2030/2040

REST FRAMEWORK

Presented to

Prashant Shukla Sir

TOPS Technologies

HTML in Python

Que.1
An API is a set of rules that allows different software system to communicate with each other.
API defines how program can request and exchange data.
It lets apps like Instagram, weather app, or payment apps gateways talk to other services without needing to know how they are built internally.

Que.2
There are two major type of apis commonly used in web services: REST and SOAP

REST : (Representational State Transfer)
Architectural style(not a protocol)
Data format: Mostly uses JSON
Communication: Over HTTP (GET,POST, PUT, DELETE)
Lightweight: Faster, more scalable and simpler than SOAP
Stateless: Each request is independent; no session is stored.

SOAP : (Simple Object Access Protocol)
Strict Rules(Protocols)
Data formation(uses only XML)
Communication : Can Use HTTP, SMTP, TCP, etc
Strict : Built in error handling, security (WS-Security)
Heavyweight : More bandwidth and processing needed

Que.3
Why APIs Are Important in web development :

1. Connect Frontend & Backend
   API allows your frontend (REACT website) to request and send data to the backend eg Django or Node.js server

2. Third - Party Integration
   Want maps, payments via Google/Facebook ? API makes that possible

3. Modular Architecture
   APIs enables microservices, where each services handles a specific tasks and communication with others via API's

Que.4

```
import requests

def get_random_joke():
 url = "https://official-joke-
api.appspot.com/random_joke"

 try:
  response = request.get(url)
  response.raise_for_status()
  joke = response.json()

  print(f"\n Here's a joke for you:
\n{joke['setup']}\n {joke['punchline']}")

 except request.exceptions.RequestException as
e:
  print(f"Failed to fetch a joke. Error : {e}")
```

Que.5

Requirements for web dev projects:

Technical Requirement

1. Frontend
   CSS, HTML, JavaScript

2. Backend
   Python (Django), NodeJs

3. Database
   MySQL

4. API integration
   REST for dynamic data

5. Hosting
   Deployment platforms

Design Requirements

1. UI/UX
   Wireframes, mockups

2. Responsive Design
   Works on all screen sizes

3. Branding
   Logos, color schemes, fonts

4. Accessibility
   Follows standards

Project Management

1. Requirement gathering
   What the client or user needs

2. Vision control
   Git + GitHub/GitLab/Bitbucket

3. Documentations
   Readme, code comments, API documents

3. Serializer

Serialization is the process of converting complex data—like Python objects or database querysets—into a format that can be easily stored or transferred, such as JSON or XML.
In web development (especially Django Rest Framework), serialization is mostly used to:

- Convert Django model instances into JSON so they can be sent over an API.
- Convert incoming JSON data back into Django model instances (for saving to the database).

Example in Django Rest Framework:

```
serializer = MyModelSerializer(my_object)
serializer.data  # returns JSON-friendly Python
dict
```

convert a Django QuerySet to JSON

1. serializers.serialize() from django.core.serializers

```
from django.core.serializers import serialize
from myapp.models import MyModel

data = MyModel.objects.all()
json_data = serialize('json', data)
print(json_data)
```

2. Using Django REST Framework Serializers (Recommended for APIs)

```
# serializers.py
from rest_framework import serializers
from myapp.models import MyModel

class
MyModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = MyModel
        fields = '__all__'
```

View:

```
# views.py
from rest_framework.response import Response
from myapp.models import MyModel
from .serializers import MyModelSerializer

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

Using serializers in Django REST Framework (DRF) is essential to convert complex data types (like Django models) to and from JSON, enabling smooth communication between your frontend and backend.

1. Create a Django Model

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

class Transaction(models.Model):
    CHOICE = (
        ('DEBIT', 'DEBIT'),
        ('CREDIT', 'CREDIT'),
    )
    title = models.CharField(max_length=200)
    amount = models.FloatField()
    type = models.CharField(max_length=6,
choices=CHOICE)
```

2. Create a Serializer

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

class
TransactionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Transaction
        fields = '__all__'  # Or list specific fields
```

3. Use Serializer in a View

```
# views.py
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Transaction
from .serializers import TransactionSerializer

@api_view(['GET'])
def transaction_list(request):
    transactions = Transaction.objects.all()
    serializer = TransactionSerializer(transactions,
many=True)
    return Response(serializer.data)
```

4. Map the View to a URL

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

urlpatterns = [
    path('transactions/', views.transaction_list),
]
```

Que.6

In web Development, HTTP request method define the type of operations you want to perform on a resource (like on a server).

1. **GET**

Purpose : Retrieve data from the server

ex : View a list of blog posts or a user profile

safe : Doesn't change any data

GET /users/1/

2. **POST**

Purpose : Submit new data to the server

ex : Create a new user or post a comment

Creates A new resource

POST /users/
Body: { "name": "Alice", "email": "alice@example.com" }

3. **PUT**
Purpose : Update an existing resource completely

ex : Replaces all fields of a user profile

Repeating the request in the same update

PUT /users/1/
Body: { "name": "Alice", "email": "new@example.com" }

4. **PATCH**
Purpose : Partially update a resource

ex : Change only the user's email

PATCH /users/1/
Body: { "email": "patch@example.com" }

5. **DELETE**
Purpose : Remove a resource from the server
ex : Delete a User

```
DELETE /users/1/
```

Que.7

In Django Rest Framework (DRF), sending and receiving responses is handled using the Response object from rest_framework.response

Import Required Classes

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Transaction
from .serializers import TransactionSerializer
```

Receiving Requests & Sending Responses
GET Request (Read)

```
@api_view(['GET'])
def get_transactions(request):
    transactions = Transaction.objects.all()
    serializer = TransactionSerializer(transactions,
many=True)
    return Response(serializer.data)
```

POST Request (Create)

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

PUT/PATCH Request (Update)

```
@api_view(['PUT'])
def update_transaction(request, pk):
    transaction = Transaction.objects.get(id=pk)
    serializer = TransactionSerializer(transaction,
data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(serializer.errors, status=400)
```

DELETE Request

```
@api_view(['DELETE'])
def delete_transaction(request, pk):
    transaction = Transaction.objects.get(id=pk)
    transaction.delete()
    return Response({"message": "Deleted
successfully"}, status=204)
```

Que.8

In Django REST Framework (DRF), views are responsible for handling incoming HTTP requests and returning responses.

1. Function-Based Views (FBVs)

These are simple Python functions. They're easy to understand and are often used for smaller or very custom APIs.

code :

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

@api_view(['GET'])
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

Pros:

- Simple and readable.
- Great for beginners or small endpoints.
- Explicit control over logic.

Cons:

- Can become repetitive and messy for large projects.
- Harder to reuse common behavior (e.g., authentication, permission checks).

2. Class-Based Views (CBVs)

These are classes that extend DRF's generic views or API View. They're more scalable and reusable.

ex :

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

class HelloWorld(APIView):
    def get(self, request):
        return Response({"message": "Hello, world!"})
```

Pros:

- More organized for large apps.
- Built-in features like mixins and generics.
- Promotes reusable and maintainable code.

Cons:

- Slightly harder to understand for beginners.
- More boilerplate for small tasks.

Que.9
In Django REST Framework (DRF), to handle HTTP requests, you need to define URLs and link them to your views

1. For Function-Based Views (FBV)

View:

```
# views.py
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

URL:

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

urlpatterns = [
    path('hello/', hello_world),
]
```

2. For Class-Based Views (CBV)

View:

```
# views.py
from rest_framework.views import APIView
from rest_framework.response import Response

class HelloWorld(APIView):
    def get(self, request):
        return Response({"message": "Hello from
class-based view"})
```

URL:
```
# urls.py
from django.urls import path
from .views import HelloWorld

urlpatterns = [
    path('hello-cbv/', HelloWorld.as_view()),  #
Note: `.as_view()` is required
]
```

3. Using Routers (with ViewSets)

ViewSet Example:
```
from rest_framework import viewsets
from .models import Transaction
from .serializers import TransactionSerializer

class
TransactionViewSet(viewsets.ModelViewSet):
    queryset = Transaction.objects.all()
    serializer_class = TransactionSerializer
```

