**🧠 Django Core Concepts (Beginner to Advanced)**

**🔹 1. Django Introduction**

* What is Django?
* Features of Django
* MVT Architecture (Model–View–Template)
* Django vs Flask

**🔹 2. Django Project Structure**

* manage.py – command-line utility
* settings.py – configuration
* urls.py – URL dispatcher
* wsgi.py / asgi.py – deployment interface

**🔹 3. Apps in Django**

* Creating apps using startapp
* Multiple apps in one project
* App modularity and reusability

**🔹 4. Routing & URLs**

* Mapping URLs to views (urls.py)
* Dynamic URLs and path converters
* Including app URLs (include())

**🔹 5. Views**

* Function-Based Views (FBV)
* Class-Based Views (CBV)
* Rendering HTML templates with data
* Returning JSON responses

**🔹 6. Templates**

* Template directory structure
* Template language: {{ variable }}, {% tags %}
* Template inheritance (base.html, extends, block)
* Using loops and conditions in templates
* Template filters and custom filters

**🔹 7. Models & Database**

* Defining models (models.Model)
* Field types: CharField, IntegerField, DateTimeField, etc.
* Relationships:
  + ForeignKey
  + ManyToManyField
  + OneToOneField
* Django ORM (Object Relational Mapping)
  + .objects.all(), .filter(), .get(), .create(), etc.
* Migrations: makemigrations, migrate

**🔹 8. Admin Panel**

* Registering models in admin.py
* Customizing the admin interface
* Adding filters, search, custom display

**🔹 9. Forms**

* Django Forms vs ModelForms
* Form validation (clean(), is\_valid())
* Custom error messages
* Handling GET and POST
* Styling with Bootstrap or django-crispy-forms

**🔹 10. Authentication and Authorization**

* User registration and login system
* User model and AbstractUser
* Login/Logout views (auth)
* Decorators: @login\_required, @permission\_required
* Group and permission-based access

**🔹 11. Class-Based Views (CBV)**

* ListView, DetailView, CreateView, UpdateView, DeleteView
* Overriding methods (get\_queryset, form\_valid)
* Mixins for reusability

**🔹 12. Static & Media Files**

* STATICFILES\_DIRS, STATIC\_ROOT
* MEDIA\_URL, MEDIA\_ROOT
* Uploading and displaying images/files
* FileField, ImageField

**🔹 13. Middleware**

* What is middleware?
* Built-in middleware (e.g., authentication, sessions)
* Writing custom middleware

**🔹 14. Messages Framework**

* Flash messages for notifications
* Tags like success, error, info

**🔹 15. Django Signals**

* Connecting model events (e.g., post\_save, pre\_delete)
* Use cases: auto-profile creation, logging

**🔹 16. Django REST Framework (DRF)**

* Serializers (ModelSerializer)
* API Views: APIView, GenericAPIView, ViewSet
* Routers and URL config
* Authentication: Token, JWT
* Permissions and throttling

**🔹 17. Testing**

* Writing tests with TestCase
* Testing views, models, forms

**🔹 18. Deployment**

* Production settings
* collectstatic for static files
* Deploying on:
  + **Heroku**
  + **Render**
  + **PythonAnywhere**
  + VPS (Gunicorn + Nginx)

**🔹 19. Third-Party Libraries**

* django-crispy-forms – better forms UI
* django-allauth – social auth
* django-filter – filtering querysets
* drf-yasg – Swagger docs for APIs
* whitenoise – static files in production

**✅ Summary Table**

| **Concept Area** | **Key Topics** |
| --- | --- |
| 🛠 Project Setup | MVT, settings, apps |
| 🌐 URLs & Views | FBV, CBV, routing |
| 🗃 Models & ORM | Fields, relationships, CRUD |
| 🎨 Templates | Inheritance, logic, filters |
| 📋 Forms | Forms vs ModelForms, validation |
| 👤 Authentication | User, login/logout, permissions |
| 📁 Static & Media | Uploads, file handling |
| 🔄 REST API | DRF, serializers, views |
| 🚀 Deployment | Heroku, staticfiles, production |
| 🔌 Third-Party Tools | crispy-forms, allauth, filters |

**📘 What is Django?**

**Django** is a **high-level Python web framework** that enables rapid development of secure and maintainable websites.  
It was created to make web development fast, scalable, and clean.

* Developed by: **Adrian Holovaty and Simon Willison**
* First released: **2005**
* Written in: **Python**
* Philosophy: **“Don’t Repeat Yourself (DRY)”** and **“Convention over Configuration”**

Django is often used to build **robust and scalable web applications** such as e-commerce sites, content management systems, social networks, and more.

**✨ Key Features of Django**

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

* Similar to MVC (Model-View-Controller)
* Separates the business logic, UI, and data handling
  + **Model** – Defines the data structure (database schema)
  + **Template** – Deals with presentation (HTML)
  + **View** – Contains business logic and interacts with the model

**2. ORM (Object Relational Mapper)**

* Allows you to interact with the database using Python objects instead of raw SQL
* Supports multiple databases (PostgreSQL, MySQL, SQLite, Oracle)
* Example:
* # Create
* user = User.objects.create(name="Teja")
* # Read
* user = User.objects.get(id=1)
* # Update
* user.name = "Teja Updated"
* user.save()
* # Delete
* user.delete()

**3. Admin Interface**

* Automatically generates a fully functional admin panel for your project
* Used to manage database models easily
* Highly customizable and secure

**4. Security Features**

* Built-in protection against:
  + SQL Injection
  + Cross Site Scripting (XSS)
  + Cross Site Request Forgery (CSRF)
  + Clickjacking
* Secure password hashing and user authentication system

**5. Scalability**

* Suitable for both small and large applications
* Can handle high traffic by scaling horizontally (e.g., load balancing, caching)

**6. Built-in Features**

* Authentication (login/logout, password management)
* Sessions, messages, and user management
* Form handling and validation
* File and media handling
* URL routing system

**7. Rapid Development**

* Encourages clean and pragmatic design
* Built-in development server for testing
* Automatic code reloading during development

**8. Reusable Components / Apps**

* Projects are organized into **apps** which are modular and reusable
* You can plug apps into multiple Django projects

**9. Rich Ecosystem**

* A large number of third-party packages available (e.g., Django REST Framework, AllAuth, Celery)
* Strong community support and documentation

**10. Internationalization (i18n)**

* Built-in support for multilingual websites
* Supports time zones, translations, formatting of dates/numbers

**🏗️ MTV Architecture in Django (Model–Template–View)**

Django follows the **MTV architecture**, which is similar to the MVC (Model–View–Controller) architecture but adapted to Django’s style.

**🔁 Comparison with MVC:**

| **MVC Component** | **Django Equivalent (MTV)** | **Responsibility** |
| --- | --- | --- |
| Model | Model | Handles data and database |
| View | Template | Handles presentation/UI |
| Controller | View | Handles business logic |

**1️ Model – Data Layer**

**📌 Definition:**

The **Model** defines the structure of the data. It is a Python class that maps to a table in your database using Django's ORM.

**📌 Responsibilities:**

* Represents and manages data
* Handles all interactions with the database (CRUD operations)
* Ensures data validation and relationships between tables

**📌 Example:**

# models.py

from django.db import models

class Student(models.Model):

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

roll\_no = models.IntegerField(unique=True)

email = models.EmailField()

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

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

return self.name

**2️ Template – Presentation Layer**

**📌 Definition:**

The **Template** is responsible for how data is presented to the user. It is the front-end (HTML + Django Template Language) part of the application.

**📌 Responsibilities:**

* Display dynamic content to users
* Embed logic like looping and conditions
* Interacts with the data passed from the view

**📌 Example:**

<!-- templates/student\_list.html -->

<h2>Student List</h2>

<ul>

{% for student in students %}

<li>{{ student.name }} - {{ student.email }}</li>

{% endfor %}

</ul>

**3️ View – Business Logic Layer**

**📌 Definition:**

The **View** is the controller of the MTV pattern. It processes the request, interacts with the model, and returns a rendered template (response).

**📌 Responsibilities:**

* Handle user requests
* Fetch/process data from the model
* Render a response (usually a template) with context data

**📌 Example:**

# views.py

from django.shortcuts import render

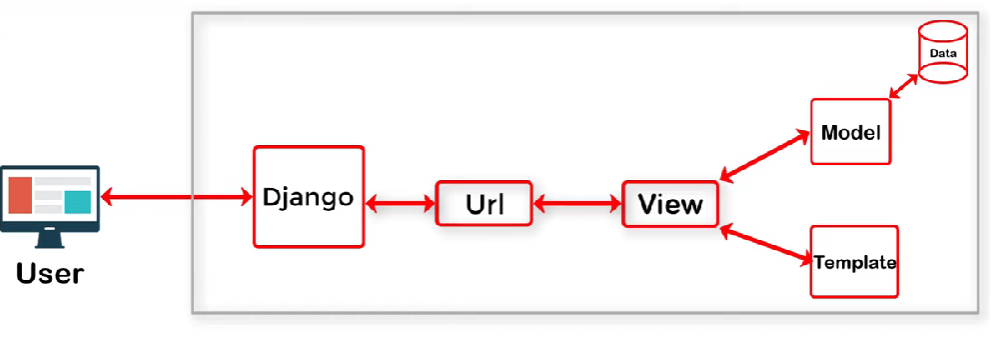
from .models import Student

def student\_list(request):

students = Student.objects.all()

return render(request, 'student\_list.html', {'students': students})

**🔁 How MTV Works Together (Flow Diagram):**



Browser Request

↓

URL Dispatcher (urls.py)

↓

View (views.py)

↓

Model (models.py)

↓

View (Fetches data)

↓

Template (HTML + context)

↓

Rendered HTML sent to browser

**🔗 URLs – Routing System**

**📌 Definition:**

The urls.py file maps a specific URL pattern to a corresponding view.

**📌 Example:**

# urls.py

from django.urls import path

from . import views

urlpatterns = [

path('students/', views.student\_list, name='student\_list'),

]

**✅ Summary Table**

| **Component** | **File** | **Purpose** |
| --- | --- | --- |
| Model | models.py | Defines database schema using classes |
| View | views.py | Handles business logic and data flow |
| Template | templates/\*.html | Renders HTML using Django Template Language |
| URL Conf | urls.py | Maps URLs to views |

**🧪 What is a Virtual Environment in Python?**

A **virtual environment** is an isolated Python environment that allows you to install specific versions of libraries without affecting the global Python setup or other projects.

**✅ Why is Virtual Environment Important?**

| **Reason** | **Explanation** |
| --- | --- |
| 🧩 **Project Isolation** | Each project can have its own dependencies and versions (e.g., Django 4.2 in one project, Django 5.0 in another). |
| 🔒 **Avoid Conflicts** | Prevents version clashes between packages used in different projects. |
| 🧼 **Cleaner Development** | Keeps your system-wide Python installation clean and untouched. |
| 🔁 **Reproducibility** | Easily replicate the same environment using requirements.txt. |
| 💼 **Deployment Readiness** | Helps in preparing consistent environments for production. |

**🛠️ Setting Up a Virtual Environment (Step-by-Step)**

**📌 Step 1: Install virtualenv**

pip install virtualenv

**📌 Step 2: Create a Virtual Environment**

virtualenv myenv

myenv is the folder name for your virtual environment.

**📌 Step 3: Navigate to the Environment Folder**

cd myenv

cd Scripts

**📌 Step 4: Activate the Virtual Environment**

**▶️ On Windows:**

activate

**▶️ On macOS/Linux:**

source myenv/bin/activate

✅ Once activated, your terminal prompt will change to show the virtual environment name like:

(myenv) C:\Users\Teja\>

**📦 Install Django in the Virtual Environment**

cd .. 🡪make sure you’re in myenv

pip install Django==5.0.0

**✅ Confirm Installation**

pip show Django

You will see details like version, location, and dependencies.

**🚀 Create a Django Project**

**📌 Step 1: Create a Django Project**

django-admin startproject myproject

**📌 Step 2: Navigate to the Project Directory**

cd myproject

**📌 Step 3: Run the Development Server**

python manage.py runserver

✅ You’ll see the development server running at http://127.0.0.1:8000/

**🔚 To Deactivate the Virtual Environment**

deactivate

**📁 Django Folder Structure – Short Notes**

**🔹 Top-Level Project Folder**

* Created using: django-admin startproject myproject

myproject/

├── manage.py

└── myproject/

├── \_\_init\_\_.py

├── settings.py

├── urls.py

├── asgi.py

└── wsgi.py

**🔸 File Descriptions**

| **File/Folder** | **Purpose** |
| --- | --- |
| manage.py | Command-line tool to run the server, create apps, migrations, etc. |
| \_\_init\_\_.py | Makes the inner folder a Python package |
| settings.py | Main config: apps, DB, templates, static files, etc. |
| urls.py | URL routing for the project |
| asgi.py | Entry point for ASGI (async support) |
| wsgi.py | Entry point for WSGI (deployment) |

**🧱 After startapp command:**

Example: python manage.py startapp myapp

myapp/

├── admin.py

├── apps.py

├── models.py

├── tests.py

├── views.py

└── migrations/

| **File/Folder** | **Purpose** |
| --- | --- |
| models.py | Define database models |
| views.py | Handles business logic |
| admin.py | Customize Django admin |
| migrations/ | Auto-generated DB migrations |
| apps.py | App configuration |

**🚀 How to Run Your First Django Project**

**✅ Step 1: Start the Development Server**

In the project directory, run:

python manage.py runserver

* Starts the local server at http://127.0.0.1:8000/
* Visit the URL in your browser to see the Django welcome page.

**🧩 Step 2: Create Views**

**🔹 Create or Edit views.py in Your App Folder (myapp/views.py)**

Add the following code:

from django.http import HttpResponse

def home(request):

return HttpResponse("This is the first Django project")

def login(request):

return HttpResponse("Welcome to login page")

**🔗 Step 3: Configure URLs**

**✅ In the project-level urls.py (myproject/urls.py):**

from django.contrib import admin

from django.urls import path

from myapp.views import home, login # Import your view functions

urlpatterns = [

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

path('home/', home, name="home"),

path('login/', login, name="login"),

]

**🔍 Now Visit in Browser**

* http://127.0.0.1:8000/home/ → shows **"This is the first Django project"**
* http://127.0.0.1:8000/login/ → shows **"Welcome to login page"**

🧱 **Creating Multiple Apps in Django – Notes**

**✅ Step 1: Create Apps Using startapp**

Run the following commands in your terminal:

python manage.py startapp coder

python manage.py startapp tester

python manage.py startapp manager

Each command creates a new Django **app folder** with default structure like views.py, models.py, etc.

**🛠 Step 2: Register Apps in settings.py**

Open myproject/settings.py and add each app to the INSTALLED\_APPS list:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

...

'coder',

'tester',

'manager',

]

✅ App names must be added as strings (in quotes) to activate them.

**🧠 Step 3: Define Views in Each App**

**Example: coder/views.py**

from django.http import HttpResponse

def jnrcoder(request):

return HttpResponse("I am junior coder")

def snrcoder(request):

return HttpResponse("I am senior coder")

You can define similar views in tester/views.py or manager/views.py depending on your logic.

**🔗 Step 4: Connect App Views to URLs**

**In project/urls.py (main URL config file):**

from django.contrib import admin

from django.urls import path

from coder import views as coder\_views

from .views import home, login # your main app views

urlpatterns = [

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

# Views from coder app

path('jnrcoder/', coder\_views.jnrcoder, name="jnrcoder"),

path('snrcoder/', coder\_views.snrcoder, name="snrcoder"),

# Main project views

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

path('login/', login, name='login'),

]

✅ Always use clear aliases when importing views from different apps (e.g., views as coder\_views) to avoid naming conflicts.

**🌐 Now Run and Test**

python manage.py runserver

Visit in browser:

* http://127.0.0.1:8000/jnrcoder/ → shows "I am junior coder"
* http://127.0.0.1:8000/snrcoder/ → shows "I am senior coder"

**multiple views from different Django apps**:

**🧩 Django – Short Notes on Multiple Views from Apps**

**✅ 1. Create Apps**

python manage.py startapp coder

python manage.py startapp tester

python manage.py startapp manager

**✅ 2. Register Apps in settings.py**

INSTALLED\_APPS = [

...

'coder',

'tester',

'manager',

]

**✅ 3. Define Views in Each App**

📄 coder/views.py

from django.http import HttpResponse

def jnrcoder(request):

return HttpResponse("I am junior coder")

def snrcoder(request):

return HttpResponse("I am senior coder")

📄 tester/views.py (example)

def testing(request):

return HttpResponse("Tester view here")

**✅ 4. Connect Views in urls.py**

📄 project/urls.py

from django.contrib import admin

from django.urls import path

from coder import views as coder\_views

from tester import views as tester\_views

urlpatterns = [

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

path('jnrcoder/', coder\_views.jnrcoder),

path('snrcoder/', coder\_views.snrcoder),

path('test/', tester\_views.testing),

]

💡 Use aliases (views as coder\_views) to avoid naming conflicts.

**✅ 5. Run Server and Test**

python manage.py runserver

**Visit in browser:**

* /jnrcoder/ → Junior coder message
* /snrcoder/ → Senior coder message
* /test/ → Tester view message

**🧩 Why We Use URL Configurations Inside Each App in Django**

**✅ Why Not Put All URLs in the Project’s urls.py?**

If you define **all routes in project/urls.py**, it becomes:

* ❌ **Hard to manage** as your app grows.
* ❌ **Cluttered and unorganized**, especially with multiple apps.
* ❌ Violates **modular design** — apps become less reusable.

**✅ Why Use URL Configurations Inside Each App?**

1. 🔄 **Modularity**: Each app handles its own routes.
2. 📦 **Reusability**: Apps can be reused across projects without changes.
3. 🧹 **Cleaner codebase**: Keeps project/urls.py clean and readable.
4. 🔧 **Easy Maintenance**: Each team/module works independently.

**🧱 App-Level URL Configuration – Example**

**📁 tester/urls.py**

from django.urls import path

from tester import views

urlpatterns = [

path('jnrtester/', views.jnrtester, name="jnrtester"),

path('snrtester/', views.snrtester, name="snrtester"),

]

**📁 coder/urls.py**

from django.urls import path

from coder import views

urlpatterns = [

path('jnrcoder/', views.jnrcoder, name="jnrcoder"),

path('snrcoder/', views.snrcoder, name="snrcoder"),

]

✅ No admin.site.urls needed in app-level URLs — it belongs only in the main project.

**🧩 Main Project URL Configuration**

**📁 project/urls.py**

from django.contrib import admin

from django.urls import path, include

from .views import home, login

urlpatterns = [

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

# