Module 16) Python DB and Framework 1. HTML in Python Theory: • Introduction to embedding HTML within Python using web frameworks like Django or Flask. • Generating dynamic HTML content using Django templates.

Ans :

When we talk about *HTML in Python*, we’re usually referring to generating or serving HTML pages using **Python web frameworks** such as **Flask** or **Django**.

**Why do we need it?**

* Python code runs on the server.
* HTML is sent to the browser so the user can see and interact with the page.
* We combine them so that we can create **dynamic web pages** — pages that change based on data, user input, or database content.

**1. Flask Example**

Flask is a lightweight Python framework. You can send HTML to the browser in two ways:

1. **Directly returning HTML as a string**
2. **Using templates (recommended)**

**Direct HTML Example:**

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def home():

return "<h1>Welcome to My Website</h1><p>This is served by Python!</p>"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

Here, HTML is inside a Python string — not ideal for big pages.

**B) Generating Dynamic HTML using Django Templates**

Django has a **template system** that separates HTML from Python logic.

**How it works in Django:**

1. **View (Python code)** — Fetches data from the database.
2. **Template (HTML file)** — Displays the data using placeholders like {{ variable }}.
3. **Django Template Engine** — Combines them and sends HTML to the browser.

**Example:**

**views.py**

from django.shortcuts import render

def home(request):

context = {

"name": "Vinay",

"products": ["Laptop", "Mouse", "Keyboard"]

}

return render(request, "home.html", context)

**home.html**

<!DOCTYPE html>

<html>

<head>

<title>My Store</title>

</head>

<body>

<h1>Welcome, {{ name }}!</h1>

<h2>Available Products:</h2>

<ul>

{% for product in products %}

<li>{{ product }}</li>

{% endfor %}

</ul>

</body>

</html>

Lab: • Write a Python program to render an HTML file using Django’s template system

Ans:

#1) Create a project & app

pip install django

django-admin startproject mysite

cd mysite

python manage.py startapp web

#2) Enable the app & templates (mysite/settings.py)

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'web', # ← add this

]

# Tell Django where templates live (default already ok, but add DIRS if you want a global folder)

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [], # or ['BASE\_DIR / "templates"'] on Django 4.x+ with pathlib

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

#3) Create a template (web/templates/home.html)

Create folders: web/templates/ and then add home.html inside.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>{{ title }}</title>

</head>

<body>

<h1>Hello, {{ user\_name }} hello </h1>

<p>Today’s topics:</p>

<ul>

{% for t in topics %}

<li>{{ t }}</li>

{% empty %}

<li>No topics yet.</li>

{% endfor %}

</ul>

</body>

</html>

#4) Write a view that renders the template (web/views.py)

from django.shortcuts import render

def home(request):

context = {

"title": "Django Template Demo",

"user\_name": "Vinay",

"topics": ["Django Templates", "Context", "Tags & Filters"]

}

return render(request, "home.html", context)

#5) Map a URL to the view  
mysite/urls.py:

from django.contrib import admin

from django.urls import path

from web.views import home

urlpatterns = [

path('admin/', admin.site.urls),

path('', home, name='home'), # root URL -> template render

]

#6) Run the server

python manage.py migrate

python manage.py runserver

Practical Example: 1. Write a Django project that renders an HTML file displaying “Welcome to Doctor Finder” on the home page.

Ans:

**Step 1 – Create the Django project**

pip install django

django-admin startproject doctorfinder

cd doctorfinder

python manage.py startapp home

**Step 2 – Add the app to settings**

doctorfinder/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'home', # ← added

]

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [], # can leave empty, we’ll store templates in the app

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

**Step 3 – Create the HTML template**

Create folder home/templates/ inside the home app.  
Inside it, create index.html:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Doctor Finder</title>

</head>

<body>

<h1>Welcome to Doctor Finder</h1>

</body>

</html>

**Step 4 – Create the view**

home/views.py

from django.shortcuts import render

def home\_page(request):

return render(request, "index.html")

**Step 5 – Map the URL**

doctorfinder/urls.py

from django.contrib import admin

from django.urls import path

from home.views import home\_page

urlpatterns = [

path('admin/', admin.site.urls),

path('', home\_page, name='home'), # Root URL → home\_page view

]

**Step 6 – Run the server**

python manage.py migrate

python manage.py runserver

2. CSS in Python Theory: • Integrating CSS with Django templates. • How to serve static files (like CSS, JavaScript) in Django.

Ans:

**1. Integrating CSS with Django templates**

Django templates are just HTML files with template syntax.  
You can link a CSS file in your HTML like this:

{% load static %}

<link rel="stylesheet" href="{% static 'css/style.css' %}">

STATIC\_URL = '/static/'

# For extra global static directory (optional)

STATICFILES\_DIRS = [

BASE\_DIR / "static"

]

**c) Using CSS in Templates**

Example template:

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

</head>

<body>

<h1>Welcome to Doctor Finder</h1>

</body>

</html>

Lab: • Create a CSS file to style a basic HTML template in Django.

Ans:

Step -1

/\* style.css \*/

body {

font-family: Arial, sans-serif;

background-color: #f4f9ff;

margin: 0;

padding: 0;

text-align: center;

}

h1 {

color: #0077cc;

margin-top: 100px;

font-size: 2.5em;

}

footer {

margin-top: 50px;

color: #666;

font-size: 0.9em;

}

Step-2

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Doctor Finder</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

</head>

<body>

<h1>Welcome to Doctor Finder</h1>

<footer>© 2025 Doctor Finder</footer>

</body>

</html>

Step-3

STATIC\_URL = '/static/'

Step-4

python manage.py runserver

Practical Example: 2) Write a Django project to display a webpage with custom CSS styling for a doctor profile page.

Ans :

**Step 1 – Create the project & app**

pip install django

django-admin startproject doctorprofile

cd doctorprofile

python manage.py startapp profileapp

**Step 2 – Add app to settings**

doctorprofile/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'profileapp', # added

]

STATIC\_URL = '/static/'

**Step 3 – Create CSS**

profileapp/static/css/style.css

body {

font-family: Arial, sans-serif;

background-color: #f8faff;

margin: 0;

padding: 0;

}

.profile-card {

max-width: 400px;

margin: 80px auto;

background: #fff;

padding: 20px;

border-radius: 12px;

box-shadow: 0 4px 12px rgba(0,0,0,0.1);

text-align: center;

}

.profile-card img {

width: 120px;

height: 120px;

border-radius: 50%;

margin-bottom: 15px;

}

.profile-card h2 {

margin: 0;

color: #0077cc;

}

.profile-card p {

color: #555;

margin: 5px 0;

}

**Step 4 – Create the HTML template**

profileapp/templates/profile.html

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Doctor Profile</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

</head>

<body>

<div class="profile-card">

<img src="https://via.placeholder.com/120" alt="Doctor Photo">

<h2>Dr. Sarah Johnson</h2>

<p>Specialist: Cardiologist</p>

<p>Experience: 12 years</p>

<p>Location: New Delhi, India</p>

</div>

</body>

</html>

**Step 5 – Create the view**

profileapp/views.py

from django.shortcuts import render

def doctor\_profile(request):

return render(request, "profile.html")

**Step 5 – Create the view**

profileapp/views.py

from django.shortcuts import render

def doctor\_profile(request):

return render(request, "profile.html")

**Step 7 – Run the server**

python manage.py migrate

python manage.py runserver

3. JavaScript with Python Theory: • Using JavaScript for client-side interactivity in Django templates. • Linking external or internal JavaScript files in Djang

Ans:

**Python/Django** handles **server-side** processing — generating HTML, fetching data, managing authentication, etc.

**JavaScript** handles **client-side** interactivity — form validation, animations, dynamic content updates without reloading the page.

Together, they create a **full-stack** web application experience.

**2. Ways to use JavaScript in Django templates**

**a) Internal JavaScript**

You can write JS directly inside your template:

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

</head>

<body>

<h1 id="welcome">Welcome to Doctor Finder</h1>

<script>

document.getElementById("welcome").style.color = "blue";

alert("Hello from JavaScript!");

</script>

</body>

</html>

**script.js**

document.addEventListener("DOMContentLoaded", function() {

document.getElementById("welcome").style.color = "green";

});

**ndex.html**

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

<script src="{% static 'js/script.js' %}"></script>

</head>

<body>

<h1 id="welcome">Welcome to Doctor Finder</h1>

</body>

</html

**4. Passing data from Python to JavaScript**

Sometimes you need Django variables inside JavaScript:

<script>

var doctorName = "{{ doctor.name }}";

console.log("Doctor:", doctorName);

</script>

Lab: • Create a Django project with JavaScript-enabled form validation.

Ans :

**Step 1 – Create project & app**

pip install django

django-admin startproject doctorform

cd doctorform

python manage.py startapp register

**Step 2 – Add app to settings**

doctorform/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'register', # added

]

STATIC\_URL = '/static/'

**Step 3 – Create the HTML template**

register/templates/register\_form.html

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Doctor Registration</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

<script src="{% static 'js/validation.js' %}"></script>

</head>

<body>

<h1>Doctor Registration Form</h1>

<form id="doctorForm" method="POST" onsubmit="return validateForm()">

{% csrf\_token %}

<label>Name:</label>

<input type="text" id="name" name="name"><br><br>

<label>Email:</label>

<input type="email" id="email" name="email"><br><br>

<label>Specialization:</label>

<input type="text" id="specialization" name="specialization"><br><br>

<button type="submit">Register</button>

</form>

</body>

</html>

**Step 4 – Create JavaScript validation**

register/static/js/validation.js

function validateForm() {

let name = document.getElementById("name").value.trim();

let email = document.getElementById("email").value.trim();

let specialization = document.getElementById("specialization").value.trim();

if (name === "") {

alert("Name is required");

return false;

}

if (email === "" || !email.includes("@")) {

alert("Valid email is required");

return false;

}

if (specialization === "") {

alert("Specialization is required");

return false;

}

return true; // form will be submitted

}

**Step 5 – Create a view**

register/views.py

from django.shortcuts import render

from django.http import HttpResponse

def register\_doctor(request):

if request.method == "POST":

name = request.POST.get("name")

email = request.POST.get("email")

specialization = request.POST.get("specialization")

return HttpResponse(f"Doctor {name} ({specialization}) registered successfully!")

return render(request, "register\_form.html")

**Step 6 – URL mapping**

doctorform/urls.py

from django.contrib import admin

from django.urls import path

from register.views import register\_doctor

urlpatterns = [

path('admin/', admin.site.urls),

path('', register\_doctor, name='register-doctor'),

]

**Step 8 – Run the server**

python manage.py migrate

python manage.py runserver

Practical Example: 3) Write a Django project where JavaScript is used to validate a patient registration form on the client side

Ans :

**Step 1 – Create project & app**

pip install django

django-admin startproject patientproject

cd patientproject

python manage.py startapp patientapp

**Step 2 – Add app to settings**

patientproject/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'patientapp', # added

]

STATIC\_URL = '/static/'

**Step 3 – Create HTML template**

patientapp/templates/patient\_form.html

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Patient Registration</title>

<link rel="stylesheet" href="{% static 'css/style.css' %}">

<script src="{% static 'js/patient\_validation.js' %}"></script>

</head>

<body>

<h1>Patient Registration Form</h1>

<form id="patientForm" method="POST" onsubmit="return validatePatientForm()">

{% csrf\_token %}

<label>Full Name:</label>

<input type="text" id="name" name="name"><br><br>

<label>Age:</label>

<input type="number" id="age" name="age" min="0"><br><br>

<label>Email:</label>

<input type="email" id="email" name="email"><br><br>

<label>Phone:</label>

<input type="text" id="phone" name="phone"><br><br>

<label>Address:</label>

<textarea id="address" name="address"></textarea><br><br>

<button type="submit">Register</button>

</form>

</body>

</html>

**Step 4 – Create JavaScript validation**

patientapp/static/js/patient\_validation.js

function validatePatientForm() {

let name = document.getElementById("name").value.trim();

let age = document.getElementById("age").value.trim();

let email = document.getElementById("email").value.trim();

let phone = document.getElementById("phone").value.trim();

let address = document.getElementById("address").value.trim();

if (name === "" || name.length < 3) {

alert("Full name must be at least 3 characters long");

return false;

}

if (age === "" || isNaN(age) || age <= 0) {

alert("Please enter a valid age");

return false;

}

if (email === "" || !email.includes("@")) {

alert("Please enter a valid email address");

return false;

}

if (phone.length !== 10 || isNaN(phone)) {

alert("Phone number must be exactly 10 digits");

return false;

}

if (address === "" || address.length < 5) {

alert("Please enter a valid address");

return false;

}

return true; // All validations passed

}

**Step 5 – Create a view**

patientapp/views.py

from django.shortcuts import render

from django.http import HttpResponse

def register\_patient(request):

if request.method == "POST":

name = request.POST.get("name")

return HttpResponse(f"Patient '{name}' registered successfully!")

return render(request, "patient\_form.html")

**Step 6 – URL mapping**

patientproject/urls.py

from django.contrib import admin

from django.urls import path

from patientapp.views import register\_patient

urlpatterns = [

path('admin/', admin.site.urls),

path('', register\_patient, name='patient-register'),

]

**Step 7 – Optional CSS**

patientapp/static/css/style.css

body {

font-family: Arial, sans-serif;

background-color: #f4f9ff;

padding: 20px;

}

h1 {

text-align: center;

color: #0077cc;

}

form {

max-width: 400px;

margin: auto;

background: white;

padding: 20px;

border-radius: 8px;

box-shadow: 0px 4px 10px rgba(0,0,0,0.1);

}

label {

font-weight: bold;

}

input, textarea, button {

width: 100%;

padding: 8px;

margin-top: 5px;

margin-bottom: 15px;

}

**Step 8 – Run the project**

python manage.py migrate

python manage.py runserver

4. Django Introduction Theory: • Overview of Django: Web development framework. • Advantages of Django (e.g.,scalability, security). • Django vs. Flask comparison: Which to choose and why.

Ans :

**1. Overview of Django**

* **Django** is a **high-level Python web framework** that encourages **rapid development** and **clean, pragmatic design**.
* It follows the **MVT (Model–View–Template)** architectural pattern:
  + **Model** → Handles database structure and data.
  + **View** → Handles business logic and returns responses.
  + **Template** → Handles presentation (HTML, CSS, JS).
* Created in 2005, maintained by the Django Software Foundation.
* Focuses on **"Don’t Repeat Yourself" (DRY)** and **"Convention over Configuration"** principles.
* Comes with **built-in features** for authentication, ORM, admin panel, form handling, and more.

**2. Advantages of Django**

 **Scalability** → Used by Instagram, Pinterest, and other large-scale sites.

 **Security** → Protects against common attacks (SQL injection, XSS, CSRF, clickjacking).

 **Rapid Development** → Built-in admin panel, ORM, form handling, and URL routing speed up work.

 **Versatile** → Can build anything from blogs to APIs to enterprise-level applications.

 **Community Support** → Huge ecosystem of plugins, libraries, and documentation.

 **Batteries Included** → Comes with most of what you need, without requiring third-party tools for basics

**3. Django vs. Flask Comparison**

| **Feature** | **Django** | **Flask** |
| --- | --- | --- |
| **Type** | Full-stack framework | Micro-framework |
| **Architecture** | MVT (Model–View–Template) | No default structure (flexible) |
| **Built-in Features** | ORM, admin panel, authentication, sessions | Minimal, need to add via extensions |
| **Scalability** | High — good for large applications | Good for small-to-medium apps |
| **Learning Curve** | Steeper (more features to learn) | Easier for beginners |
| **Flexibility** | More opinionated — enforces structure | Highly flexible, developer decides |
| **Use Case** | Enterprise apps, large projects | Small projects, quick prototypes |

* **Choose Django if**:
  + You want a **full-featured, ready-to-use framework**.
  + You’re building **large, complex, or long-term projects**.
  + You want built-in **security, admin, and ORM** without installing many extra packages.
* **Choose Flask if**:
  + You want **full control** over architecture.
  + You’re building a **small project or prototype** quickly.
  + You prefer adding components yourself rather than using built-in ones.

Lab: • Write a short project using Django’s built-in tools to render a simple webpage.

Ans :

**Step 1 – Create project & app**

pip install django

django-admin startproject hellodjango

cd hellodjango

python manage.py startapp home

**Step 2 – Add app to settings**

hellodjango/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'home', # our app

]

**Step 3 – Create template**

home/templates/index.html

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello Django</title>

</head>

<body>

<h1>Hello, Django World!</h1>

</body>

</html>

**Step 4 – Create the view**

home/views.py

from django.shortcuts import render

def home\_page(request):

return render(request, "index.html")

**Step 5 – Map the URL**

hellodjango/urls.py

from django.contrib import admin

from django.urls import path

from home.views import home\_page

urlpatterns = [

path('admin/', admin.site.urls),

path('', home\_page, name='home'),

]

**Step 6 – Run the server**

python manage.py migrate

python manage.py runserver

Practical Example: 4) Write a Python program to create a Django project and understand its directory structure.

Ans :

**Step 1 – Install Django**

pip install django

**Step 2 – Create a Django project**

django-admin startproject myproject

**Step 3 – Understanding the directory structure**

**Top level**

* **manage.py** → Command-line utility to runserver, migrate, create apps, etc.

**Inner myproject/ folder**

1. **\_\_init\_\_.py**
   * Makes this folder a Python package.
2. **settings.py**
   * Main configuration file:
     + Installed apps
     + Database settings
     + Static files
     + Middleware
3. **urls.py**
   * URL dispatcher (maps URLs to views).
4. **asgi.py**
   * Entry point for ASGI-compatible web servers (for async support).
5. **wsgi.py**
   * Entry point for WSGI-compatible web servers (traditional deployment).

**Step 4 – Run the project**

cd myproject

python manage.py runserver

**Step 5 – Create an app**

python manage.py startapp home

Now you’ll see:

home/

admin.py

apps.py

models.py

tests.py

views.py

migrations/

\_\_init\_\_.py

* **Project** → The main configuration and settings for your site.
* **App** → A specific module or feature inside your project.
* You can have **many apps** inside one Django project.

5. Virtual Environment Theory: • Understanding the importance of a virtual environment in Python projects. • Using venv or virtualenv to create isolated environments.

Ans:

**1. Importance of a Virtual Environment**

A **virtual environment** is an isolated space where you can install Python packages without affecting the **global Python installation** on your computer.

**Why it’s important:**

1. **Dependency Isolation** → Different projects can have different versions of the same package without conflicts.
2. **Avoid Permission Issues** → No need for admin/root rights to install packages.
3. **Reproducibility** → The environment can be shared with others using a requirements.txt file.
4. **Project Portability** → You can easily replicate the exact environment on another system.

**Example Problem Without Virtual Env:**  
Project A needs Django 4.2, Project B needs Django 3.2 — installing one globally will break the other.

**2. Using venv (built-in)**

venv comes with Python 3.3+.

**Create a virtual environment:**

python -m venv myenv

**3. Using virtualenv (third-party)**

If you want more features or Python 2 support:

pip install virtualenv

virtualenv myenv

**4. Installing Packages Inside Virtual Env**

Once activated:

pip install django

Packages will be installed only in myenv/ and won’t affect global Python.

**5. Saving and Sharing Environment**

**Export dependencies:**

pip freeze > requirements.txt

**Recreate environment:**

pip install -r requirements.txt

Lab: • Set up a virtual environment for a Django project.

Ans :

**1. Create a project folder**

mkdir doctor\_finder

cd doctor\_finder

**2. Create a virtual environment**

python -m venv env

This will create:

doctor\_finder/

env/

Scripts/ (Windows) or bin/ (Mac/Linux)

**3. Activate the virtual environment**

* **Windows (cmd):**
* env\Scripts\activate
* **Windows (PowerShell):**
* env\Scripts\Activate.ps1
* **Mac/Linux:**
* source env/bin/activate

You’ll see (env) at the start of your terminal prompt, meaning it’s active.

**4. Install Django inside the virtual environment**

pip install django

**5. Verify Django installation**

django-admin --version

**6. Create a new Django project**

django-admin startproject doctor\_finder .

**7. Run the development server**

python manage.py runserver

Practical Example: 5) Write a Python program to create and activate a virtual environment, then install Django in it.

Ans :

import os

import subprocess

import sys

# Step 1: Create virtual environment

env\_name = "env"

print(f"Creating virtual environment '{env\_name}'...")

subprocess.run([sys.executable, "-m", "venv", env\_name])

# Step 2: Install Django inside the virtual environment

# On Windows: env\Scripts\pip, On Mac/Linux: env/bin/pip

pip\_path = os.path.join(env\_name, "Scripts", "pip") if os.name == "nt" else os.path.join(env\_name, "bin", "pip")

print("Installing Django...")

subprocess.run([pip\_path, "install", "django"])

# Step 3: Show installed Django version

print("Checking Django version...")

subprocess.run([pip\_path, "show", "django"])

print("\n✅ Virtual environment created and Django installed successfully.")

print(f"To activate it, run:\n Windows: {env\_name}\\Scripts\\activate\n Mac/Linux: source {env\_name}/bin/activate")

1. Project and App Creation

Theory: • Steps to create a Django project and individual apps within the project. • Understanding the role of manage.py, urls.py, and views.py

Ans :

**1. Steps to Create a Django Project and Apps**

**A. Create a Django Project**

1. **Install Django**
2. pip install django
3. **Start a project**

django-admin startproject projectname

**Run development server**

python manage.py runserver

**B. Create an App Inside the Project**

1. **Create app**

python manage.py startapp appname

1. **Register the app** in settings.py → INSTALLED\_APPS list:
2. INSTALLED\_APPS = [
3. 'appname',
4. ...
5. ]

**2. Role of Key Django Files**

**A. manage.py**

* Command-line utility for interacting with your project.
* Lets you run commands like:
  + runserver → Start development server.
  + makemigrations → Create migration files for DB changes.
  + migrate → Apply migrations to DB.
  + startapp → Create new app.

**B. urls.py**

* Stores **URL patterns** that map a web address to a specific view.
* Acts as the “routing system” of Django.
* Example:
* from django.urls import path
* from . import views
* urlpatterns = [
* path('', views.home, name='home'),
* ]

**C. views.py**

* Contains functions or classes (**views**) that handle requests and return responses (HTML, JSON, etc.).
* Example:
* from django.http import HttpResponse
* def home(request):

return HttpResponse("Welcome to Doctor Finder")

Lab: • Create a Django project with an app to manage doctor profiles.

**Lab: Django Project – Doctor Profiles**

**1. Create a Virtual Environment (optional but recommended)**

python -m venv env

env\Scripts\activate # Windows

# OR

source env/bin/activate # Mac/Linux

**2. Install Django**

pip install django

**3. Create a Django Project**

django-admin startproject doctor\_finder

cd doctor\_finder

**4. Create an App for Doctor Profiles**

python manage.py startapp doctor\_profiles

**5. Register the App in settings.py**

Open doctor\_finder/settings.py and add:

INSTALLED\_APPS = [

'doctor\_profiles',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

**6. Define a Model for Doctor Profiles (doctor\_profiles/models.py)**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.PositiveIntegerField(help\_text="Years of experience")

contact = models.CharField(max\_length=15)

def \_\_str\_\_(self):

return self.name

**7. Create and Apply Migrations**

python manage.py makemigrations

python manage.py migrate

**8. Create a View (doctor\_profiles/views.py)**

from django.http import HttpResponse

def home(request):

return HttpResponse("<h1>Welcome to Doctor Finder</h1><p>Manage Doctor Profiles here.</p>")

**9. Configure URLs**

* **In doctor\_finder/urls.py:**

from django.contrib import admin

from django.urls import path

from doctor\_profiles import views

urlpatterns = [

path('admin/', admin.site.urls),

path('', views.home, name='home'),

]

**10. Run the Server**

python manage.py runserver

Practical Example: 6) Write a Python program to create a Django project and a new app within the project called doctor.

Ans:

import subprocess

import os

import sys

# Project and App names

project\_name = "myproject"

app\_name = "doctor"

# Step 1: Install Django (if not already installed)

print("Installing Django...")

subprocess.run([sys.executable, "-m", "pip", "install", "django"])

# Step 2: Create Django project

print(f"Creating Django project '{project\_name}'...")

subprocess.run(["django-admin", "startproject", project\_name])

# Step 3: Navigate into project folder

os.chdir(project\_name)

# Step 4: Create Django app

print(f"Creating Django app '{app\_name}'...")

subprocess.run([sys.executable, "manage.py", "startapp", app\_name])

print("\n✅ Django project and app created successfully!")

print(f"Project directory structure:\n")

for root, dirs, files in os.walk(".", topdown=True):

level = root.replace(".", "").count(os.sep)

indent = " " \* 4 \* level

print(f"{indent}{os.path.basename(root)}/")

sub\_indent = " " \* 4 \* (level + 1)

for f in files:

print(f"{sub\_indent}{f}")

print("\nNext steps:")

print(f"1. Open {project\_name}/settings.py and add '{app\_name}' to INSTALLED\_APPS.")

print("2. Run: python manage.py runserver")

1. Run it:
2. python create\_django\_doctor\_app.py
3. It will:
   * Install Django (if missing).
   * Create a project named myproject.
   * Create an app named doctor.
   * Show you the generated folder structure.

7. MVT Pattern Architecture Theory: • Django’s MVT (Model-View-Template) architecture and how it handles request-response cycles.

Ans :

**7. MVT Pattern Architecture in Django**

**What is MVT?**

Django follows the **MVT** (Model–View–Template) design pattern, which is a variation of MVC (Model–View–Controller) tailored for web development.

**1. Components of MVT**

1. **Model**
   * Represents the **data layer**.
   * Defines the structure of your database tables using Python classes.
   * Handles all database operations (create, read, update, delete).
   * Example:
   * class Doctor(models.Model):
   * name = models.CharField(max\_length=100)
   * specialization = models.CharField(max\_length=100)
2. **View**
   * Acts as the **business logic layer**.
   * Handles HTTP requests, interacts with the Model, and returns a response.
   * Usually returns an **HTML page** using templates or JSON data for APIs.
   * Example:
   * def home(request):
   * return render(request, 'home.html')
3. **Template**
   * Represents the **presentation layer**.
   * HTML files that define how data is displayed to the user.
   * Can include **Django template language** for dynamic content.
   * Example:
   * <h1>Welcome, {{ doctor.name }}</h1>

**2. Request-Response Cycle in Django (MVT Flow)**

1. **User sends a request** (e.g., GET /doctors).
2. **URL Dispatcher** matches the request to a specific view function (via urls.py).
3. **View function** processes the request:
   * Fetches data from the **Model** if needed.
   * Passes data to the **Template**.
4. **Template** renders HTML with the provided data.
5. **Response** is sent back to the user's browser.

**3. MVT vs MVC**

* In MVC, the **Controller** handles request flow.
* In MVT, Django’s **View** acts like the Controller, and the Template takes care of the presentation.
* Django automates much of the “controller” work internally.

Lab: • Build a simple Django app showcasing how the MVT architecture works.

Ans :

**1. Create a Django Project and App**

In your terminal:

django-admin startproject mvt\_demo

cd mvt\_demo

python manage.py startapp doctor

**2. Define the Model (Model)**

**doctor/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

def \_\_str\_\_(self):

return self.name

**3. Register the Model in Admin**

**doctor/admin.py**

from django.contrib import admin

from .models import Doctor

admin.site.register(Doctor)

**4. Add App to Settings**

**mvt\_demo/settings.py**

INSTALLED\_APPS = [

# Default apps...

'doctor',

]

**5. Create the View (View)**

**doctor/views.py**

from django.shortcuts import render

from .models import Doctor

def doctor\_list(request):

doctors = Doctor.objects.all()

return render(request, 'doctor\_list.html', {'doctors': doctors})

**6. Create the URL Mapping**

**mvt\_demo/urls.py**

from django.contrib import admin

from django.urls import path

from doctor import views

urlpatterns = [

path('admin/', admin.site.urls),

path('doctors/', views.doctor\_list, name='doctor\_list'),

]

**7. Create the Template (Template)**

Make a folder **doctor/templates/** and inside it create **doctor\_list.html**:

<!DOCTYPE html>

<html>

<head>

<title>Doctor List</title>

</head>

<body>

<h1>List of Doctors</h1>

<ul>

{% for doctor in doctors %}

<li>{{ doctor.name }} - {{ doctor.specialization }}</li>

{% empty %}

<li>No doctors available</li>

{% endfor %}

</ul>

</body>

</html>

**8. Apply Migrations and Add Data**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

Practical Example: 7) Write a Django project with models, views, and templates to display doctor information.

Ans:

**1. Create Project & App**

django-admin startproject doctor\_info

cd doctor\_info

python manage.py startapp doctor

**2. Define the Model (Model)**

**doctor/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.PositiveIntegerField(help\_text="Experience in years")

contact = models.CharField(max\_length=20)

def \_\_str\_\_(self):

return self.name

**3. Register the Model in Admin**

**doctor/admin.py**

from django.contrib import admin

from .models import Doctor

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ('name', 'specialization', 'experience', 'contact')

**4. Add App to Settings**

**doctor\_info/settings.py**

INSTALLED\_APPS = [

# Default Django apps...

'doctor',

]

**5. Create the View (View)**

**doctor/views.py**

from django.shortcuts import render

from .models import Doctor

def doctor\_list(request):

doctors = Doctor.objects.all()

return render(request, 'doctor\_list.html', {'doctors': doctors})

**6. URL Configuration**

**doctor\_info/urls.py**

from django.contrib import admin

from django.urls import path

from doctor import views

urlpatterns = [

path('admin/', admin.site.urls),

path('doctors/', views.doctor\_list, name='doctor\_list'),

]

**7. Create the Template (Template)**

Create folder: **doctor/templates/** and file **doctor\_list.html**:

<!DOCTYPE html>

<html>

<head>

<title>Doctor Information</title>

<style>

body { font-family: Arial, sans-serif; margin: 20px; }

table { border-collapse: collapse; width: 70%; }

th, td { border: 1px solid #333; padding: 8px; text-align: left; }

th { background-color: #f4f4f4; }

</style>

</head>

<body>

<h1>Available Doctors</h1>

<table>

<tr>

<th>Name</th>

<th>Specialization</th>

<th>Experience (Years)</th>

<th>Contact</th>

</tr>

{% for doctor in doctors %}

<tr>

<td>{{ doctor.name }}</td>

<td>{{ doctor.specialization }}</td>

<td>{{ doctor.experience }}</td>

<td>{{ doctor.contact }}</td>

</tr>

{% empty %}

<tr>

<td colspan="4">No doctors found.</td>

</tr>

{% endfor %}

</table>

</body>

</html>

**8. Run Migrations & Add Data**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

8. Django Admin Panel Theory: • Introduction to Django’s built-in admin panel. • Customizing the Django admin interface to manage database records.

Ans :

**1. Introduction**

* The **Django Admin Panel** is a **built-in interface** that allows developers and administrators to **manage database records** without writing SQL queries.
* It’s automatically generated from your Django models.
* Accessible at the /admin/ URL after enabling it in your project.

**2. Key Features**

* **Create, read, update, delete (CRUD)** operations on models.
* Search and filter database records.
* Inline editing for related models.
* Permission management for different admin users.

**3. Enabling the Admin Panel**

1. Ensure 'django.contrib.admin' is included in **INSTALLED\_APPS** in settings.py.
2. Run database migrations:
3. python manage.py migrate
4. Create a superuser (admin account):
5. python manage.py createsuperuser
6. Start the server and log in at:
7. http://127.0.0.1:8000/admin/

**4. Registering Models**

* Models must be **registered** to appear in the admin panel.
* Example:
* from django.contrib import admin
* from .models import Doctor
* admin.site.register(Doctor)

**5. Customizing the Admin Panel**

You can customize:

* **list\_display** → Columns shown in the list view.
* **search\_fields** → Searchable fields.
* **list\_filter** → Add filters by field.
* **ordering** → Default sort order.

**Example:**

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ('name', 'specialization', 'experience')

search\_fields = ('name', 'specialization')

list\_filter = ('specialization',)

ordering = ('name',)

**6. Advantages**

* No need to build a separate admin interface.
* Saves time in database management.
* Highly customizable for large projects.
* Secure and permission-based.

Lab: • Set up and customize the Django admin panel to manage a "Doctor Finder" project

Ans :

**1. Create a Django Project & App**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctor

**2. Define the Model**

**doctor/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.PositiveIntegerField(help\_text="Experience in years")

contact = models.CharField(max\_length=20)

def \_\_str\_\_(self):

return self.name

**3. Add App to Installed Apps**

**doctor\_finder/settings.py**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'doctor', # our app

]

**4. Run Migrations**

python manage.py makemigrations

python manage.py migrate

**5. Create a Superuser**

python manage.py createsuperuser

* Enter **username**, **email** (optional), and **password**.

**6. Register Model in Admin**

**doctor/admin.py**

from django.contrib import admin

from .models import Doctor

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ('name', 'specialization', 'experience', 'contact')

search\_fields = ('name', 'specialization')

list\_filter = ('specialization',)

ordering = ('name',)

**7. Run the Server**

python manage.py runserver

Practical Example: 8) Write a Django project to create an admin panel and add custom fields for managing doctor information.

Ans :

**Step 1: Create Django Project & App**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctor

**Step 2: Define the Doctor Model**

**doctor/models.py**

from django.db import models

class Doctor(models.Model):

GENDER\_CHOICES = [

('M', 'Male'),

('F', 'Female'),

('O', 'Other'),

]

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.PositiveIntegerField(help\_text="Experience in years")

contact = models.CharField(max\_length=20)

email = models.EmailField(unique=True)

gender = models.CharField(max\_length=1, choices=GENDER\_CHOICES)

available\_days = models.CharField(max\_length=100, help\_text="e.g. Mon, Wed, Fri")

fee = models.DecimalField(max\_digits=6, decimal\_places=2)

def \_\_str\_\_(self):

return f"{self.name} - {self.specialization}"

**Step 3: Add App to Installed Apps**

**doctor\_finder/settings.py**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'doctor',

]

**Step 4: Run Migrations**

python manage.py makemigrations

python manage.py migrate

**Step 5: Create a Superuser**

python manage.py createsuperuser

* Enter **username**, **email**, and **password**.

**Step 6: Customize the Admin Panel**

**doctor/admin.py**

from django.contrib import admin

from .models import Doctor

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ('name', 'specialization', 'experience', 'fee', 'available\_days')

list\_filter = ('specialization', 'gender', 'available\_days')

search\_fields = ('name', 'specialization', 'email')

ordering = ('name',)

fieldsets = (

('Personal Information', {

'fields': ('name', 'gender', 'email', 'contact')

}),

('Professional Details', {

'fields': ('specialization', 'experience', 'available\_days', 'fee')

}),

)

**Step 7: Run the Server**

python manage.py runserver

9. URL Patterns and Template Integration Theory: • Setting up URL patterns in urls.py for routing requests to views. • Integrating templates with views to render dynamic HTML content.

Ans :

**1. URL Patterns in Django**

* Django uses a **URL dispatcher** to map incoming requests to the correct **view function**.
* URL patterns are defined in **urls.py** using Python lists of path() or re\_path() functions.
* Syntax:

from django.urls import path

from . import views

urlpatterns = [

path('home/', views.home, name='home'),

]

* **Components of path()**:
  + route: URL string (e.g., "home/")
  + view: Function/class to handle the request
  + name: (Optional) Name for referencing this URL in templates

**2. How Django Processes a Request**

1. Browser sends a request (e.g., /home/).
2. Django checks the **root urls.py** for a matching pattern.
3. If matched, it calls the assigned **view function**.
4. The view function:
   * Processes data
   * Loads a **template**
   * Returns an **HTTP Response**

**3. Template Integration**

* Templates in Django are **HTML files** stored inside a **templates** folder.
* They can contain **Django Template Language (DTL)** to display dynamic content.
* Example template:

<!-- templates/home.html -->

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

</head>

<body>

<h1>Welcome, {{ user\_name }}!</h1>

</body>

</html>

**4. Linking Templates with Views**

* Use **render()** to load a template and pass data.

from django.shortcuts import render

def home(request):

context = {'user\_name': 'Vinay'}

return render(request, 'home.html', context)

* context is a Python dictionary where:
  + **Key** → variable name in template
  + **Value** → data to display

**5. Template Directory Setup**

* In **settings.py**, ensure:

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [BASE\_DIR / "templates"], # global templates folder

'APP\_DIRS': True,

},

]

* You can store templates **globally** or **inside each app**.

**6. Example Workflow**

* **URL** → /doctor/
* **urls.py**:

path('doctor/', views.doctor\_profile, name='doctor\_profile')

* **views.py**:

def doctor\_profile(request):

data = {'name': 'Dr. John', 'specialization': 'Cardiology'}

return render(request, 'doctor\_profile.html', data)

* **doctor\_profile.html**:

<h2>{{ name }} - {{ specialization }}</h2>

Lab: • Create a Django project with URL patterns and corresponding views and templates.

Ans :

**1. Create the Django Project & App**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctor

**2. Add App to Installed Apps**

**doctor\_finder/settings.py**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'doctor', # our app

]

**3. Create View Functions**

**doctor/views.py**

from django.shortcuts import render

def home(request):

return render(request, 'home.html', {'title': 'Welcome to Doctor Finder'})

def doctor\_profile(request):

doctor\_data = {

'name': 'Dr. John Smith',

'specialization': 'Cardiologist',

'experience': 12

}

return render(request, 'doctor\_profile.html', doctor\_data)

**4. Set Up URL Patterns**

**doctor/urls.py**

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'),

path('profile/', views.doctor\_profile, name='doctor\_profile'),

]

**doctor\_finder/urls.py**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

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

]

**5. Create Templates**

**Folder structure**:

doctor/

templates/

home.html

doctor\_profile.html

**home.html**

<!DOCTYPE html>

<html>

<head>

<title>{{ title }}</title>

</head>

<body>

<h1>{{ title }}</h1>

<a href="{% url 'doctor\_profile' %}">View Doctor Profile</a>

</body>

</html>

**doctor\_profile.html**

<!DOCTYPE html>

<html>

<head>

<title>Doctor Profile</title>

</head>

<body>

<h1>{{ name }}</h1>

<p>Specialization: {{ specialization }}</p>

<p>Experience: {{ experience }} years</p>

<a href="{% url 'home' %}">Back to Home</a>

</body>

</html>

**6. Run the Server**

python manage.py runserver

Practical Example: 9) Write a Django project where URL routing is used to navigate between different pages of a “Doctor Finder” site (home, profile, contact).

Ans :

**1. Create Project & App**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctor

**2. Add App in Settings**

**doctor\_finder/settings.py**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'doctor', # our app

]

**3. Create Views**

**doctor/views.py**

from django.shortcuts import render

def home(request):

return render(request, 'home.html', {'title': 'Welcome to Doctor Finder'})

def profile(request):

doctor\_info = {

'name': 'Dr. Sarah Lee',

'specialization': 'Neurologist',

'experience': 8

}

return render(request, 'profile.html', doctor\_info)

def contact(request):

return render(request, 'contact.html', {'email': 'contact@doctorfinder.com'})

**4. Set Up URLs**

**doctor/urls.py**

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'),

path('profile/', views.profile, name='profile'),

path('contact/', views.contact, name='contact'),

]

**doctor\_finder/urls.py**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

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

]

**5. Create Templates**

**Folder structure:**

doctor/

templates/

base.html

home.html

profile.html

contact.html

**base.html** (Layout template)

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

</head>

<body>

<nav>

<a href="{% url 'home' %}">Home</a> |

<a href="{% url 'profile' %}">Profile</a> |

<a href="{% url 'contact' %}">Contact</a>

</nav>

<hr>

{% block content %}{% endblock %}

</body>

</html>

**home.html**

{% extends "base.html" %}

{% block content %}

<h1>{{ title }}</h1>

<p>Find the best doctors in your city.</p>

{% endblock %}

**profile.html**

{% extends "base.html" %}

{% block content %}

<h2>{{ name }}</h2>

<p>Specialization: {{ specialization }}</p>

<p>Experience: {{ experience }} years</p>

{% endblock %}

**contact.html**

{% extends "base.html" %}

{% block content %}

<h2>Contact Us</h2>

<p>Email: {{ email }}</p>

{% endblock %}

**6. Run the Server**

python manage.py runserver

10. Form Validation using JavaScript Theory: • Using JavaScript for front-end form validation.

Ans:

**1. What is Form Validation?**

Form validation ensures that the data entered by a user into an HTML form is **correct, complete, and in the correct format** before it is sent to the server.

In Django, you can validate data **server-side** (in Python), but JavaScript enables **client-side validation** — which means the browser checks the form before sending it to the server.

**2. Why Use JavaScript for Form Validation?**

* **Faster Feedback** – Errors are shown instantly without waiting for a server response.
* **Reduced Server Load** – Only valid data is sent to the server.
* **Improved User Experience** – Users can correct mistakes immediately.

**3. Common Types of Form Validation**

* **Required Field Validation** – Ensures a field is not empty.
* **Email Format Validation** – Checks if the input is a valid email format.
* **Password Validation** – Checks for length, complexity, and matching confirmation.
* **Number Validation** – Ensures numbers fall within a given range.

**4. JavaScript Validation Techniques**

* **HTML5 Attributes** (e.g., required, pattern, minlength, maxlength) – basic validation without extra code.
* **JavaScript Event Listeners** (e.g., onsubmit, onblur, oninput) – custom validation logic.
* **Regular Expressions** – match patterns such as email, phone number, etc.

**5. Example Workflow in Django**

1. **HTML Form in Template**
   * Create a form in your Django template.
   * Add id attributes for JavaScript to target.
2. **JavaScript Validation Script**
   * Write a script that runs before form submission.
   * If validation fails, prevent the form from submitting.
3. **Server-side Validation**
   * Even with JavaScript validation, **always validate again in Django views** for security.

**6. Advantages & Limitations**

**Advantages:**

* Instant feedback
* Better user experience
* Less server load

**Limitations:**

* Can be disabled in the browser (not secure alone)
* Must always be complemented by **server-side validation**

Lab: • Write a Django project to implement JavaScript form validation for a user registration form.

Ans :

**1. Create the Django Project**

django-admin startproject js\_form\_project

cd js\_form\_project

python manage.py startapp accounts

**2. Add accounts to INSTALLED\_APPS**

In js\_form\_project/settings.py:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'accounts',

]

**3. Set Up URLs**

**js\_form\_project/urls.py**

from django.contrib import admin

from django.urls import path

from accounts import views

urlpatterns = [

path('admin/', admin.site.urls),

path('register/', views.register, name='register'),

]

**4. Create the View**

**accounts/views.py**

from django.shortcuts import render

def register(request):

return render(request, 'register.html')

**5. Create the Template**

**accounts/templates/register.html**

<!DOCTYPE html>

<html>

<head>

<title>User Registration</title>

<script>

function validateForm(event) {

event.preventDefault(); // Stop form from submitting

let username = document.getElementById("username").value.trim();

let email = document.getElementById("email").value.trim();

let password = document.getElementById("password").value;

let confirmPassword = document.getElementById("confirm\_password").value;

if (username === "") {

alert("Username is required.");

return false;

}

let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (!emailPattern.test(email)) {

alert("Enter a valid email address.");

return false;

}

if (password.length < 6) {

alert("Password must be at least 6 characters long.");

return false;

}

if (password !== confirmPassword) {

alert("Passwords do not match.");

return false;

}

alert("Form submitted successfully!");

document.getElementById("registrationForm").submit();

}

</script>

</head>

<body>

<h2>User Registration</h2>

<form id="registrationForm" method="post" onsubmit="validateForm(event)">

{% csrf\_token %}

<label>Username:</label>

<input type="text" id="username" name="username"><br><br>

<label>Email:</label>

<input type="email" id="email" name="email"><br><br>

<label>Password:</label>

<input type="password" id="password" name="password"><br><br>

<label>Confirm Password:</label>

<input type="password" id="confirm\_password" name="confirm\_password"><br><br>

<button type="submit">Register</button>

</form>

</body>

</html>

**6. Run the Server**

python manage.py runserver

Practical Example: 10) Write a Django project that uses JavaScript to validate fields like email and phone number in a registration form.

Ans :

**1. Create the Django Project & App**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp accounts

**2. Add App in Settings**

In **doctor\_finder/settings.py**:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'accounts',

]

**3. URL Routing**

**doctor\_finder/urls.py**

from django.contrib import admin

from django.urls import path

from accounts import views

urlpatterns = [

path('admin/', admin.site.urls),

path('register/', views.register, name='register'),

]

**4. View Function**

**accounts/views.py**

from django.shortcuts import render

def register(request):

return render(request, 'register.html')

**5. HTML + JavaScript Validation**

**accounts/templates/register.html**

<!DOCTYPE html>

<html>

<head>

<title>Patient Registration</title>

<script>

function validateForm(event) {

event.preventDefault(); // Stop default form submission

let email = document.getElementById("email").value.trim();

let phone = document.getElementById("phone").value.trim();

// Email validation regex

let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (!emailPattern.test(email)) {

alert("Please enter a valid email address.");

return false;

}

// Phone number validation (10 digits)

let phonePattern = /^[0-9]{10}$/;

if (!phonePattern.test(phone)) {

alert("Please enter a valid 10-digit phone number.");

return false;

}

alert("Registration successful!");

document.getElementById("registrationForm").submit();

}

</script>

</head>

<body>

<h2>Patient Registration</h2>

<form id="registrationForm" method="post" onsubmit="validateForm(event)">

{% csrf\_token %}

<label>Email:</label>

<input type="text" id="email" name="email"><br><br>

<label>Phone Number:</label>

<input type="text" id="phone" name="phone"><br><br>

<button type="submit">Register</button>

</form>

</body>

</html>

**6. Run the Server**

python manage.py runserver

11. Django Database Connectivity (MySQL or SQLite) Theory: • Connecting Django to a database (SQLite or MySQL). • Using the Django ORM for database queries.

Ans :

**11. Django Database Connectivity (MySQL or SQLite)**

**1. Django’s Default Database**

* By default, Django uses **SQLite** (a lightweight file-based database) — no installation required.
* Database settings are in **settings.py** under the DATABASES dictionary.

