# Python DB and Framework

## 1. HTML in Python

- **Introduction to embedding HTML within Python using web frameworks like Django or Flask.**
- Web frameworks like **Django** and **Flask** let you build dynamic websites using Python. To display content to users, these frameworks use **HTML templates** that can include Python-like code to make the pages interactive and dynamic.

1. **Flask and HTML Templates**

   In Flask, you use the **Jinja2 templating engine** to embed Python logic in HTML files.

   Example folder structure

```
/my_flask_app
  /templates
    home.html
  app.py
```

   app.py(python code)

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("home.html", name="Alice")
```

   home.html (HTML Template)

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello, {{ name }}!</h1>
  </body>
</html>
```

   O/P : **{{ name }}** is replaced with "Alice" from Python.

2. **Django and HTML Template**

   Django uses its own templating language (similar to Jinja2) to create reusable and dynamic HTML pages.

   Example folder structure

```
/my_django_project
  /my_app
    /templates
```

home.html
views.py
urls.py

views.py

```
from django.shortcuts import render

def home(request):
    return render(request, 'home.html', {'name': 'Alice'})
```

urls.py

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

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

home.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Welcome, {{ name }}!</h1>
  </body>
</html>
```

**Features of these templates:**

Variable : {{name}}

Logic : {% if user %}…{% endif %}, {% for item in list %}…{% endfor %}

Reusable blocks with {% block content %} and {% extends 'base.html' %}

- **Generating dynamic HTML content using Django templates.**
- generate dynamic HTML content using Django templates is one of the most powerful features of Django.
- It lets you inject Python data into your HTML to build interactive and data-driven web pages.

- How Django Generates Dynamic HTML with Templates?
- step 1 : Define a View That Passes Data
- In views.py, you prepare data (from a database, logic, etc.) and pass it to the template.

```
from django.shortcuts import render

def blog_list(request):
    blogs = [
```

```
        {'title': 'New Home Sales Picked Up in December', 'author': 'Admin', 'date': 'Dec. 01, 2020'},
        {'title': 'AI in 2025: What to Expect', 'author': 'Jane', 'date': 'Apr. 15, 2025'},
    ]
    return render(request, 'blog_list.html', {'blogs': blogs})
```

step 2 : Create the Template with Dynamic Tags
Inside templates/blog_list.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Blog List</title>
</head>
<body>
  <h1>Recent Blog Posts</h1>
  <ul>
    {% for blog in blogs %}
      <li>
        <strong>{{ blog.title }}</strong><br>
        <em>by {{ blog.author }} on {{ blog.date }}</em>
      </li>
    {% endfor %}
  </ul>
</body>
</html>
```

- {% for blog in blogs %} loops through each blog item.
- {{ blog.title }}, {{ blog.author }}, and {{ blog.date }} insert the values dynamically.

step 3 : Hook Up the View to a URL
In urls.py

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

urlpatterns = [
    path('blogs/', views.blog_list, name='blog-list'),
]
```

Now when you go to /blogs/, Django renders blog_list.html and injects the blog data.

**More Dynamic features you can use**

**Loops**

```
{% for item in list %}
  {{ item }}
{% endfor %}
```

**Conditionals**

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}</p>
{% else %}
  <p>Please log in.</p>
{% endif %}
```

**Template inheritance**

create a base template

```
<!-- base.html -->
<html>
  <body>
    <header>My Site</header>
    {% block content %}{% endblock %}
  </body>
</html>
```

Then Extend it :

```
<!-- blog_list.html -->
{% extends "base.html" %}
{% block content %}
  <h1>Blog Posts</h1>
  ...
{% endblock %}
```

## 2. CSS in Python

- **Integrating CSS with Django templates.**
- Django uses the {% static %} template tag to handle CSS, JavaScript, images, etc.
- **Step-by-Step: Add CSS to Your Django Templates**
- **step 1 :** Create a static Folder in Your App
- Inside your app (e.g., home), create a folder called static, and then inside it, another folder named after your app for better organization.

  Ex :

```
home/
├── static/
│   └── home/
│       └── styles.css
```

style.css

```
body {
    background-color: #f2f2f2;
    font-family: Arial, sans-serif;
    text-align: center;
    margin-top: 100px;
}
```

```
h1 {
    color: #0066cc;
}
```

**step 2 :** Configure Static Files in settings.py
Make sure these are set (they usually are by default):
setting.py
STATIC_URL = '/static/'

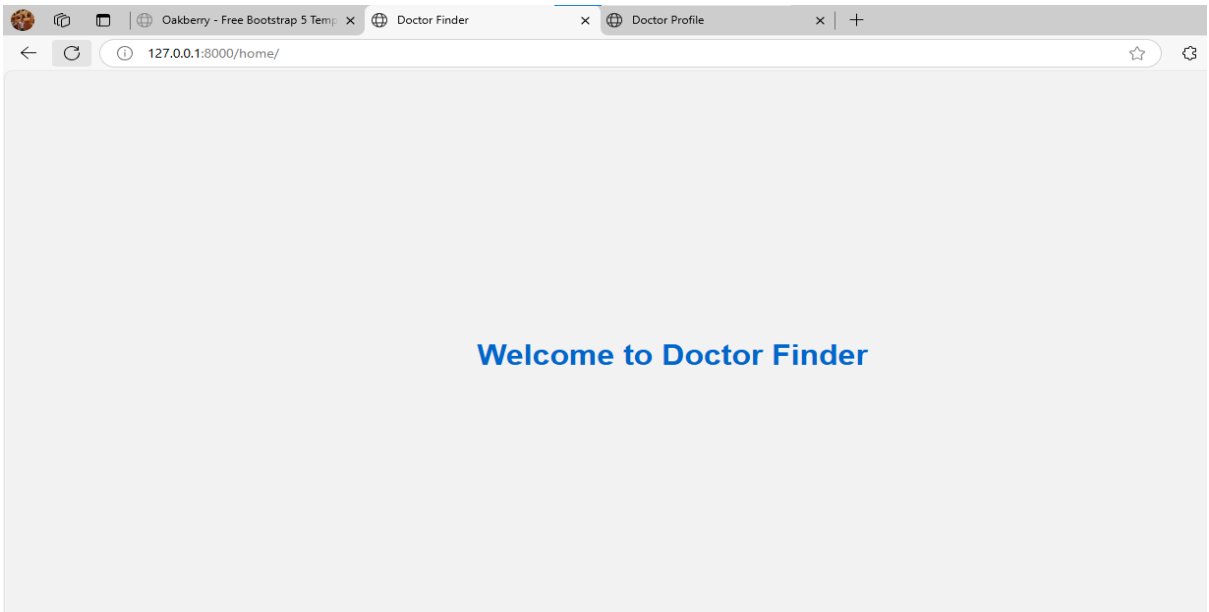
**step 3 :** Load and Link the CSS in Your Template
In your template file (e.g., home.html):

```
<!-- home/templates/home.html -->
{% load static %}

<!DOCTYPE html>
<html>
<head>
    <title>Doctor Finder</title>
    <link rel="stylesheet" type="text/css" href="{% static 'home/styles.css' %}">
</head>
<body>
    <h1>Welcome to Doctor Finder</h1>
</body>
</html>
```

- {% load static %} enables use of the {% static %} tag.
- The href points to your CSS file.

