MODULE 10) REST FRAMEWORK

◇ Introduction to APIs

1. What is an API (Application Programming Interface)?

- An **API** is like a **bridge** that allows two different software applications to talk to each other.
- It defines a **set of rules and protocols** for how software components should interact.
- Example:
    - When you log in to a website using **Google or Facebook**, that site uses Google's/Facebook's API to authenticate you.
    - When you use a **weather app**, it fetches live data from a weather API.

Think of an API like a **waiter in a restaurant**:

- You (the user) tell the waiter (API) what you want.
- The waiter passes your request to the kitchen (server).
- The kitchen prepares the food (data) and the waiter brings it back to you.

---

2. Types of APIs: REST, SOAP

a) REST (Representational State Transfer)

- The most widely used API type in modern web development.
- Works over **HTTP/HTTPS** using methods like:
    - GET → Retrieve data
    - POST → Send data
    - PUT/PATCH → Update data
    - DELETE → Remove data
- Data is usually sent in **JSON** (lightweight, human-readable).
- **Example:**
- GET https://api.example.com/users/1

→ Fetches details of user with ID 1.

b) SOAP (Simple Object Access Protocol)

- An older protocol for APIs, often used in enterprise systems.
- Uses **XML** format for communication.
- More **secure and strict**, but also **heavier** compared to REST.
- **Example:**

```
<soap:Envelope>
  <soap:Body>
    <GetUserDetails>
      <UserID>1</UserID>
    </GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

---

3. Why are APIs Important in Web Development?

APIs are **essential** because they:

Connect Systems – Allow different applications (frontend, backend, mobile apps, third-party services) to communicate.
Save Time – Developers can use prebuilt APIs (like payment APIs – Razorpay, Stripe, PayPal) instead of writing everything from scratch.
Scalability – APIs allow apps to grow by integrating with new services.
Reusability – Once built, the same API can be used by web apps, mobile apps, or even other services.
Security – APIs can control what data is shared and protect sensitive information through authentication methods (OAuth, API keys, JWT).

Diagram – API Workflow

[Client (Browser / Mobile App)]

|

(API Request: GET /users)

|

[API (Waiter)]

|

[Server / Database]

|

(API Response: JSON Data)

|

[Client Displays Data on Screen]

◇ 2. Requirements for Web Development Projects

1. Understanding Project Requirements

Before starting any project, first we should **understand what the project needs**.

- **What should it do?** (features) → Example: An online shop should let users see products, add to cart, and buy.
- **How should it work?** (speed, design, security) → Example: The website should open fast and work on mobile also.
- **What do clients/users want?** → Example: Easy login with Google or Facebook.
- **Which tools will be used?** → Example: Python Django + REST Framework + MySQL..

---

2. Setting up the Environment and Installing Packages

Once we know the requirements, we must **prepare our computer** for development.

- **Install software** →
    - Code Editor (VS Code, PyCharm)
    - Programming Language (Python, JavaScript, PHP)
    - Database (MySQL, PostgreSQL)
    - Git (for saving code)
- **Install packages/libraries** (ready-made tools that make coding faster) →
  Example for Python Django REST:
- # Create virtual environment
- python -m venv env

- # Activate environment
- env\Scripts\activate # (Windows)
- source env/bin/activate # (Linux/Mac)

- # Install Django + REST Framework
- pip install django djangorestframework

◇ 3. Serialization in Django REST Framework

1. What is Serialization?

- **Serialization** means **converting data into a format that can be shared or stored easily**.
- In Django REST Framework, serialization is mostly used to convert **Python objects (like models, querysets)** into **JSON format** (which is used in APIs).
- Example:
    - Python object → {"id": 1, "name": "Jay", "email": "jay@example.com"} (JSON)
    - JSON is easy for browsers, mobile apps, and other systems to understand.

Think of it like **translating data into a common language** so everyone (frontend, backend, mobile apps) can understand.

---

2. Converting Django QuerySets to JSON

- In Django, when we fetch data from the database using ORM, we get a **QuerySet**.
- Example:
- users = User.objects.all()

Here users is a QuerySet (Python object).

- But APIs don't understand QuerySets directly → They need **JSON format**.
- **Serialization helps us convert QuerySets into JSON.**

Example conversion:

QuerySet: <QuerySet [<User: Jay>, <User: Ansh>]>

After Serialization (JSON):

