Module 10) Rest Framework

Introduction to APIs

**Theory:**

**What is an API (Application Programming Interface)?**

An **API** is a set of rules and protocols that allows one software application to interact with another. It defines how different software components should communicate, making it easier for developers to integrate and use functionalities from other programs, services, or platforms without knowing their internal details.

In simple terms, an API is like a waiter in a restaurant: you tell the waiter (API) what you want, and the waiter communicates with the kitchen (software) to get your order and bring it back.

**Types of APIs: REST, SOAP.**

### Types of APIs

Two common types used in web development are:

**1.REST (Representational State Transfer) API:**

Uses standard HTTP methods like GET, POST, PUT, DELETE.

Works with resources identified by URLs (endpoints).

Often returns data in JSON or XML format.

Lightweight, easy to use, and widely adopted for web and mobile apps.

Stateless: Each request from client to server must contain all necessary information.

**2.SOAP (Simple Object Access Protocol) API:**

Uses XML-based messaging protocol.

More rigid and standardized with built-in error handling.

Uses protocols like HTTP, SMTP, TCP for message transport.

Supports complex operations and higher security standards.

Often used in enterprise systems and legacy applications.

**Why are APIs important in web development?**

**Integration:** APIs let different systems and services communicate easily, enabling developers to combine functionalities (like payment gateways, social media login, or maps) into their apps without building from scratch.

**Reusability:** APIs expose certain features so developers can reuse them in multiple projects, speeding up development.

**Modularity:** By separating backend services (via APIs) from frontend interfaces, it’s easier to update, maintain, and scale applications.

**Automation:** APIs allow automatic data exchange and operations between applications without manual intervention.

**Ecosystem Growth:** Many platforms offer public APIs, encouraging developers to build on top of them, fostering innovation and rich ecosystems.

**Lab:**

Write a Python program that consumes a simple public API (e.g., a joke API).

**Practical Example:**

1. Write a Python script to fetch a random joke from an API and display it on the console.2. Requirements for Web Development Projects

**Theory:**

**Understanding project requirements.**

Before you start coding, it’s crucial to fully understand **what the project needs to do**. This includes:

**Gathering requirements:** Talk with stakeholders (clients, users, team members) to find out what features and functionalities are needed.

**Defining scope:** Clarify what will be included in the project and what won’t, to avoid scope creep.

**Functional requirements:** What the system **should do** — e.g., user login, data display, form submission.

**Non-functional requirements:** Performance, security, usability, scalability.

**Constraints & assumptions:** Budget limits, technology stack preferences, deadlines.

**Use cases or user stories:** Step-by-step descriptions of how users will interact with the system.

**Documentation:** Write clear requirement documents or briefs to refer back to during development.

**Why it matters:** Proper understanding saves time, reduces errors, and ensures the final product meets expectations.

**Setting up the environment and installing necessary packages.**

Once the requirements are clear, the next step is to **prepare your development environment** so you can start building efficiently.

**Choose the right tools:** Decide on the programming languages, frameworks, libraries, and databases based on the project needs.

**Install software and dependencies:**

For example, if you’re working on a Python web project, install Python, a code editor like VSCode, and packages like Django or Flask using package managers (e.g., pip).

For JavaScript projects, install Node.js and use npm or yarn to add libraries like React, Express, etc.

**Set up version control:** Initialize Git repository to track code changes.

**Configure environment variables:** Store sensitive info like API keys or database credentials securely.

**Create project structure:** Organize files and folders logically.

**Test the setup:** Run simple test scripts to ensure everything is working.

**Lab:**



Write a requirements.txt file for a Django project that includes all necessary dependencies.

**Practical Example:** 2) Write a Python script to set up a Django project and install packages

like django, djangorestframework, requests, etc.

3. Serialization in Django REST Framework

**Theory:**

**What is Serialization?**

**Serialization** is the process of converting complex data types — such as objects, querysets, or custom data structures — into a format that can be easily stored or transmitted. Common formats include JSON, XML, or YAML.

**Why serialize?**  
When you want to send data over the web (e.g., from your backend to frontend or API clients), you need to convert Python objects into JSON or another standard format that can be understood universally.

**Deserialization** is the reverse: converting data from JSON or another format back into Python objects.

**Converting Django QuerySets to JSON.**

Django QuerySets represent collections of database objects. To send QuerySet data as JSON (e.g., in an API response), you need to serialize it.

**Basic way using Django's built-in serializers:**

from django.core import serializersfrom django.http import JsonResponsefrom myapp.models import Book

def books\_json(request):

books = Book.objects.all()

data = serializers.serialize('json', books)

return JsonResponse(data, safe=False) # safe=False because data is a JSON string

serializers.serialize('json', queryset) converts the QuerySet to a JSON string.