**step 4 :** Run the Server and View the Result



- **How to serve static files (like CSS, JavaScript) in Django**.
- Serving **static files** (like **CSS**, **JavaScript**, images, etc.) in Django is essential for styling and interactive features.
- Static files are **assets that don't change dynamically**, such as:
- CSS
- JavaScript
- Images
- Fonts

**Step-by-Step: How to Serve Static Files in Django (During Development)**

**step 1 :** Create a static/ Folder in Your App

Ex :

```
your_app/
├── static/
│   └── your_app/
│       ├── styles.css
│       ├── script.js
```

Naming the second folder after your app helps keep things organized, especially with multiple apps.

**step 2** : Reference the Static File in Your Template

```
{% load static %}

<!DOCTYPE html>
<html>
<head>
    <title>Example</title>
    <link rel="stylesheet" href="{% static 'your_app/styles.css' %}">
</head>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

**step 3** : Ensure Settings Are Configured (in settings.py)

```
STATIC_URL = '/static/'
```

**step 4** : Serve Static Files in Development (Optional but common)

If you're not using django.contrib.staticfiles (enabled by default), add this to urls.py in your **project.**

```
from django.conf import settings
```

```
from django.conf.urls.static import static

urlpatterns = [
    # your other paths here
]

# Only for development
if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATIC_ROOT)
```

**When You Deploy (Production)**

1. Run : python manage.py collectstatic
2. Configure your web server (e.g., **Nginx**, **Apache**) to serve files from the STATIC_ROOT directory.

**Common static tag :**

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

<img src="{% static 'your_app/images/logo.png' %}" alt="Logo">
```

## 3. JavaScript with Python

- **Using JavaScript for client-side interactivity in Django templates.**
- JavaScript is what brings your Django-powered pages to life—handling things like clicks, form validation, animations, and AJAX.
- **Basic Flow : JavaScript + Django Templates**

  **step 1 :** Create a JavaScript File

  Inside your app's static folder:

```
your_app/
└── static/
    └── your_app/
        └── script.js
```

  **step 2 :** Add JavaScript to Your Template

  Use the {% static %} tag to link the JS file:

```
{% load static %}

<!DOCTYPE html>
<html>
<head>
    <title>Interactive Page</title>
    <script src="{% static 'your_app/script.js' %}" defer></script>
```

```
</head>
<body>
   <h1 id="greeting">Welcome!</h1>
   <button onclick="changeGreeting()">Click Me</button>
</body>
</html>
```

**step 3 :** Write the JavaScript Logic
your_app/static/your_app/script.js:

```
function changeGreeting() {
   document.getElementById("greeting").innerText = "Hello from JavaScript!";
}
```

Now when you click the button, the message updates instantly—no page reload!

- **Linking external or internal JavaScript files in Django.**
  **1. Internal JavaScript Files** (Stored in Your Project):
  File Structure :

```
your_app/
├── static/
│   └── your_app/
│       └── script.js
├── templates/
│   └── your_template.html
```

In Template :

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
   <title>Internal JS Example</title>
   <script src="{% static 'your_app/script.js' %}" defer></script>
</head>
<body>
   <button onclick="sayHello()">Click Me</button>
</body>
</html>
```

script.js

```
function sayHello() {
   alert("Hello from internal JavaScript!");
}
```

The defer attribute makes sure the script runs after the HTML loads.

**2.External JavaScript Files** (Like CDN links):
Just include them directly in your template <head> or before the </body> tag.

Ex :

```
<!DOCTYPE html>
<html>
<head>
   <title>External JS Example</title>
   <script src="https://code.jquery.com/jquery-3.6.0.min.js" defer></script>
</head>
<body>
   <button id="btn">Click Me</button>
   <script>
      document.addEventListener("DOMContentLoaded", () => {
         $('#btn').click(() => {
            alert("Hello from jQuery!");
         });
      });
   </script>
</body>
</html>
```

Note :

- Use {% load static %} at the top of your templates before using {% static %}.
- Keep your JS files organized in static/your_app/ for easy reference.
- For large scripts, link to a file. For quick logic, inline script tags work fine.

## 4. Django Introduction

- **Overview of Django: Web development framework.**
  **Django** is a **high-level Python web framework** that enables **rapid development** of secure and maintainable websites.
- **Feature :**
- **Fast Development**: Comes with ready-to-use components (authentication, admin panel, forms, ORM, etc.)
- **Secure**: Protects against common threats like SQL injection, XSS, CSRF, etc.
- **Scalable**: Powers large sites like Instagram, Pinterest, and Disqus.
- **DRY Principle**: "Don't Repeat Yourself" – encourages clean and reusable cod

**Django is made up of:**

1. **MVT Architecture (Model-View-Template)**

| Component | Role |
|---|---|
| Model | Define database structure |
| View | Python function/classes that handle request and return responses |
| Template | HTML files for the front-end |

2. **Built-in Features**
    - **Admin Interface** – Auto-generated dashboard to manage data
    - **ORM (Object-Relational Mapper)** – Write Python instead of SQL
    - **Routing** – Maps URLs to views
    - **Authentication** – Login, logout, permissions
    - **Form Handling** – With validation and CSRF protection

**Django Workflow**

1. User makes a request (e.g., visits /register/)
2. URLConf (URL dispatcher) maps it to a view
3. The **view** pulls/updates data via **models**
4. Data is passed to a **template** (HTML) and rendered
5. The response is returned to the browser

**Ex :**

A simple Django view

```
from django.shortcuts import render

def home(request):
    return render(request, 'home.html', {'name': 'Doctor Finder'})
```

**Common Django commands**

```
django-admin startproject mysite
python manage.py startapp blog
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

**Real world Use cases** :

Blogs
eCommerce sites
Health management systems
Social networks
APIs (using Django REST Framework)

- **Advantages of Django (e.g., scalability, security).**
    1. **Security**
    - Django helps developers avoid common security pitfalls.
    - Protects against :
        - SQL injection
        - Cross-Site Scripting(XSS)
        - Cross-site Request Forgery (CSRF)

    - Clickingjacking
- secure password hashing, session management, and user authentication.

2. **Rapid Development**

- Django's philosophy is **"batteries included"** – it comes with everything you need (ORM, admin panel, forms, etc.).
- Developers can build and deploy apps **faster** compared to frameworks where you need to assemble everything manually.

3. **Scalability**

- Designed to handle high-traffic applications (e.g., Instagram, Disqus).
- Supports caching, load balancing, and database optimization techniques.
- Works well with large teams and big projects.

4. **Versatile and fully Loaded**

- **Out-of-the box tools :**
    - Admin interface
    - ORM
    - Authentication system
    - Form handling
    - Middleware support

You can build anything from simple blogs to enterprise-level apps.

5. **Object-Relational Mapper (ORM)**

- Django's ORM lets you interact with the database using Python code.
- No need to write raw SQL for most operations.

6. **Built-in Admin Interface**

- Automatically generated from your models.
- Used to manage users, content, and data without writing custom backend tools.

7. **Great Community and Documentation**

- Large community of developers.
- Tons of third-party packages and plugins.
- Excellent documentation and tutorials.

8. **Testing Framework**

- Django comes with a built-in unit testing framework.
- Makes test-driven development (TDD) easier.

9. **Reusable and Maintainable code**

- Follows **DRY (Don't Repeat Yourself)** principle.
- Encourages modular, reusable app structures.

10. **Good for both Small and Large project**

- **can be used for :**
    - Personal blogs
    - Startup MVPs
    - Full-Scale enterprise software

- **Django vs. Flask comparison: Which to choose and why.**
    - **Django vs. Flask: Key Differences**

| **Feature** | **Django** | **Flask** |
|---|---|---|
| Type | Full-stack web framework | Lightweight, micro-framework |
| Philosophy | "Batteries included" – comes with built-in tools | Minimalistic – you build things yourself |
| Admin Panel | Built-in, auto-generated | Not included (can add with extensions) |
| ORM | Built-in ORM (Django ORM) | Not included (use SQLAlchemy or others) |
| Routing | Built-in URL routing | Manual, more flexible |
| Form Handling | Built-in (forms.Form, ModelForm) | Manual or via third-party libraries |
| Security | Built-in protections (CSRF, XSS, SQLi, etc.) | Needs to be manually added or extended |
| Scalability | Scales well with large apps | Also scalable, but more effort is needed |
| Community & Support | Large, mature, with many plugins | Large and active, very Pythonic |
| Learning Curve | Medium – lots of built-in structure to learn | Easy – great for beginners or quick projects |

- **Choose Django if :**
- You want to build a feature-rich web app fast (e.g., admin panel, authentication).
- You prefer convention over configuration.
- You're building a large, scalable app with complex database models.
- You want built-in security features and form handling.

**Use cases :**

CMS
eCommerce platforms
Healthcare systems
Social networks

Dashboards and portals

**choose flask if :**
You want full control over your architecture.
You're building an API or microservice.
Your app is simple and lightweight (e.g., single-page apps).
You like assembling tools yourself or using third-party libraries.

**Use cases :**
REST API
Prototypes / MVPs
Microservices
Lightweight web apps

## 5. Virtual Environment

- **Understanding the importance of a virtual environment in Python projects.**
- A **virtual environment** is an isolated workspace that contains its own **Python interpreter** and **set of packages**, separate from your system's global Python installation.
- **Uses of virtual Environment :**
  1. Avoid Dependency Conflicts
  - Different projects often need different versions of libraries. A virtual environment ensures:
  - Project A can use Django 4.2
  - Project B can use Django 3.2
  - Both projects work perfectly without interfering with each other.

  2. Clean Project Environment
  - Keeps your system clean by installing packages only within your project's environment—not globally.

  3. Easy Project Setup
  - You can easily reproduce the same environment using requirements.txt:
    **pip freeze > requirements.txt**

  - Other users (or your future self) can recreate the environment using:
    **pip install -r requirements.txt**

  4. Better Security
  - By not installing packages system-wide, you reduce the risk of unintentionally affecting system-level tools or scripts.

- **Create and Use a Virtual Environment :**
  1. **Create it :**
     python -m venv venv

  2. **Activate it :**
     **Windows:**
     venv\Scripts\activate

  3. **Install Packages locally**
     pip install django

  4. **Deactivate when done**
     deactivate

     Using a virtual environment ensures your project's Python and Django versions are predictable and isolated—especially important in team environments or deployments.

- **Using venv or virtualenv to create isolated environments.**
- Using **venv** or **virtualenv** is essential for creating **isolated Python environments**, especially in Django and other project-based work.

| Tool | Description |
|---|---|
| **Venv** | Built-in since Python 3.3; recommended for most users. |
| **Virtualenv** | An external tool (install via pip) with more features, supports older versions of Python. |

**Using venv (Built-in) :**

1. **Create a Virtual Environment**
   python -m venv venv

   - venv is the folder name where the environment will live.
   - You can name it anything (e.g., .env, myenv, etc.).

2. **Activate the Environment**
   **Windows:**
   venv\Scripts\activate

   Once activated, your terminal will show something like:
   (venv) C:\YourProject>

3. **Install Packages**
   Inside the environment:
   pip install django

4. **Save Dependencies (Optional but useful)**
   pip freeze > requirements.txt

5. **Deactivate When Done**
   deactivate

**Using virtualenv (Alternative)**

If you want to use virtualenv, install it first:
pip install virtualenv

Then create a virtual environment:
virtualenv venv

Activation and usage are the same as venv.

## 6. Project and App Creation

- **Steps to create a Django project and individual apps within the project.**
- Steps to Create a Django Project and Apps :

1.Set Up a Virtual Environment (Recommended)

```
python -m venv venv
source venv/bin/activate   # or venv\Scripts\activate on Windows
```

2.Install Django

```
pip install django
```

**3.**Create a Django Project

```
django-admin startproject myproject
cd myproject
```

Structure Created:

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

- manage.py: Command-line tool to manage your project
- settings.py: Project configuration
- urls.py: Global URL routing

Run the Development Server (Optional Test)

python manage.py runserver

Visit http://127.0.0.1:8000 – you should see Django's welcome page.

1. Create an App Inside the Project
   Apps are where your actual business logic lives (models, views, templates).
   python manage.py startapp myapp

   Structure of an App:
   myapp/
   ├── admin.py
   ├── apps.py
   ├── models.py
   ├── tests.py
   ├── views.py
   ├── urls.py  # (You create this manually)
   └── migrations/

2. Register the App in settings.py
   Open myproject/settings.py and add your app to INSTALLED_APP
   INSTALLED_APPS = [
   ...
   'myapp',
   ]

3. Create URLs for Your App
   Create urls.py inside myapp/:
   from django.urls import path
   from . import views

   urlpatterns = [
   path('', views.home, name='home'),
   ]

   In views.py:
   from django.http import HttpResponse

   def home(request):
   return HttpResponse("Hello from my app!")

Now, include the app URLs in your project's urls.py:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),

]
```

4. Migrate and Run

```
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

- **Understanding the role of manage.py, urls.py, and views.py.**
- the **roles of manage.py, urls.py, and views.py** in a Django project. These are some of the most important files you'll interact with when building any Django application.

1. **manage.py: Project Management Tool**
   Acts as a **command-line utility** to interact with your Django project.

   Lets you run commands like:
   runserver: start the development server
   makemigrations & migrate: manage database changes
   createsuperuser: create an admin user

   Ex:

   ```
   python manage.py runserver
   python manage.py makemigrations
   ```

2. **urls.py: URL Routing Configuration**
   Maps **URLs to views**.
   You define what happens when a user visits a specific path.

   **Two Types :**
   - **Project-level urls.py** (in the main project folder): root URL configuration.
   - **App-level urls.py** (inside each app): handles routes specific to that app.

   Ex :

   ```
   # project/urls.py
   from django.contrib import admin
   from django.urls import path, include

   urlpatterns = [
       path('admin/', admin.site.urls),
       path('', include('myapp.urls')),  # app-level routing
   ]
   ```

```
# myapp/urls.py
from django.urls import path
from . import views

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

3. **views.py: Defines Logic for Each URL**

- Contains **Python functions (or classes)** that handle requests and return responses.
- Each view is like a controller in MVC – it pulls data and renders templates.

Ex:

```
# myapp/views.py
from django.http import HttpResponse
from django.shortcuts import render

def home(request):
    return HttpResponse("Hello from Django!")  # or render a template
```

Or render a full template

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

## 7. MVT Pattern Architecture

- **Django's MVT (Model-View-Template) architecture and how it handles request-response cycles.**
- Django follows the **MVT architecture**—**Model-View-Template**—which is similar to MVC (Model-View-Controller) but tailored for Django's design.
- Django's MVT Architecture :

1. **Model**

- Handles all **data-related logic**.
- Represents your database structure using Python classes.
- Uses Django's **ORM (Object-Relational Mapper)** to interact with the database.

Ex :