```
[
  {"id": 1, "name": "Jay"},
  {"id": 2, "name": "Ansh"}
]
```

---

3. Using Serializers in Django REST Framework (DRF)

- DRF gives us a **Serializer class** to easily convert model data into JSON and back.
- Example:

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

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
# serializers.py
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = '__all__'
# views.py
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .models import Student
from .serializers import StudentSerializer
```

```
@api_view(['GET'])
def student_list(request):
    students = Student.objects.all()        # QuerySet
    serializer = StudentSerializer(students, many=True)  # Convert to JSON
    return Response(serializer.data)
```

- Now, when we call API endpoint /students/, it will return:
- [
- {"id": 1, "name": "Jay", "age": 22},
- {"id": 2, "name": "Ansh", "age": 25}
-]

◇ 4. Requests and Responses in Django REST Framework

1. HTTP Request Methods (GET, POST, PUT, DELETE)

In APIs, the client (browser, mobile app, or frontend) sends an **HTTP request** to the server. The server processes it and sends back a **response**.

The main HTTP methods are:

- **GET** → Used to **fetch data** from the server.
  Example: Get a list of all students → /students/
- **POST** → Used to **send new data** (create record).
  Example: Add a new student → { "name": "Jay", "age": 22 }
- **PUT** → Used to **update/replace existing data**.
  Example: Update student details → Change Jay's age from 22 to 23.
- **DELETE** → Used to **delete data**.
  Example: Remove student with ID 1.

These are called **CRUD operations**:

- Create → POST
- Read → GET

- Update → PUT/PATCH
- Delete → DELETE

---

2. Sending and Receiving Responses in DRF

In Django REST Framework, we use:

- **Request Object (request)** → to read incoming data from the client.
- **Response Object (Response)** → to send back data in JSON format.

Example: Handling GET and POST in DRF

```
# views.py
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .models import Student
from .serializers import StudentSerializer

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

    elif request.method == 'POST':  # Add new student
        serializer = StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=201)
        return Response(serializer.errors, status=400)
```

Example: Handling PUT and DELETE in DRF

```
@api_view(['GET', 'PUT', 'DELETE'])
def student_detail(request, pk):
    try:
        student = Student.objects.get(pk=pk)
    except Student.DoesNotExist:
        return Response({"error": "Student not found"}, status=404)

    if request.method == 'GET':   # Fetch one student
        serializer = StudentSerializer(student)
        return Response(serializer.data)

    elif request.method == 'PUT':  # Update student
        serializer = StudentSerializer(student, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=400)

    elif request.method == 'DELETE':  # Delete student
        student.delete()
        return Response({"message": "Student deleted"}, status=204)
```

◇ 5. Views in Django REST Framework

1. Understanding Views in DRF

In Django REST Framework, a **view** is the part of the code that **handles the request** from the client and **returns a response**.

- You can think of a **view as the brain of the API**:

- It receives the request.
- Talks to the database using models.
- Converts data into JSON using serializers.
- Sends the response back to the client.

---

2. Function-Based Views (FBVs)

- Views written as **simple Python functions**.
- Easy to understand for beginners.
- We use the @api_view decorator.

Example: Function-Based View

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

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

- Here:
- A request comes → View fetches students → Serializes them → Sends JSON response.

---

3. Class-Based Views (CBVs)

- Views written as **Python classes**.
- More powerful, reusable, and organized.
- DRF provides base classes like APIView, GenericAPIView, and ViewSets.

Example: Class-Based View

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


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

📌 Here:

- A **class** handles the request.
- Inside it, we define methods like get(), post(), put(), delete() for different HTTP requests.

Function-Based Views vs Class-Based Views

Feature	**Function-Based Views (FBVs)**	**Class-Based Views (CBVs)**
Code Style	Simple, short functions	Organized in classes
Best For	Small APIs, beginners	Large projects, reusable code
Flexibility	Less reusable	More reusable & scalable
Example	@api_view(['GET', 'POST'])	class MyView(APIView)

Great 👍 Let's cover **URL Routing in Django REST Framework (DRF)** in simple and clear words with examples:

◇ 6. URL Routing in Django REST Framework

1. Defining URLs and Linking Them to Views

a) Function-Based Views (FBVs) Example

```
# urls.py

from django.urls import path

from . import views


urlpatterns = [

   path('students/', views.student_list, name='student-list'),

]
```

Here:

- URL /students/ is linked to the function student_list in views.py.