URL with Router:
```
from django.urls import path, include
from rest_framework.routers import
DefaultRouter
from .views import TransactionViewSet
```

```
router = DefaultRouter()
router.register(r'transactions',
TransactionViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

Que.10

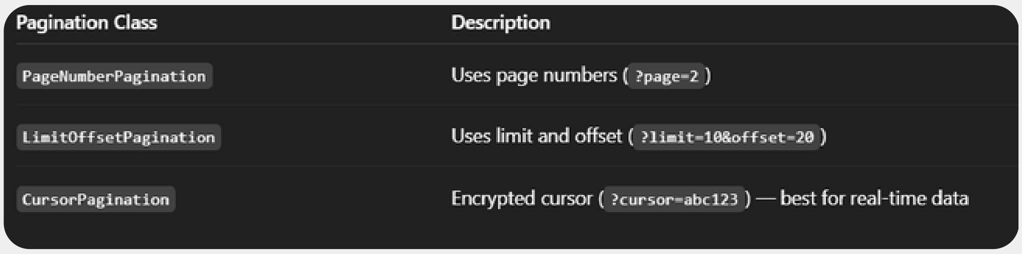
1. Set Global Pagination (Recommended for most cases)

Update your settings.py:

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

This adds pagination to all views that return a queryset.

2. Types of Pagination Classes in DRF



Pagination Class	Description
`PageNumberPagination`	Uses page numbers (`?page=2`)
`LimitOffsetPagination`	Uses limit and offset (`?limit=10&offset=20`)
`CursorPagination`	Encrypted cursor (`?cursor=abc123`) — best for real-time data

3. Customize Pagination per View

If you want pagination only for certain views:

```
from rest_framework.pagination import
PageNumberPagination
from rest_framework.generics import ListAPIView
from .models import Transaction
from .serializers import TransactionSerializer

class CustomPagination(PageNumberPagination):
    page_size = 5

class TransactionList(ListAPIView):
    queryset = Transaction.objects.all()
    serializer_class = TransactionSerializer
    pagination_class = CustomPagination
```

4. Response Format

```
{
    "count": 42,
    "next": "http://example.com/api/items/?
page=2",
    "previous": null,
    "results": [
        { "id": 1, "name": "Item 1" },
        ...
    ]
}
```

Que.11

Configuring Django settings properly is essential for development and production environments.

1. Database Configuration

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

2. Static Files Configuration

```
STATIC_URL = '/static/'

# For collecting static files (e.g., in production)
STATICFILES_DIRS = [BASE_DIR / "static"]  #
during development
STATIC_ROOT = BASE_DIR / "staticfiles"   # for
collectstatic
```

3. API Keys and Secrets

```
import os

API_KEY = os.getenv('API_KEY')  # Reads from
system environment
```

And in your .env file:

```
API_KEY=your_secret_api_key
```

Que.12

1. Create 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 Django Project

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

4. Create a Django App

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

5. Update settings.py

```
# settings.py

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

6. Create a Model

```
# myapp/models.py

from django.db import models

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

7. Create a Serializer

```
# myapp/serializers.py

from rest_framework import serializers
from .models import Item

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

8. Create a View

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Item
from .serializers import ItemSerializer

@api_view(['GET'])
def item_list(request):
    items = Item.objects.all()
    serializer = ItemSerializer(items, many=True)
    return Response(serializer.data)
```

9. Define URL Patterns

```
# myapp/urls.py

from django.urls import path
from .views import item_list

urlpatterns = [
    path('items/', item_list),
]
```

And include this in your main urls.py:

```
# myproject/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('myapp.urls')),
]
```

10. Run the Server

```
python manage.py runserver
```

Que.13
To implement social authentication (Google, Facebook, etc.) in Django, the easiest and most popular way is to use the django-allauth package, often combined with dj-rest-auth for Django REST Framework.

Step-by-Step: Social Login with Google/Facebook

Install Required Packages

