- DELETE → Remove something

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

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

What is CRUD?

CRUD stands for:

- **C**reate – Add new data
- **R**ead – Retrieve existing data
- **U**pdate – Modify existing data
- **D**elete – Remove data

These are the **four basic operations** you perform on a database — pretty much the foundation of how most backend systems work.

Why is CRUD Fundamental to Backend Development?

Because **every backend system is about managing data** — and CRUD is the simplest, most effective model for that.

Let's say you're building a blog platform:

- Create a new post
- Read posts to display to users
- Update posts when users edit them
- Delete posts if users remove them

That's CRUD in action — whether you're using SQL, NoSQL, Django, Flask, Node.js, Rails, etc.

CRUD in Practice

In RESTful APIs:

- POST → Create
- GET → Read
- PUT / PATCH → Update
- DELETE → Delete

13. Authentication and Authorization API

Difference between authentication and authorization:-

Authentication

"Who are you?"

It's the process of **verifying identity**.

When a user logs in with a username/email and password (or OAuth, biometrics, etc.), that's authentication. The system checks: *"Is this person really who they say they are?"*

Example:
Logging into your email with your password — the system verifies that *you* are the owner of the account.

Authorization

"What are you allowed to do?"

This happens **after authentication** and determines what **access or permissions** a user has.

Example:
After logging in (authenticated), can you:

- View admin pages?
- Edit someone else's post?
- Download certain files?

That's authorization — defining *what actions you're allowed to perform* based on your role or privileges.

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

1. Install DRF (if not already)

pip install djangorestframework

Add it to INSTALLED_APPS in your settings.py:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework.authtoken',  # token auth support
]
```

2. Migrate to create the token model

bash

CopyEdit

python manage.py migrate

3. Add Token Authentication in Settings

In settings.py:

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

4. Create Token for Users (automatically)

You can generate a token when a user is created using Django signals or the built-in view.

Add this to a file like signals.py inside your app:

```
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
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)
```

Then connect signals in apps.py:

```
def ready(self):
    import yourapp.signals
```

5. Enable Token Authentication View

In your urls.py:

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

urlpatterns = [
    path('api/token/', obtain_auth_token),  # Login to get token
]
```

6. How It Works

- **User logs in via** POST /api/token/ with username and password.
- DRF returns a token like:

```
{"token": "d3c15d2a3a9f56a1e78dc0b84c08d9b2748b8ec1"}
```

- Use this token in Authorization header in all subsequent API calls:

```
Authorization: Token d3c15d2a3a9f56a1e78dc0b84c08d9b2748b8ec1
```

7. Protecting Views

Use DRF's @authentication_classes and @permission_classes, or set globally in settings.py.

Example for a class-based view:

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

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:-

Introduction to OpenWeatherMap API

OpenWeatherMap is a popular weather data service that provides:

- **Current weather**
- **5-day / 16-day forecasts**
- **Historical weather**
- **Air pollution, UV index**, and more

It's widely used in apps, websites, and dashboards that need real-time weather information.

🔑 **Step 1: Get an API Key**

1. Go to https://openweathermap.org/api
2. Sign up (free).
3. Go to **"My API Keys"** in your account.
4. Copy your **API key** (you'll use it in every request).

🔎 **Step 2: Choose an Endpoint**

For **current weather**, the base URL is:

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

You can query by:

- **City name**
- **City ID**
- **Geographic coordinates**
- **ZIP code**

🧪 **Step 3: Make a Request (Example in Python)**

Here's a simple example using the requests library to get the weather in **London**:

```
import requests

API_KEY = 'your_api_key_here'

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"Weather: {data['weather'][0]['description']}")
```

🌍 **Optional: Use Coordinates**

```
lat = 28.6139   # Example: New Delhi latitude

lon = 77.2090   # Longitude

url = f'https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API_KEY}&units=metric'
```

Sample JSON Response (Trimmed)


```
{
  "name": "London",
  "main": {
    "temp": 18.34,
    "humidity": 77
  },
  "weather": [
    {
      "description": "broken clouds"
    }
```

```
  ]
}
```

Use in Django

You can easily integrate this in a Django view:

```
from django.http import JsonResponse
import requests

def weather_view(request):
    city = request.GET.get('city', 'London')
    url = f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid=your_api_key&units=metric'
    response = requests.get(url)
    data = response.json()
    return JsonResponse({
        'city': data['name'],
        'temp': data['main']['temp'],
        'description': data['weather'][0]['description'],
    })
```

15. Google Maps Geocoding API

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

What is Google Maps Geocoding API?