---

b) Class-Based Views (CBVs) Example

```
# urls.py

from django.urls import path

from .views import StudentList


urlpatterns = [

   path('students/', StudentList.as_view(), name='student-list'),

]
```

Here:

- URL /students/ is linked to the class StudentList.
- Note: For CBVs we use .as_view() to make the class callable.

---

c) Using Routers with ViewSets (Shortcut in DRF)

- DRF provides **routers** which automatically create URL routes for **ViewSets** (like CRUD APIs).

```
# urls.py

from django.urls import path, include

from rest_framework.routers import DefaultRouter
```

```
from .views import StudentViewSet


router = DefaultRouter()

router.register('students', StudentViewSet)


urlpatterns = [

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

]
```

Here:

- The router automatically creates:
    - GET /students/ → list all students
    - POST /students/ → add a new student
    - GET /students/{id}/ → get one student
    - PUT /students/{id}/ → update student
    - DELETE /students/{id}/ → delete student

Pagination in Django REST Framework

1. **Adding pagination to APIs to handle large data sets**

- **Pagination** means breaking a **large dataset into smaller pages**.
- It helps when an API has too much data (like thousands of records).
- Instead of sending everything at once, the API sends data **page by page** (like 10 or 20 records per page).
- This makes the API **faster, lighter, and easier to use**.

Example:

- Page 1 → /students/?page=1 → Shows first 10 students.
- Page 2 → /students/?page=2 → Shows next 10 students.

◇ 8. Settings Configuration in Django

Jay sompura

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

In Django, **settings.py** is where we configure important project settings.

1. Database Configuration

- Django needs to know which **database** to use.
- Example (MySQL):

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'mydatabase',
      'USER': 'root',
      'PASSWORD': 'password123',
      'HOST': 'localhost',
      'PORT': '3306',
   }
}
```

- This connects Django to the database.

---

2. Static Files Configuration

- **Static files** are CSS, JS, and images used in your project.
- You need to tell Django where to find them:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / "static"]  # for development
STATIC_ROOT = BASE_DIR / "staticfiles"    # for production
```

---

3. API Keys Configuration

- When using **external services** (like Google API, payment gateway), you need **API keys**.

- Store them in settings.py or better in environment variables:

```
GOOGLE_API_KEY = "your_google_api_key_here"

PAYPAL_CLIENT_ID = "your_paypal_client_id_here"
```

- **Tip:** For security, don't put API keys directly in code for production; use .env files.

◇ 9. Project Setup

1: Setting up a Django REST Framework project

Answer (Simple English):

To start a Django REST Framework (DRF) project, you need to **create a Django project, install DRF, and set up basic structure**.

Steps to Setup a DRF Project

1. **Install Python and Django**

```
pip install django
```

2. **Install Django REST Framework**

```
pip install djangorestframework
```

3. **Create a Django Project**

```
django-admin startproject myproject

cd myproject
```

4. **Create a Django App**

```
python manage.py startapp myapp
```

5. **Add the App and REST Framework to settings.py**

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    ...
    'rest_framework',  # Add this
```

```
    'myapp',          # Add your app
]
```

6. **Run Initial Migrations**

```
python manage.py migrate
```

7. **Run Development Server**

```
python manage.py runserver
```

- Now your Django project is running at http://127.0.0.1:8000/.
- You can start creating **models, serializers, views, and URLs** for your API.

Here's a **simple and clear theory** for Social Authentication, Email, and OTP APIs in Django REST Framework:

◇ 10. Social Authentication, Email, and OTP Sending API

1. Social Authentication in Django

- Social authentication allows users to **log in using social accounts** like Google or Facebook instead of creating a new account.
- In Django, this is usually done using **third-party packages** like django-allauth or social-auth-app-django.

Steps:

1. Install the package:

```
pip install django-allauth
```

2. Add apps to settings.py:

```
INSTALLED_APPS = [
    ...
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',  # For Google
    'allauth.socialaccount.providers.facebook', # For Facebook
```

]

3. Configure URLs and redirect settings.
4. Users can now log in using Google/Facebook.

---

2. Sending Emails in Django

- Django can send emails using built-in **EmailMessage** class or external APIs like **SendGrid**.
- Example using Django's SMTP:

```
from django.core.mail import send_mail


send_mail(
   'Subject here',
   'Hello! This is a test email.',
   'from@example.com',
   ['to@example.com'],
   fail_silently=False,
)
```

- Using **SendGrid API**: You send email via API instead of SMTP for more reliability and features.

---

3. Sending OTPs using APIs

- OTPs (One-Time Passwords) are used for **verification/login**.
- Third-party APIs like **Twilio** or **MSG91** can send OTPs via SMS or Email.

Example using Twilio:

```
from twilio.rest import Client


account_sid = 'your_account_sid'
auth_token = 'your_auth_token'
client = Client(account_sid, auth_token)
```

```
message = client.messages.create(
    body="Your OTP is 123456",
    from_='+1234567890',
    to='+919876543210'
)
print(message.sid)
```

◇ 11. RESTful API Design

1. What is REST?

- **REST** (Representational State Transfer) is an **architecture style** for designing APIs.
- It makes APIs **simple, scalable, and easy to use**.

---

2. REST Principles

a) Statelessness

- Each API request must contain **all the information** needed for the server to process it.
- Server **does not store client session data** between requests.
- Example: If you call /students/1, server does not remember previous requests; it responds independently.

b) Resource-Based URLs

- Everything in REST is treated as a **resource** (like students, books, orders).
- URLs represent resources.
- Example:

/students/ → All students

/students/1/ → Student with ID 1

/books/ → All books

/books/23/ → Book with ID 23

c) Using HTTP Methods for CRUD Operations

- REST APIs use **HTTP methods** to perform operations on resources:

HTTP Method	Action	Example URL
GET	Read data	/students/
POST	Create data	/students/
PUT	Update data	/students/1/
PATCH	Partial update	/students/1/
DELETE	Delete data	/students/1/

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

1. What is CRUD?

- **CRUD** stands for the four basic operations we perform on data in any application:

Operation	Meaning	HTTP Method
Create	Add new data	POST
Read	Retrieve data	GET
Update	Modify existing data	PUT/PATCH
Delete	Remove data	DELETE

---

2. Why is CRUD Fundamental?

- CRUD operations are **the backbone of backend development** because almost every application needs to **manage data**.

- Examples:
  - A **blog** app → Create posts, Read posts, Update posts, Delete posts.
  - An **e-commerce** app → Create orders, Read products, Update cart, Delete items.
- Without CRUD, APIs cannot perform basic data management.

◇ 13. Authentication and Authorization API

1. Difference Between Authentication and Authorization

Term	Meaning
Authentication	Verifying **who the user is** (login, identity check).
Authorization	Checking **what the user is allowed to do** (permissions, access level).

Example:

- Authentication → You log in with username & password → Server knows you are "Jay".
- Authorization → Can "Jay" edit a post or only read it? → Server checks permissions.

---

2. Token-Based Authentication in Django REST Framework

- DRF provides **token-based authentication** to secure APIs.
- When a user logs in successfully, they get a **token**.
- Every future API request must include this token to access protected endpoints.

Steps to Implement:

1. **Install DRF auth package (if not already):**

```
pip install djangorestframework
```

2. **Add Token Authentication to settings.py**

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

```
    ]
}
```

3. **Create tokens for users**

```
python manage.py drf_create_token <username>
```

4. **Use token in API requests**

- Include in header:

```
Authorization: Token <user_token_here>
```

Example in Views:

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


class StudentList(APIView):
    permission_classes = [IsAuthenticated]


    def get(self, request):
        return Response({"message": "You can see this because you are authenticated!"})
```

◇ 14. OpenWeatherMap API Integration

1. What is OpenWeatherMap API?

- **OpenWeatherMap** is a service that provides **weather data** (current weather, forecasts, temperature, humidity, wind, etc.) via APIs.
- Developers can **retrieve weather information** for any city or location in real-time.

---

2. How to Retrieve Weather Data

1. **Sign Up & Get API Key**
    - Go to OpenWeatherMap → Register → Get a free API key.
2. **API Endpoint Example**
    - Current weather for a city:

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

3. **HTTP Request Method**
    - Use **GET** method to request data.
4. **Example Response (JSON)**

```
{
  "weather": [{"description": "clear sky"}],
  "main": {"temp": 298.15, "humidity": 50},
  "wind": {"speed": 3.6},
  "name": "London"
}
```

5. **Integrate in Django**
    - Use Python requests library to fetch weather data:

```
import requests

api_key = "YOUR_API_KEY"
city = "London"
url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}"

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

◇ 15. Google Maps Geocoding API

1. What is Google Maps Geocoding API?

- The **Geocoding API** converts **addresses into geographic coordinates** (latitude & longitude).
- It can also do the reverse: convert **coordinates into addresses** (reverse geocoding).
- Useful for maps, location tracking, delivery apps, etc.

---

2. How to Use the API

1. **Get API Key**
    - Go to Google Cloud Console → Enable Geocoding API → Get API key.
2. **API Endpoint Example**
    - Convert an address to coordinates:

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

3. **HTTP Request Method**
    - Use **GET** to request data.
4. **Example Response (JSON)**


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

