Module 16) Python DB and Framework

1. HTML in Python (Using Django)

Theory

- HTML can be embedded in Python applications using **web frameworks** like **Django** or **Flask**.
- **Django** uses a powerful **template system** that allows dynamic data to be passed from Python views to HTML files.
- Django templates support:
    - Variables ({{ }})
    - Template tags ({% %})
- This separation of logic (Python) and presentation (HTML) makes web applications clean and maintainable.

Lab Program

Python program to render an HTML file using Django's template system

Step 1: Create Django Project

```
django-admin startproject doctorfinder

cd doctorfinder

python manage.py startapp home
```

Step 2: Add App in settings.py

```
INSTALLED_APPS = [
    'home',
]
```

Step 3: Create View (home/views.py)

```
from django.shortcuts import render


def index(request):
    return render(request, "index.html")
```

Step 4: URL Configuration

home/urls.py

```
from django.urls import path
from .views import index

urlpatterns = [
    path('', index, name='home'),
]
```

doctorfinder/urls.py

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

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

Step 5: Create Template

templates/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Doctor Finder</title>
</head>
<body>
    <h1>Welcome to Doctor Finder</h1>
</body>
</html>
```

Practical Example 1

Django project that renders "Welcome to Doctor Finder" on the home page

When the server runs and the home URL is opened, the page displays:

Welcome to Doctor Finder

2. CSS in Python (Django Templates)

Theory

- Django allows CSS integration using **static files**.
- Static files include:
    - CSS
    - JavaScript
    - Images
- Django uses the {% static %} template tag to link static files.
- CSS files are stored inside the **static** directory of an app.

Lab Program

Create a CSS file to style a basic HTML template in Django

Step 1: Create Static Folder Structure

```
home/
── static/
   └── css/
      └── style.css
```

Step 2: CSS File (style.css)

```
body {
   background-color: #f2f2f2;
   font-family: Arial;
}

h1 {
   color: #2c3e50;
   text-align: center;
}
```

Step 3: Load Static Files in HTML

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
   <title>Doctor Profile</title>
   <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
   <h1>Doctor Profile</h1>
</body>
</html>
```

Practical Example 2

Django project to display a webpage with custom CSS styling for a doctor profile page

Doctor Profile Template (doctor.html)

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
   <title>Doctor Profile</title>
   <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
   <h1>Dr. John Smith</h1>
   <p>Specialization: Cardiologist</p>
   <p>Experience: 10 Years</p>
</body>
</html>
```

View (views.py)

```
from django.shortcuts import render
def doctor_profile(request):
```

```
    return render(request, "doctor.html")
```

Output

- A **styled doctor profile webpage**
- Background color applied
- Text formatted using CSS

One-Line Exam Conclusion

Django allows dynamic HTML rendering using templates and supports CSS integration through static files for styling web pages.

Below are **neatly arranged, copy-paste-ready answers** written in **lab record / exam format**, consistent with your previous sections.

3. JavaScript with Python (Django)

Theory

- JavaScript is used in Django applications for **client-side interactivity** such as:
    - Form validation
    - Dynamic content updates
    - User interaction without reloading the page
- JavaScript files can be:
    - Embedded directly in Django templates
    - Linked as external static files
- Django uses the {% static %} tag to include JavaScript files.

Lab Program

Django project with JavaScript-enabled form validation

Step 1: Create Django Project & App

```
django-admin startproject hospital

cd hospital

python manage.py startapp patient
```

Step 2: Register App in settings.py

```
INSTALLED_APPS = [
    'patient',
]
```

Step 3: Create View (patient/views.py)

```
from django.shortcuts import render

def register(request):
    return render(request, "register.html")
```

Step 4: URL Configuration

patient/urls.py

```
from django.urls import path
from .views import register

urlpatterns = [
    path('', register, name='register'),
]
```

hospital/urls.py

```
from django.urls import path, include

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

Step 5: JavaScript File

static/js/validate.js

```
function validateForm() {
    let name = document.forms["regForm"]["name"].value;
    let age = document.forms["regForm"]["age"].value;

    if (name === "" || age === "") {
        alert("All fields are required");
        return false;
    }
    if (age < 0) {
        alert("Invalid age");
```

```
        return false;
    }
    return true;
}
```

Step 6: HTML Template

templates/register.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Patient Registration</title>
    <script src="{% static 'js/validate.js' %}"></script>
</head>
<body>
    <h2>Patient Registration</h2>
    <form name="regForm" onsubmit="return validateForm()">
        Name: <input type="text" name="name"><br><br>
        Age: <input type="number" name="age"><br><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

Practical Example 3

Django project using JavaScript to validate a patient registration form

- JavaScript checks empty fields and invalid age
- Validation happens **before form submission**
- Improves user experience

4. Django Introduction

Theory

Overview of Django

- Django is a **high-level Python web framework** used to build secure and scalable web applications quickly.
- It follows the **Model-View-Template (MVT)** architecture.

Advantages of Django

- Built-in security features
- Scalable and robust
- Fast development using reusable components
- Built-in admin panel

Django vs Flask

Feature	**Django**	**Flask**
Framework Type	Full-stack	Micro-framework
Built-in Tools	Yes	Minimal
Learning Curve	Moderate	Easy
Best For	Large projects	Small applications

Lab Program

Short Django project to render a simple webpage

View (views.py)

```
from django.shortcuts import render


def home(request):
    return render(request, "home.html")
```

Template (home.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Django Page</title>
```

```
</head>
<body>
  <h1>Hello from Django!</h1>
</body>
</html>
```

Practical Example 4

Python program to create a Django project and understand its directory structure

Create Project

django-admin startproject mysite

Directory Structure

```
mysite/
|
├── manage.py
├── mysite/
|   ├── __init__.py
|   ├── settings.py
|   ├── urls.py
|   ├── asgi.py
|   └── wsgi.py
```

Directory Explanation

- manage.py → Command-line utility
- settings.py → Project configuration
- urls.py → URL routing
- wsgi.py → Web server gateway
- asgi.py → Asynchronous server gateway

One-Line Exam Conclusion

Django is a powerful Python web framework that simplifies web development using built-in tools, templates, and security features.

5. Virtual Environment

Theory

- A **virtual environment** is an isolated Python environment used to manage project-specific dependencies.
- It prevents conflicts between different project libraries and Python versions.
- Virtual environments make projects **portable, clean, and easy to maintain**.
- Common tools used:
    - venv (built-in Python module)
    - virtualenv (external package)

Lab Program

Set up a virtual environment for a Django project

Step 1: Create Virtual Environment

python -m venv myenv

Step 2: Activate Virtual Environment

Windows

myenv\Scripts\activate

Linux / macOS

source myenv/bin/activate

Step 3: Install Django

pip install django

Step 4: Verify Installation

django-admin --version

Practical Example 5

Python commands to create and activate a virtual environment and install Django

python -m venv venv

venv\Scripts\activate

pip install django

One-Line Exam Answer

A virtual environment is used to isolate project dependencies and avoid version conflicts.

6. Project and App Creation

Theory

- A **Django project** is the overall web application configuration.
- A **Django app** is a module that performs a specific function.
- Important files:
    - manage.py → Command-line utility for Django tasks
    - urls.py → Maps URLs to views
    - views.py → Contains request-handling logic

Lab Program

Create a Django project with an app to manage doctor profiles

Step 1: Create Django Project

django-admin startproject hospital

cd hospital

Step 2: Create App

python manage.py startapp doctor

Step 3: Register App in settings.py

```
INSTALLED_APPS = [
    'doctor',
]
```

Step 4: Create View (doctor/views.py)

```
from django.shortcuts import render
```

```
def doctor_home(request):
    return render(request, "doctor.html")
```

Step 5: URL Configuration

doctor/urls.py

```
from django.urls import path
from .views import doctor_home

urlpatterns = [
    path('', doctor_home),
]
```

hospital/urls.py

```
from django.urls import path, include

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

Step 6: Create Template

templates/doctor.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Doctor Profile</title>
</head>
<body>
    <h1>Doctor Profile Page</h1>
</body>
</html>
```

Practical Example 6

Python commands to create a Django project and a new app called doctor

django-admin startproject clinic

cd clinic

python manage.py startapp doctor

python manage.py runserver

One-Line Exam Conclusion

Django projects organize applications using modular apps that handle specific functionality such as doctor profile management.

7. MVT Pattern Architecture

Theory

- Django follows the **MVT (Model–View–Template)** architecture.
- **Model**
    - Defines the database structure using Python classes.
- **View**
    - Handles business logic and processes HTTP requests.
- **Template**
    - Defines how data is displayed using HTML.
- **Request–Response Cycle**:

1. User sends a request (URL)
2. URL is mapped to a View
3. View interacts with Model (if required)
4. Data is passed to Template
5. Response is returned to the browser

Lab Program

Build a simple Django app showcasing MVT architecture

Step 1: Create Project and App

django-admin startproject doctorfinder

cd doctorfinder

python manage.py startapp doctor

Step 2: Create 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.IntegerField()

    def __str__(self):
        return self.name
```

Step 3: Register App in settings.py

```
INSTALLED_APPS = [
    'doctor',
]
```

Step 4: Create 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})
```

Step 5: URL Configuration

doctor/urls.py

```
from django.urls import path
from .views import doctor_list

urlpatterns = [
```

```
    path('', doctor_list),
]
```

doctorfinder/urls.py

```
from django.urls import path, include

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

Step 6: Template (templates/doctor_list.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Doctor List</title>
</head>
<body>
    <h1>Doctors Information</h1>
    <ul>
        {% for doctor in doctors %}
            <li>
                {{ doctor.name }} - {{ doctor.specialization }} ({{ doctor.experience }} Years)
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Practical Example 7

Django project using MVT to display doctor information

- Model stores doctor data
- View fetches data

- Template displays doctor information

8. Django Admin Panel

Theory

- Django provides a **built-in admin panel** to manage application data.
- It allows:
    - Create, update, delete records
    - Manage users and permissions
- The admin interface can be customized using admin.py.

Lab Program

Set up and customize the Django admin panel for a "Doctor Finder" project

Step 1: Create Superuser

python manage.py createsuperuser

Step 2: Register Model in Admin (doctor/admin.py)

```
from django.contrib import admin
from .models import Doctor

class DoctorAdmin(admin.ModelAdmin):
    list_display = ('name', 'specialization', 'experience')
    search_fields = ('name', 'specialization')

admin.site.register(Doctor, DoctorAdmin)
```

Step 3: Run Server

python manage.py runserver

Step 4: Access Admin Panel

http://127.0.0.1:8000/admin/

Practical Example 8

Django project with admin panel and custom fields for managing doctor information

- Admin panel shows:
    - Doctor Name
    - Specialization
    - Experience
- Search and filter options enabled

One-Line Exam Conclusion

Django's MVT architecture cleanly separates data, logic, and presentation, while the admin panel provides an efficient interface for managing application data.

9. URL Patterns and Template Integration

Theory

- Django uses **URL patterns** in urls.py to map browser requests to specific views.
- Each URL pattern is connected to a **view function**.
- Views interact with **templates** to render dynamic HTML content.
- This allows navigation between different pages in a web application.

Lab Program

Create a Django project with URL patterns, views, and templates

Step 1: Create Project and App

```
django-admin startproject doctorfinder

cd doctorfinder

python manage.py startapp doctor
```

Step 2: Register App in settings.py

```
INSTALLED_APPS = [
    'doctor',
]
```

Step 3: Create Views (doctor/views.py)

```
from django.shortcuts import render

def home(request):
    return render(request, "home.html")

def profile(request):
    return render(request, "profile.html")

def contact(request):
    return render(request, "contact.html")
```

Step 4: URL Configuration

doctor/urls.py

```
from django.urls import path
from .views import home, profile, contact

urlpatterns = [
    path('', home, name='home'),
    path('profile/', profile, name='profile'),
    path('contact/', contact, name='contact'),
]
```

doctorfinder/urls.py

```
from django.urls import path, include

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

Step 5: Templates

templates/home.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Finder</title>
</head>
<body>
  <h1>Welcome to Doctor Finder</h1>
  <a href="/profile/">Profile</a> |
  <a href="/contact/">Contact</a>
</body>
</html>
```

templates/profile.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Profile</title>
</head>
<body>
  <h2>Dr. John Smith</h2>
  <p>Specialization: Cardiologist</p>
  <a href="/">Home</a>
</body>
</html>
```

templates/contact.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Contact</title>
```

```
</head>
<body>
  <h2>Contact Us</h2>
  <p>Email: contact@doctorfinder.com</p>
  <a href="/">Home</a>
</body>
</html>
```

Practical Example 9

Doctor Finder site navigation using URL routing

- Home page → /
- Profile page → /profile/
- Contact page → /contact/
- URLs route requests to views and templates

10. Form Validation using JavaScript

Theory

- **JavaScript** is used for **client-side form validation** to prevent incorrect data submission.
- Validation improves user experience by providing immediate feedback.
- JavaScript validation occurs before data is sent to the server.

Lab Program

Django project implementing JavaScript form validation

Step 1: View (doctor/views.py)

```
from django.shortcuts import render

def register(request):
  return render(request, "register.html")
```

Step 2: URL Configuration (doctor/urls.py)

```
path('register/', register, name='register'),
```

Step 3: JavaScript File

static/js/validate.js

```
function validateForm() {
    let email = document.forms["regForm"]["email"].value;
    let phone = document.forms["regForm"]["phone"].value;

    let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    let phonePattern = /^[0-9]{10}$/;

    if (!emailPattern.test(email)) {
        alert("Invalid email format");
        return false;
    }

    if (!phonePattern.test(phone)) {
        alert("Phone number must be 10 digits");
        return false;
    }

    return true;
}
```

Step 4: Registration Template

templates/register.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Register</title>
    <script src="{% static 'js/validate.js' %}"></script>
</head>
<body>
    <h2>User Registration</h2>
    <form name="regForm" onsubmit="return validateForm()">
        Email: <input type="text" name="email"><br><br>
        Phone: <input type="text" name="phone"><br><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

Practical Example 10

JavaScript validation for email and phone number

- Email format validated using regex
- Phone number restricted to 10 digits
- Form submission blocked if validation fails

One-Line Exam Conclusion

Django URL routing connects views and templates, while JavaScript ensures efficient front-end form validation.

11. Django Database Connectivity (SQLite / MySQL)

Theory

- Django supports database connectivity using **SQLite**, **MySQL**, **PostgreSQL**, etc.
- **SQLite** is the default database in Django and is file-based, lightweight, and easy to use.
- **MySQL** is used for large, production-level applications.
- Django interacts with databases using the **ORM (Object Relational Mapper)**, which allows database operations using Python code instead of SQL queries.

Lab Program

Set up database connectivity for a Django project

Step 1: Database Configuration (SQLite – Default)

settings.py

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

(No extra installation required)

Optional: MySQL Configuration (Theory Purpose)

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'doctor_db',
      'USER': 'root',
      'PASSWORD': 'password',
      'HOST': 'localhost',
      'PORT': '3306',
   }
}
```

Step 2: Create App

```
python manage.py startapp doctor
```

Step 3: Create 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.IntegerField()

    def __str__(self):
        return self.name
```

Step 4: Apply Migrations

python manage.py makemigrations

python manage.py migrate

Practical Example 11

Django project to connect to SQLite/MySQL and manage doctor records

- Database connected using Django settings
- Doctor records stored using Django ORM
- Tables created automatically using migrations

12. ORM and QuerySets

Theory

- **Django ORM** allows developers to interact with the database using Python objects.
- A **QuerySet** is a collection of database records retrieved using ORM queries.
- Common ORM operations:
    - Create
    - Read
    - Update
    - Delete (CRUD)

Lab Program

Perform CRUD operations using Django ORM

Create (Insert Data)

```
Doctor.objects.create(
    name="Dr. Alice",
    specialization="Dermatologist",
    experience=8
)
```

Read (Fetch Data)

```
doctors = Doctor.objects.all()

for doc in doctors:
    print(doc.name, doc.specialization, doc.experience)
```

Update (Modify Data)

```
doc = Doctor.objects.get(id=1)
doc.experience = 10
doc.save()
```

Delete (Remove Data)

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

Practical Example 12

Django project demonstrating CRUD operations on doctor profiles

- **Create** doctor profile using ORM
- **Read** records using QuerySets
- **Update** doctor details
- **Delete** doctor records from database

13. Django Forms and Authentication

Theory

- Django provides a **built-in form system** to handle user input, validation, and rendering.

- Django's **authentication system** supports:
    - User registration (sign up)
    - Login and logout
    - Password reset and change
- Authentication ensures secure access to application features.

Lab Program

Create a Django project for user registration and login functionality

Step 1: Create Project and App

```
django-admin startproject userauth
cd userauth
python manage.py startapp accounts
```

Step 2: Register App

settings.py

```
INSTALLED_APPS = [
    'accounts',
    'django.contrib.auth',
    'django.contrib.sessions',
]
```

Step 3: User Registration Form (accounts/forms.py)

```
from django import forms
from django.contrib.auth.models import User

class SignupForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'email', 'password']
```

Step 4: Views (accounts/views.py)

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from .forms import SignupForm

def signup(request):
    form = SignupForm(request.POST or None)
    if form.is_valid():
        user = form.save(commit=False)
        user.set_password(form.cleaned_data['password'])
        user.save()
        return redirect('login')
    return render(request, "signup.html", {'form': form})

def user_login(request):
    if request.method == "POST":
        user = authenticate(
            username=request.POST['username'],
            password=request.POST['password']
        )
        if user:
            login(request, user)
            return redirect('profile')
    return render(request, "login.html")

def user_logout(request):
    logout(request)
    return redirect('login')

def profile(request):
```

```
    return render(request, "profile.html")
```

Step 5: URL Configuration

accounts/urls.py

```
from django.urls import path
from .views import signup, user_login, user_logout, profile

urlpatterns = [
    path('signup/', signup, name='signup'),
    path('login/', user_login, name='login'),
    path('logout/', user_logout, name='logout'),
    path('profile/', profile, name='profile'),
]
```

Step 6: Templates

- signup.html
- login.html
- profile.html

(Use {{ form.as_p }} for forms)

Practical Example 13

Django project handling sign up, login, password reset, and profile update

- User registration using Django forms
- Secure login/logout using authentication system
- Profile page after login

14. CRUD Operations using AJAX

Theory

- **AJAX (Asynchronous JavaScript and XML)** allows data exchange with the server without reloading the page.
- In Django, AJAX is commonly implemented using:
    - JavaScript fetch() or XMLHttpRequest

- JSON responses

- AJAX improves user experience by enabling dynamic updates.

Lab Program

Implement AJAX for CRUD operations in a Django project

Step 1: View (doctor/views.py)

```
from django.http import JsonResponse
from .models import Doctor

def add_doctor(request):
    if request.method == "POST":
        Doctor.objects.create(
            name=request.POST['name'],
            specialization=request.POST['specialization']
        )
        return JsonResponse({'status': 'success'})
```

Step 2: URL Configuration

```
path('add/', add_doctor, name='add_doctor'),
```

Step 3: Template with AJAX

templates/doctor.html

```
<script>
function addDoctor() {
    fetch("/add/", {
        method: "POST",
        headers: {
            "X-CSRFToken": "{{ csrf_token }}"
        },
        body: new URLSearchParams({
```

```
        name: document.getElementById("name").value,
        specialization: document.getElementById("spec").value
      })
    })
    .then(res => res.json())
    .then(data => alert("Doctor added successfully"));
}
</script>

<input type="text" id="name" placeholder="Name">
<input type="text" id="spec" placeholder="Specialization">
<button onclick="addDoctor()">Add Doctor</button>
```

Practical Example 14

AJAX-based CRUD operations without page refresh

- Doctor records added dynamically
- Server responds using JSON
- Page does not reload

15. Customizing the Django Admin Panel

15.1 Theory: Techniques for Customizing the Django Admin Panel

The Django admin panel can be customized using a ModelAdmin class where properties like list_display, list_filter, search_fields, ordering, and list_editable control which fields appear, how they are ordered, and what filters and search boxes are available in the list view. Custom methods can also be added to list_display to show computed or formatted values, and ModelAdmin options like fieldsets, readonly_fields, and filter_horizontal can improve the detail/edit page layout and usability.

The admin site itself can be branded using admin.site.site_header, admin.site.site_title, and admin.site.index_title so the panel looks like a custom dashboard instead of the generic “Django administration” interface. Actions can also be added for bulk updates, and permissions can be tuned so specific user groups can only see or modify certain models or fields.

15.2 Lab Task: Customize the Admin for Better Management of Records

Lab Goal:
Customize the Django admin so an admin user can easily manage doctor records in a Doctor Finder project, with list view showing specialty and availability and filters for faster search.

Steps:

1. Create a Django project and app (for example, project doctor_finder, app doctors).
2. Define a Doctor model with fields like name, specialty, availability, city, and phone in models.py.
3. Register the Doctor model in admin.py using a custom DoctorAdmin class that sets list_display, list_filter, search_fields, and list_editable to expose more information directly in the list view.
4. Customize site branding using admin.site.site_header, etc., in admin.py to match the "Doctor Finder" theme.
5. Run makemigrations, migrate, create a superuser, and log in to /admin/ to verify the customized interface.

15.3 Practical Example 15: Django Project with Detailed Doctor Info in Admin

Below is a minimal example you can copy and adapt. This assumes:

- Project name: doctor_finder
- App name: doctors

1) doctors/models.py

python

```
from django.db import models


class Doctor(models.Model):
    AVAILABILITY_CHOICES = [
        ('MORNING', 'Morning'),
        ('AFTERNOON', 'Afternoon'),
        ('EVENING', 'Evening'),
        ('FULL_DAY', 'Full day'),
        ('OFF', 'Not available'),
    ]

    name = models.CharField(max_length=100)
```

```
    specialty = models.CharField(max_length=100)
    city = models.CharField(max_length=100, blank=True)
    phone = models.CharField(max_length=20, blank=True)
    availability = models.CharField(
        max_length=20,
        choices=AVAILABILITY_CHOICES,
        default='FULL_DAY'
    )
    consultation_fee = models.DecimalField(
        max_digits=8,
        decimal_places=2,
        default=0
    )
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return f"{self.name} ({self.specialty})"
```

2) doctors/admin.py

```
python
from django.contrib import admin
from .models import Doctor

# Site-wide branding
admin.site.site_header = "Doctor Finder Admin"
admin.site.site_title = "Doctor Finder Admin Portal"
admin.site.index_title = "Doctor Finder Administration"

@admin.register(Doctor)
class DoctorAdmin(admin.ModelAdmin):
    # Columns shown in list view
    list_display = (
```

```
    "name",
    "specialty",
    "city",
    "availability",
    "consultation_fee",
    "is_active",
    "availability_status",
)

# Fields that can be edited directly in list view
list_editable = (
    "availability",
    "consultation_fee",
    "is_active",
)

# Filters on right side
list_filter = (
    "specialty",
    "city",
    "availability",
    "is_active",
)

# Search box fields
search_fields = (
    "name",
    "specialty",
    "city",
    "phone",
)
```

```
    # Default ordering
    ordering = ("specialty", "name")

    # Show read-only computed field in detail view
    readonly_fields = ("availability_status",)

    fieldsets = (
        ("Basic Information", {
            "fields": ("name", "specialty", "city", "phone")
        }),
        ("Schedule & Status", {
            "fields": ("availability", "consultation_fee", "is_active", "availability_status")
        }),
    )

    def availability_status(self, obj):
        if not obj.is_active or obj.availability == "OFF":
            return "Not taking appointments"
        if obj.availability == "FULL_DAY":
            return "Available all day"
        return f"Available in the {obj.get_availability_display()}"

    availability_status.short_description = "Availability Status"
```

3) doctor_finder/settings.py

Add the app:

python

```
INSTALLED_APPS = [
    # ...
    'doctors',
]
```

Run:

bash

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver

Then open http://127.0.0.1:8000/admin/ and manage detailed doctor information through the customized admin panel

16. Payment Integration Using Paytm

16.1 Theory: Integrating Paytm (Payment Gateway) in Django

Payment gateway integration in Django typically involves three parts: creating an order/transaction record, redirecting the user to the payment gateway with signed parameters (including checksum), and handling the callback/response from the gateway to mark the payment as successful or failed. Paytm provides APIs and an official Python/REST SDK that allow generation and verification of checksums and interaction with their Order/Transaction APIs; a Django app generally wraps these calls in views and models so each payment attempt is recorded.

For development, Paytm offers a staging environment with test credentials; in this mode, payments are simulated and the status is returned to the callback URL without money transfer, making it safe to build and test the integration. The core security rule is that secret keys and checksum generation must stay on the server side, never exposed in JavaScript or templates.

16.2 Lab Task: Implement Paytm Payment Gateway in a Django Project

Lab Goal:
Add Paytm integration to a Django project so users can pay consultation fees when booking a doctor.

High-level steps:

1. Create Paytm merchant account and obtain Merchant ID (MID), Merchant Key, and website/channel IDs for staging.
2. Install required dependencies (for example, a Paytm checksum helper or SDK) and configure credentials in Django settings.py as environment variables.
3. Create a Transaction or Payment model to store order ID, user, amount, status, and Paytm response data.
4. Build a view that:
    - Creates a transaction record.
    - Generates parameters required by Paytm, including checksum.

- Posts them to Paytm payment URL via an auto-submit HTML form.

5. Build a callback view that receives Paytm's POST data, verifies checksum, checks transaction status, updates the Transaction record, and shows success/failure to the user.
6. Add URLs and basic templates for the "Pay" page, the redirect form to Paytm, and the callback result page.

16.3 Practical Example 16: Integrate Paytm in the "Doctor Finder" Project

Below is a simplified, educational example wired to a Doctor and a Transaction model. It does not include real merchant keys; you must fill your own staging credentials from Paytm dashboard.

Assumptions:

- Project: doctor_finder
- Apps: doctors (as above), payments

1) payments/models.py

python