The **Geocoding API** lets you convert:

- 📍 **Addresses** → into **Coordinates** (Latitude/Longitude)
- 🧭 **Coordinates** → back into **Addresses** (Reverse Geocoding)

It’s perfect for location-based services like maps, delivery apps, or weather apps.

Step 1: Get a Google API Key

1. Go to the Google Cloud Console.
2. Create a new project (or use an existing one).

3. Enable the **"Geocoding API"**.
4. Go to **APIs & Services > Credentials**, and generate an **API key**.

Tip: Restrict your API key by domain or IP to avoid misuse.

Step 2: Build the Geocoding Request URL

Basic request format:

https://maps.googleapis.com/maps/api/geocode/json?address=YOUR_ADDRESS&key=YOUR_API_KEY

Step 3: Python Code to Get Coordinates

```
import requests

def get_coordinates(address):
    api_key = 'YOUR_GOOGLE_API_KEY'
    base_url = 'https://maps.googleapis.com/maps/api/geocode/json'

    params = {'address': address, 'key': api_key}
    response = requests.get(base_url, params=params)
    data = response.json()

    if data['status'] == 'OK':
        location = data['results'][0]['geometry']['location']
        return location['lat'], location['lng']
    else:
        return None, None

# Example usage
lat, lng = get_coordinates('1600 Amphitheatre Parkway, Mountain View, CA')
print(f"Latitude: {lat}, Longitude: {lng}")
```

Sample JSON Response (Trimmed)

```
{
  "results": [
    {
      "geometry": {
        "location": {
          "lat": 37.4224764,
          "lng": -122.0842499
        }
      },
      ...
    }
  ],
  "status": "OK"
}
```


16. GitHub API Integration

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

The **GitHub REST API** lets you programmatically interact with GitHub — almost anything you can do in the GitHub web UI, you can do with the API:

- List/create/edit/delete repositories
- Manage pull requests and issues
- Collaborate, comment, review code
- Authenticate with OAuth or tokens

Docs: https://docs.github.com/en/rest

Step 1: Authentication

Create a **Personal Access Token (PAT)**:

1. Go to GitHub → Settings → Developer settings → Personal access tokens.
2. Click **"Generate new token"** (select scopes like repo, user, etc.).
3. Copy the token — you'll use it in headers:

```
Authorization: token YOUR_TOKEN
```

Step 2: Make Basic Requests

Example: Get Your Repositories

```
import requests

headers = {
   "Authorization": "token YOUR_GITHUB_TOKEN"
}
url = "https://api.github.com/user/repos"
response = requests.get(url, headers=headers)

for repo in response.json():
   print(repo["name"], repo["html_url"])
```

Interacting with Repos, Pull Requests, and Issues

1. Repositories

- **List all**: GET /user/repos
- **Create one**: POST /user/repos

```
requests.post("https://api.github.com/user/repos", headers=headers, json={
   "name": "new-repo",
   "private": False,
   "description": "Created via API"
})
```

2. Issues

- **List issues**: GET /repos/:owner/:repo/issues
- **Create issue**:

```
requests.post("https://api.github.com/repos/OWNER/REPO/issues", headers=headers, json={
   "title": "Bug in signup flow",
   "body": "Steps to reproduce..."
})
```

3. Pull Requests

- **List PRs**: GET /repos/:owner/:repo/pulls
- **Create PR**:

```
requests.post("https://api.github.com/repos/OWNER/REPO/pulls", headers=headers, json={
  "title": "Fix typo in docs",
  "head": "fix-branch",
  "base": "main",
  "body": "This PR fixes a minor typo."
})
```

17. Twitter API Integration

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

What is the Twitter API?