```
# models.py
class Doctor(models.Model):
    name = models.CharField(max_length=100)
    specialization = models.CharField(max_length=100)
```

2. **View**

- Contains the **business logic**.
- Processes user requests, interacts with models, and passes data to templates.
- Returns an HTTP response (could be HTML, JSON, etc.).

Ex :

```
# 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})
```

3. **Template**

- Responsible for **presentation**.
- Uses Django's templating language to display data from the view.
- Keeps HTML separate from business logic

Ex :

```
<!-- doctor_list.html -->
<h1>Doctors</h1>
<ul>
    {% for doctor in doctors %}
        <li>{{ doctor.name }} – {{ doctor.specialization }}</li>
    {% endfor %}
</ul>
```

**Django Request-Response Cycle**

1. **User sends a request** (e.g., GET /doctors/)
2. **URL dispatcher (urls.py)** maps the URL to the right view function
3. **View** processes the request, talks to the **Model** if needed
4. View passes data to a **Template**
5. Template renders the HTML with dynamic content
6. **Django returns the response** (HTML page) to the user

Visual summary :

Browser (Request)

↓

urls.py → View → Model (optional) → Template → View

↓

Browser (Response)

## 8. Django Admin Panel

- **Introduction to Django's built-in admin panel.**
- Django, a high-level Python web framework, comes with a powerful **built-in admin panel** that makes managing application data straightforward and efficient. This admin interface is one of Django's most celebrated features because it allows developers and

non-technical users alike to interact with the database through a user-friendly web interface.

- The **Django admin panel** is a web-based interface that is automatically generated from your models. It allows site administrators to :
    - Add, update, and delete data.
    - Manage users and permissions
    -
- **Key Features:**
    - **Automatic Interface Generation**: Once you define your models in Django, the admin interface is generated with minimal setup.
    - **Authentication and Authorization**: It integrates with Django's authentication system to manage user permissions.
    - **Customizability**: You can customize how models are displayed, add search fields, filters, and even override templates to suit your needs.
    - **Inline Editing**: Related models can be edited directly within another model's edit page.

- **Getting Standard :**
    - **Enable the Admin App**: Ensure 'django.contrib.admin' is included in INSTALLED_APPS in settings.py.
    - **Migrate the Database**: Run python manage.py migrate to create necessary tables.
    - **Create a Superuser**: Use python manage.py createsuperuser to create an admin user.
    - **Register Models**: In your admin.py file, register the models you want to manage.

      Ex :

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

      admin.site.register(MyModel)
      ```

    - **Access the Admin Panel**: Run your server with python manage.py runserver and navigate to http://127.0.0.1:8000/admin.

- **Customizing the Django admin interface to manage database records.**
- Django's admin panel is powerful out of the box, but one of its greatest strengths is how easy it is to **customize the interface** to better manage your database records.

1. **Customizing List Display**
   You can specify which fields appear in the list view of the admin using list_display.

   ```
   class ProductAdmin(admin.ModelAdmin):
   ```

```
    list_display = ('name', 'price', 'available', 'created_at')
```

2. **Adding Filters**
   Use list_filter to add sidebar filters for fields like booleans, dates, and foreign keys.
```
class ProductAdmin(admin.ModelAdmin):
    list_filter = ('available', 'created_at')
```

3. **Search Functionality**
   Enable searching across specified fields using search_fields.
```
class ProductAdmin(admin.ModelAdmin):
    search_fields = ('name', 'description')
```

4. **Editable List Fields**
   Make fields editable directly from the list view with list_editable.
```
class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'available')
    list_editable = ('price', 'available')
```

5. **Organizing the Edit Form**
   Use fieldsets to group fields in the detail view for better form layout.
```
class ProductAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Basic Info', {
            'fields': ('name', 'price')
        }),
        ('Availability', {
            'fields': ('available', 'stock')
        }),
    )
```

6. **Inline Models**
   Manage related models on the same page using inlines.
```
class ReviewInline(admin.TabularInline):  # or admin.StackedInline
    model = Review
    extra = 1

class ProductAdmin(admin.ModelAdmin):
    inlines = [ReviewInline]
```

7. **Custom Admin Actions**
   Define custom actions to apply to selected records.
```
def mark_as_unavailable(modeladmin, request, queryset):
    queryset.update(available=False)
mark_as_unavailable.short_description = "Mark selected products as unavailable"

class ProductAdmin(admin.ModelAdmin):
    actions = [mark_as_unavailable]
```

## 9. URL Patterns and Template Integration

- **Setting up URL patterns in urls.py for routing requests to views.**
- In Django, the urls.py file is used to define **URL patterns** that route incoming web requests to the appropriate **views**. This routing system allows your application to respond to different URLs with specific logic.

1. **Basic URL Pattern Setup**

   Each Django project has a root urls.py (usually in the project folder), and each app can have its own urls.py. Here's a simple setup:

   Project-level urls.py:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),  # Includes app-level URLs
]
```

   App-level urls.py (myapp/urls.py):

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

urlpatterns = [
path('', views.home, name='home'),
path('about/', views.about, name='about'),
]
```

2. **Connecting to Views**

   Define the corresponding views in views.py:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to the homepage!")

def about(request):
    return HttpResponse("About us page")
```

3. **Using Path Converters**

   You can pass dynamic data through the URL using path converters.

```
urlpatterns = [
    path('post/<int:post_id>/', views.post_detail, name='post_detail'),
]
```

And in views.py:

```
def post_detail(request, post_id):
    return HttpResponse(f"Viewing post {post_id}")
```

Common path converters:

- <int:val> – Integer
- <str:val> – String (default)
- <slug:val> – Slug string
- <uuid:val> – UUID
- <path:val> – A string including slashes

4. **Using re_path for Regex URLs**
   If you need more control, use re_path for regular expression URLs:

```
from django.urls import re_path

urlpatterns = [
    re_path(r'^archive/(?P<year>[0-9]{4})/$', views.archive, name='archive'),
]
```

5. **Naming URL Patterns**
   Use the name parameter to easily reference URLs in templates and redirect() or reverse() functions.

```
# urls.py
path('contact/', views.contact, name='contact')

# Template or Python code
<a href="{% url 'contact' %}">Contact Us</a>
```

- **Integrating templates with views to render dynamic HTML content**.
- Django uses a powerful templating engine to render dynamic HTML content. The basic idea is to use **views** to process data and **templates** to present that data in a user-friendly format.
  1. **Creating Templates Directory**
     Create a folder named templates inside your app directory. Inside that, create your HTML files.

```
myapp/
│
```

```
templates/
└── myapp/
    ├── home.html
    └── about.html
```

2. **Writing a Template File**

   **Ex : home.html**

```
<!DOCTYPE html>
<html>
<head>
   <title>Home</title>
</head>
<body>
   <h1>Welcome, {{ username }}!</h1>
   <p>This is the homepage.</p>
</body>
</html>
```

3. **Updating Views to Use Templates**

   Use render() in your views to link data to templates.

   Ex : views.py

```
from django.shortcuts import render

def home(request):
   context = {'username': 'Alice'}
   return render(request, 'myapp/home.html', context)

def about(request):
   return render(request, 'myapp/about.html')
```

   The render() function takes request, the template name, and a context dictionary.

   Template variables like {{ username }} are dynamically replaced using the context.

4. **Linking Between Pages**

   Use the {% url %} template tag for clean internal links.

   Ex : home.html

```
<a href="{% url 'about' %}">About Us</a>
```

5. **Template Inheritance (Optional)**

   You can use base templates to avoid repeating HTML structure.

   Ex : base.html

```
<!DOCTYPE html>
```

```
<html>
<head>
   <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
   <header><h1>My Website</h1></header>
   {% block content %}{% endblock %}
</body>
</html>
```

home.html

```
{% extends 'myapp/base.html' %}

{% block title %}Home{% endblock %}

{% block content %}
   <h2>Welcome, {{ username }}!</h2>
{% endblock %}
```

## 10. Form Validation using JavaScript

- **Using JavaScript for front-end form validation.**
- Front-end form validation with JavaScript enhances user experience by providing instant feedback before the form is submitted to the server. It helps catch errors early and reduces unnecessary server load.
- **Basic JavaScript Form Validation Example**

Ex : HTML Form

```
<form id="myForm" onsubmit="return validateForm()">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required>
  <br>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <br>
  <input type="submit" value="Submit">
</form>
<p id="error" style="color:red;"></p>
```

Javascript validation

```
<script>
function validateForm() {
   let name = document.getElementById("name").value.trim();
   let email = document.getElementById("email").value.trim();
   let error = document.getElementById("error");

   error.textContent = "";  // Clear previous errors
```

```
    if (name === "") {
        error.textContent = "Name is required.";
        return false;
    }

    if (!validateEmail(email)) {
        error.textContent = "Invalid email format.";
        return false;
    }

    return true;
}

function validateEmail(email) {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(email);
}
</script>
```

**You can validate :**

- **Required fields** – Ensure the user doesn't leave essential fields empty.
- **Email format** – Check if the email matches standard formatting.
- **Password rules** – Enforce minimum length, character types, etc.
- **Numeric ranges** – For age, price, quantity inputs.
- **Custom conditions** – Like password confirmation or agreement to terms.

**Benefits of JavaScript Validation:**

- Instant feedback to users.
- Reduces round-trips to the server.
- Enhances user experience.

Note : Front-end validation should always be backed up with server-side validation (like Django forms or serializers) to ensure security, since users can bypass JavaScript.

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

- **Connecting Django to a database (SQLite or MySQL).**
- Django supports multiple databases, with **SQLite** being the default. You can easily switch to **MySQL** or others like PostgreSQL depending on your project needs.
  1. **Using SQLite (Default)**
     SQLite is lightweight and comes preconfigured with Django. It's great for development and small projects.

- **step-by-step :**
  - In settings.py, the default configuration looks like this:

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

To set up :

```
python manage.py migrate
```

  - This creates a db.sqlite3 file in your project directory.

2. **Connecting to MySQL**

For production or larger-scale apps, you might prefer MySQL.

**step 1 : Install MySQL Client**

Install the MySQL Python adapter:

```
pip install mysqlclient
```

If it fails (especially on Windows), use:

```
pip install pymysql
```

**Step 2: Update settings.py**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'your_db_name',
        'USER': 'your_db_user',
        'PASSWORD': 'your_db_password',
        'HOST': 'localhost',  # Or the IP of your DB server
        'PORT': '3306',
    }
}
```

**Step 3: Create Database in MySQL**

Log into MySQL and run:

```
CREATE DATABASE your_db_name CHARACTER SET UTF8;
```

**Step 4: Migrate Your Schema**

```
python manage.py migrate
```

3. **Test the Connection**

Run the development server:

python manage.py runserver

- **Using the Django ORM for database queries.**
- Django's **Object-Relational Mapper (ORM)** is one of its most powerful features. It lets you interact with your database using Python code instead of raw SQL. You define your data models in Python, and Django handles all the SQL behind the scenes.

  - **Basic Model Example**
    Let's say you have a model like this in models.py:

    ```
    from django.db import models

    class Book(models.Model):
        title = models.CharField(max_length=100)
        author = models.CharField(max_length=100)
        published_date = models.DateField()
        available = models.BooleanField(default=True)

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

  - **Creating Records**

    ```
    book = Book(title="Django Basics", author="John Doe",
    published_date="2024-01-01")
    book.save()
    ```

    Or use `.create()`:

    ```
    Book.objects.create(title="Django Pro", author="Jane Smith",
    published_date="2024-05-01")
    ```

  - **Querying Records**

    All Records

    ```
    books = Book.objects.all()
    ```

    Filtering

    ```
    available_books = Book.objects.filter(available=True)
    ```

    Single object

    ```
    book = Book.objects.get(id=1)  # Raises error if not found
    ```

    Conditional Lookup

    ```
    recent_books = Book.objects.filter(published_date__year=2024)
    ```

- **Updating Records**
  book = Book.objects.get(id=1)
  book.title = "Updated Title"
  book.save()
  Or bulk update:
  Book.objects.filter(available=True).update(available=False)

- **Deleting Records**
  book = Book.objects.get(id=1)
  book.delete()

  Or bulk delete:
  Book.objects.filter(available=False).delete()

- **Advanced Queries**
  ordering:
  Book.objects.order_by('published_date')

  Chaining**:**
  Book.objects.filter(available=True).order_by('-published_date')

  Q objects for complex lookups**:**
  from django.db.models import Q
  Book.objects.filter(Q(title__icontains="Django") |
  Q(author__icontains="John"))

## 12. ORM and QuerySets

- **Understanding Django's ORM and how QuerySets are used to interact with the database.**
- Django's **Object-Relational Mapper (ORM)** allows you to work with your database using Python objects. Instead of writing raw SQL queries, you use **models** and **QuerySets** to interact with your data in an intuitive way.
- A **QuerySet** is a collection of database records (rows) that Django creates from your model. It's like a list of objects, but it's lazy and only hits the database when needed.
- You get a QuerySet by calling Model.objects and using methods like .all(), .filter(), or .exclude().

  Ex :
  from myapp.models import Book

  # All books
  books = Book.objects.all()

```
# Filtered books
recent_books = Book.objects.filter(published_date__year=2024)

# One specific book
book = Book.objects.get(id=1)
```

**Common QuerySet Methods :**

| Method | Description |
|---|---|
| .all( ) | Returns all records |
| . filter(**kwargs) | Filters records based on conditions |
| .exclude(**kwargs) | Opposite of filter |
| .get(**kwargs) | Returns a single record, or error |
| .count() | Counts matching records |
| .order_by() | Orders results by a field |
| .values() | Returns dictionaries instead of model instances |
| .exists() | Returns True if any records match |

**QuerySet Filtering with Lookups :**
Django supports powerful **field lookups**:

```
Book.objects.filter(title__icontains="django")  # Case-insensitive match
Book.objects.filter(published_date__year=2025)  # Filter by year
Book.objects.filter(id__in=[1, 2, 3])           # Filter by list of IDs
```

**Chaining QuerySets:**
QuerySets can be **chained** together for more complex queries:
```
Book.objects.filter(available=True).order_by('-published_date')
```

Each filter returns a new QuerySet, allowing a functional style of building queries.

**Lazy Evaluation**
QuerySets are **lazy**—they don't hit the database until needed:books =
```
Book.objects.filter(author="Alice")  # No query yet
for book in books:  # Query is executed here
   print(book.title)
```

**Combining Queries with Q Objects :**
For more complex logic, use Q:
```
from django.db.models import Q
Book.objects.filter(Q(title__icontains="django") | Q(author__icontains="john"))
```

- **Use the ORM and QuerySets :**
  Cleaner and safer than raw SQL
  Portable across databases

Integrated with Django forms and admin
Easy to maintain and debug

## 13.Django Forms and Authentication

- **Using Django's built-in form handling.**
- Django Form Handling (Step-by-Step):

step 1: Create a Form class

First, create a form in forms.py inside your app:

Ex :

```
from django import forms

class MyForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

step 2: Create a View to handle the form

In your views.py

Ex :

```
from django.shortcuts import render
from .forms import MyForm

def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            # Process the data
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            # Do something with the data (save to DB, send email, etc.)
            return render(request, 'thanks.html')
    else:
        form = MyForm()

    return render(request, 'my_form.html', {'form': form})
```

step 3 : Create a Template

Make a template my_form.html:

```
<form method="post">
    {% csrf_token %}
```

```
    {{ form.as_p }}
    <button type="submit">Send</button>
</form>
```

step 4 : Add URL in urls.py

Ex:

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

urlpatterns = [
    path('form/', views.my_view, name='my_form'),
]
```

- **GET request**: Django displays an empty form.
- **POST request**: Django validates and processes the form data.
- **form.is_valid()**: checks if the form is correct.
- **form.cleaned_data**: gets the cleaned input data.

- **creating forms from models (ModelForm)**

  If you have a model and want to create a form from it, use ModelForm:

  Ex :

```
from django.forms import ModelForm
from .models import MyModel

class MyModelForm(ModelForm):
    class Meta:
        model = MyModel
        fields = ['name', 'email', 'message']
```

- **Implementing Django's authentication system (sign up, login, logout, password management).**

  step 1 : Create URLs in your urls.py

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

urlpatterns = [
    path('home/', views.home, name='home'),
    path('profile-page/', views.profile, name='profile'),
    path('patient_form/', views.patient_form, name='patient_form'),
    path('signup/', views.signup, name='signup'),
    path('login_view/', views.login_view, name='login_view'),
    path('logout/', views.logout, name='logout'),
    path('password_change/', views.password_change_view, name='password_change'),
    path('password_change_done/', views.pass_change_done, name='password_change_done'),
]
```

step 2 : Create your Views in views.py:

```python
from django.shortcuts import render,redirect
from . forms import PatientForm
from django.contrib.auth.forms import UserCreationForm,AuthenticationForm,PasswordChangeForm
from django.contrib.auth.decorators import login_required
from django.contrib.auth import authenticate,login,update_session_auth_hash,logout as auth_logout
from django.contrib import messages
```

```python
def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('login_view')

        else:
            form = UserCreationForm()
            return render(request,'signup.html',{'form':form})
    else:
        return render(request,'signup.html')
```

```python
def login_view(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)   #django automatically bind request data to the form
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password)   #Django checks if a user exists with the given user
            if user is not None:
                login(request, user)     #django create session  and user stay logged in page
                messages.success(request, f"Welcome {username}!")
                return redirect('home')  # Redirect to your homepage or dashboard
            else:
                messages.error(request, "Invalid username or password.")
        else:
            messages.error(request, "Invalid username or password.")
    else:
        form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

```python
# Logout View
@login_required
def logout(request):
    auth_logout(request)  # This clears the session
    return redirect('login_view')  # Redirect to login page after logout

# Password Change View
@login_required
def password_change_view(request):
    if request.method == 'POST':
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            user = form.save()
            update_session_auth_hash(request, user)  # Important! Keeps the user logged in after password change
            return redirect('password_change_done')
    else:
        form = PasswordChangeForm(user=request.user)
    return render(request, 'password_change.html', {'form': form})

# Password Change Done View
@login_required
def pass_change_done(request):
    return render(request, 'pass_change_done.html')
```

step 3 : Create Templates

signup.html

myapp > templates > signup.html

```html
<h2>Sign Up</h2>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Sign Up</button>
</form>
```

login.html

myapp > templates > login.html

```html
{% if messages %}
    {% for message in messages %}
        <div>{{ message }}</div>
    {% endfor %}
{% endif %}
<h2>Login</h2>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Login</button>
</form>
```

logout.html

```
myapp > templates > logout_html
1   <h2>You have been logged out.</h2>
2   <a href="{% url 'login_view' %}">Login Again</a>
3
```

password_change.html

```
myapp > templates > password_change.html
1   <h2>Change Password</h2>
2   <form method="post">
3       {% csrf_token %}
4       {{ form.as_p }}
5       <button type="submit">Change Password</button>
6   </form>
7
```

pass_change_done.html

```
myapp > templates > pass_change_done.html
1   <h2>Password changed successfully!</h2>
2   <a href="{% url 'home' %}">Return to home</a>
3
```

## 14. CRUD Operations using AJAX

- **Using AJAX for making asynchronous requests to the server without reloading the page.**
- **AJAX** = Asynchronous JavaScript And XML
- It lets your webpage **talk to the server** and **update** without refreshing the full page.
- In Django, we use AJAX usually with **views** that return **JSON** data.

Ex : Submitting a Form using AJAX in Django

1.Create a Django View to Handle AJAX Request

```
# views.py
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt  # Only for testing — better to use CSRF token properly in production
def ajax_submit(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        # You can save data or process it here

        return JsonResponse({'status': 'success', 'message': f'Hello {name}! We received your email {email}.'})
```

```
    return JsonResponse({'status': 'fail', 'message': 'Only POST requests allowed'})
```

2.Add URL in urls.py

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

urlpatterns = [
    path('ajax-submit/', views.ajax_submit, name='ajax_submit'),
]
```

3.Create a Simple HTML Form + JavaScript AJAX Code

```
<!-- templates/ajax_form.html -->
<h2>AJAX Form</h2>

<form id="myForm">
    {% csrf_token %}
    Name: <input type="text" name="name" id="name"><br>
    Email: <input type="text" name="email" id="email"><br>
    <button type="submit">Submit</button>
</form>

<div id="response"></div>

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

<script>
    $('#myForm').submit(function(event) {
        event.preventDefault();  // Stop the form from submitting normally

        $.ajax({
            url: "{% url 'ajax_submit' %}",
            type: "POST",
            data: {
                'name': $('#name').val(),
                'email': $('#email').val(),
                'csrfmiddlewaretoken': '{{ csrf_token }}'
            },
            success: function(response) {
                if (response.status == 'success') {
                    $('#response').html('<p style="color:green;">' + response.message +
'</p>');
                } else {
                    $('#response').html('<p style="color:red;">' + response.message + '</p>');
                }
```

```
        }
      });
  });
</script>
```

- When user fills the form and clicks submit:
- JavaScript captures the form without reloading the page.
- It sends a **POST request** to /ajax-submit/.
- Django view processes the data and returns a **JSON** response.
- JavaScript updates part of the page (inside ) without full refresh

## 15. Customizing the Django Admin Panel

- **Techniques for customizing the Django admin panel.**
- Customize Django Admin :
- To make it **look better** (professional).
- To **control** what fields are visible.
- To **add search**, **filters**, **group fields**, **custom actions**, etc.
- To make admin easier for non-technical staff (marketing, sales, HR).
- Techniques for Customizing Django Admin Panel :

  1. Change Admin Display (List View)

  **Show specific columns in the list view** using list_display.

```
# admin.py
from django.contrib import admin
from .models import Product

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'price', 'created_at')  # Fields shown in admin list page
```

  Now the admin list page shows id, name, price, and created_at columns.

  2.Add Search in Admin

  **Enable searching by certain fields** with search_fields.

```
class ProductAdmin(admin.ModelAdmin):
    search_fields = ('name', 'description')
```

  Now you get a search box at the top!