```
pip install django-allauth dj-rest-auth
```

For Google and Facebook:

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

Update settings.py

```
INSTALLED_APPS = [
    ...
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',  # or
'facebook'
    'dj_rest_auth',
    'dj_rest_auth.registration',
]
```

```
SITE_ID = 1

AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',

'allauth.account.auth_backends.AuthenticationB
ackend',
)

REST_USE_JWT = True  # Optional for JWT
support
```

Include URLs

```
# project/urls.py

from django.urls import path, include

urlpatterns = [
    path('auth/', include('dj_rest_auth.urls')),
    path('auth/registration/',
include('dj_rest_auth.registration.urls')),
    path('auth/social/',
include('allauth.socialaccount.urls')),
]
```

Get OAuth Credentials

- Google: Go to Google Cloud Console
- Facebook: Go to Facebook Developers

Create an app and get:

- Client ID
- Client Secret
- Set redirect URI: http://localhost:8000/accounts/google/login/callback/

Add Credentials to settings.py

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

Run Migrations

```
python manage.py migrate
```

Test Authentication

Now, you can hit endpoints like:

- GET /auth/social/login/google/
- Or use frontend SDKs (React, etc.) to redirect to Google, then send the token to DRF for authentication

Que.14

A. Sending Emails via SendGrid

Install SendGrid Package

```
pip install sendgrid
```

Configure SendGrid in settings.py

```
EMAIL_BACKEND =
"sendgrid_backend.SendgridBackend"
SENDGRID_API_KEY = "your_sendgrid_api_key"
SENDGRID_SANDBOX_MODE_IN_DEBUG = False
# Optional
```

Send Email (Example)

```
from django.core.mail import send_mail

send_mail(
    subject="Your OTP Code",
    message="Your OTP is 123456",
    from_email="your@email.com",
    recipient_list=["user@example.com"],
)
```

B. Sending OTP via Twilio SMS

Install Twilio SDK

```
pip install twilio
```

Send SMS (View Example)

```
from twilio.rest import Client

def send_otp(phone_number, otp):
    account_sid = "your_twilio_account_sid"
    auth_token = "your_twilio_auth_token"
    client = Client(account_sid, auth_token)

    message = client.messages.create(
        body=f"Your OTP is {otp}",
        from_="+1XXXXXXXXXX",  # Your Twilio phone
number
        to=phone_number
    )

    return message.sid
```

C. Generating OTP (Optional)

```
import random

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

D. Security Tips

- Never hardcode API keys—use .env + python-dotenv.
- OTPs should expire (store them temporarily in DB or cache like Redis).

Que.15

REST PRINCIPLES

1. Statelessness

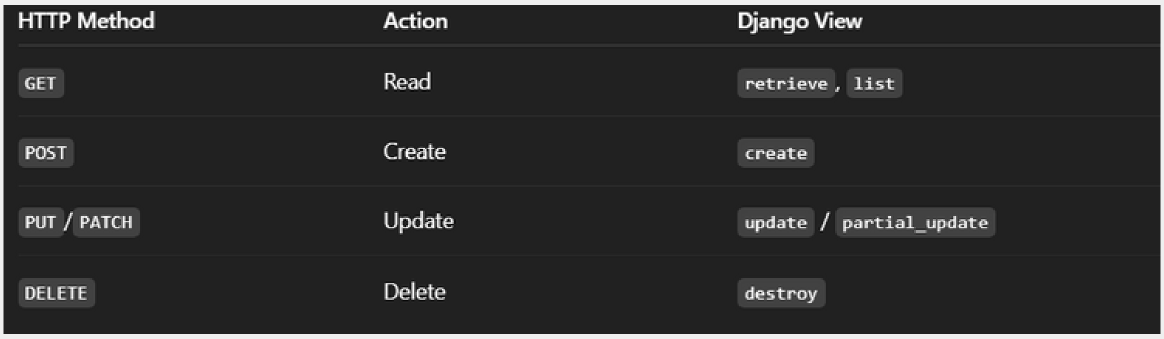
- Every HTTP request must contain all the information needed for the server to understand and process it.
- The server does not store any session info between requests.

