Name:- Vishvash Limbasiya

**Python - Rest Framework**

Q.1:- Using Django’s built-in form handling.

Ans:-

1. Defining Forms:

* Forms are defined as Python classes, typically within a forms.py file in your Django app.
* Each form field is defined as an instance of a Field class (e.g., CharField, IntegerField, EmailField).
* You can specify attributes like label, max\_length, required, and widget for each field.

2. Rendering Forms:

* In your Django template, you can render the form using template tags.
* Django automatically generates the HTML for the form, including labels, input fields, and error messages.

3. Processing Form Data:

* In your Django view, you can process form data submitted by the user.
* You create an instance of your form class, passing in the request.POST data.
* Django automatically validates the submitted data based on the field definitions.
* If the data is valid, you can access it through the form.cleaned\_data dictionary.
* If the data is invalid, Django re-renders the form with error messages.

Q.2:- Implementing Django’s authentication system (sign up, login, logout, password management).

Ans:-

**Django Sign Up and login with confirmation Email | Python**

1. Step 1: Create a New Django Project and App. ...
2. Step 2: Set Up Templates Folder. ...
3. Step 3: Configure settings.py. ...
4. Step 4: Configure Project URLs. ...
5. Step 5: Configure App URLs. ...
6. Step 6: Create User Registration Form. ...
7. Step 7: Define Views for Registration and Login.

Q.3:- Techniques for customizing the Django admin panel.

Ans:-

:- ModelAdmin Class:

* Create a ModelAdmin class to control how a model is displayed and managed:
  + list\_display: Specify which model fields to show in the list view.
  + list\_filter: Add filters to the list view based on field values.
  + search\_fields: Enable searching through model fields.
  + ordering: Control the default ordering of records in the list view.
  + readonly\_fields: Make certain fields read-only in the edit form.
  + fieldsets: Organize fields into logical groups in the edit form.
  + inlines: Display related models inline within the edit form (using TabularInline or StackedInline).
  + actions: Add custom actions for bulk operations on selected records.
  + form: Use a custom form for the "add" and "change" views.
  + save\_model: Override the save method to perform custom actions before or after saving.
  + get\_queryset: Annotate calculated fields to optimize queries.

Q.4:- Steps to push a Django project to GitHub.

Ans:-

git init

git add .

git commit -m "Initial commit"

git remote add origin https://github.com/vishvash3228/Final\_pro.git

git branch -M main

git push -u origin main

Q.5 What is Serialization?

Ans:-

Serialization is the process of converting an object's state into a byte stream, making it possible to store the object or transmit it over a network.

Q.6:- Converting Django QuerySets to JSON.

Ans:-

1. This is the most common and recommended way and Import the necessary modules.

from django.core.serializers import serialize  
 from django.http import JsonResponse

2. Return the JSON data in a JsonResponse:

return JsonResponse(json\_data, safe=False)

* **Foreign Keys and Related Objects:**

When serializing models with foreign keys, the default serialize() function might not include all the data from related models. You may need to use use\_natural\_foreign\_keys=True or customize the output.

* **Large QuerySets:**

For very large QuerySets, consider using pagination to avoid performance issues.

* **Security:**

Be cautious when serializing sensitive data. Ensure that you only include the necessary fields in your JSON response.

* **Error Handling:**

Implement proper error handling to gracefully manage situations where serialization fails.

Q.7:- Using serializers in Django REST Framework (DRF).

Ans:-

**1. Define a Serializer**

In serializers.py:

from rest\_framework import serializers

from .models import MyModel

class MyModelSerializer(serializers.ModelSerializer):

class Meta:

model = MyModel

fields = '\_\_all\_\_'

**2. Use It in a View**

In views.py:

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

from .models import MyModel

from .serializers import MyModelSerializer

@api\_view(['GET'])

def mymodel\_list(request):

data = MyModel.objects.all()

serializer = MyModelSerializer(data, many=True)

return Response(serializer.data)

**Add URL**

In urls.py:

from django.urls import path

from .views import mymodel\_list

urlpatterns = [

path('api/mymodel/', mymodel\_list),

]

Q.8:- HTTP request methods (GET, POST, PUT, DELETE).

Ans:-

**HTTP Request Methods**

| **Method** | **Purpose** | **Typical Use Case** |
| --- | --- | --- |
| **GET** | Retrieve data from server | Fetch a webpage, get API data |
| **POST** | Send data to server to create | Submit forms, create new records |
| **PUT** | Update existing resource completely | Update a whole record |
| **DELETE** | Delete a resource on the server | Remove an item |

Q.9:- Sending and receiving responses in DRF.

Ans:-

* 1. Sending Responses:-

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

@api\_view(['GET'])

def example\_view(request):

data = {"message": "Hello, DRF!"}

return Response(data, status=200)

Q.10:- Understanding views in DRF: Function-based views vs Class-based views.

Ans:-

1. Function-Based Views (FBV):-

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

@api\_view(['GET'])

def hello(request):

return Response({"message": "Hello"})

2. Class-Based Views (CBV):-

from rest\_framework.views import APIView

from rest\_framework.response import Response

class HelloView(APIView):

def get(self, request):

return Response({"message": "Hello "})

Q.11:- Defining URLs and linking them to views.

Ans:-

1. **1. Understanding URLs and Views:**
   * **URLs (Uniform Resource Locators):** These are the addresses that users type into their web browsers or use to navigate between pages on a website.
   * **Views:** These are Python functions that handle incoming web requests and return responses. They typically process user input, retrieve data from databases, render templates, and send back HTTP responses.
   * **URL Routing:** This is the process of matching incoming URLs to specific view functions, ensuring the correct logic is executed for a given request.

Q.12:- Configuring Django settings for database, static files, and API keys.

Ans:-

**Database**

In settings.py:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql', # or 'sqlite3', 'mysql', etc.

'NAME': 'API',

'USER': 'root',

'PASSWORD': '’,

'HOST': 'localhost',

'PORT': '3306',

}

}

1. **Static Files:-**

STATIC\_URL = '/static/'

STATIC\_ROOT = BASE\_DIR / 'staticfiles'

STATICFILES\_DIRS = [BASE\_DIR / 'static'

Q.13:- Setting up a Django REST Framework project.

Ans:-

* **Python:** Ensure you have Python installed.
* **pip:** Python's package installer should also be installed.
* **Virtual Environment:** It's recommended to use a virtual environment to manage dependencies for your project.

Q.14:- What is CRUD, and why is it fundamental to backend development?

Ans:-

1. **Create:** Add new data (e.g., user signup, new blog post)

2. **Read:** Retrieve data (e.g., display user profile, list posts)

3.**Update:** Modify existing data (e.g., edit profile, update settings)

4. **Delete:** Remove data (e.g., delete account, remove comment)