```
}
```

5. **Integrate in Django**
    - Use Python requests library to fetch coordinates:

```
import requests


api_key = "YOUR_API_KEY"
address = "New York"
url = f"https://maps.googleapis.com/maps/api/geocode/json?address={address}&key={api_key}"


response = requests.get(url)
data = response.json()
lat = data['results'][0]['geometry']['location']['lat']
lng = data['results'][0]['geometry']['location']['lng']


print(f"Latitude: {lat}, Longitude: {lng}")
```

◇ 16. GitHub API Integration

1. What is GitHub API?

- GitHub provides a **RESTful API** to interact with repositories, pull requests, issues, and other features programmatically.
- It allows developers to **read, create, update, and delete GitHub data** without using the web interface.

---

2. How to Interact with GitHub API

1. **Get a Personal Access Token (PAT)**
    - Go to GitHub → Settings → Developer settings → Personal access tokens → Generate new token.
    - This token is used for **authentication** in API requests.
2. **API Endpoint Examples**

- Get user repositories:

GET https://api.github.com/users/<username>/repos

- Create an issue:

POST https://api.github.com/repos/<username>/<repo>/issues

- List pull requests:

GET https://api.github.com/repos/<username>/<repo>/pulls

3. **HTTP Request Methods**

- GET → Read data (repos, issues, pull requests)
- POST → Create new data (issues, comments)
- PATCH → Update existing data (issue, PR)
- DELETE → Remove data (e.g., repository files)

4. **Example using Python requests**

```
import requests

token = "YOUR_PERSONAL_ACCESS_TOKEN"
headers = {"Authorization": f"token {token}"}

# Get user repositories
url = "https://api.github.com/users/jaysompura/repos"
response = requests.get(url, headers=headers)
repos = response.json()

for repo in repos:
    print(repo['name'])
```

◇ 17. Twitter API Integration

Jay sompura

1. What is Twitter API?

- Twitter provides a **RESTful API** to interact with its platform programmatically.
- You can **fetch tweets, post tweets, and get user information** without using the Twitter website.
- Useful for building apps that **analyze trends, automate posting, or display tweets**.

---

2. How to Use Twitter API

1. **Create a Twitter Developer Account**
    - Go to Twitter Developer Portal → Create a project → Get **API Key, API Secret, Access Token, and Access Token Secret**.
2. **API Endpoint Examples**

- Get user tweets:

GET https://api.twitter.com/2/users/:id/tweets

- Post a tweet:

POST https://api.twitter.com/2/tweets

- Get user details:

GET https://api.twitter.com/2/users/:id

3. **Authentication**

- Use **OAuth 2.0 Bearer Token** or **OAuth 1.0a** for authentication.

4. **Example using Python requests**

```
import requests

bearer_token = "YOUR_BEARER_TOKEN"
headers = {"Authorization": f"Bearer {bearer_token}"}

# Get user tweets
user_id = "123456789"
url = f"https://api.twitter.com/2/users/{user_id}/tweets"
response = requests.get(url, headers=headers)
```

```
tweets = response.json()


for tweet in tweets['data']:
    print(tweet['text'])
```

◇ 18. REST Countries API Integration

1. What is REST Countries API?

- REST Countries API is a **free API** that provides **country-specific data** such as name, capital, population, region, currencies, languages, and more.
- Useful for building apps that need **geographic or demographic information**.

---

2. How to Retrieve Country Data

1. **API Endpoint Examples**

- Get all countries:

```
GET https://restcountries.com/v3.1/all
```

- Get country by name:

```
GET https://restcountries.com/v3.1/name/{country}
```

- Get country by code:

```
GET https://restcountries.com/v3.1/alpha/{code}
```

2. **HTTP Request Method**

- Use **GET** requests to fetch data.

3. **Example using Python requests**

```
import requests


# Get data for India

url = "https://restcountries.com/v3.1/name/India"

response = requests.get(url)
```

```
data = response.json()


print(f"Country: {data[0]['name']['common']}")
print(f"Capital: {data[0]['capital'][0]}")
print(f"Region: {data[0]['region']}")
print(f"Population: {data[0]['population']}")
```

◇ 19. Email Sending APIs (SendGrid, Mailchimp)

1. What are Email Sending APIs?

- Email sending APIs allow your app to **send emails programmatically** without using manual email clients.
- Common use cases:
  - Transactional emails (order confirmations, password resets)
  - Newsletters
  - Notifications
- Popular services: **SendGrid, Mailchimp**

---

2. How to Use Email APIs

a) SendGrid Example

1. Sign up at SendGrid → Get API key.
2. Install Python library:

```
pip install sendgrid
```

3. Send email using Python:

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


message = Mail(
```

```
    from_email='from@example.com',
    to_emails='to@example.com',
    subject='Test Email',
    html_content='<strong>Hello! This is a test email.</strong>'
)
sg = SendGridAPIClient('YOUR_SENDGRID_API_KEY')
response = sg.send(message)

print(response.status_code)
```

b) Mailchimp Example

1. Sign up at Mailchimp → Get API key and audience ID.
2. Use API to send emails to a mailing list.

◇ 20. SMS Sending APIs (Twilio)

1. What is Twilio API?

- **Twilio** is a cloud service that allows apps to **send SMS messages, OTPs, and make calls programmatically**.
- Useful for notifications, user verification, alerts, and 2FA (Two-Factor Authentication).

---

2. How to Use Twilio API

1. **Sign Up & Get Credentials**
    - Go to Twilio → Sign up → Get **Account SID, Auth Token, and a Twilio phone number**.
2. **Install Twilio Python Library**

```
pip install twilio
```

3. **Send SMS using Python**

```
from twilio.rest import Client
```

```
account_sid = 'YOUR_ACCOUNT_SID'
auth_token = 'YOUR_AUTH_TOKEN'
client = Client(account_sid, auth_token)

message = client.messages.create(
    body="Your OTP is 123456",
    from_='+1234567890',  # Twilio phone number
    to='+919876543210'    # Recipient number
)

print(message.sid)
```

◇ 21. Payment Integration (PayPal, Stripe)

1. What is Payment Integration?

- Payment integration allows your app or website to **accept online payments** securely.
- Popular payment gateways: **PayPal** and **Stripe**.
- Useful for e-commerce, subscriptions, donations, and other transactions.

---

2. How to Integrate Payment Gateways

a) PayPal Integration

1. Sign up for a PayPal Developer Account → Get **Client ID & Secret**.
2. Use **PayPal SDK or REST API** to process payments.
3. Example flow:
    - Create a payment → Redirect user to PayPal → User pays → PayPal sends confirmation → Update your app.

b) Stripe Integration

1. Sign up for Stripe → Get **API keys (Publishable & Secret key)**.

2. Use Stripe SDK or REST API to create payment intents and process payments.
3. Example flow:
    - Create a PaymentIntent → Collect card details → Confirm payment → Receive webhook → Update your app.

◇ 22. Google Maps API Integration

1. What is Google Maps API?

- Google Maps API allows apps to **display interactive maps** and perform **location-based operations** like calculating distances or getting directions.
- Useful for delivery apps, travel apps, location tracking, and geospatial services.

---

2. How to Use Google Maps API

1. **Get API Key**
    - Go to Google Cloud Console → Enable Maps API → Get **API key**.
2. **API Features**

- **Maps Display** → Show interactive map on your website or app.
- **Distance Matrix** → Calculate distance & travel time between multiple locations.
- **Directions API** → Get routes between two points.

3. **Example: Display Map**

```
<!DOCTYPE html>
<html>
 <head>
  <title>Simple Map</title>
  <script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY"></script>
  <script>
   function initMap() {
    var location = {lat: 40.7128, lng: -74.0060}; // New York
```

```
      var map = new google.maps.Map(document.getElementById('map'), {
        zoom: 10,
        center: location
      });
      var marker = new google.maps.Marker({position: location, map: map});
    }
  </script>
 </head>
 <body onload="initMap()">
  <div id="map" style="height:500px;width:100%;"></div>
 </body>
</html>
```

4. **Example: Distance Calculation (Python)**

```
import requests

api_key = "YOUR_API_KEY"
origins = "New York, NY"
destinations = "Los Angeles, CA"
url = f"https://maps.googleapis.com/maps/api/distancematrix/json?origins={origins}&destinations={destinations}&key={api_key}"

response = requests.get(url)
data = response.json()
print(data['rows'][0]['elements'][0]['distance']['text'])
```