However, this method includes extra metadata and the JSON format is not always easy to customize.

**Using serializers in Django REST Framework (DRF).**

Django REST Framework provides a powerful and flexible **Serializer** class to convert complex data (models, querysets) to JSON and validate incoming data.

**Lab:**



Create a Django REST API with serialization for a Doctor model.

**Practical Example:** 3) Write a Django REST API to serialize a Doctor model with fields

like name, specialty, and contact details.

4. Requests and Responses in Django REST Framework

**Theory:**

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

### HTTP Request Methods

These are the main HTTP methods used in web APIs to perform different actions on resources:

**1.GET:**

Retrieves data from the server.

Should NOT change any data (safe and idempotent).

Example: Fetch a list of users or details of a specific user.

**2.POST:**

Sends data to the server to create a new resource.

Typically used for form submissions or creating new records.

Example: Create a new user.

**3.PUT:**

Updates an existing resource by replacing it with the provided data.

Idempotent (multiple identical requests result in the same state).

Example: Update user details completely.

**4.DELETE:**

Deletes the specified resource.

Example: Remove a user from the database.

**Sending and receiving responsesin DRF.**

DRF provides tools to easily handle HTTP requests and send responses in API views.

**Lab:**



Create a Django REST API that accepts POST requests to add new doctor profiles.

**Practical Example:** 4) Write a Django project where the API accepts a POST request to add

a doctor’s details to the database.5. Views in Django REST Framework

**Theory:**

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

### 1. Function-Based Views (FBVs)

**Definition:**  
Simple Python functions decorated with @api\_view that handle HTTP requests.

**How it works:**  
You explicitly check the HTTP method (GET, POST, etc.) inside the function and write logic accordingly.

**Pros:**

Easy to understand and write for simple APIs.

Clear flow for beginners.

**Cons:**

Can get messy and repetitive as complexity grows.

Less reusable and harder to extend.

### 2. Class-Based Views (CBVs)

**Definition:**  
Classes that inherit from DRF base classes like APIView or generic views.

**How it works:**  
You define methods like get(), post(), put(), delete() inside the class, each corresponding to HTTP methods.

**Pros:**

More organized and easier to maintain for complex APIs.

Supports inheritance and mixins for reusable behavior.

Works well with DRF’s generic views and viewsets.

**Cons:**

Slightly more complex to grasp for beginners.

**Lab:**

**Implement a class-based view in DRF for managing doctor profiles.**

**Practical Example:** 5) Write a Django project that implements a class-based view to handle

doctor profile creation, reading, updating, and deletion (CRUD operations).

6. URL Routing in Django REST Framework

**Theory:**

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

### 1. ****Basic URL Configuration in Django****

URLs in Django are defined using the urls.py file. You link specific URL patterns (paths) to view functions or classes.

### 2. ****Using DRF’s Routers with ViewSets (Advanced/Recommended)****

When using ViewSets, DRF provides a **Router** that automatically handles URL routing for common actions like list, retrieve, create, update, delete.

**Lab:**



Set up URL routing in a Django project to link to CRUD API endpoints for doctors.

**Practical Example:** 6) Write a Django project that routes URLs to the views handling doctor

CRUD operations (/doctors, /doctors/<id>).

7. Pagination in Django REST Framework

**Theory:**

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

Adding **pagination** in Django REST Framework (DRF) is a smart way to handle large datasets by returning only a portion of the results at a time, rather than everything at once.

## ✅ Why Pagination?

Improves performance (especially on large databases).

Reduces network load and response time.

Gives clients control to fetch data page by page.

## Step-by-Step: Adding Pagination in DRF

### 1. ****Set Pagination Globally (Recommended)****

Edit your settings.py file:

# settings.py

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10, # Number of items per page

}

This automatically applies pagination to all list views in your API.

### 2. ****Use in ViewSets or Generic Views****

# views.py

from rest\_framework import viewsetsfrom .models import Bookfrom .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

queryset = Book.objects.all()

serializer\_class = BookSerializer

When you hit the /books/ endpoint, your JSON response will now look like this:

{

"count": 100,

"next": "http://localhost:8000/books/?page=2",

"previous": null,

"results": [

{

"id": 1,

"title": "Book 1",

...

},

...

]}

### 3. ****Other Pagination Options in DRF****

You can also use these classes instead of PageNumberPagination:

| **Pagination Class** | **Description** |
| --- | --- |
| LimitOffsetPagination | Use limit/offset params (?limit=5&offset=10) |
| CursorPagination | More secure, stable, and efficient for large data sets |

**Example with** LimitOffsetPagination**:**

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

'PAGE\_SIZE': 5,

}

**Lab:**



Implement pagination in a Django REST API for fetching doctor profiles.