The **Twitter API v2** (now managed under **X Developer Platform**) lets developers access Twitter data programmatically. You can:

- Read user timelines (tweets)
- Post tweets
- Search tweets by keywords
- Get user profile data
- Analyze trends, followers, mentions, etc.

Docs: https://developer.x.com/en/docs

Step 1: Get Access

1. Go to https://developer.x.com and apply for a developer account.
2. Create a **project** and **app**.
3. Generate the **Bearer Token** (for read) and/or **OAuth 2.0 Access Token** (for read/write).

Step 2: Install tweepy (Python SDK)

```
pip install tweepy
```

Example 1: Fetch Recent Tweets by User

```
import tweepy

bearer_token = 'YOUR_BEARER_TOKEN'

client = tweepy.Client(bearer_token=bearer_token)

# Get recent tweets from a user by username
response = client.get_users_tweets(
    id=client.get_user(username="elonmusk").data.id,
    max_results=5
)

for tweet in response.data:
    print(tweet.text)
```

Example 2: Post a Tweet

For write access, use OAuth2 with access token/secret:

```
import tweepy

consumer_key = 'API_KEY'
consumer_secret = 'API_SECRET'
access_token = 'ACCESS_TOKEN'
access_token_secret = 'ACCESS_SECRET'

auth = tweepy.OAuth1UserHandler(consumer_key, consumer_secret, access_token, access_token_secret)
api = tweepy.API(auth)

api.update_status("Hello from the Twitter API!")
```

Example 3: Get User Profile Info

```
user = client.get_user(username="nasa", user_fields=["created_at", "description", "public_metrics"])


print("Username:", user.data.username)

print("Followers:", user.data.public_metrics["followers_count"])

print("Bio:", user.data.description)
```

18. REST Countries API Integration

Introduction to REST Countries API and how to retrieve country-specific data:-

The **REST Countries API** is a free, public API that provides detailed information about countries worldwide — perfect for projects involving geography, travel, localization, or global data visualization.

Base URL:

https://restcountries.com/

Docs:
https://restcountries.com/#api-endpoints-v3

What Data Can You Get?

- Country name, capital, region
- Languages, currencies
- Population, area
- Flags, maps
- Country codes (ISO 3166)
- Borders, timezones, etc.

Example 1: Get All Countries

```
import requests


url = "https://restcountries.com/v3.1/all"

response = requests.get(url)
```

```
countries = response.json()

for c in countries[:5]:  # Print first 5 countries
    print(c['name']['common'], '-', c['capital'][0])
```

Example 2: Get Country by Name

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

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

print("Country:", data['name']['common'])
print("Capital:", data['capital'][0])
print("Population:", data['population'])
print("Currency:", list(data['currencies'].keys())[0])
```

Other Query Examples

- By **full name**:

https://restcountries.com/v3.1/name/india?fullText=true

- By **country code (ISO Alpha-2/3)**:

https://restcountries.com/v3.1/alpha/IN

Example Output (JSON snippet)

```
{
  "name": { "common": "India" },
  "capital": ["New Delhi"],
  "region": "Asia",
  "population": 1380004385,
```

```
  "currencies": {
    "INR": { "name": "Indian rupee", "symbol": "₹" }
  }
}
```

19. Email Sending APIs (SendGrid, Mailchimp)

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

What Are Transactional Emails?

Transactional emails are automated, user-specific messages triggered by actions — such as:

- Password reset links
- Order confirmations
- Email verification
- Welcome emails

Unlike marketing emails, **they must be timely and reliable**.

Top Email API Providers

1. SendGrid

- Owned by Twilio
- Great for transactional and marketing emails
- Offers a **powerful REST API** and SMTP support

2. Mailchimp (Transactional API)

- Formerly known as Mandrill
- Focused on high-volume transactional email
- Better suited for transactional than general campaigns

Step-by-Step: Sending Email via SendGrid API in Python

Step 1: Setup

- Sign up at https://sendgrid.com
- Go to **Settings > API Keys > Create API Key**
- Choose "Full Access" or "Restricted Access" for mail sending

Install SendGrid Python SDK:

```
pip install sendgrid
```

Step 2: Code to Send Email

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

def send_email(to_email, subject, content):
    message = Mail(
        from_email='you@example.com',
        to_emails=to_email,
        subject=subject,
        html_content=content
    )
    try:
        sg = SendGridAPIClient('YOUR_SENDGRID_API_KEY')
        response = sg.send(message)
        print(response.status_code)
    except Exception as e:
        print(e)
```

Using Mailchimp Transactional (Mandrill)

Step 1: Setup

- Go to https://mailchimp.com/developer/transactional/
- Sign up → Enable **Transactional Email (Mandrill)**
- Generate an **API key**

Install the Python SDK:

```
pip install mailchimp-transactional
```

Step 2: Code to Send Email

```
import mailchimp_transactional as MailchimpTransactional
from mailchimp_transactional.api_client import ApiClientError
```

```
def send_mailchimp_email(to_email, subject, content):
    try:
        client = MailchimpTransactional.Client(api_key='YOUR_MANDRILL_API_KEY')
        message = {
            "from_email": "you@example.com",
            "subject": subject,
            "html": content,
            "to": [{"email": to_email, "type": "to"}]
        }
        response = client.messages.send({"message": message})
        print(response)
    except ApiClientError as error:
        print("An exception occurred: {}".format(error.text))
```

20. SMS Sending APIs (Twilio)

Introduction to Twilio API for sending SMS and OTPs.

Step-by-Step: Sending SMS or OTP using Twilio API in Python

✅ **Step 1: Set Up Twilio Account**

1. Go to https://twilio.com and sign up.
2. Verify your phone number (for testing).
3. Go to **Console → Account Info** to find your:
    - **Account SID**
    - **Auth Token**
4. Get a **Twilio phone number** with SMS capability.

✅ **Step 2: Install Twilio SDK**

```
pip install twilio
```

✅ **Step 3: Send a Basic SMS**

```
from twilio.rest import Client

# Your Twilio credentials
account_sid = 'YOUR_ACCOUNT_SID'
auth_token = 'YOUR_AUTH_TOKEN'
twilio_number = '+1234567890'  # Your Twilio number

client = Client(account_sid, auth_token)

message = client.messages.create(
    body="Hello! This is a test SMS from Twilio.",
    from_=twilio_number,
    to='+919999999999'  # Replace with recipient's number
)

print("Message SID:", message.sid)
```

🔐 Sending OTP (One-Time Password)

You can either generate your own OTP or use Twilio's **Verify API**, which is designed for secure OTP flows.

✅ Option 1: Custom OTP via SMS

```
import random

otp = random.randint(100000, 999999)

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

✅ Option 2: Use Twilio Verify API (Recommended for production)

1. Go to Twilio Console > Verify
2. Create a **Verify Service** and get its **Service SID**

Step A: Send OTP using Verify

```
verify_sid = 'YOUR_VERIFY_SERVICE_SID'


client.verify.services(verify_sid).verifications.create(
   to='+919999999999',
   channel='sms'  # or 'call'
)
```

Step B: Check the OTP

```
client.verify.services(verify_sid).verification_checks.create(
   to='+919999999999',
   code='123456'  # user input
)
```

If the code is valid, the .status will be "approved".

21. Payment Integration (PayPal, Stripe)

Introduction to integrating payment gateways like PayPal and Stripe:-

What Is a Payment Gateway?

A **payment gateway** securely processes payment information between your app and financial institutions. It handles:

- User authentication (card, wallet, etc.)
- Payment authorization
- Transaction notifications

Why PayPal and Stripe?

Stripe

- Developer-first, modern API
- Supports cards, wallets (Apple Pay, Google Pay), bank transfers
- Excellent documentation and SDKs
- Ideal for SaaS, subscriptions, and marketplaces

PayPal

- Widely trusted by users globally
- Supports PayPal accounts, cards, bank transfers
- Great for consumer-facing apps and eCommerce
- Also offers **subscription and payout APIs**

Basic Integration Flow (Common to Both)

1. **Frontend** collects payment info using a secure widget (Stripe Elements or PayPal Checkout).
2. **Backend** creates a payment session or order.
3. User is redirected to pay → success/failure callback.
4. Backend verifies payment → updates your database.

Stripe: Accepting One-Time Card Payment (Python + JS)

1. Install Stripe SDK:

```
pip install stripe
```

2. Backend (Python)

```
import stripe
stripe.api_key = 'your_stripe_secret_key'


def create_checkout_session():
    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
            },
            'quantity': 1,
        }],
```

```
    mode='payment',
    success_url='https://yourapp.com/success',
    cancel_url='https://yourapp.com/cancel',
  )
  return session.url  # Redirect user here
```

PayPal: Accepting Payment via REST API

1. Setup

- Go to https://developer.paypal.com
- Create sandbox app → get **Client ID** and **Secret**

2. Backend: Generate Access Token

```
import requests
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('CLIENT_ID', 'SECRET')
res = requests.post(
  'https://api-m.sandbox.paypal.com/v1/oauth2/token',
  auth=auth,
  data={'grant_type': 'client_credentials'}
)
token = res.json()['access_token']
```

3. Create an Order

```
headers = {'Authorization': f'Bearer {token}'}
data = {
  "intent": "CAPTURE",
  "purchase_units": [{
    "amount": {
      "currency_code": "USD",
      "value": "20.00"
    }
  }],
```

```
    "application_context": {
        "return_url": "https://yourapp.com/success",
        "cancel_url": "https://yourapp.com/cancel"
    }
}
order = requests.post(
    "https://api-m.sandbox.paypal.com/v2/checkout/orders",
    json=data,
    headers=headers
)
approve_url = order.json()['links'][1]['href']
```

22. Google Maps API Integration

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

What Is the Google Maps API?

Google Maps offers a set of APIs to work with maps, geolocation, routes, distances, and places.

Core APIs you'll use:

- **Maps JavaScript API** – Display maps in your frontend
- **Distance Matrix API** – Calculate travel distances/duration
- **Geocoding API** – Convert addresses ↔ coordinates (optional)

Step 1: Get a Google Maps API Key

1. Go to https://console.cloud.google.com
2. Create a project
3. Enable:
    - Maps JavaScript API
    - Distance Matrix API
4. Go to **APIs & Services > Credentials** → Get your **API key**

Step 2: Display a Map with Markers (HTML + JS)

```
<!DOCTYPE html>
<html>
 <head>
  <title>Simple Map</title>
  <style>
   #map { height: 400px; width: 100%; }
  </style>
 </head>
 <body>
  <h2>My Google Map</h2>
  <div id="map"></div>

  <script>
   function initMap() {
    const loc1 = { lat: 28.6139, lng: 77.2090 }; // New Delhi
    const map = new google.maps.Map(document.getElementById("map"), {
     zoom: 7,
     center: loc1
    });

    new google.maps.Marker({ position: loc1, map: map, title: "New Delhi" });
   }
  </script>

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

Step 3: Calculate Distance Between Two Locations

Use the **Distance Matrix API** (from backend or JavaScript):

Example using Python (backend):

```
import requests


def get_distance(origin, destination):
    API_KEY = 'YOUR_API_KEY'
    url = 
f'https://maps.googleapis.com/maps/api/distancematrix/json?origins={origin}&destinations={des
tination}&key={API_KEY}'


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


    if data['status'] == 'OK':
        distance = data['rows'][0]['elements'][0]['distance']['text']
        duration = data['rows'][0]['elements'][0]['duration']['text']
        return distance, duration
    return None, None


# Example:
dist, time = get_distance("New Delhi", "Jaipur")
print(f"Distance: {dist}, Duration: {time}")
```

Sample API Response

```
{
  "rows": [
    {
      "elements": [
        {
          "distance": { "text": "268 km", "value": 267834 },
```

```
        "duration": { "text": "5 hours 10 mins", "value": 18623 },
        "status": "OK"
       }
     ]
    }
  ]
}
```