1. What is REST API? State different REST API request methods.

REST API (Representational State Transfer API): A REST API is an architectural style for building web services that allow communication between a client and a server. It uses standard HTTP methods and is stateless, meaning that each request from the client to the server must contain all the necessary information to understand and process the request. REST APIs are widely used for creating web services because they are simple, scalable, and flexible.

Core Principles of REST:

- **Stateless**: Each request from the client must contain all the information the server needs to fulfill it.
- **Client-Server**: The client and server are separate entities, and communication is done over HTTP.
- **Uniform Interface**: A consistent way of interacting with resources through a set of standard HTTP methods.
- **Resource-Based**: REST APIs use URLs (Uniform Resource Identifiers) to identify resources.

Different REST API Request Methods:

1. **GET**: Retrieves data from the server. It does not modify any data.
   - Example: GET /api/users/ - Fetches a list of users.
2. **POST**: Sends data to the server to create a new resource.
   - Example: POST /api/users/ - Creates a new user.

3. **PUT**: Sends data to the server to update an existing resource. It replaces the entire resource.
   - Example: PUT /api/users/1/ - Updates the user with ID 1.
4. **PATCH**: Similar to PUT, but only updates specific fields of an existing resource, rather than replacing the entire resource.
   - Example: PATCH /api/users/1/ - Updates specific fields of the user with ID 1.
5. **DELETE**: Deletes a resource from the server.
   - Example: DELETE /api/users/1/ - Deletes the user with ID 1.
6. **OPTIONS**: Describes the communication options for the target resource. It is often used to check supported methods.
   - Example: OPTIONS /api/users/ - Returns which methods (GET, POST, etc.) are allowed for /api/users/.

---

2. What is Django Rest Framework?

Django Rest Framework (DRF) is a powerful toolkit for building Web APIs in Django. It provides features such as serialization, authentication, permissions, and view handling, making it easier to build RESTful APIs on top of Django's models and views.

Key Features of Django Rest Framework (DRF):

1. **Serialization**: Converts complex data types (like Django models or querysets) into native Python data types (like dictionaries) that can then be rendered as JSON or XML.
2. **Authentication**: DRF supports several authentication methods like token-based authentication, OAuth, and session authentication.
3. **Permissions**: DRF includes a flexible permission system to control who can access the API.
4. **ViewSets & Routers**: DRF provides powerful abstractions for defining views, such as ModelViewSet, and automatically generates URLs using routers.
5. **Browsable API**: DRF provides a built-in web interface for interacting with the API, making it easier for developers to test and explore the API.

Example of a Simple API View in DRF:

from rest_framework.views import APIView from rest_framework.response import Response from rest_framework import status
class HelloWorld(APIView): def get(self, request): return Response({"message": "Hello, World!"}, status=status.HTTP_200_OK)

---

3. What are Serializers and De-serializers in Django Rest Framework?

Serializers in DRF: Serializers in Django Rest Framework are used to convert complex data types like Django models or querysets into Python data types (like dictionaries) that can easily be rendered into JSON, XML, or other content types. They are also used to validate incoming data when creating or updating resources.

Key Functions of Serializers:

- **Serialization**: Converting complex objects into native Python datatypes.
- **Deserialization**: Converting incoming data (like JSON) into Python data types or model instances.

Example of a Serializer:

from rest_framework import serializers from .models import User class UserSerializer(serializers.ModelSerializer): class Meta: model = User fields = ['id', 'name', 'email']

In this example, UserSerializer converts User model instances to Python dictionaries and vice versa.

De-serialization in DRF: De-serialization is the process of taking data from an external format (like JSON) and converting it into Python objects or Django model instances. When data is received via a POST or PUT request, it is typically de-serialized to create or update a model instance.

Example of De-serialization (Creating a new User from JSON data):

from rest_framework.parsers import JSONParser from .models import User from .serializers import UserSerializer def create_user(request): data = JSONParser().parse(request) serializer = UserSerializer(data=data) if serializer.is_valid(): serializer.save() # Saves the new user to the

database return Response(serializer.data, status=status.HTTP_201_CREATED) return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

Summary:

- **Serializers** convert complex data (e.g., Django models) into Python native data types for easy rendering (like JSON).
- **De-serializers** convert incoming data (like JSON or XML) into Python data structures or Django model instances.

Both serialization and de-serialization are integral to handling incoming and outgoing data in Django Rest Framework APIs.