**Practical Example:** 7) Write a Django API that returns paginated results for a list of doctors.8. Settings Configuration in Django

**Theory:**

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

| **Setting Type** | **Configuration** |
| --- | --- |

|  |  |
| --- | --- |
| Database | DATABASES in settings.py |

|  |  |
| --- | --- |
| Static Files | STATIC\_URL, STATICFILES\_DIRS, STATIC\_ROOT |

|  |  |
| --- | --- |
| API Keys & Secrets | Use .env + python-decouple |

**Lab:**



Modify settings.py to connect Django to a MySQL or SQLite database.

**Practical Example:** 8) Write a Django project that connects to an SQLite database and stores

doctor profiles.

9. Project Setup

**Theory:**

Setting up a Django REST Framework project.

**Lab:**



Create a new Django project and app for managing doctor profiles.

**Practical Example:** 9) Write a Django project to set up a new app called doctor\_finder

and create models, serializers, and views.

10. Social Authentication, Email, and OTP Sending API

**Theory:**



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



Sending emails and OTPs using third-party APIslike Twilio, SendGrid.

**Lab:**



Add Google login to a Django project using django-allauth.

**Practical Example:** 10) Write a Django project that integrates Google login and sends OTPs

to users using Twilio.11. RESTful API Design

**Theory:**



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

operations.

**Lab:**



Design a REST API for managing doctor profiles using Django REST Framework.

**Practical Example:** 11) Write a Django REST API with endpoints for creating, reading,

updating, and deleting doctors.

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

**Theory:**



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

**Lab:**



Implement a CRUD API using Django REST Framework for doctor profiles.

**Practical Example:** 12) Write a Django project that allows users to create, read, update, and

delete doctor profiles using API endpoints.

13. Authentication and Authorization API

**Theory:**



Difference between authentication and authorization.



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

**Lab:**



Implement user login, logout, and registration APIs in a Django project.

**Practical Example:** 13) Write a Django project that uses token-based authentication for

users and restricts access to certain API endpoints.14. OpenWeatherMap API Integration

**Theory:**



Introduction to OpenWeatherMap API and how to retrieve weather data.

**Lab:**



Create a Django project that fetches weather data for a given location.

**Practical Example:** 14) Write a Django project to fetch current weather data for a location

using the OpenWeatherMap API.

15. Google Maps Geocoding API

**Theory:**



Using Google Maps Geocoding API to convert addresses into coordinates.

**Lab:**



Create a Django project that takes an address as input and returns the latitude and longitude.

**Practical Example:** 15) Write a Django project that uses Google Maps API to find the

coordinates of a given address.

16. GitHub API Integration

**Theory:**



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

**Lab:**



Use GitHub API to create a repository and retrieve user data.

**Practical Example:** 16) Write a Django project that interacts with the GitHub API to create a

new repository and list all repositories for a given user.17. Twitter API Integration

**Theory:**



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

**Lab:**



Create a Django project that fetches recent tweets of a specific user.

**Practical Example:** 17) Write a Django project to fetch and display the latest 5 tweets from a

Twitter user using the Twitter API.

18. REST Countries API Integration

**Theory:**



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

**Lab:**



Use REST Countries API to fetch data for a specific country.

**Practical Example:** 18) Write a Django project that displays details (population, language,

currency) of a country entered by the user using the REST Countries API.

19. Email Sending APIs (SendGrid, Mailchimp)

**Theory:**



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

**Lab:**



Implement email sending functionality in a Django project using SendGrid.

**Practical Example:** 19) Write a Django project to send a confirmation email to a user using

the SendGrid API after successful registration.20. SMS Sending APIs (Twilio)

**Theory:**



Introduction to Twilio API forsending SMS and OTPs.

**Lab:**



Use Twilio API to send OTP to a user’s phone.

**Practical Example:** 20) Write a Django project that sends an OTP to the user's mobile

number during registration using Twilio API.

21. Payment Integration (PayPal, Stripe)

**Theory:**



Introduction to integrating payment gateways like PayPal and Stripe.

**Lab:**



Add Stripe payment functionality to a Django project.

**Practical Example:** 21) Write a Django project to allow users to make payments via Stripe

for booking doctor appointments.

22. Google Maps API Integration

**Theory:**



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

**Lab:**



Use Google Maps API to display doctor locations on a map.

**Practical Example:** 23) Write a Django project that integrates Google Maps API to show doctor locations in a specific city.