2. Resource-Based URLs

- REST treats everything as a resource (e.g., users, posts, transactions).
- URLs should refer to nouns, not actions.

Ex:
GET   /transactions/   → list all transactions
POST   /transactions/   → create a new transaction
GET   /transactions/5/   → get transaction with ID 5
PUT   /transactions/5/   → update transaction with ID 5
DELETE /transactions/5/   → delete transaction with ID 5

3. HTTP Methods for CRUD



HTTP Method	Action	Django View
`GET`	Read	`retrieve`, `list`
`POST`	Create	`create`
`PUT` / `PATCH`	Update	`update` / `partial_update`
`DELETE`	Delete	`destroy`

Que.16
CRUD stands for Create, Read, Update, Delete — the four basic operations that any database or backend system must support to manage data effectively.

Why CRUD Is Fundamental in Backend Development:

- Core to Data Handling: Everything from user registration to product catalogs involves CRUD operations.
- Database Interaction: CRUD maps directly to SQL:
- Create → INSERT
- Read → SELECT
- Update → UPDATE
- Delete → DELETE
- Standardized API Design: RESTful APIs are built around CRUD using appropriate HTTP methods.

Que.17

The difference between authentication and authorization is fundamental in backend and API security:

1. Authentication – "Who are you?"
    - Definition: Verifies the identity of the user.
    - Purpose: Confirms the user is who they claim to be.
    - Examples:
        - Logging in with username and password.
        - Using an API token or JWT.

2. Authorization – "What can you do?"
    - Definition: Determines the permissions of an authenticated user.
    - Purpose: Controls access to resources and actions.
    - Examples:
        - A regular user cannot delete other users.
        - Only admins can access /admin/ or perform certain operations.

Que.18

To implement token-based authentication in Django REST Framework (DRF), you can use DRF’s built-in TokenAuthentication.

1. Install DRF Token Auth Module

```
pip install djangorestframework
```

Then add 'rest_framework.authtoken' to INSTALLED_APPS in settings.py:

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

2. Run Migrations

```
python manage.py migrate
```

3. Configure Authentication in settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [

'rest_framework.authentication.TokenAuthenticat
ion',
    ]
}
```

4. Create Token on User Creation (Optional)

```
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 in apps.py, connect the signal:

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

5. Create Login Endpoint to Get Token

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

urlpatterns = [
    path('api/token/', obtain_auth_token),  # POST
with username & password
]
```

6. Protect a View with Token Authentication

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

class ProtectedView(APIView):
    authentication_classes = [TokenAuthentication]
    permission_classes = [IsAuthenticated]
    def get(self, request):
        return Response({"message": f"Hello
{request.user.username}, you're authenticated!"})
```

7. Using the Token in Requests

```
Authorization: Token your_token_here
```

Que.19

The OpenWeatherMap API provides real-time weather data like temperature, humidity, wind speed, and forecasts. It’s a popular and free API (with paid tiers) often used in web and mobile apps.

Sign Up and Get API Key

- Visit https://openweathermap.org/api
- Create an account and get your API key.

Base API URL Structure

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

Example API Call (in Python)

```
import requests

def get_weather(city):
    API_KEY = 'your_api_key'
    url =
f"https://api.openweathermap.org/data/2.5/weat
her?q={city}&appid={API_KEY}&units=metric"
```

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

    if response.status_code == 200:
        return {
            'temperature': data['main']['temp'],
            'description': data['weather'][0]
['description'],
            'city': data['name']
        }
    else:
        return {'error': data.get('message',
'Something went wrong')}
```

Que.20
The Google Maps Geocoding API allows you to convert human-readable addresses into geographic coordinates (latitude & longitude), and vice versa.

Get a Google Maps API Key

- Go to: https://console.cloud.google.com/
- Create a project, enable "Geocoding API", and get your API key.

Make a Request to the Geocoding API

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

Example:

```
import requests

def geocode_address(address):
    API_KEY = 'your_api_key'
    url =
f"https://maps.googleapis.com/maps/api/geocod
e/json?address={address}&key={API_KEY}"