Example (SQLite default):

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

**2. Connecting Django to MySQL**

To use MySQL, install the MySQL client:

pip install mysqlclient

Update **settings.py**:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'doctor\_finder\_db',

'USER': 'root',

'PASSWORD': 'your\_password',

'HOST': 'localhost',

'PORT': '3306',

}

}

**3. Running Migrations**

Once database settings are configured:

python manage.py makemigrations

python manage.py migrate

* **makemigrations** → Prepares SQL for changes in models.
* **migrate** → Executes SQL to update the database schema.

**4. Django ORM (Object-Relational Mapping)**

Django ORM lets you interact with the database **without writing SQL**.

Example Model:

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

**5. Common ORM Queries**

# Create

Doctor.objects.create(name="Dr. Smith", specialization="Cardiology")

# Read all

doctors = Doctor.objects.all()

# Filter

cardiologists = Doctor.objects.filter(specialization="Cardiology")

# Update

Doctor.objects.filter(name="Dr. Smith").update(specialization="Neurology")

# Delete

Doctor.objects.filter(name="Dr. Smith").delete()

Lab: • Set up database connectivity for a Django project.

Ans :

**1. Create a Django Project**

django-admin startproject doctor\_finder

cd doctor\_finder

**2. Install Database Driver**

**For SQLite** → No installation needed (Django default).  
**For MySQL** → Install the MySQL client:

pip install mysqlclient

**3. Configure Database in settings.py**

**SQLite (default)**

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

**MySQL**

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'doctor\_finder\_db',

'USER': 'root',

'PASSWORD': 'your\_password',

'HOST': 'localhost',

'PORT': '3306',

}

}

**4. Create Database (MySQL Only)**

CREATE DATABASE doctor\_finder\_db;

**5. Create an App**

python manage.py startapp doctor

**6. Register App in settings.py**

INSTALLED\_APPS = [

...,

'doctor',

]

**7. Create a Model in doctor/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

def \_\_str\_\_(self):

return self.name

**8. Apply Migrations**

python manage.py makemigrations

python manage.py migrate

**9. Test ORM**

Open Django shell:

python manage.py shell

Run:

from doctor.models import Doctor

Doctor.objects.create(name="Dr. John", specialization="Cardiology")

print(Doctor.objects.all())

Practical Example: 11) Write a Django project to connect to an SQLite/MySQL database and manage doctor records.

Ans :

**Step 1: Create Project**

django-admin startproject doctor\_finder

cd doctor\_finder

**Step 2: Create App**

python manage.py startapp doctor

**Step 3: Configure Database in settings.py**

**For SQLite (default):**

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

**For MySQL:**

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'doctor\_finder\_db',

'USER': 'root',

'PASSWORD': 'your\_password',

'HOST': 'localhost',

'PORT': '3306',

}

}

📌 For MySQL, create the database first:

CREATE DATABASE doctor\_finder\_db;

and install:

pip install mysqlclient

**Step 4: Add App to INSTALLED\_APPS**

In settings.py:

INSTALLED\_APPS = [

...,

'doctor',

]

**Step 5: Create Model in doctor/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

contact = models.CharField(max\_length=15)

def \_\_str\_\_(self):

return self.name

**Step 6: Migrate Database**

python manage.py makemigrations

python manage.py migrate

**Step 7: Register Model in Admin**

In doctor/admin.py:

from django.contrib import admin

from .models import Doctor

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ('name', 'specialization', 'contact')

Create a superuser:

python manage.py createsuperuser

**Step 8: Create Views in doctor/views.py**

from django.shortcuts import render

from .models import Doctor

def doctor\_list(request):

doctors = Doctor.objects.all()

return render(request, 'doctor\_list.html', {'doctors': doctors})

**Step 9: URL Routing**

In doctor\_finder/urls.py:

from django.contrib import admin

from django.urls import path

from doctor.views import doctor\_list

urlpatterns = [

path('admin/', admin.site.urls),

path('doctors/', doctor\_list, name='doctor\_list'),

]

**Step 10: Create Template**

Create folder doctor/templates/doctor\_list.html:

<!DOCTYPE html>

<html>

<head>

<title>Doctor Records</title>

</head>

<body>

<h1>Doctor List</h1>

<ul>

{% for doctor in doctors %}

<li>{{ doctor.name }} - {{ doctor.specialization }} - {{ doctor.contact }}</li>

{% empty %}

<li>No doctors available</li>

{% endfor %}

</ul>

</body>

</html>

**Step 11: Run Server**

python manage.py runserver

12. ORM and QuerySets Theory: • Understanding Django’s ORM and how QuerySets are used to interact with the database.

Ans :

**1. What is ORM?**

* **ORM** stands for **Object Relational Mapper**.
* In Django, ORM allows developers to interact with the database using **Python objects and methods** instead of writing raw SQL queries.
* Each **model class** in Django corresponds to a **table** in the database.
* Each **model object (instance)** corresponds to a **row** in that table.

👉 Example:  
If you have a model like this:

from django.db import models

class Student(models.Model):

name = models.CharField(max\_length=100)

age = models.IntegerField()

course = models.CharField(max\_length=50)

* Django will create a table like:

student(id, name, age, course)

* An object like Student(name="Vinay", age=21, course="Python") will represent one row in that table.

**2. QuerySets**

* A **QuerySet** is a collection of objects retrieved from the database.
* It can contain **zero, one, or many records**.
* QuerySets are built on top of Django ORM and allow filtering, ordering, and combining queries.

👉 Example:

students = Student.objects.all() # Fetches all students

**3. Common QuerySet Methods**

Here are the most used QuerySet operations:

1. **Retrieve all objects**

Student.objects.all()

1. **Filter objects**

Student.objects.filter(course="Python") # WHERE course = 'Python'

Student.objects.filter(age\_\_gte=18) # WHERE age >= 18

1. **Get a single object**

Student.objects.get(id=1) # Fetch student with id=1

⚠️ If no record or multiple records are found → raises error.

1. **Create new record**

student = Student.objects.create(name="Amit", age=22, course="Django")

1. **Update record**

student = Student.objects.get(id=1)

student.age = 23

student.save()

1. **Delete record**

student = Student.objects.get(id=1)

student.delete()

1. **Ordering**

Student.objects.all().order\_by('name') # ASC

Student.objects.all().order\_by('-age') # DESC

Slicing (limit)

python

Copy

Edit

Student.objects.all()[:5] # LIMIT 5

Aggregation

python

Copy

Edit

from django.db.models import Avg, Max, Min, Count

Student.objects.all().aggregate(Avg('age'))

Student.objects.all().aggregate(Max('age'))

Chaining

python

Copy

Edit

Student.objects.filter(course="Python").order\_by('name')

4. Advantages of Django ORM

No need to write raw SQL queries.

Works across different databases (SQLite, MySQL, PostgreSQL).

Cleaner, more readable code.

Prevents SQL injection.

Lab: • Perform CRUD operations using Django ORM. Practical Example: 12) Write a Django project that demonstrates CRUD operations (Create, Read, Update, Delete) on doctor profiles using Django ORM

Ans :

**Step 1: Create Django Project**

django-admin startproject hospital

cd hospital

python manage.py startapp doctors

**Step 2: Create Doctor Model**

In doctors/models.py:

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.IntegerField() # in years

email = models.EmailField(unique=True)

def \_\_str\_\_(self):

return f"{self.name} - {self.specialization}"

Run migrations:

python manage.py makemigrations

python manage.py migrate

**Step 3: Perform CRUD Operations using ORM**

We’ll use the **Django shell** to test.

Open shell:

python manage.py shell

**1. CREATE (Insert doctor records)**

from doctors.models import Doctor

# Method 1: Using create()

doc1 = Doctor.objects.create(name="Dr. Amit", specialization="Cardiologist", experience=10, email="amit@hospital.com")

# Method 2: Using save()

doc2 = Doctor(name="Dr. Neha", specialization="Neurologist", experience=8, email="neha@hospital.com")

doc2.save()

**2. READ (Fetch doctor records)**

# Fetch all

Doctor.objects.all()

# Fetch by condition

Doctor.objects.filter(specialization="Cardiologist")

# Fetch single record

Doctor.objects.get(id=1)

**3. UPDATE (Modify doctor record)**

doc = Doctor.objects.get(id=1)

doc.experience = 12

doc.save()

**4. DELETE (Remove doctor record)**

doc = Doctor.objects.get(id=2)

doc.delete()

**Step 4: Use QuerySet Features**

# Order doctors by experience

Doctor.objects.all().order\_by('-experience')

# Count doctors

Doctor.objects.count()

# Filter by multiple conditions

Doctor.objects.filter(experience\_\_gte=5, specialization="Cardiologist")

13. Django Forms and Authentication Theory: • Using Django’s built-in form handling. • Implementing Django’s authentication system (sign up, login, logout, password management).

Ans :

**1. Django Forms**

* Django provides a powerful **Form API** to handle user input securely.
* Forms are Python classes that map to HTML <form> elements.
* Django takes care of:
  + Generating HTML form fields.
  + Validating input.
  + Protecting against CSRF (Cross-Site Request Forgery).
  + Converting data to Python objects.

👉 Example:

from django import forms

class ContactForm(forms.Form):

name = forms.CharField(max\_length=100)

email = forms.EmailField()

message = forms.CharField(widget=forms.Textarea)

In a **view**:

from django.shortcuts import render

from .forms import ContactForm

def contact\_view(request):

if request.method == "POST":

form = ContactForm(request.POST)

if form.is\_valid():

print(form.cleaned\_data) # dictionary of validated form data

else:

form = ContactForm()

return render(request, "contact.html", {"form": form})

In **template** (contact.html):

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Submit</button>

</form>

**2. Django Authentication System**

Django has a **built-in authentication framework** for handling users.

* **Key features:**
  + User registration (sign up).
  + Login / Logout.
  + Password hashing (secure storage).
  + Password reset & change.
  + Permissions & groups.

**3. Common Authentication Functions**

1. **Sign Up (User Registration)**

from django.contrib.auth.models import User

user = User.objects.create\_user(username="vinay", password="mypassword123", email="vinay@example.com")

1. **Login**

from django.contrib.auth import authenticate, login

user = authenticate(username="vinay", password="mypassword123")

if user is not None:

login(request, user) # starts session

1. **Logout**

from django.contrib.auth import logout

logout(request)

1. **Password Change**

user.set\_password("newpassword123")

user.save()

1. **Authentication Forms**  
   Django provides ready-made forms in django.contrib.auth.forms:

* UserCreationForm → Sign up
* AuthenticationForm → Login
* PasswordChangeForm → Change password
* PasswordResetForm → Reset password (via email)

**4. Example Workflow**

* **Sign up page:** Users register with UserCreationForm.
* **Login page:** Users authenticate using AuthenticationForm.
* **Logout:** Ends session.
* **Password management:** Use Django’s built-in forms.

Lab: • Create a Django project for user registration and login functionality.

Ans :

**Step 1: Create Project and App**

django-admin startproject authsystem

cd authsystem

python manage.py startapp accounts

Add "accounts" in **INSTALLED\_APPS** inside authsystem/settings.py.

**Step 2: URLs Configuration**

In authsystem/urls.py:

from django.contrib import admin

from django.urls import path

from accounts import views

urlpatterns = [

path('admin/', admin.site.urls),

path('signup/', views.signup\_view, name='signup'),

path('login/', views.login\_view, name='login'),

path('logout/', views.logout\_view, name='logout'),

path('', views.home, name='home'), # homepage

]

**Step 3: Views (Registration, Login, Logout)**

In accounts/views.py:

from django.shortcuts import render, redirect

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

from django.contrib.auth import login, logout, authenticate

from django.contrib import messages

# Home Page

def home(request):

return render(request, "home.html")

# Signup (Register)

def signup\_view(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

login(request, user) # auto-login after registration

messages.success(request, "Registration successful!")

return redirect("home")

else:

form = UserCreationForm()

return render(request, "signup.html", {"form": form})

# Login

def login\_view(request):

if request.method == "POST":

form = AuthenticationForm(data=request.POST)

if form.is\_valid():

user = form.get\_user()

login(request, user)

messages.success(request, "Login successful!")

return redirect("home")

else:

form = AuthenticationForm()

return render(request, "login.html", {"form": form})

# Logout

def logout\_view(request):

logout(request)

messages.info(request, "Logged out successfully!")

return redirect("login")

**Step 4: Templates**

Inside accounts/templates/ create:

**home.html**

<!DOCTYPE html>

<html>

<head>

<title>Home</title>

</head>

<body>

<h2>Welcome {{ user.username|default:"Guest" }}</h2>

{% if user.is\_authenticated %}

<p>You are logged in!</p>

<a href="{% url 'logout' %}">Logout</a>

{% else %}

<a href="{% url 'login' %}">Login</a> |

<a href="{% url 'signup' %}">Signup</a>

{% endif %}

</body>

</html>

**signup.html**

<!DOCTYPE html>

<html>

<head>

<title>Signup</title>

</head>

<body>

<h2>User Registration</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Register</button>

</form>

<p>Already have an account? <a href="{% url 'login' %}">Login</a></p>

</body>

</html>

**login.html**

<!DOCTYPE html>

<html>

<head>

<title>Login</title>

</head>

<body>

<h2>User Login</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Login</button>

</form>

<p>Don't have an account? <a href="{% url 'signup' %}">Signup</a></p>

</body>

</html>

**Step 5: Run Server**

python manage.py runserver

Practical Example: 13) Write a Django project to handle user sign up, login, password reset, and profile updates.

Ans :

**Step 1: Create Project & App**

django-admin startproject userauth

cd userauth

python manage.py startapp accounts

Add "accounts" in INSTALLED\_APPS inside userauth/settings.py.

Also add:

EMAIL\_BACKEND = "django.core.mail.backends.console.EmailBackend"

👉 This will print password reset emails in the terminal (good for testing).

**Step 2: URL Configuration**

In userauth/urls.py:

from django.contrib import admin

from django.urls import path

from accounts import views

from django.contrib.auth import views as auth\_views

urlpatterns = [

path('admin/', admin.site.urls),

# Custom Views

path('', views.home, name='home'),

path('signup/', views.signup\_view, name='signup'),

path('login/', views.login\_view, name='login'),

path('logout/', views.logout\_view, name='logout'),

path('profile/', views.profile\_view, name='profile'),

path('profile/update/', views.profile\_update, name='profile\_update'),

# Password Reset (Django built-in views)

path('password\_reset/', auth\_views.PasswordResetView.as\_view(template\_name="password\_reset.html"), name='password\_reset'),

path('password\_reset/done/', auth\_views.PasswordResetDoneView.as\_view(template\_name="password\_reset\_done.html"), name='password\_reset\_done'),

path('reset/<uidb64>/<token>/', auth\_views.PasswordResetConfirmView.as\_view(template\_name="password\_reset\_confirm.html"), name='password\_reset\_confirm'),

path('reset/done/', auth\_views.PasswordResetCompleteView.as\_view(template\_name="password\_reset\_complete.html"), name='password\_reset\_complete'),

]

**Step 3: Views**

In accounts/views.py:

from django.shortcuts import render, redirect

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

from django.contrib.auth import login, logout

from django.contrib.auth.decorators import login\_required

from django.contrib import messages

from django.contrib.auth.models import User

from django import forms

# Profile Update Form

class ProfileUpdateForm(forms.ModelForm):

class Meta:

model = User

fields = ['username', 'email', 'first\_name', 'last\_name']

# Home Page

def home(request):

return render(request, "home.html")

# Signup

def signup\_view(request):

if request.method == "POST":

form = UserCreationForm(request.POST)

if form.is\_valid():

user = form.save()

login(request, user)

messages.success(request, "Signup successful!")

return redirect("home")

else:

form = UserCreationForm()

return render(request, "signup.html", {"form": form})

# Login

def login\_view(request):

if request.method == "POST":

form = AuthenticationForm(data=request.POST)

if form.is\_valid():

user = form.get\_user()

login(request, user)

messages.success(request, "Login successful!")

return redirect("home")

else:

form = AuthenticationForm()

return render(request, "login.html", {"form": form})

# Logout

def logout\_view(request):

logout(request)

messages.info(request, "Logged out successfully!")

return redirect("login")

# Profile

@login\_required

def profile\_view(request):

return render(request, "profile.html")

# Profile Update

@login\_required

def profile\_update(request):

if request.method == "POST":

form = ProfileUpdateForm(request.POST, instance=request.user)

if form.is\_valid():

form.save()

messages.success(request, "Profile updated!")

return redirect("profile")

else:

form = ProfileUpdateForm(instance=request.user)

return render(request, "profile\_update.html", {"form": form})

**Step 4: Templates**

Inside accounts/templates/:

**home.html**

<h2>Welcome {{ user.username|default:"Guest" }}</h2>

{% if user.is\_authenticated %}

<p><a href="{% url 'profile' %}">Profile</a></p>

<a href="{% url 'logout' %}">Logout</a>

{% else %}

<a href="{% url 'login' %}">Login</a> |

<a href="{% url 'signup' %}">Signup</a>

{% endif %}

**signup.html**

<h2>Sign Up</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Register</button>

</form>

<a href="{% url 'login' %}">Already have an account? Login</a>

**login.html**

<h2>Login</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Login</button>

</form>

<a href="{% url 'password\_reset' %}">Forgot Password?</a>

**profile.html**

<h2>User Profile</h2>

<p><b>Username:</b> {{ user.username }}</p>

<p><b>Email:</b> {{ user.email }}</p>

<p><b>Name:</b> {{ user.first\_name }} {{ user.last\_name }}</p>

<a href="{% url 'profile\_update' %}">Edit Profile</a> |

<a href="{% url 'logout' %}">Logout</a>

**profile\_update.html**

<h2>Update Profile</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Save Changes</button>

</form>

<a href="{% url 'profile' %}">Back to Profile</a>

**Password Reset Templates**

* password\_reset.html

<h2>Password Reset</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Reset Password</button>

</form>

* password\_reset\_done.html

<p>Password reset link sent! Check your email.</p>

* password\_reset\_confirm.html

<h2>Enter New Password</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Change Password</button>

</form>

* password\_reset\_complete.html

<p>Password has been reset! <a href="{% url 'login' %}">Login</a></p>

**Step 5: Run Project**

python manage.py runserver

14. CRUD Operations using AJAX Theory: • Using AJAX for making asynchronous requests to the server without reloading the page.

Ans :

**1. What is AJAX?**

* **AJAX (Asynchronous JavaScript and XML)** is a technique that allows web applications to **send and receive data from the server without reloading the entire page**.
* In Django, AJAX is commonly used with **jQuery** or the modern **Fetch API**.
* Instead of refreshing the page, AJAX dynamically updates parts of the page → better user experience.

**2. How AJAX Works**

1. A user performs an action (e.g., click a button, submit a form).
2. JavaScript sends an **AJAX request** to the server (via a Django view).
3. Django processes the request and returns a response (usually JSON).
4. JavaScript updates the webpage dynamically with the new data.

**3. Benefits of AJAX**

* **Faster interactions** → only fetch required data.
* **Improved UX** → no full page reloads.
* **Dynamic updates** → great for CRUD apps, search, chat apps, etc.

**4. AJAX in Django CRUD**

Using AJAX, we can:

* **Create**: Submit forms asynchronously → add new data to database and update page instantly.
* **Read**: Fetch data from server (e.g., list of items) without reloading.
* **Update**: Edit items inline (like in Google Sheets).
* **Delete**: Remove records instantly with confirmation popup.

**5. Example Workflow**

Suppose we have a **Book model**:

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.CharField(max\_length=100)

* **Without AJAX**: Adding or deleting a book requires a full page refresh.
* **With AJAX**: User clicks "Add Book", the form sends data via AJAX → server saves it → JS updates the book list instantly.

**6. JSON Response in Django**

For AJAX, Django often returns **JSON data** instead of HTML.  
Example:

from django.http import JsonResponse

def add\_book(request):

if request.method == "POST":

title = request.POST.get("title")

author = request.POST.get("author")

book = Book.objects.create(title=title, author=author)

return JsonResponse({"id": book.id, "title": book.title, "author": book.author})

The **frontend** (JavaScript) will receive this JSON and update the DOM.

Lab: • Implement AJAX in a Django project for performing CRUD operations

Ans :

**Step 1: Create Django Project & App**

django-admin startproject ajaxcrud

cd ajaxcrud

python manage.py startapp books

Add "books" in INSTALLED\_APPS inside ajaxcrud/settings.py.

**Step 2: Create Model**

In books/models.py:

from django.db import models

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.CharField(max\_length=100)

def \_\_str\_\_(self):

return self.title

Run:

python manage.py makemigrations

python manage.py migrate

**Step 3: URLs**

In ajaxcrud/urls.py:

from django.contrib import admin

from django.urls import path

from books import views

urlpatterns = [

path('admin/', admin.site.urls),

path('', views.book\_list, name="book\_list"),

path('add/', views.add\_book, name="add\_book"),

path('update/<int:pk>/', views.update\_book, name="update\_book"),

path('delete/<int:pk>/', views.delete\_book, name="delete\_book"),

]

**Step 4: Views (AJAX CRUD)**

In books/views.py:

from django.shortcuts import render, get\_object\_or\_404

from django.http import JsonResponse

from .models import Book

# Home - List Books

def book\_list(request):

books = Book.objects.all()

return render(request, "book\_list.html", {"books": books})

# Create

def add\_book(request):

if request.method == "POST":

title = request.POST.get("title")

author = request.POST.get("author")

book = Book.objects.create(title=title, author=author)

return JsonResponse({"id": book.id, "title": book.title, "author": book.author})

# Update

def update\_book(request, pk):

book = get\_object\_or\_404(Book, pk=pk)

if request.method == "POST":

book.title = request.POST.get("title")

book.author = request.POST.get("author")

book.save()

return JsonResponse({"id": book.id, "title": book.title, "author": book.author})

# Delete

def delete\_book(request, pk):

book = get\_object\_or\_404(Book, pk=pk)

book.delete()

return JsonResponse({"deleted": True})

**Step 5: Template with AJAX**

Create books/templates/book\_list.html:

<!DOCTYPE html>

<html>

<title>AJAX CRUD - Books</title>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

</head>

<body>

<h2>Book List</h2>

<!-- Add Book Form -->

<form id="addBookForm">

{% csrf\_token %}

<input type="text" name="title" placeholder="Title" required>

<input type="text" name="author" placeholder="Author" required>

<button type="submit">Add Book</button>

</form>

<ul id="bookList">

{% for book in books %}

<li id="book-{{ book.id }}">

<b>{{ book.title }}</b> by {{ book.author }}

<button onclick="deleteBook({{ book.id }})">Delete</button>

<button onclick="editBook({{ book.id }}, '{{ book.title }}', '{{ book.author }}')">Edit</button>

</li>

{% endfor %}

</ul>

<!-- Edit Form (hidden) -->

<div id="editForm" style="display:none;">

<h3>Edit Book</h3>

<form id="updateBookForm">

{% csrf\_token %}

<input type="hidden" id="editId">

<input type="text" id="editTitle" name="title" required>

<input type="text" id="editAuthor" name="author" required>

<button type="submit">Update</button>

</form>

</div>

<script>

// CSRF setup for AJAX

function getCookie(name) {

let cookieValue = null;

if (document.cookie && document.cookie !== '') {

const cookies = document.cookie.split(';');

for (let cookie of cookies) {

cookie = cookie.trim();

if (cookie.substring(0, name.length + 1) === (name + '=')) {

cookieValue = decodeURIComponent(cookie.substring(name.length + 1));

break;

}

}

}

return cookieValue;

}

const csrftoken = getCookie('csrftoken');

// Add Book

$("#addBookForm").submit(function(e) {

e.preventDefault();

$.ajax({

url: "/add/",

type: "POST",

data: $(this).serialize(),

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

$("#bookList").append(

`<li id="book-${data.id}">

<b>${data.title}</b> by ${data.author}

<button onclick="deleteBook(${data.id})">Delete</button>

<button onclick="editBook(${data.id}, '${data.title}', '${data.author}')">Edit</button>

</li>`

);

$("#addBookForm")[0].reset();

}

});

});

// Delete Book

function deleteBook(id) {

$.ajax({

url: `/delete/${id}/`,

type: "POST",

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

if (data.deleted) {

$(`#book-${id}`).remove();

}

}

});

}

// Show Edit Form

function editBook(id, title, author) {

$("#editForm").show();

$("#editId").val(id);

$("#editTitle").val(title);

$("#editAuthor").val(author);

}

// Update Book

$("#updateBookForm").submit(function(e) {

e.preventDefault();

let id = $("#editId").val();

$.ajax({

url: `/update/${id}/`,

type: "POST",

data: {

"title": $("#editTitle").val(),

"author": $("#editAuthor").val(),

},

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

$(`#book-${data.id}`).html(

`<b>${data.title}</b> by ${data.author}

<button onclick="deleteBook(${data.id})">Delete</button>

<button onclick="editBook(${data.id}, '${data.title}', '${data.author}')">Edit</button>`

);

$("#editForm").hide();

}

});

});

</script>

</body>

</html>

Step 6: Run Project

bash

Copy

Edit

python manage.py runserver

Practical Example: 14) Write a Django project that uses AJAX to add, edit, or delete doctor profiles without refreshing the page?

Ans :

**Step 1: Create Project & App**

django-admin startproject doctorajax

cd doctorajax

python manage.py startapp doctors

Add "doctors" to INSTALLED\_APPS in doctorajax/settings.py.

**Step 2: Create Model**

In doctors/models.py:

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.IntegerField() # in years

def \_\_str\_\_(self):

return f"{self.name} ({self.specialization})"

Run:

python manage.py makemigrations

python manage.py migrate

**Step 3: URLs**

In doctorajax/urls.py:

from django.contrib import admin

from django.urls import path

from doctors import views

urlpatterns = [

path('admin/', admin.site.urls),

path('', views.doctor\_list, name="doctor\_list"),

path('add/', views.add\_doctor, name="add\_doctor"),

path('update/<int:pk>/', views.update\_doctor, name="update\_doctor"),

path('delete/<int:pk>/', views.delete\_doctor, name="delete\_doctor"),

]

**Step 4: Views (AJAX Handlers)**

In doctors/views.py:

from django.shortcuts import render, get\_object\_or\_404

from django.http import JsonResponse

from .models import Doctor

# Show all doctors

def doctor\_list(request):

doctors = Doctor.objects.all()

return render(request, "doctor\_list.html", {"doctors": doctors})

# Add doctor

def add\_doctor(request):

if request.method == "POST":

name = request.POST.get("name")

specialization = request.POST.get("specialization")

experience = request.POST.get("experience")

doctor = Doctor.objects.create(name=name, specialization=specialization, experience=experience)

return JsonResponse({

"id": doctor.id,

"name": doctor.name,

"specialization": doctor.specialization,

"experience": doctor.experience

})

# Update doctor

def update\_doctor(request, pk):

doctor = get\_object\_or\_404(Doctor, pk=pk)

if request.method == "POST":

doctor.name = request.POST.get("name")

doctor.specialization = request.POST.get("specialization")

doctor.experience = request.POST.get("experience")

doctor.save()

return JsonResponse({

"id": doctor.id,

"name": doctor.name,

"specialization": doctor.specialization,

"experience": doctor.experience

})

# Delete doctor

def delete\_doctor(request, pk):

doctor = get\_object\_or\_404(Doctor, pk=pk)

doctor.delete()

return JsonResponse({"deleted": True})

**Step 5: Template with AJAX**

Create doctors/templates/doctor\_list.html:

<!DOCTYPE html>

<html>

<head>

<title>Doctor Profiles - AJAX CRUD</title>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

</head>

<body>

<h2>Doctor Profiles</h2>

<!-- Add Doctor Form -->

<form id="addDoctorForm">

{% csrf\_token %}

<input type="text" name="name" placeholder="Name" required>

<input type="number" name="experience" placeholder="Experience (years)" required>

<button type="submit">Add Doctor</button>

</form>

<hr>

<ul id="doctorList">

{% for doc in doctors %}

<li id="doctor-{{ doc.id }}">

<b>{{ doc.name }}</b> - {{ doc.specialization }} ({{ doc.experience }} years)

<button onclick="deleteDoctor({{ doc.id }})">Delete</button>

<button onclick="editDoctor({{ doc.id }}, '{{ doc.name }}', '{{ doc.specialization }}', '{{ doc.experience }}')">Edit</button>

</li>

{% endfor %}

</ul>

<!-- Edit Form -->

<div id="editForm" style="display:none;">

<h3>Edit Doctor</h3>

<form id="updateDoctorForm">

{% csrf\_token %}

<input type="hidden" id="editId">

<input type="text" id="editName" name="name" required>

<input type="text" id="editSpecialization" name="specialization" required>

<input type="number" id="editExperience" name="experience" required>

<button type="submit">Update</button>

</form>

</div>

<script>

// CSRF setup

function getCookie(name) {

let cookieValue = null;

if (document.cookie && document.cookie !== '') {

const cookies = document.cookie.split(';');

for (let cookie of cookies) {

cookie = cookie.trim();

if (cookie.startsWith(name + '=')) {

cookieValue = decodeURIComponent(cookie.split('=')[1]);

break;

}

}

}

return cookieValue;

}

const csrftoken = getCookie('csrftoken');

// Add Doctor

$("#addDoctorForm").submit(function(e) {

e.preventDefault();

$.ajax({

url: "/add/",

type: "POST",

data: $(this).serialize(),

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

$("#doctorList").append(

`<li id="doctor-${data.id}">

<b>${data.name}</b> - ${data.specialization} (${data.experience} years)

<button onclick="deleteDoctor(${data.id})">Delete</button>

<button onclick="editDoctor(${data.id}, '${data.name}', '${data.specialization}', '${data.experience}')">Edit</button>

</li>`

);

$("#addDoctorForm")[0].reset();

}

});

});

// Delete Doctor

function deleteDoctor(id) {

$.ajax({

url: `/delete/${id}/`,

type: "POST",

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

if (data.deleted) {

$(`#doctor-${id}`).remove();

}

}

});

}

// Show Edit Form

function editDoctor(id, name, specialization, experience) {

$("#editForm").show();

$("#editId").val(id);

$("#editName").val(name);

$("#editSpecialization").val(specialization);

$("#editExperience").val(experience);

}

// Update Doctor

$("#updateDoctorForm").submit(function(e) {

e.preventDefault();

let id = $("#editId").val();

$.ajax({

url: `/update/${id}/`,

type: "POST",

data: {

"name": $("#editName").val(),

"specialization": $("#editSpecialization").val(),

"experience": $("#editExperience").val()

},

headers: {"X-CSRFToken": csrftoken},

success: function(data) {

$(`#doctor-${data.id}`).html(

`<b>${data.name}</b> - ${data.specialization} (${data.experience} years)

<button onclick="deleteDoctor(${data.id})">Delete</button>

<button onclick="editDoctor(${data.id}, '${data.name}', '${data.specialization}', '${data.experience}')">Edit</button>`

);

$("#editForm").hide();

}

});

});

</script>

</body>

</html>

Step 6: Run Project

bash

Copy

Edit

python manage.py runserver

15. Customizing the Django Admin Panel Theory: • Techniques for customizing the Django admin panel

Ans :

**1. Registering Models**

* By default, you need to register a model in admin.py:
* from django.contrib import admin
* from .models import Doctor
* admin.site.register(Doctor)
* This makes the model accessible in the admin panel.

**2. Using ModelAdmin Class**

* Customize how a model appears in the admin by extending admin.ModelAdmin.

Example:

class DoctorAdmin(admin.ModelAdmin):

list\_display = ("id", "name", "specialization", "experience")

search\_fields = ("name", "specialization")

list\_filter = ("specialization",)

ordering = ("-experience",)

admin.site.register(Doctor, DoctorAdmin)

✅ Features added:

* **list\_display** → Show selected fields in the list view.
* **search\_fields** → Add a search box for searching by fields.
* **list\_filter** → Add filter options (sidebar).
* **ordering** → Default ordering in list view.

**3. Editable List Page**

* Make fields editable directly in the list page:
* list\_editable = ("experience",)

**4. Fieldsets & Form Customization**

* Organize fields in the **detail (edit) page**:
* fieldsets = (
* ("Personal Info", {"fields": ("name", "specialization")}),
* ("Professional Info", {"fields": ("experience",)}),
* )
* Alternatively, specify order with:
* fields = ("name", "specialization", "experience")

**5. Read-only Fields**

* Prevent editing for certain fields:
* readonly\_fields = ("id",)

**6. Custom Forms in Admin**

* Use Django forms for validation or styling:
* from django import forms
* from .models import Doctor
* class DoctorForm(forms.ModelForm):
* class Meta:
* model = Doctor
* fields = "\_\_all\_\_"
* def clean\_experience(self):
* exp = self.cleaned\_data["experience"]
* if exp < 0:
* raise forms.ValidationError("Experience cannot be negative!")
* return exp
* class DoctorAdmin(admin.ModelAdmin):
* form = DoctorForm
* admin.site.register(Doctor, DoctorAdmin)

**7. Inline Models**

* Manage related models **within the parent model’s admin**.

Example (Doctor has multiple Patients):

class PatientInline(admin.TabularInline): # or StackedInline

model = Patient

extra = 1 # show empty rows for new patients

class DoctorAdmin(admin.ModelAdmin):

inlines = [PatientInline]

**8. Custom Actions**

* Add bulk actions in the list page:
* def mark\_experienced(modeladmin, request, queryset):
* queryset.update(experience=10)
* mark\_experienced.short\_description = "Set experience to 10 years"
* class DoctorAdmin(admin.ModelAdmin):
* actions = [mark\_experienced]

**9. Branding the Admin**

* Change admin title, header, and index:
* admin.site.site\_header = "Hospital Management Admin"
* admin.site.site\_title = "Hospital Admin"
* admin.site.index\_title = "Welcome to Hospital Dashboard"

**10. Using list\_display\_links & prepopulated\_fields**

* **Click only certain fields for editing**:
* list\_display\_links = ("name",)
* **Auto-fill slug fields**:
* prepopulated\_fields = {"slug": ("name",)}

Lab: • Customize the Django admin panel for better management of records.

Ans :  
**Step 1: Create a Model (Doctor Example)**

In models.py:

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.IntegerField()

email = models.EmailField(unique=True)

def \_\_str\_\_(self):

return f"{self.name} ({self.specialization})"

**Step 2: Register the Model with Custom Admin**

In admin.py:

from django.contrib import admin

from .models import Doctor

# 1) Custom Action

def reset\_experience(modeladmin, request, queryset):

queryset.update(experience=0)

reset\_experience.short\_description = "Reset experience to 0"

# 2) Doctor Admin Customization

class DoctorAdmin(admin.ModelAdmin):

list\_display = ("id", "name", "specialization", "experience", "email") # Show fields in list view

search\_fields = ("name", "specialization") # Search bar

list\_filter = ("specialization",) # Filter sidebar

ordering = ("-experience",) # Default order

list\_editable = ("experience",) # Edit directly in list

readonly\_fields = ("email",) # Make email read-only

actions = [reset\_experience] # Custom bulk action

# Organize fields in sections

fieldsets = (

("Personal Information", {"fields": ("name", "email")}),

("Professional Details", {"fields": ("specialization", "experience")}),

)

admin.site.register(Doctor, DoctorAdmin)

**Step 3: Customize Admin Branding**

In admin.py (below the registrations):

admin.site.site\_header = "Hospital Management Admin"

admin.site.site\_title = "Hospital Admin Portal"

admin.site.index\_title = "Welcome to the Hospital Management Dashboard"

**Step 4: Run Server**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

Practical Example: 15) Write a Django project that customizes the admin panel to display more detailed doctor information (e.g., specialties, availability)

Ans :

**Step 1: Models**

In models.py (inside an app like hospital):

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialization = models.CharField(max\_length=100)

experience = models.PositiveIntegerField()

email = models.EmailField(unique=True)

available\_days = models.CharField(max\_length=100, help\_text="E.g., Mon, Wed, Fri")

available\_time = models.TimeField()

def \_\_str\_\_(self):

return f"Dr. {self.name} ({self.specialization})"

class Patient(models.Model):

doctor = models.ForeignKey(Doctor, on\_delete=models.CASCADE, related\_name="patients")

name = models.CharField(max\_length=100)

age = models.IntegerField()

disease = models.CharField(max\_length=100)

def \_\_str\_\_(self):

return f"{self.name} (Patient of {self.doctor.name})"

**Step 2: Admin Customization**

In admin.py:

from django.contrib import admin

from .models import Doctor, Patient

# Custom Inline (Patients under Doctor)

class PatientInline(admin.TabularInline):

model = Patient

extra = 1

# Custom Action

def mark\_experienced(modeladmin, request, queryset):

queryset.update(experience=10)

mark\_experienced.short\_description = "Set experience to 10 years"

# Custom Doctor Admin

class DoctorAdmin(admin.ModelAdmin):

list\_display = ("id", "name", "specialization", "experience", "email", "available\_days", "available\_time")

search\_fields = ("name", "specialization", "email")

list\_filter = ("specialization", "available\_days")

ordering = ("-experience",)

list\_editable = ("experience", "available\_days", "available\_time")

readonly\_fields = ("email",)

actions = [mark\_experienced]

inlines = [PatientInline]

# Organize fields

fieldsets = (

("Personal Information", {"fields": ("name", "email")}),

("Professional Details", {"fields": ("specialization", "experience")}),

("Availability", {"fields": ("available\_days", "available\_time")}),

)

# Register models

admin.site.register(Doctor, DoctorAdmin)

admin.site.register(Patient)

# Branding Admin Panel

admin.site.site\_header = "Hospital Management Admin"

admin.site.site\_title = "Hospital Admin Portal"

admin.site.index\_title = "Welcome to Hospital Dashboard"

**Step 3: Apply Migrations & Create Superuser**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

**Step 4: Test in Admin Panel**

* Open http://127.0.0.1:8000/admin/
* Login as superuser
* Navigate to **Doctors**:
  + Doctors are listed with **name, specialization, email, experience, availability**.
  + Search and filter by specialization/days.
  + Edit experience and availability directly from the list view.
  + Patients are managed **inline inside each doctor profile**.
  + Bulk action available → *Mark as Experienced (10 years)*.
* Branding changes: custom **header, title, and index page**

16. Payment Integration Using Paytm Theory: • Introduction to integrating payment gateways (like Paytm) in Django projects.

Ans :

**Introduction**

In modern web applications, **payment gateways** are essential for handling secure financial transactions. Django supports payment gateway integration through APIs and SDKs provided by service providers like **Paytm, Razorpay, Stripe, and PayPal**.

**Why Paytm?**

* Widely used in India.
* Provides **Merchant APIs** for web and mobile apps.
* Supports **UPI, Wallet, Net Banking, Debit/Credit Cards**.
* Secure (uses **checksum/hash-based encryption**).

**How Payment Integration Works (Flow)**

1. **Customer initiates payment** → Selects an amount and clicks "Pay".
2. **Django Server creates an order** → Generates transaction/order ID.
3. **Redirect to Paytm Gateway** → Django app sends required details (Merchant ID, Order ID, Amount, Customer ID, etc.) with checksum.
4. **Customer completes payment** → Using UPI/Wallet/NetBanking.
5. **Paytm redirects back to Django app** → With transaction status (success/failure) and checksum for verification.
6. **Django verifies checksum** → Ensures payment authenticity using Paytm’s SDK.
7. **Transaction saved in DB** → Application updates order/payment status accordingly.

**Key Components in Django Paytm Integration**

* **Merchant ID & Key**: Provided by Paytm Dashboard after creating a merchant account.
* **Checksum Utility**: A digital signature to ensure security (Paytm provides paytmchecksum Python package).
* **Callback URL**: Django endpoint where Paytm sends the transaction response.
* **Transaction Status API**: Used to verify if the payment was successful.

**Steps to Integrate Paytm in Django**

1. **Install Paytm Checksum Library**
2. pip install paytmchecksum
3. **Configure Paytm Credentials** in settings.py:
4. PAYTM\_MERCHANT\_ID = "YourMerchantID"
5. PAYTM\_MERCHANT\_KEY = "YourMerchantKey"
6. PAYTM\_WEBSITE = "WEBSTAGING"
7. PAYTM\_CHANNEL\_ID = "WEB"
8. PAYTM\_INDUSTRY\_TYPE\_ID = "Retail"
9. PAYTM\_CALLBACK\_URL = "http://127.0.0.1:8000/callback/"
10. **Create Payment Order View**
    * Collect order details.
    * Generate checksum.
    * Redirect to Paytm Payment Gateway.
11. **Handle Callback URL (Response)**
    * Receive Paytm response (order ID, txn ID, status).
    * Verify checksum.
    * Update transaction status in database.

Lab: • Implement Paytm payment gateway in a Django project.

Ans :

**Step 1: Install Required Packages**

pip install paytmchecksum

**Step 2: Add Paytm Credentials in settings.py**

# settings.py

PAYTM\_MERCHANT\_ID = "YOUR\_MERCHANT\_ID"

PAYTM\_MERCHANT\_KEY = "YOUR\_MERCHANT\_KEY"

PAYTM\_WEBSITE = "WEBSTAGING" # For testing, use "DEFAULT" in production

PAYTM\_CHANNEL\_ID = "WEB"

PAYTM\_INDUSTRY\_TYPE\_ID = "Retail"

PAYTM\_CALLBACK\_URL = "http://127.0.0.1:8000/callback/"

PAYTM\_TXN\_URL = "https://securegw-stage.paytm.in/order/process" # staging URL

**Step 3: Create Payment Model**

In models.py:

from django.db import models

class Payment(models.Model):

order\_id = models.CharField(max\_length=100, unique=True)

amount = models.DecimalField(max\_digits=10, decimal\_places=2)

txn\_id = models.CharField(max\_length=100, blank=True, null=True)

status = models.CharField(max\_length=50, default="Pending")

created\_at = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return f"Order {self.order\_id} - {self.status}"

**Step 4: Create Views**

In views.py:

import uuid

import requests

from django.conf import settings

from django.shortcuts import render, redirect

from django.views.decorators.csrf import csrf\_exempt

from .models import Payment

import paytmchecksum

# Step 1: Payment Page

def initiate\_payment(request):

if request.method == "POST":

amount = request.POST.get("amount")

order\_id = str(uuid.uuid4())

# Save Payment Record

payment = Payment.objects.create(order\_id=order\_id, amount=amount)

# Create Paytm Params

paytm\_params = {

"MID": settings.PAYTM\_MERCHANT\_ID,

"ORDER\_ID": order\_id,

"CUST\_ID": "CUST001",

"TXN\_AMOUNT": str(amount),

"CHANNEL\_ID": settings.PAYTM\_CHANNEL\_ID,

"WEBSITE": settings.PAYTM\_WEBSITE,

"INDUSTRY\_TYPE\_ID": settings.PAYTM\_INDUSTRY\_TYPE\_ID,

"CALLBACK\_URL": settings.PAYTM\_CALLBACK\_URL,

}

checksum = paytmchecksum.generateSignature(paytm\_params, settings.PAYTM\_MERCHANT\_KEY)

paytm\_params["CHECKSUMHASH"] = checksum

return render(request, "paytm\_redirect.html", {"paytm\_params": paytm\_params, "txn\_url": settings.PAYTM\_TXN\_URL})

return render(request, "payment.html")

# Step 2: Callback URL

@csrf\_exempt

def callback(request):

received\_data = dict(request.POST.items())

paytm\_checksum = received\_data.pop("CHECKSUMHASH", None)

is\_valid\_checksum = paytmchecksum.verifySignature(received\_data, settings.PAYTM\_MERCHANT\_KEY, paytm\_checksum)

order\_id = received\_data.get("ORDERID")

payment = Payment.objects.get(order\_id=order\_id)

if is\_valid\_checksum:

if received\_data.get("RESPCODE") == "01": # Success

payment.status = "Success"

payment.txn\_id = received\_data.get("TXNID")

else:

payment.status = "Failed"

else:

payment.status = "Checksum Mismatch"

payment.save()

return render(request, "payment\_status.html", {"response": received\_data, "payment": payment})

**Step 5: Create Templates**

**templates/payment.html**

<h2>Make a Payment</h2>

<form method="POST">

{% csrf\_token %}

<label>Amount:</label>

<input type="text" name="amount" required>

<button type="submit">Pay with Paytm</button>

</form>

**templates/paytm\_redirect.html**

<h2>Redirecting to Paytm...</h2>

<form method="post" action="{{ txn\_url }}" name="paytm\_form">

{% for key, value in paytm\_params.items %}

<input type="hidden" name="{{ key }}" value="{{ value }}">

{% endfor %}

<noscript><input type="submit" value="Click here if not redirected"></noscript>

</form>

<script>document.paytm\_form.submit();</script>

**templates/payment\_status.html**

<h2>Payment Status</h2>

<p>Order ID: {{ payment.order\_id }}</p>

<p>Transaction ID: {{ payment.txn\_id }}</p>

<p>Status: {{ payment.status }}</p>

**Step 6: Add URLs**

In urls.py:

from django.urls import path

from . import views

urlpatterns = [

path("payment/", views.initiate\_payment, name="initiate\_payment"),

path("callback/", views.callback, name="callback"),

]

**Step 7: Run Server**

python manage.py makemigrations

python manage.py migrate

python manage.py runserver

* Visit **http://127.0.0.1:8000/payment/**
* Enter amount → Redirects to **Paytm Gateway**
* After payment → Returns to Django app with **status** updated in DB

Practical Example: 16) Write a Django project that integrates Paytm for handling payments in the "Doctor Finder" project.

Ans :  
**Step 1: Update Models**

In appointments/models.py (or wherever you store appointments):

from django.db import models

from django.contrib.auth.models import User

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialty = models.CharField(max\_length=100)

fee = models.DecimalField(max\_digits=10, decimal\_places=2)

def \_\_str\_\_(self):

return f"{self.name} - {self.specialty}"

class Appointment(models.Model):

patient = models.ForeignKey(User, on\_delete=models.CASCADE)

doctor = models.ForeignKey(Doctor, on\_delete=models.CASCADE)

date = models.DateTimeField()

is\_paid = models.BooleanField(default=False)

def \_\_str\_\_(self):

return f"{self.patient.username} → {self.doctor.name} on {self.date}"

class Payment(models.Model):

order\_id = models.CharField(max\_length=100, unique=True)

appointment = models.OneToOneField(Appointment, on\_delete=models.CASCADE)

amount = models.DecimalField(max\_digits=10, decimal\_places=2)

txn\_id = models.CharField(max\_length=100, blank=True, null=True)

status = models.CharField(max\_length=50, default="Pending")

created\_at = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return f"{self.order\_id} - {self.status}"

**Step 2: Add Paytm Settings**

In settings.py:

PAYTM\_MERCHANT\_ID = "YOUR\_MERCHANT\_ID"

PAYTM\_MERCHANT\_KEY = "YOUR\_MERCHANT\_KEY"

PAYTM\_WEBSITE = "WEBSTAGING" # "DEFAULT" in production

PAYTM\_CHANNEL\_ID = "WEB"

PAYTM\_INDUSTRY\_TYPE\_ID = "Retail"

PAYTM\_CALLBACK\_URL = "http://127.0.0.1:8000/payment/callback/"

PAYTM\_TXN\_URL = "https://securegw-stage.paytm.in/order/process"

**Step 3: Payment Views**

In appointments/views.py:

import uuid

import paytmchecksum

from django.conf import settings

from django.shortcuts import render, redirect, get\_object\_or\_404

from django.views.decorators.csrf import csrf\_exempt

from .models import Appointment, Payment

def initiate\_payment(request, appointment\_id):

appointment = get\_object\_or\_404(Appointment, id=appointment\_id)

order\_id = str(uuid.uuid4())

payment = Payment.objects.create(

order\_id=order\_id,

appointment=appointment,

amount=appointment.doctor.fee

)

paytm\_params = {

"MID": settings.PAYTM\_MERCHANT\_ID,

"ORDER\_ID": order\_id,

"CUST\_ID": str(request.user.id),

"TXN\_AMOUNT": str(appointment.doctor.fee),

"CHANNEL\_ID": settings.PAYTM\_CHANNEL\_ID,

"WEBSITE": settings.PAYTM\_WEBSITE,

"INDUSTRY\_TYPE\_ID": settings.PAYTM\_INDUSTRY\_TYPE\_ID,

"CALLBACK\_URL": settings.PAYTM\_CALLBACK\_URL,

}

checksum = paytmchecksum.generateSignature(paytm\_params, settings.PAYTM\_MERCHANT\_KEY)

paytm\_params["CHECKSUMHASH"] = checksum

return render(request, "appointments/paytm\_redirect.html", {"paytm\_params": paytm\_params, "txn\_url": settings.PAYTM\_TXN\_URL})

@csrf\_exempt

def payment\_callback(request):

received\_data = dict(request.POST.items())

paytm\_checksum = received\_data.pop("CHECKSUMHASH", None)

is\_valid\_checksum = paytmchecksum.verifySignature(received\_data, settings.PAYTM\_MERCHANT\_KEY, paytm\_checksum)

order\_id = received\_data.get("ORDERID")

payment = Payment.objects.get(order\_id=order\_id)

appointment = payment.appointment

if is\_valid\_checksum:

if received\_data.get("RESPCODE") == "01": # Success

payment.status = "Success"

payment.txn\_id = received\_data.get("TXNID")

appointment.is\_paid = True

appointment.save()

else:

payment.status = "Failed"

else:

payment.status = "Checksum Mismatch"

payment.save()

return render(request, "appointments/payment\_status.html", {"response": received\_data, "payment": payment})

**Step 4: Templates**

**appointments/templates/appointments/paytm\_redirect.html**

<h2>Redirecting to Paytm...</h2>

<form method="post" action="{{ txn\_url }}" name="paytm\_form">

{% for key, value in paytm\_params.items %}

<input type="hidden" name="{{ key }}" value="{{ value }}">

{% endfor %}

<noscript><input type="submit" value="Click here if not redirected"></noscript>

</form>

<script>document.paytm\_form.submit();</script>

**appointments/templates/appointments/payment\_status.html**

<h2>Payment Status</h2>

<p>Order ID: {{ payment.order\_id }}</p>

<p>Transaction ID: {{ payment.txn\_id }}</p>

<p>Status: {{ payment.status }}</p>

{% if payment.status == "Success" %}

<p>Appointment confirmed with Dr. {{ payment.appointment.doctor.name }}</p>

{% else %}

<p>Payment failed. Please try again.</p>

{% endif %}

**Step 5: URLs**

In appointments/urls.py:

from django.urls import path

from . import views

urlpatterns = [

path("pay/<int:appointment\_id>/", views.initiate\_payment, name="initiate\_payment"),

path("callback/", views.payment\_callback, name="payment\_callback"),

]

Include in main urls.py:

path("payment/", include("appointments.urls")),

Step 6: Workflow

Patient books an appointment.

Redirects to payment page → Pay with Paytm.

On success → Appointment.is\_paid = True.

Patient + doctor both see confirmed appointment.

Failed payments don’t confirm appointment.

17. GitHub Project Deployment Theory: • Steps to push a Django project to GITHUB

Ans :

**Theory**

Deploying a Django project to GitHub involves pushing your entire project code to a GitHub repository so it can be version-controlled, shared, and collaborated on.

**Steps to Push a Django Project to GitHub**

1. **Install Git**
   * Make sure Git is installed:
   * git --version
2. **Initialize Git in your Django Project Folder**  
   Navigate to your Django project root (where manage.py exists):
3. git init
4. **Create a .gitignore File**  
   To avoid pushing unnecessary files (like migrations, db, secrets):

.gitignore example for Django:

\*.pyc

\_\_pycache\_\_/

db.sqlite3

/venv/

/env/

.DS\_Store

\*.log

staticfiles/

media/

.env

1. **Add Project Files to Git**
2. git add .
3. **Commit the Files**
4. git commit -m "Initial commit - Django project"
5. **Create a Repository on GitHub**
   * Go to [GitHub](https://github.com) → New Repository
   * Give it a name (e.g., doctor-finder-django)
   * Leave it empty (don’t add README/License at this step if you already have files locally).
6. **Connect Local Project with GitHub Repo**
7. git remote add origin https://github.com/USERNAME/REPO\_NAME.git
8. **Push Project to GitHub**
9. git branch -M main
10. git push -u origin main
11. **Verify on GitHub**
    * Refresh your repository page.
    * You should now see your Django project hosted on GitHub.

Lab: • Deploy a Django project to GitHub for version control.

Ans :

**Steps**

**1. Initialize Git in Your Project**

Navigate to your Django project folder (where manage.py is located):

cd path/to/your/django\_project

git init

**2. Create a .gitignore File**

Inside the project root, create a .gitignore file to exclude unnecessary files:

**.gitignore**

\*.pyc

\_\_pycache\_\_/

db.sqlite3

/venv/

/env/

.DS\_Store

\*.log

staticfiles/

media/

.env

**3. Add Files to Git**

git add .

**4. Commit Files**

git commit -m "Initial commit - Django project"

**5. Create a Repository on GitHub**

* Go to [GitHub](https://github.com) → **New Repository**.
* Name it (e.g., doctor-finder-django).
* Don’t initialize with a README (to avoid conflicts).

**6. Connect Local Repo with GitHub**

git remote add origin https://github.com/YOUR\_USERNAME/doctor-finder-django.git

**7. Push the Project to GitHub**

git branch -M main

git push -u origin main

**8. Verify**

* Open your GitHub repository link in a browser.

Practical Example: 17) Write a step-by-step guide to deploying the “Doctor Finder” project to GitHub

Ans :

**Step 1: Prepare Your Project**

Make sure your **Doctor Finder** project is working locally.  
The structure may look like this:

doctor\_finder/

│── doctor\_finder/ # Main project folder

│── doctors/ # App (doctor profiles, appointments, etc.)

│── templates/ # HTML templates

│── static/ # Static files (CSS, JS)

│── manage.py

**Step 2: Initialize Git**

Open your terminal inside the project root (where manage.py is):

git init

**Step 3: Create .gitignore**

Inside the root folder, create a .gitignore file to exclude unnecessary files:

**.gitignore**

\*.pyc

\_\_pycache\_\_/

db.sqlite3

/venv/

/env/

.DS\_Store

\*.log

staticfiles/

media/

.env

**Step 4: Stage and Commit Files**

git add .

git commit -m "Initial commit - Doctor Finder project"

**Step 5: Create GitHub Repository**

1. Go to [GitHub](https://github.com).
2. Click **New Repository**.
3. Name it: doctor-finder-django.
4. Leave it empty (don’t add README or .gitignore).
5. Copy the repo URL (HTTPS). Example:
6. https://github.com/YOUR\_USERNAME/doctor-finder-django.git

**Step 6: Connect Local Repo with GitHub**

git remote add origin https://github.com/YOUR\_USERNAME/doctor-finder-django.git

**Step 7: Push Project to GitHub**

git branch -M main

git push -u origin main

**Step 8: Verify on GitHub**

* Open your GitHub repository link.
* You should now see your **Doctor Finder Django project** uploaded.

18. Live Project Deployment (PythonAnywhere) Theory: • Introduction to deploying Django projects to live servers like PythonAnywhere.

Ans :

**Theory**

Once your Django project is version-controlled on GitHub, the next step is to **deploy it to a live server** so others can access it online.  
**PythonAnywhere** is a popular platform that provides free and paid hosting for Python-based web apps, including Django.

**Why Deploy to PythonAnywhere?**

* ✅ Free tier available for small projects.
* ✅ Pre-installed Python and Django support.
* ✅ Simple web-based interface (no server setup required).
* ✅ Easy integration with GitHub for deployment.

**General Steps in Deployment**

1. **Push Project to GitHub** (already done in Step 17).
2. **Create a PythonAnywhere Account** → pythonanywhere.com.
3. **Clone Project from GitHub** inside PythonAnywhere.
4. **Create a Virtual Environment** and install dependencies (requirements.txt).
5. **Configure Database** (default: SQLite; can also connect to MySQL).
6. **Collect Static Files**:
7. python manage.py collectstatic
8. **Configure WSGI file** to point to Django’s wsgi.py.
9. **Set Up Web App** inside PythonAnywhere’s **Web tab** → Choose manual configuration → Link to your project.
10. **Reload the App** and test it online at your-username.pythonanywhere.com.

**Key Points**

* You need a **requirements.txt** file so PythonAnywhere knows which packages to install.
* pip freeze > requirements.txt
* Django settings may need adjustments:
  + Set DEBUG = False in settings.py.
  + Add your domain to ALLOWED\_HOSTS. Example:
  + ALLOWED\_HOSTS = ['your-username.pythonanywhere.com']
* Free tier allows only **1 web app** and limited CPU usage.

Lab: • Deploy a Django project to PythonAnywhere.

Ans :

**Objective**

To take a Django project (e.g., **Doctor Finder**) live on PythonAnywhere so it is accessible via a public URL.

**Step 1: Push Your Project to GitHub**

Make sure your Django project is already uploaded to GitHub (done in Step 17).  
Also create a **requirements.txt** file:

pip freeze > requirements.txt

**Step 2: Create a PythonAnywhere Account**

1. Go to PythonAnywhere.
2. Create a **free account** (for learning/testing).

**Step 3: Open a Bash Console on PythonAnywhere**

* Log in → Go to **Dashboard → Consoles → Bash**.

**Step 4: Clone Your Project from GitHub**

Inside the console, run:

git clone https://github.com/YOUR\_USERNAME/doctor-finder-django.git

cd doctor-finder-django

**Step 5: Create a Virtual Environment**

python3.10 -m venv venv

source venv/bin/activate

pip install -r requirements.txt

**Step 6: Configure Django Settings**

* Open your settings.py.
* Change:

DEBUG = False

ALLOWED\_HOSTS = ['your-username.pythonanywhere.com']

**Step 7: Apply Migrations**

python manage.py migrate

python manage.py collectstatic

Press **yes** when asked to overwrite.

**Step 8: Configure the Web App in PythonAnywhere**

1. Go to the **Web tab** in PythonAnywhere.
2. Click **Add a new web app** → Choose **Manual configuration** → Select **Python 3.10** (or your version).
3. Set the **source code path** to your project directory.
4. Point the **WSGI configuration file** to your project’s wsgi.py. Example:
5. /home/username/doctor-finder-django/doctor\_finder/wsgi.py

**Step 9: Reload the App**

* In the **Web tab**, click **Reload**.

**Step 10: Access Your Site**

Open:

https://your-username.pythonanywhere.com

Practical Example: 18) Write a Django project and deploy it on PythonAnywhere, making it accessible online

Ans :

**Step 1: Create Django Project Locally**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctors

**doctor\_finder/settings.py**

Add the app:

INSTALLED\_APPS = [

...

'doctors',

]

**doctors/models.py**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialty = models.CharField(max\_length=100)

availability = models.CharField(max\_length=50)

def \_\_str\_\_(self):

return self.name

**doctors/views.py**

from django.shortcuts import render

from .models import Doctor

def doctor\_list(request):

doctors = Doctor.objects.all()

return render(request, "doctors/doctor\_list.html", {"doctors": doctors})

**doctors/templates/doctors/doctor\_list.html**

<!DOCTYPE html>

<html>

<head>

<title>Doctor Finder</title>

</head>

<body>

<h1>Available Doctors</h1>

<ul>

{% for doctor in doctors %}

<li>{{ doctor.name }} - {{ doctor.specialty }} ({{ doctor.availability }})</li>

{% endfor %}

</ul>

</body>

</html>

**doctor\_finder/urls.py**

from django.contrib import admin

from django.urls import path

from doctors.views import doctor\_list

urlpatterns = [

path('admin/', admin.site.urls),

path('', doctor\_list, name='doctor\_list'),

]

**Run locally**

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

Visit → http://127.0.0.1:8000 ✅ Doctors list working locally.

**Step 2: Push Project to GitHub**

echo "venv/" > .gitignore

pip freeze > requirements.txt

git init

git add .

git commit -m "Initial Doctor Finder project"

git remote add origin https://github.com/YOUR\_USERNAME/doctor-finder-django.git

git push -u origin main

**Step 3: Deploy on PythonAnywhere**

1. **Sign up** at pythonanywhere.com.
2. **Open a Bash console**.
3. Clone repo:
4. git clone https://github.com/YOUR\_USERNAME/doctor-finder-django.git
5. cd doctor-finder-django

**Step 4: Setup Virtual Environment**

python3.10 -m venv venv

source venv/bin/activate

pip install -r requirements.txt

**Step 5: Configure Django for Deployment**

In **settings.py**:

DEBUG = False

ALLOWED\_HOSTS = ['your-username.pythonanywhere.com']

**Step 6: Run Migrations and Collect Static Files**

python manage.py migrate

python manage.py collectstatic

**Step 7: Configure Web App in PythonAnywhere**

1. Go to **Web tab** → Add New Web App → Manual config → Python 3.10.
2. Set **working directory** to your project folder.
3. Edit the **WSGI file** to point to:
4. /home/your-username/doctor-finder-django/doctor\_finder/wsgi.py

**Step 8: Reload & Test**

* Click **Reload** in Web tab.
* Open in browser:
* https://your-username.pythonanywhere.com

You should see the **Doctor Finder app live** 🎉

19. Social Authentication Theory: • Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.

Ans :

**Theory**

**🔹 What is Social Authentication?**

Social authentication allows users to log in to your Django application using their **social media accounts** (Google, Facebook, GitHub, etc.) instead of creating a new username/password.

It is implemented using the **OAuth2 protocol**, which works like this:

1. User clicks “Login with Google/Facebook”.
2. The app redirects the user to the provider’s (Google/Facebook) login page.
3. User grants permission → provider redirects back with an **access token**.
4. Django uses that token to authenticate the user and create a session.