```
from django.db import models
from django.contrib.auth.models import User
from doctors.models import Doctor


class Transaction(models.Model):
    STATUS_CHOICES = [
        ('PENDING', 'Pending'),
        ('SUCCESS', 'Success'),
        ('FAILED', 'Failed'),
    ]

    user = models.ForeignKey(User, on_delete=models.CASCADE)
    doctor = models.ForeignKey(Doctor, on_delete=models.CASCADE)
    order_id = models.CharField(max_length=64, unique=True)
    txn_amount = models.DecimalField(max_digits=8, decimal_places=2)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='PENDING')
    paytm_txn_id = models.CharField(max_length=64, blank=True)
    resp_message = models.CharField(max_length=255, blank=True)
```

```
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.order_id} - {self.user} - {self.status}"
```

2) doctor_finder/settings.py – Paytm config (staging)

```
python
import os

PAYTM_MERCHANT_ID = os.environ.get("PAYTM_MERCHANT_ID", "YOUR_MID_HERE")
PAYTM_MERCHANT_KEY = os.environ.get("PAYTM_MERCHANT_KEY",
"YOUR_MERCHANT_KEY_HERE")
PAYTM_WEBSITE = os.environ.get("PAYTM_WEBSITE", "WEBSTAGING")
PAYTM_CHANNEL_ID = os.environ.get("PAYTM_CHANNEL_ID", "WEB")
PAYTM_INDUSTRY_TYPE_ID = os.environ.get("PAYTM_INDUSTRY_TYPE_ID", "Retail")
PAYTM_CALLBACK_URL = os.environ.get(
    "PAYTM_CALLBACK_URL",
    "http://127.0.0.1:8000/payments/callback/"
)
PAYTM_TXN_URL = "https://securegw-stage.paytm.in/theia/processTransaction"
```

These keys and URLs follow the pattern in Paytm's staging documentation.

3) Simple checksum helper – payments/paytm.py

For learning, a minimal wrapper is shown; in a real app, use Paytm's official checksum utility or SDK.

```
python
import base64
import hashlib
import hmac
from django.conf import settings

def generate_checksum(params, merchant_key=None):
    if not merchant_key:
```

```
    merchant_key = settings.PAYTM_MERCHANT_KEY
  data = "|".join(str(params[key]) for key in sorted(params.keys()))
  return hmac.new(
    merchant_key.encode(),
    data.encode(),
    hashlib.sha256
  ).hexdigest()

def verify_checksum(params, checksum, merchant_key=None):
  generated = generate_checksum(params, merchant_key)
  return generated == checksum
```

4) payments/views.py

python

```
import uuid
from django.shortcuts import render, redirect
from django.views.decorators.csrf import csrf_exempt
from django.conf import settings
from django.contrib.auth.decorators import login_required
from .models import Transaction
from doctors.models import Doctor
from .paytm import generate_checksum, verify_checksum

@login_required
def initiate_payment(request, doctor_id):
  doctor = Doctor.objects.get(id=doctor_id)

  if request.method == "GET":
    context = {"doctor": doctor}
    return render(request, "payments/pay.html", context)

  order_id = str(uuid.uuid4())
```

```
    amount = doctor.consultation_fee

    transaction = Transaction.objects.create(
        user=request.user,
        doctor=doctor,
        order_id=order_id,
        txn_amount=amount,
    )

    params = {
        "MID": settings.PAYTM_MERCHANT_ID,
        "ORDER_ID": transaction.order_id,
        "CUST_ID": str(request.user.id),
        "TXN_AMOUNT": str(transaction.txn_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 = generate_checksum(params)
    params["CHECKSUMHASH"] = checksum

    context = {
        "paytm_txn_url": settings.PAYTM_TXN_URL,
        "params": params,
    }
    return render(request, "payments/redirect_to_paytm.html", context)

@csrf_exempt
def callback(request):
```

```
    if request.method == "POST":
        received_data = dict(request.POST.items())
        paytm_checksum = received_data.pop("CHECKSUMHASH", "")

        is_valid = verify_checksum(received_data, paytm_checksum)

        order_id = received_data.get("ORDER_ID")
        resp_msg = received_data.get("RESPMSG", "")
        resp_code = received_data.get("RESPCODE", "")
        txn_id = received_data.get("TXNID", "")

        transaction = Transaction.objects.get(order_id=order_id)

        if is_valid and resp_code == "01":
            transaction.status = "SUCCESS"
        else:
            transaction.status = "FAILED"

        transaction.paytm_txn_id = txn_id
        transaction.resp_message = resp_msg
        transaction.save()

        context = {
            "transaction": transaction,
            "is_valid_checksum": is_valid,
            "received_data": received_data,
        }
        return render(request, "payments/callback.html", context)

    return redirect("/")
```

5) payments/urls.py

python

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

app_name = "payments"

urlpatterns = [
    path("pay/<int:doctor_id>/", views.initiate_payment, name="pay"),
    path("callback/", views.callback, name="callback"),
]
```

Include in doctor_finder/urls.py:

python

```
from django.urls import path, include

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

6) Templates

templates/payments/pay.html (user confirms payment):

xml

```
<h2>Pay Consultation Fee</h2>
<p>Doctor: {{ doctor.name }} ({{ doctor.specialty }})</p>
<p>City: {{ doctor.city }}</p>
<p>Consultation Fee: {{ doctor.consultation_fee }}</p>

<form method="post">
    {% csrf_token %}
    <button type="submit">Proceed to Pay with Paytm</button>
</form>
```

templates/payments/redirect_to_paytm.html (auto-post to Paytm):

xml

```
<html>
  <body onload="document.forms[0].submit()">
    <p>Redirecting to Paytm, please wait...</p>
    <form method="post" action="{{ paytm_txn_url }}">
      {% for key, value in params.items %}
        <input type="hidden" name="{{ key }}" value="{{ value }}">
      {% endfor %}
    </form>
  </body>
</html>
```

templates/payments/callback.html (show result):

xml

```
<h2>Payment Status</h2>

{% if transaction.status == 'SUCCESS' %}
  <p>Payment successful for order: {{ transaction.order_id }}</p>
  <p>Paytm Transaction ID: {{ transaction.paytm_txn_id }}</p>
{% else %}
  <p>Payment failed for order: {{ transaction.order_id }}</p>
{% endif %}

<p>Message: {{ transaction.resp_message }}</p>
```

With this, your Doctor Finder project now has:

- A customized admin panel to manage detailed doctor information.
- A Paytm-based payment flow where patients can pay consultation fees linked to specific doctors.

Here is a concise, arranged answer you can copy-paste into your document.

---

17. GitHub Project Deployment

17.1 Theory: Steps to Push a Django Project to GitHub

Pushing a Django project to GitHub uses Git for version control so that every change is tracked and the full project is safely stored online and shareable. The core steps are: initialize a Git repository in your project folder, stage and commit the files, create a GitHub repository, connect your local repo to GitHub using git remote, and finally push your code to GitHub using git push

Using a .gitignore file for Django avoids committing unnecessary files such as virtual environments, __pycache__, and local database files, which keeps the repository clean and lightweight. After the first push, routine work is committing new changes and pushing them again so GitHub always has the latest state of the project.

17.2 Lab: Deploy a Django Project to GitHub (Version Control)

Lab Goal:
Upload the full “Doctor Finder” Django project to GitHub and enable version control using Git.

Lab Steps (generic Django project):

1. **Install Git (once per machine)**
    - Download and install Git from https://git-scm.com/ and confirm with:
      git --version in terminal/Command Prompt.
2. **Prepare the Django project folder**
    - Open a terminal in the folder that contains manage.py of your Doctor Finder project (for example doctor_finder/).
3. **Initialize Git and create .gitignore**
    - Initialize repository:

bash

git init

    - Create .gitignore and add common Django ignores, for example:

text

env/

venv/

*.pyc

__pycache__/

db.sqlite3

*.log

.DS_Store

staticfiles/

4. **Stage and commit the project**

- Stage files:

bash

```
git add .
```

- Commit with a message:

bash

```
git commit -m "Initial Doctor Finder project"
```

5. **Create a new GitHub repository**
   - Log in to GitHub, click **New repository**, give it a name like doctor-finder, choose Public/Private, and create it **without** adding an initial README (or adjust commands if you do).
6. **Connect local Git to GitHub and push**
   - In the GitHub "Quick setup" section, copy your HTTPS URL, like: https://github.com/your-username/doctor-finder.git.
   - Run:

bash

```
git remote add origin https://github.com/your-username/doctor-finder.git
git branch -M main
git push -u origin main
```

- After authentication, refresh the GitHub repo page to see all project files.

17.3 Practical Example 17: Step-by-Step Guide for "Doctor Finder" Project to GitHub

Use these exact steps for your Doctor Finder project.

1. **Open terminal in project root**
   - Navigate to folder that contains manage.py, for example:

bash

```
cd path/to/doctor_finder
```

2. **Initialize Git**

bash

```
git init
```

3. **Create .gitignore (Doctor Finder specific)**
   Create a file named .gitignore in the project root with:

text

```
# Python / Django
__pycache__/
*.py[cod]
.DS_Store

# Virtual environment
env/
venv/

# SQLite database
db.sqlite3

# VS Code and IDE settings
.vscode/
.idea/

# Static collection
staticfiles/
```

4. **Stage all files**

```
bash
git add .
```

5. **Commit the Doctor Finder project**

```
bash
git commit -m "Initial commit - Doctor Finder Django project"
```

6. **Create GitHub repository named doctor-finder**
    - On GitHub:
        - Click **New** → Repository name: doctor-finder.
        - Optional: add description.
        - Choose Public/Private.
        - Skip auto README (or handle merge if you add it).
7. **Connect and push to GitHub**

- Back in terminal:

bash

git remote add origin https://github.com/your-username/doctor-finder.git

git branch -M main

git push -u origin main

8. **Verify**
    - Open https://github.com/your-username/doctor-finder and confirm that manage.py, app folders, settings.py, templates, etc., are present.

Now your “Doctor Finder” project is fully deployed to GitHub for version control.

---

18. Live Project Deployment (PythonAnywhere)

18.1 Theory: Deploying Django Projects to PythonAnywhere

PythonAnywhere is a cloud platform that hosts Python/Django applications by providing a Linux environment, web server, and tools like virtualenvs and a web dashboard, so you can run your Django site online without managing your own server. Deploying a Django project to PythonAnywhere usually involves uploading or cloning your code, creating a virtual environment, installing dependencies, configuring a web app with a WSGI file, and setting up static files and environment variables.

For Django specifically, PythonAnywhere recommends a manual configuration where you point the WSGI file to your Django wsgi.py, set ALLOWED_HOSTS to include your PythonAnywhere domain, run migrations, and configure static files via STATIC_ROOT and a static files mapping in the Web tab.

18.2 Lab: Deploy a Django Project to PythonAnywhere

Lab Goal:
Take a Django project (such as “Doctor Finder”), deploy it to PythonAnywhere, and make it accessible via username.pythonanywhere.com.

Lab Steps (generic Django project):

1. **Prepare project and push to GitHub**
    - Ensure your project is working locally and already on GitHub, as described in section 17
2. **Create PythonAnywhere account and open Bash console**
    - Sign up at https://www.pythonanywhere.com, log in, and open a **Bash** console from the **Consoles** tab.
3. **Clone project from GitHub**
    - In Bash:

bash

```
git clone https://github.com/your-username/doctor-finder.git
cd doctor-finder
```

4. **Create virtual environment and install requirements**
    - Either using mkvirtualenv or local venv approach:

bash

```
mkvirtualenv --python=/usr/bin/python3.10 doctorfinder-venv
workon doctorfinder-venv
pip install -r requirements.txt
```

5. **Update ALLOWED_HOSTS in settings**
    - In settings.py, set:

python

```
ALLOWED_HOSTS = ['yourusername.pythonanywhere.com']
```

6. **Create web app (manual config)**
    - Go to **Web** tab → **Add a new web app** → choose your free domain (yourusername.pythonanywhere.com) → select **Manual configuration** → pick same Python version as virtualenv.
7. **Configure WSGI file**
    - On Web tab, click the WSGI file link for your new web app, then edit to point to your Django project, for example:

python

```
import os
import sys

path = '/home/yourusername/doctor-finder'
if path not in sys.path:
   sys.path.append(path)

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'doctor_finder.settings')

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

8. **Set virtualenv for web app**
    - In Web tab, under **Virtualenv**, put the path, for example: /home/yourusername/.virtualenvs/doctorfinder-venv and save.
9. **Configure static files**
    - In settings.py, add absolute STATIC_ROOT, e.g.:

python

STATIC_URL = '/static/'

STATIC_ROOT = '/home/yourusername/doctor-finder/staticfiles'

    - Back in Bash:

bash

python manage.py collectstatic

    - In Web tab → Static files:
        - URL: /static/
        - Directory: /home/yourusername/doctor-finder/staticfiles and save.
10. **Run migrations and create superuser**
    - In Bash (with virtualenv active):

bash

python manage.py migrate

python manage.py createsuperuser

11. **Reload the web app**
    - Click **Reload** button on Web tab, then visit https://yourusername.pythonanywhere.com to see your live Django site.

18.3 Practical Example 18: Deploy "Doctor Finder" on PythonAnywhere

This is the full step-by-step guide for your "Doctor Finder" project.

1. **Confirm project on GitHub**
    - Make sure your Doctor Finder repo is online, for example: https://github.com/your-username/doctor-finder.
2. **Clone into PythonAnywhere**
    - Log in to PythonAnywhere → **Consoles** → **Bash**.
    - Run:

bash

git clone https://github.com/your-username/doctor-finder.git

cd doctor-finder

3. **Create virtualenv and install packages**

bash

mkvirtualenv --python=/usr/bin/python3.10 doctorfinder-venv

workon doctorfinder-venv

pip install -r requirements.txt

4. **Configure ALLOWED_HOSTS**
    - Edit doctor_finder/settings.py:

python

ALLOWED_HOSTS = ['yourusername.pythonanywhere.com']

5. **Set STATIC_ROOT**
    - In the same settings.py:

python

STATIC_URL = '/static/'

STATIC_ROOT = '/home/yourusername/doctor-finder/staticfiles'

6. **Run migrations and create superuser**

bash

python manage.py migrate

python manage.py createsuperuser

7. **Collect static files**

bash

python manage.py collectstatic

8. **Create PythonAnywhere web app**
    - Go to **Web** → **Add a new web app** → choose yourusername.pythonanywhere.com → **Manual configuration** → Python 3.10 (or your chosen version).
9. **Set virtualenv path**
    - In Web tab → Virtualenv path: /home/yourusername/.virtualenvs/doctorfinder-venv and save.
10. **Edit WSGI file**

- Click the WSGI file link (something like /var/www/yourusername_pythonanywhere_com_wsgi.py) and replace contents with:

```
python

import os

import sys


path = '/home/yourusername/doctor-finder'

if path not in sys.path:

  sys.path.append(path)


os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'doctor_finder.settings')


from django.core.wsgi import get_wsgi_application

application = get_wsgi_application()
```

11. **Set static files mapping**
    - In Web tab → Static files:
      - URL: /static/
      - Directory: /home/yourusername/doctor-finder/staticfiles and save.
12. **Reload and test**
    - Click **Reload** on Web tab.
    - Open https://yourusername.pythonanywhere.com in the browser.
    - Visit /admin/ to log in with your superuser and manage your live “Doctor Finder” project.

Here is a clean, arranged answer you can copy-paste into your document.

---

19. Social Authentication

19.1 Theory: Social Login (Google, Facebook, GitHub) with OAuth2 in Django

Social authentication in Django is commonly implemented using OAuth2 providers like Google, Facebook, and GitHub, where the user logs in through the provider and Django receives a verified identity token instead of handling the password directly. A popular approach is to use the django-allauth package, which provides ready-made views, URLs, and models to integrate multiple providers

by configuring them in INSTALLED_APPS, adding authentication backends, and registering client IDs and secrets in provider-specific admin consoles.dev+3

The basic flow is: user clicks “Login with Google/Facebook/GitHub”, gets redirected to the provider’s consent page, approves access, and the provider redirects back to a callback URL with an authorization code that Django exchanges for an access token and profile data. After successful authentication, django-allauth creates or links a local user account and logs the user in, then redirects to a post-login page defined by LOGIN_REDIRECT_URL or similar settings.allauth+2

19.2 Lab: Implement Google and Facebook Login in a Django Project

Lab Goal:
Add “Login with Google” and “Login with Facebook” to a Django project using django-allauth.

Lab Steps (high level):

1. **Install and configure django-allauth**
    - Install:

bash

```
pip install django-allauth
```

- In settings.py, add apps:

python

```
INSTALLED_APPS = [
    # django apps
    'django.contrib.sites',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    # allauth apps
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
```

]

SITE_ID = 1

- Add authentication backends:

python

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
]
```

2. **Include allauth URLs**
    - In urls.py:

python

```
from django.urls import path, include


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

3. **Create Google OAuth2 credentials**
    - In Google Cloud Console, create a project and an OAuth 2.0 Client ID (Web application), set authorized redirect URI to something like: http://127.0.0.1:8000/accounts/google/login/callback/.
    - Copy Client ID and Client Secret and configure them in Django either via admin (Social Applications) or environment variables.
4. **Create Facebook App and credentials**
    - In Facebook Developers dashboard, create an app, enable Facebook Login, configure valid OAuth redirect URIs such as: http://127.0.0.1:8000/accounts/facebook/login/callback/.
    - Obtain App ID and App Secret, and configure them as a Social Application for the Facebook provider.
5. **Add login links to templates**
    - Use provider-specific URLs like {% provider_login_url "google" %} and {% provider_login_url "facebook" %} in your login page template to show clickable social login buttons.

19.3 Practical Example 19: Django Project with Google and Facebook Login

Below is an example setup you can adapt in your “Doctor Finder” project.

1) Install and configure django-allauth

In terminal (inside virtualenv):

bash

```
pip install django-allauth
```

doctor_finder/settings.py (only the relevant parts):

python

```
INSTALLED_APPS = [
    # 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',
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
]

SITE_ID = 1

AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
```

]

LOGIN_REDIRECT_URL = '/'

LOGOUT_REDIRECT_URL = '/'

ACCOUNT_EMAIL_VERIFICATION = 'none' *# for local testing*

ACCOUNT_AUTHENTICATION_METHOD = 'username_email'

ACCOUNT_EMAIL_REQUIRED = True

doctor_finder/urls.py:

python

from django.contrib **import** admin

from django.urls **import** path, include

urlpatterns = [

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

  path('accounts/', include('allauth.urls')),

]

This mirrors typical django-allauth quickstart setup for social authentication.

2) Configure Google provider

Option A: Use Django admin

- Run python manage.py migrate to create allauth tables, then python manage.py createsuperuser and log into /admin/.
- In admin, go to **Sites** → ensure domain is localhost:8000 or 127.0.0.1:8000 for local testing.
- Go to **Social applications** → **Add**:
    - Provider: **Google**
    - Name: Google Login
    - Client id: (from Google Cloud)
    - Secret key: (client secret)
    - Add your Site to “Chosen sites”.

Option B: Environment variables (advanced)

- Store SOCIALACCOUNT_PROVIDERS config and secrets in environment, as shown in many tutorials.

3) Configure Facebook provider

- In Facebook Developers console, create app and enable Facebook Login Web.
- Add OAuth redirect URI: http://127.0.0.1:8000/accounts/facebook/login/callback/ and add your localhost domain in App Domains.
- In Django admin → **Social applications** → Add:
    - Provider: **Facebook**
    - Name: Facebook Login
    - Client id: (App ID)
    - Secret key: (App Secret)
    - Attach Site.

4) Add social login buttons in template

templates/account/login.html (simplified example):

xml

```
<h2>Login</h2>

<form method="post" action="{% url 'account_login' %}">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Login</button>
</form>

<hr>

<h3>Or login with</h3>

<a href="{% provider_login_url 'google' %}">
  <button type="button">Login with Google</button>
</a>

<a href="{% provider_login_url 'facebook' %}">
  <button type="button">Login with Facebook</button>
</a>
```

Now users can log in using Google or Facebook accounts, and django-allauth will handle creating and linking local user records.

---

20. Google Maps API Integration

20.1 Theory: Integrating Google Maps API into Django

Integrating Google Maps API into Django usually means rendering a map in a template using the JavaScript Maps API and feeding it location data (coordinates or addresses) from Django models via context or JSON endpoints. The typical steps are: create or use a model storing location data (latitude/longitude or address), get a Google Maps API key from Google Cloud Console, include the Maps JavaScript API script in your template, and use JavaScript to initialize the map and place markers based on the data coming from Django.

Django can output the location list directly in the template using template tags, or it can serve a JSON endpoint that JavaScript fetches to dynamically add markers, which is helpful when there are many or frequently changing locations. For more advanced setups, libraries like django-google-maps or GeoDjango can help with managing and geocoding locations, but a basic Doctor Finder project can work with simple float latitude/longitude fields.

20.2 Lab: Display Doctor Locations with Google Maps API in "Doctor Finder"

Lab Goal:
Use Google Maps API to display doctors on a map in the Doctor Finder project based on their stored coordinates.

Lab Steps (high level):

1. **Extend Doctor model with coordinates**
    - Add latitude and longitude fields (or address + geocoding) to the Doctor model so each doctor has a map position
2. **Get a Google Maps JavaScript API key**
    - In Google Cloud Console, enable "Maps JavaScript API" and create an API key; restrict it to your domain when going to production.
3. **Create a view that sends doctor data to template**
    - In Django, query all active doctors, pass their names and coordinates to a template where the map is rendered.
4. **Render a map template with markers**
    - In the template, include the Google Maps script using your API key, initialize a JavaScript map, loop over the doctor list (in JS) and place a marker for each doctor.
5. **Optional: Use info windows and filters**
    - Add info windows to show doctor name/specialty and maybe filter doctors by specialty or city via JavaScript and Django views.

20.3 Practical Example 20: Django Project Showing Doctor Locations on Google Maps

Below is a minimal example to show doctor markers on a Google Map in the Doctor Finder project.

1) Update Doctor model with coordinates

doctors/models.py:

python

```
from django.db import models


class Doctor(models.Model):
    name = models.CharField(max_length=100)
    specialty = models.CharField(max_length=100)
    city = models.CharField(max_length=100, blank=True)
    latitude = models.FloatField(null=True, blank=True)
    longitude = models.FloatField(null=True, blank=True)


    def __str__(self):
        return f"{self.name} ({self.specialty})"
```

You can manually set latitude/longitude in admin for each doctor for demo purposes, or later integrate geocoding to convert addresses to coordinates.

2) Add Google Maps API key in settings

doctor_finder/settings.py:

python

```
import os


GOOGLE_MAPS_API_KEY = os.environ.get("GOOGLE_MAPS_API_KEY", "YOUR_API_KEY_HERE")
```

This is a typical way to keep the key configurable via environment variables.

3) Create a view to list doctors with coordinates

doctors/views.py:

python

```
from django.shortcuts import render
from django.conf import settings
```

```
from .models import Doctor


def doctor_map(request):
    doctors = Doctor.objects.filter(latitude__isnull=False, longitude__isnull=False)
    context = {
        "doctors": doctors,
        "google_maps_api_key": settings.GOOGLE_MAPS_API_KEY,
    }
    return render(request, "doctors/doctor_map.html", context)
```

4) Add URL for the map page

doctors/urls.py:

python

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


app_name = "doctors"


urlpatterns = [
    path('map/', views.doctor_map, name='doctor_map'),
]
```

Include in main doctor_finder/urls.py:

python

```
from django.urls import path, include


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

5) Create template to display the Google Map and doctor markers

templates/doctors/doctor_map.html:

xml

```
<!DOCTYPE html>
<html>
<head>
  <title>Doctor Finder - Map</title>
  <style>
    #map {
      height: 500px;
      width: 100%;
    }
  </style>
</head>
<body>
  <h2>Nearby Doctors</h2>
  <div id="map"></div>

  <script>
    // Build a JavaScript array of doctor data from Django context
    const doctors = [
      {% for doctor in doctors %}
        {
          name: "{{ doctor.name|escapejs }}",
          specialty: "{{ doctor.specialty|escapejs }}",
          city: "{{ doctor.city|escapejs }}",
          lat: {{ doctor.latitude }},
          lng: {{ doctor.longitude }}
        }{% if not forloop.last %},{% endif %}
      {% endfor %}
    ];

    function initMap() {
      // Default center; can be set dynamically or to first doctor
```

```
let center = { lat: 20.5937, lng: 78.9629 };  // Example: India

if (doctors.length > 0) {
  center = { lat: doctors[0].lat, lng: doctors[0].lng };
}

const map = new google.maps.Map(document.getElementById("map"), {
  zoom: 5,
  center: center,
});

const infoWindow = new google.maps.InfoWindow();

doctors.forEach(function(doctor) {
  const marker = new google.maps.Marker({
    position: { lat: doctor.lat, lng: doctor.lng },
    map: map,
    title: doctor.name,
  });

  marker.addListener("click", function() {
    const content = `
      <div>
        <strong>${doctor.name}</strong><br>
        ${doctor.specialty}<br>
        ${doctor.city}
      </div>
    `;
    infoWindow.setContent(content);
    infoWindow.open(map, marker);
  });
```

```
    });
  }
</script>

<script async defer
  src="https://maps.googleapis.com/maps/api/js?key={{ google_maps_api_key }}&callback=initMap">
</script>
</body>
</html>
```

This structure matches common patterns where Django passes serialized location data to a template and the Google Maps JavaScript API renders the map and markers.

6) Use in the “Doctor Finder” project

- Add several doctors with latitude/longitude through the admin panel.
- Run the development server and open:
  http://127.0.0.1:8000/doctors/map/
- You will see an interactive Google Map displaying doctor markers with info windows when clicked, effectively integrating Google Maps into the Doctor Finder project.