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

    if data['status'] == 'OK':
        location = data['results'][0]['geometry']
['location']
        return {
            'latitude': location['lat'],
            'longitude': location['lng'],
            'formatted_address': data['results'][0]
['formatted_address']
        }
    else:
        return {'error': data.get('error_message',
'Address not found')}
```

Que.21

The GitHub API is a RESTful (and GraphQL) interface that allows developers to programmatically interact with GitHub — including repositories, pull requests, issues, commits, and users.

Common Use Cases

1. List repositories for a user
2. Create or comment on issues
3. Get details or merge pull requests
4. Automate workflows (e.g., CI/CD)

1. Authentication

- For basic access, use public endpoints.
- For writing/creating content, use a Personal Access Token (PAT).

```
Authorization: Bearer YOUR_GITHUB_TOKEN
```

2. Example: List Public Repositories of a User

```
import requests

def list_repos(username):
    url =
f"https://api.github.com/users/{username}/repos
"
    response = requests.get(url)
    return response.json()
```

3. Create an Issue in a Repo

```
def create_issue(owner, repo, token, title, body):
    url =
f"https://api.github.com/repos/{owner}/{repo}/is
sues"
    headers = {
        "Authorization": f"Bearer {token}",
        "Accept": "application/vnd.github.v3+json"
    }
    data = {
        "title": title,
        "body": body
    }
    response = requests.post(url,
headers=headers, json=data)
    return response.json()
```

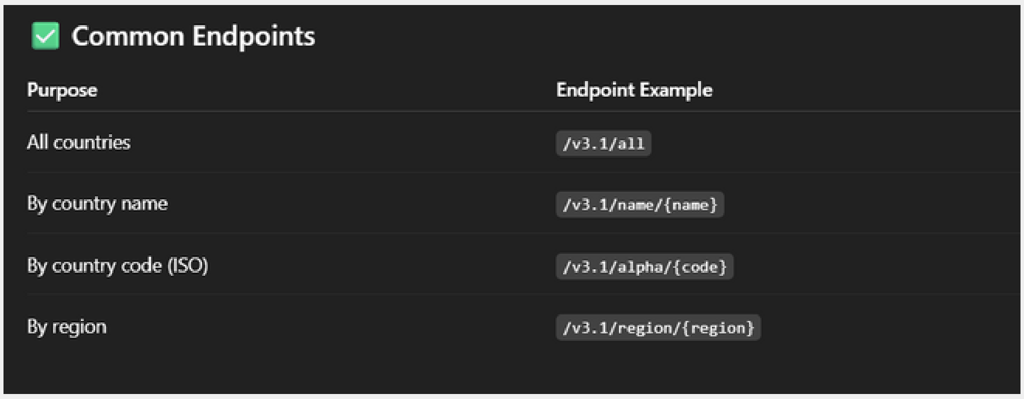
4. Get Pull Request Info

```
def get_pull_requests(owner, repo):
    url =
f"https://api.github.com/repos/{owner}/{repo}/p
ulls"
    response = requests.get(url)
    return response.json()
```

GitHub REST API Docs:
https://docs.github.com/en/rest

Que.22
The REST Countries API is a free public API that provides detailed information about countries — such as name, capital, currency, population, languages, flags, and more. It's great for educational apps, travel tools, or learning REST concepts.

Base URL
https://restcountries.com/



✅ Common Endpoints

Purpose	Endpoint Example
All countries	`/v3.1/all`
By country name	`/v3.1/name/{name}`
By country code (ISO)	`/v3.1/alpha/{code}`
By region	`/v3.1/region/{region}`

Example: Get Info by Country Name

```
import requests

def get_country_info(name):
    url =
f"https://restcountries.com/v3.1/name/{name}"
    response = requests.get(url)
    data = response.json()

    if response.status_code == 200:
        country = data[0]
        return {
            'name': country['name']['common'],
            'capital': country.get('capital', ['N/A'])[0],
            'population': country.get('population'),
            'currency': list(country['currencies'].keys())
[0],
            'flag': country['flags']['png']
        }
    else:
        return {"error": "Country not found"}
```

Que.23

SendGrid API – Sending Transactional Emails

Install SendGrid SDK:

```
pip install sendgrid
```

Send Email with Python:

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

def send_transactional_email(to_email, subject,
body):
    message = Mail(
        from_email='your@email.com',
        to_emails=to_email,
        subject=subject,
        html_content=body
    )

    try:
        sg =
SendGridAPIClient(os.getenv('SENDGRID_API_KE
Y'))
        response = sg.send(message)
        return response.status_code
    except Exception as e:
        return str(e)
```

Que.24
The Twilio API is a powerful and reliable service for sending SMS, voice messages, and OTPs (One-Time Passwords) programmatically. It's widely used in authentication, alerts, and real-time communication systems.

Sign Up and Get Credentials

- Go to https://www.twilio.com/
- Create a free account.
- Get:
- Account SID
- Auth Token
- A Twilio phone number (to send SMS)

Install Twilio Python SDK

```
pip install twilio
```

Send SMS/OTP Example in Python

```
from twilio.rest import Client

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

    message = client.messages.create(
        body=f"Your OTP is {otp}",
        from_='+1234567890',  # Your Twilio phone
number
        to=phone_number
    )
    return message.sid  # or message.status
```

Generate OTP (Simple)

```
import random

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

Que.25

Integrating payment gateways like PayPal and Stripe allows your application to securely process online payments — for subscriptions, products, services, or donations.

1. Stripe Integration Overview

What Stripe Offers:

- One-time and subscription payments
- Easy API and SDKs (Python, JS, etc.)
- Webhooks for real-time payment status

Basic Steps:

1. Sign up at https://dashboard.stripe.com
2. Get your test API keys
3. Install Stripe SDK:

```
pip install stripe
```

Example Payment Intent in Django:

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

def create_payment_intent(amount):
    intent = stripe.PaymentIntent.create(
        amount=int(amount * 100),  # Stripe uses
cents
        currency='usd',
        payment_method_types=['card'],
    )
    return intent.client_secret
```

2. PayPal Integration Overview

What PayPal Offers:

- Trusted payment system globally
- Supports PayPal accounts, cards, and subscriptions
- REST APIs and SDKs for Python, Node.js, etc.

Basic Steps:

1. Create developer account at https://developer.paypal.com
2. Create sandbox app and get:
    - Client ID
    - Secret

Install SDK:

```
pip install paypalrestsdk
```

Example Payment Creation:

```
import paypalrestsdk

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

payment = paypalrestsdk.Payment({
    "intent": "sale",
    "payer": {"payment_method": "paypal"},
    "redirect_urls": {
        "return_url":
"http://localhost:8000/payment-success",
        "cancel_url":
"http://localhost:8000/payment-cancel"
    },
    "transactions": [{
        "amount": {"total": "10.00", "currency":
"USD"},
        "description": "Test Payment"
    }]
})

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

Que.26
The Google Maps API lets you display maps, place markers, and calculate distances between locations — making it ideal for delivery apps, travel sites, or geolocation tools.

Key Google Maps APIs for This Use Case

1. Maps JavaScript API — to render interactive maps on websites
2. Directions API or Distance Matrix API — to calculate distance/time between places

1. Displaying a Map with Markers (JavaScript)

```
<!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: 28.6139, lng: 77.2090 }; //
New Delhi
        var map = new
google.maps.Map(document.getElementById('ma
p'), {
          zoom: 10,
          center: location
        });
       >
```

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

2. Calculate Distance Using Distance Matrix API

```
import requests

def get_distance(origin, destination, api_key):
    url =
"https://maps.googleapis.com/maps/api/distance
matrix/json"
    params = {
        "origins": origin,
        "destinations": destination,
        "key": api_key
    }

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

    if data['status'] == 'OK':
        return {
            'distance': data['rows'][0]['elements'][0]
['distance']['text'],
            'duration': data['rows'][0]['elements'][0]
['duration']['text']
        }
    else:
        return {"error": data.get("error_message",
"Request failed")}
```