**🔹 Why Use Social Login?**

* ✅ Easier & faster signup (no need to remember passwords).
* ✅ Increases trust and reduces friction.
* ✅ Provides verified emails directly from providers.
* ✅ Reduces chances of fake accounts.

**🔹 Popular Django Libraries for Social Login**

1. **django-allauth** → Most popular, supports multiple providers (Google, Facebook, GitHub, Twitter, etc.).
2. **social-auth-app-django** → Flexible but requires more setup.

⚡ **django-allauth** is recommended for beginners and widely used in production.

**🔹 Basic Setup with django-allauth**

1. Install:
2. pip install django-allauth
3. Add to **settings.py**:
4. INSTALLED\_APPS = [
5. ...
6. 'django.contrib.sites',
7. 'allauth',
8. 'allauth.account',
9. 'allauth.socialaccount',
10. 'allauth.socialaccount.providers.google',
11. 'allauth.socialaccount.providers.facebook',
12. ]
13. SITE\_ID = 1
14. AUTHENTICATION\_BACKENDS = [
15. 'django.contrib.auth.backends.ModelBackend',
16. 'allauth.account.auth\_backends.AuthenticationBackend',
17. ]
18. Add **URLs**:
19. from django.urls import path, include
20. urlpatterns = [
21. path('accounts/', include('allauth.urls')),
22. ]
23. Run migrations:
24. python manage.py migrate
25. Configure **OAuth Credentials**:
    * Go to Google/Facebook developer console.
    * Create a project → Get **Client ID** and **Secret Key**.
    * Add them in Django admin → **Social Applications**.
26. Now users can log in via Google/Facebook at:
27. /accounts/login/

**Lab**

**Lab Task:**

**Implement Google and Facebook login for the Django project.**

**Steps:**

1. Install django-allauth.
2. Update settings.py and urls.py as shown above.
3. Run python manage.py migrate.
4. Go to **Google Developer Console** → create **OAuth client ID** → set redirect URI:
5. http://127.0.0.1:8000/accounts/google/login/callback/
6. Go to **Facebook Developers Console** → create app → enable Facebook login → set redirect URI:
7. http://127.0.0.1:8000/accounts/facebook/login/callback/
8. In **Django Admin → Social Applications**:
   * Add Google → enter Client ID and Secret.
   * Add Facebook → enter App ID and Secret.
9. Restart server → Visit /accounts/login/ → You will see **Login with Google** and **Login with Facebook** buttons.

Lab: • Implement Google and Facebook login for the Django project

Ans :

**Step 1: Install Dependencies**

pip install django-allauth

**Step 2: Update settings.py**

INSTALLED\_APPS = [

# Default Django apps

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'django.contrib.sites',

# Allauth apps

'allauth',

'allauth.account',

'allauth.socialaccount',

# Providers

'allauth.socialaccount.providers.google',

'allauth.socialaccount.providers.facebook',

]

# Site ID (needed for django-allauth)

SITE\_ID = 1

# Authentication backends

AUTHENTICATION\_BACKENDS = [

'django.contrib.auth.backends.ModelBackend',

'allauth.account.auth\_backends.AuthenticationBackend',

]

# Login settings

LOGIN\_REDIRECT\_URL = '/'

LOGOUT\_REDIRECT\_URL = '/'

**Step 3: Add URLs in urls.py**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('accounts/', include('allauth.urls')), # Allauth routes

]

Now /accounts/login/ will show login options.

**Step 4: Run Migrations**

python manage.py migrate

**Step 5: Configure Google OAuth**

1. Go to Google Cloud Console.
2. Create a project → Enable **OAuth consent screen**.
3. Create **OAuth client ID** (Web application).
4. Set **Authorized redirect URI**:
5. http://127.0.0.1:8000/accounts/google/login/callback/
6. Copy **Client ID** and **Client Secret**.

**Step 6: Configure Facebook OAuth**

1. Go to Facebook Developers Console.
2. Create an app → Add **Facebook Login** product.
3. Set **Valid OAuth Redirect URI**:
4. http://127.0.0.1:8000/accounts/facebook/login/callback/
5. Copy **App ID** and **App Secret**.

**Step 7: Add Social Applications in Django Admin**

1. Run:
2. python manage.py createsuperuser
3. Log in to /admin/.
4. Under **Social Applications** → Add:
   * **Google** → enter Client ID & Secret.
   * **Facebook** → enter App ID & Secret.
   * Select **Sites → example.com** (or your domain).

**Step 8: Test the Login**

* Visit:
* http://127.0.0.1:8000/accounts/login/
* You will see **Login with Google** and **Login with Facebook**.
* After login, you are redirected to / (homepage).

Practical Example: 19) Write a Django project to allow users to log in using Google or Facebook.

Ans :

**Step 1: Start a Django Project**

django-admin startproject social\_login\_project

cd social\_login\_project

python manage.py startapp accounts

**Step 2: Install Required Package**

pip install django-allauth

**Step 3: Update settings.py**

INSTALLED\_APPS = [

# Default Django apps

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'django.contrib.sites',

# Allauth

'allauth',

'allauth.account',

'allauth.socialaccount',

# Providers

'allauth.socialaccount.providers.google',

'allauth.socialaccount.providers.facebook',

# Local app

'accounts',

]

SITE\_ID = 1

AUTHENTICATION\_BACKENDS = [

'django.contrib.auth.backends.ModelBackend',

'allauth.account.auth\_backends.AuthenticationBackend',

]

LOGIN\_REDIRECT\_URL = '/'

LOGOUT\_REDIRECT\_URL = '/'

**Step 4: Add URLs**

soclal\_login\_project/urls.py

from django.contrib import admin

from django.urls import path, include

from accounts.views import home

urlpatterns = [

path('admin/', admin.site.urls),

path('', home, name="home"),

path('accounts/', include('allauth.urls')), # Allauth URLs

]

**Step 5: Create a Simple View**

accounts/views.py

from django.shortcuts import render

from django.contrib.auth.decorators import login\_required

def home(request):

return render(request, "home.html")

**Step 6: Create a Template**

Create templates/home.html:

<!DOCTYPE html>

<html>

<head>

<title>Social Login Example</title>

</head>

<body>

<h1>Welcome to Social Login Project</h1>

{% if user.is\_authenticated %}

<p>Hello, {{ user.username }}!</p>

<a href="{% url 'account\_logout' %}">Logout</a>

{% else %}

<a href="{% url 'account\_login' %}">Login</a>

{% endif %}

</body>

</html>

**Step 7: Run Migrations**

python manage.py migrate

python manage.py createsuperuser

**Step 8: Configure Google Login**

1. Go to Google Cloud Console.
2. Create OAuth credentials → Add redirect URI:
3. http://127.0.0.1:8000/accounts/google/login/callback/
4. Copy **Client ID** & **Secret**.

**Step 9: Configure Facebook Login**

1. Go to Facebook Developer Console.
2. Create an App → Add **Facebook Login**.
3. Set redirect URI:
4. http://127.0.0.1:8000/accounts/facebook/login/callback/
5. Copy **App ID** & **Secret**.

**Step 10: Add Social Apps in Django Admin**

1. Run python manage.py runserver.
2. Log in to /admin/.
3. Under **Social Applications** → Add **Google** and **Facebook** providers.
   * Paste **Client ID & Secret**.
   * Select **Site: example.com**.

**Step 11: Test the Project**

1. Visit → http://127.0.0.1:8000/.
2. Click **Login**.
3. Choose **Google** or **Facebook**.
4. Authenticate → You’ll be redirected back and logged in.

20. Google Maps API Theory: • Integrating Google Maps API into Django projects.

Ans :

**Introduction**

The **Google Maps API** allows developers to embed interactive maps, location markers, routes, and geolocation features into web applications. In Django, it can be integrated to display doctor clinic locations, patient addresses, or directions in real-time.

**Key Features of Google Maps API**

1. **Maps Embed API** – Embed simple maps using <iframe>.
2. **Maps JavaScript API** – Add dynamic and interactive maps with markers and routes.
3. **Geocoding API** – Convert addresses into latitude/longitude (and vice versa).
4. **Places API** – Find nearby hospitals, doctors, or pharmacies.
5. **Directions API** – Show travel directions between two points.

**Steps to Integrate Google Maps API in Django**

1. **Get API Key**
   * Visit Google Cloud Console.
   * Enable **Maps JavaScript API** (and optional: Places, Directions, Geocoding).
   * Generate an **API Key**.
2. **Add API Key to Django**
   * Store securely in settings.py (or .env file).
3. GOOGLE\_MAPS\_API\_KEY = "your\_api\_key\_here"
4. **Pass API Key to Templates**
   * Example:
5. from django.conf import settings
6. def map\_view(request):
7. return render(request, "map.html", {"google\_maps\_api\_key": settings.GOOGLE\_MAPS\_API\_KEY})
8. **Use JavaScript in Template**
   * Embed Google Maps:
9. <div id="map" style="height:400px; width:100%;"></div>
10. <script src="https://maps.googleapis.com/maps/api/js?key={{ google\_maps\_api\_key }}"></script>
11. <script>
12. function initMap() {
13. var clinic = {lat: 23.0225, lng: 72.5714}; // Example: Ahmedabad
14. var map = new google.maps.Map(document.getElementById('map'), {
15. zoom: 14,
16. center: clinic
17. });
18. var marker = new google.maps.Marker({position: clinic, map: map});
19. }
20. window.onload = initMap;
21. </script>

**Use Cases in Django Projects**

* **Doctor Finder App** → Show doctors’ clinic locations.
* **E-commerce** → Display delivery location tracking.
* **Logistics** → Real-time vehicle movement on maps.

Lab: • Use Google Maps API to display doctor locations in the "Doctor Finder" project.

Ans :

**Step 1: Get Google Maps API Key**

1. Go to Google Cloud Console.
2. Enable **Maps JavaScript API** (and optionally **Geocoding API** if you want to use addresses).
3. Create credentials → **API Key**.
4. Copy the API key (we’ll use it in Django).

**Step 2: Add API Key to settings.py**

# settings.py

GOOGLE\_MAPS\_API\_KEY = "your\_api\_key\_here"

**Step 3: Update Models (models.py)**

Each doctor has a clinic location (latitude & longitude).

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialty = models.CharField(max\_length=100)

latitude = models.FloatField()

longitude = models.FloatField()

def \_\_str\_\_(self):

return self.name

Run:

python manage.py makemigrations

python manage.py migrate

**Step 4: Create View (views.py)**

from django.shortcuts import render

from django.conf import settings

from .models import Doctor

def doctor\_map(request):

doctors = Doctor.objects.all()

return render(request, "doctor\_map.html", {

"doctors": doctors,

"google\_maps\_api\_key": settings.GOOGLE\_MAPS\_API\_KEY

})

**Step 5: Configure URL (urls.py)**

from django.urls import path

from . import views

urlpatterns = [

path("map/", views.doctor\_map, name="doctor\_map"),

]

**Step 6: Create Template (doctor\_map.html)**

<!DOCTYPE html>

<html>

<head>

<title>Doctor Locations</title>

<style>

#map {

height: 500px;

width: 100%;

}

</style>

</head>

<body>

<h2>Doctor Locations on Map</h2>

<div id="map"></div>

<!-- Google Maps API -->

<script src="https://maps.googleapis.com/maps/api/js?key={{ google\_maps\_api\_key }}"></script>

<script>

function initMap() {

var map = new google.maps.Map(document.getElementById('map'), {

zoom: 12,

center: {lat: 23.0225, lng: 72.5714} // Default center (Ahmedabad)

});

{% for doctor in doctors %}

var marker = new google.maps.Marker({

position: {lat: {{ doctor.latitude }}, lng: {{ doctor.longitude }}},

map: map,

title: "{{ doctor.name }} ({{ doctor.specialty }})"

});

{% endfor %}

}

window.onload = initMap;

</script>

</body>

</html>

**Step 7: Add Sample Doctor Data**

Open Django shell:

python manage.py shell

from app.models import Doctor

Doctor.objects.create(name="Dr. Sharma", specialty="Cardiologist", latitude=23.0338, longitude=72.5850)

Doctor.objects.create(name="Dr. Patel", specialty="Dermatologist", latitude=23.0205, longitude=72.5714)

**Step 8: Run the Project**

python manage.py runserver

Visit:  
👉 http://127.0.0.1:8000/map/

You’ll see Google Maps with markers for each doctor’s client

Practical Example: 20) Write a Django project to display doctor locations using Google Maps API.

Ans :

**Step 1: Create Django Project**

django-admin startproject doctor\_finder

cd doctor\_finder

python manage.py startapp doctors

Add doctors app in **settings.py**:

INSTALLED\_APPS = [

...,

'doctors',

]

**Step 2: Add Google Maps API Key in settings.py**

GOOGLE\_MAPS\_API\_KEY = "your\_google\_maps\_api\_key\_here"

**Step 3: Define Doctor Model (doctors/models.py)**

from django.db import models

class Doctor(models.Model):

name = models.CharField(max\_length=100)

specialty = models.CharField(max\_length=100)

latitude = models.FloatField()

longitude = models.FloatField()

def \_\_str\_\_(self):

return f"{self.name} - {self.specialty}"

Apply migrations:

python manage.py makemigrations

python manage.py migrate

**Step 4: Register in Admin (doctors/admin.py)**

from django.contrib import admin

from .models import Doctor

@admin.register(Doctor)

class DoctorAdmin(admin.ModelAdmin):

list\_display = ("name", "specialty", "latitude", "longitude")

Now, add doctors via **Django Admin Panel**.

**Step 5: Create View (doctors/views.py)**

from django.shortcuts import render

from django.conf import settings

from .models import Doctor

def doctor\_map(request):

doctors = Doctor.objects.all()

return render(request, "doctor\_map.html", {

"doctors": doctors,

"google\_maps\_api\_key": settings.GOOGLE\_MAPS\_API\_KEY

})

**Step 6: Configure URL (doctor\_finder/urls.py)**

from django.contrib import admin

from django.urls import path

from doctors import views

urlpatterns = [

path('admin/', admin.site.urls),

path('map/', views.doctor\_map, name='doctor\_map'),

]

**Step 7: Create Template (doctors/templates/doctor\_map.html)**

<!DOCTYPE html>

<html>

<head>

<title>Doctor Locations</title>

<style>

#map {

height: 500px;

width: 100%;

}

</style>

</head>

<body>

<h2>Doctor Locations on Map</h2>

<div id="map"></div>

<!-- Google Maps API Script -->

<script src="https://maps.googleapis.com/maps/api/js?key={{ google\_maps\_api\_key }}"></script>

<script>

function initMap() {

var map = new google.maps.Map(document.getElementById('map'), {

zoom: 12,

center: {lat: 23.0225, lng: 72.5714} // Default center (Ahmedabad)

});

{% for doctor in doctors %}

var marker = new google.maps.Marker({

position: {lat: {{ doctor.latitude }}, lng: {{ doctor.longitude }}},

map: map,

title: "{{ doctor.name }} ({{ doctor.specialty }})"

});

{% endfor %}

}

window.onload = initMap;

</script>

</body>

</html>

**Step 8: Add Doctor Data**

Open Django shell:

python manage.py shell

from doctors.models import Doctor

Doctor.objects.create(name="Dr. Sharma", specialty="Cardiologist", latitude=23.0338, longitude=72.5850)

Doctor.objects.create(name="Dr. Patel", specialty="Dermatologist", latitude=23.0205, longitude=72.5714)

**Step 9: Run the Project**

python manage.py runserver