  3.Add Filters

**Filter data easily** using list_filter.

```
class ProductAdmin(admin.ModelAdmin):
    list_filter = ('category', 'created_at')
```

Adds filter sidebar in admin panel.

4.Customize Form Layout (Fieldsets)

**Group fields into sections** with fieldsets.

```
class ProductAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Basic Information', {
            'fields': ('name', 'category', 'price')
        })
        ('More Details', {
            'fields': ('description', 'created_at')
        }),
    )
```

Admin form will have **sections** now.

4. Add Inline Editing

**Edit related models directly inside parent model's page** using TabularInline or StackedInline.

Example: Add **multiple Images** inside a **Product** page:

```
from .models import ProductImage

class ProductImageInline(admin.TabularInline):
    model = ProductImage
    extra = 1  # Number of empty images shown

class ProductAdmin(admin.ModelAdmin):
    inlines = [ProductImageInline]
```

Now when you edit a Product, you can add Images directly.

5. Add Custom Actions

**Create custom buttons to run actions** on multiple items.

```
class ProductAdmin(admin.ModelAdmin):
    actions = ['mark_as_featured']

    def mark_as_featured(self, request, queryset):
        queryset.update(featured=True)
    mark_as_featured.short_description = "Mark selected products as Featured"
```

Now you can select multiple products and mark them Featured with one click.

6. Change Admin Branding

Customize Admin Title, Header, and Index title in settings.py:

```
# settings.py
ADMIN_SITE_HEADER = "My Company Admin"
ADMIN_SITE_TITLE = "My Admin Portal"
ADMIN_INDEX_TITLE = "Welcome to Admin Area"
```

## 16. Payment Integration Using Paytm

- **Introduction to integrating payment gateways (like Paytm) in Django projects.**
- Payment gateway : It is a service that allows your app to collect money from users securely (credit cards, UPI, net banking, etc).
- Popular Payment gateway :
- Paytm
- Razorpay
- Stripe
- PayPal
- PhonePe

- Main Steps for Integration:
  - Create a merchant account with the payment gateway (example: Paytm Business).
  - Get API keys: Merchant ID (MID), Secret Key, Website, Industry Type.
  - Install any required SDK or library.
  - Create Django views that:
    - Generate payment request.
    - Redirect user to Paytm payment page.
    - Handle Paytm's **response** (success or failure).
  - Verify the payment securely using checksum or signature.
  - Show success/failure page to the user.

**How Paytm Integration works :**

| Step | Action |
|---|---|
| 1. | User clicks Pay button on your site |
| 2. | Django creates a payment request with transaction details |
| 3. | Redirect to Paytm's payment gateway |
| 4. | User completes payment |
| 5. | Paytm sends a response (success/failure) to your server |
| 6. | Your Django server verifies the response |
| 7. | You update your database (order paid / failed) |

Example Paytm Integration Flow :

**Step 1**: **Install the Paytm checksum package (helps verify the payment):**

pip install paytmchecksum

**Step 2**: **Django view to create payment request:**

**# views.py**

```
import paytmchecksum
from django.shortcuts import render, redirect
from django.conf import settings
import requests

def initiate_payment(request):
    if request.method == "POST":
        order_id = "ORDER" + str(random.randint(10000, 99999))
        amount = "500"  # payment amount
        data = {
            "MID": settings.PAYTM_MID,
            "ORDER_ID": order_id,
            "CUST_ID": "customer_id",
            "TXN_AMOUNT": amount,
            "CHANNEL_ID": "WEB",
            "WEBSITE": settings.PAYTM_WEBSITE,
            "INDUSTRY_TYPE_ID": settings.PAYTM_INDUSTRY_TYPE_ID,
            "CALLBACK_URL": "http://127.0.0.1:8000/callback/",
        }

        checksum = paytmchecksum.generateSignature(data,
settings.PAYTM_SECRET_KEY)
        data["CHECKSUMHASH"] = checksum

        return render(request, 'paytm_redirect.html', {'paytm_data': data})
    return render(request, 'payment_form.html')
```

**Step 3**: **HTML form to auto-submit data to Paytm**

```
<!-- paytm_redirect.html -->
<form method="post" action="https://securegw-stage.paytm.in/theia/processTransaction"
name="paytm_form">
    {% for key, value in paytm_data.items %}
        <input type="hidden" name="{{ key }}" value="{{ value }}">
    {% endfor %}
</form>
<script type="text/javascript">
    document.paytm_form.submit();
</script>
```

This auto-submits to Paytm’s payment gateway.

**Step 4**: **Handle Paytm response (Callback URL)**

# views.py

```
@csrf_exempt
def payment_callback(request):
    if request.method == "POST":
        received_data = dict(request.POST)
        paytm_checksum = received_data.pop('CHECKSUMHASH', [None])[0]

        is_valid_checksum = paytmchecksum.verifySignature(received_data,
settings.PAYTM_SECRET_KEY, paytm_checksum)

        if is_valid_checksum:
            if received_data['RESPCODE'][0] == '01':
                # Payment successful
                return render(request, 'payment_success.html', {'data': received_data})
            else:
                # Payment failed
                return render(request, 'payment_failed.html', {'data': received_data})
        else:
            return HttpResponse("Checksum Mismatch")
```

**Step 5**: **Add these in your settings.py**

```
PAYTM_MID = 'YourMerchantID'
PAYTM_SECRET_KEY = 'YourSecretKey'
PAYTM_WEBSITE = 'WEBSTAGING'
PAYTM_INDUSTRY_TYPE_ID = 'Retail'
```

In **production**, you must switch URLs to live server URLs (not staging).

## 17. GitHub Project Development

- **Steps to push a Django project to GitHub.**

  **1.Initialize a Git Repository:**

  Open your Django project folder in terminal or command prompt:

  ```
  cd your_project_folder/
  git init
  ```

  This initializes a **local Git repo** inside your Django project.

  **2.Create a .gitignore File**

  Before adding files, you **must** create a .gitignore file to tell Git **what not to track** (important for Django).

  Create .gitignore file and add this:

```
# .gitignore

*.pyc
__pycache__/
db.sqlite3
/static/
.env
*.log
/media/
```

- This avoids uploading:
  - Python cache files
  - SQLite database
  - Static/media folders
  - Secret environment files
  - Logs

**3.Add Files to Git**

git add .

This **adds all files** (except what's inside .gitignore) to Git's staging area.

**4.Commit the Files**

git commit -m "Initial commit - Django project setup"

This **saves** your files into the Git history.

**5.Create a Repository on GitHub**

Go to GitHub
Click **New repository**
Name it (example: my-django-project)
**DO NOT** initialize with README or .gitignore on GitHub (you already have locally)
Click **Create repository**

GitHub gives you some **URL** like:

https://github.com/your-username/my-django-project.git

**6.Add GitHub Remote Origin**

Now, connect your local repo to GitHub repo:
git remote add origin https://github.com/your-username/my-django-project.git
origin is the nickname for your GitHub repo.

**7.Push Your Project**

git push -u origin master
or if you're on main branch (new Git versions):
git push -u origin main

This **uploads your entire Django project** to GitHub!

Note : If you later add **new files**, you repeat:
git add .
git commit -m "Added new feature"
git push

## 18. Live Project Deployment (PythonAnywhere)

- **Introduction to deploying Django projects to live servers like PythonAnywhere.**
- **PythonAnywhere** is an easy, cloud-based hosting service.
- You can host **Python/Django projects** without setting up complicated servers.
- You get a **free plan** to deploy small projects easily.
- Perfect for learning, testing, and small websites.
- Steps to Deploy Django Project to PythonAnywhere :

**1.Prepare Your Django Project for Production**
Before uploading:

**Install requirements file**:
In your project root:
pip freeze > requirements.txt
(So the server knows what libraries your project needs.)

**Update settings.py**:
Add your domain to ALLOWED_HOSTS:
ALLOWED_HOSTS = ['your-username.pythonanywhere.com']
Make sure DEBUG = False for production.

**Collect static files**:
python manage.py collectstatic
(This gathers all CSS/JS files into /static/ ready for deployment.)

**2.Create an Account on PythonAnywhere**
Go to PythonAnywhere
Create a **free account**.
Login.

**3.Upload Your Django Project**
Open **Files** tab on PythonAnywhere dashboard.
Upload your Django project **folder by folder** (or zip and unzip it inside).

Best place to put your project:
/yourusername/mysite/

## 4.Create a Virtual Environment on PythonAnywhere

Open **Bash console** on PythonAnywhere.
Create a virtualenv (Python 3.8, 3.10 etc.)
mkvirtualenv myenv --python=/usr/bin/python3.10

Activate it:
workon myenv

Install Django and your project libraries:
pip install -r /home/yourusername/mysite/requirements.txt

## 5.Set Up the Web App

Go to **Web** tab on PythonAnywhere.
Click **Add a new web app ➔ Manual configuration ➔** Choose **Django**.

Set these settings:

- **Source code**: /home/yourusername/mysite/
- **Virtualenv**: /home/yourusername/.virtualenvs/myenv
- **WSGI file**: Edit WSGI to point to your wsgi.py file.

Example WSGI config:

```
import sys
import os

path = '/home/yourusername/mysite'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'your_project_name.settings'

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

Save and reload!

## 6.Final Steps

Apply migrations:
Open **Bash console** again:
python manage.py migrate

Create superuser:
python manage.py createsuperuser

Now visit:

https://your-username.pythonanywhere.com/
Your Django app is **LIVE on the Internet**

## 19. Social Authentication

- **Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.**
- Instead of filling registration forms, users can **sign in using their Google/Facebook/GitHub accounts**.
- It uses **OAuth2** protocol to securely allow login without sharing passwords.
- Much faster and trusted for users.
- Best Way: Use django-allauth for social login
- django-allauth is a powerful Django package that easily handles:
    - Email login
    - Google, Facebook, GitHub login
    - Social account linking
    - Account management

- Steps to Set up Google, Facebook, GitHub Login using django-allauth
  **1.Install django-allauth**

  pip install django-allauth

  Add it to your INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
    ...
    'django.contrib.sites',  # Required!
    'allauth',
    'allauth.account',
    'allauth.socialaccount',

    # Providers (Google, Facebook, GitHub)
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
    'allauth.socialaccount.providers.github',
]
```

  Also add:

```
SITE_ID = 1
```

  Authentication settings:

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',  # Default
    'allauth.account.auth_backends.AuthenticationBackend',  # Needed for social login
)
```

Set login redirects:
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'

**2.Update URLs**

In your project's urls.py:

```
from django.urls import path, include

urlpatterns = [
    path('accounts/', include('allauth.urls')),  # <-- Important
    ...
]
```

Now, /accounts/login/, /accounts/logout/, etc. will work automatically.

**3.Configure Social App Providers (Google / Facebook / GitHub)**

Go to Django Admin panel.
Under **Social applications**, click **Add**.
Fill the fields:

- **Provider**: (Google / Facebook / GitHub)
- **Name**: Whatever you like.
- **Client id** and **Secret key** (you'll get these from Google/Facebook/GitHub developer console)
- **Sites**: Select example.com (or your dev site)

**How to Get Client IDs and Secrets?**

**A. Google Login:**

- Go to Google Developers Console.
- Create new Project ➔ Credentials ➔ OAuth Client ID.
- Add Authorized Redirect URIs:
  http://127.0.0.1:8000/accounts/google/login/callback/
  Copy **Client ID** and **Secret Key** into Django Admin.

**B. Facebook Login:**

- Go to Facebook Developers.
- Create App ➔ Set up Facebook Login ➔ Web.
- Add Valid OAuth Redirect URIs:
  http://127.0.0.1:8000/accounts/facebook/login/callback/
  Copy App ID and App Secret into Django Admin.

**C. GitHub Login:**

- Go to GitHub Developer Settings.
- Create OAuth App ➔ Set Authorization callback URL:
  http://127.0.0.1:8000/accounts/github/login/callback/
  Copy Client ID and Secret into Django Admin.

After setup, you can visit:

- /accounts/login/
- Django will show you **Google / Facebook / GitHub** login buttons.
- User clicks ➔ gets redirected ➔ authenticates ➔ redirected back to your site logged-in!
  You can create your own login template to add fancy social login buttons like:
  <a href="{% provider_login_url 'google' %}">Login with Google</a><br>
  <a href="{% provider_login_url 'facebook' %}">Login with Facebook</a><br>
  <a href="{% provider_login_url 'github' %}">Login with GitHub</a>

  These buttons redirect users to respective OAuth2 providers.

## 20. Google Maps API

- **Integrating Google Maps API into Django projects.**
- **Google Maps API** allows you to **embed** maps, **show markers**, **draw routes**, **autocomplete locations**, etc.
- You can use it **inside your Django templates** by calling Google Maps scripts.
- Google requires you to get an **API key** to use Maps on your site.
- Steps to Integrate Google Maps API into Django :

**1.Get Google Maps API Key**

- Go to Google Cloud Console.
- Create a **new project**.
- Enable **Maps JavaScript API**.
- Go to **Credentials** ➔ Create Credentials ➔ API Key.
- (Optional but recommended) **Restrict the API Key** to only your domains.
- Now you have your **Google Maps API Key**.

**2.Add the API Key to Django settings.py**

Inside settings.py, add:

GOOGLE_MAPS_API_KEY = 'YOUR_API_KEY_HERE' This keeps the key cleanly organized.

**3.Create a Template to Show the Map**

In your Django app, create a simple HTML file:

```
<!-- templates/show_map.html -->
<!DOCTYPE html>
<html>
<head>
  <title>My Google Map</title>
  <script src="https://maps.googleapis.com/maps/api/js?key={{
GOOGLE_MAPS_API_KEY }}"></script>
```

```
    <script>
      function initMap() {
        var location = {lat: 28.6139, lng: 77.2090};  // Example: New Delhi coordinates
        var map = new google.maps.Map(document.getElementById('map'), {
          zoom: 10,
          center: location
        });
        var marker = new google.maps.Marker({
          position: location,
          map: map
        });
      }
    </script>
</head>
<body onload="initMap()">
    <h2>My Location on Map</h2>
    <div id="map" style="height: 500px; width: 100%;"></div>
</body>
</html>
```

This will display a map centered at the coordinates you give.

**4.Pass the API Key to Template**

In your Django views.py:

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

def show_map(request):
    context = {
        'GOOGLE_MAPS_API_KEY': settings.GOOGLE_MAPS_API_KEY
    }
    return render(request, 'show_map.html', context)
```

Now your API key dynamically goes into your template!

**5.URL Configuration**

In your app's urls.py:

```
from . import views

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

Now visiting `/map/` will show your Google Map!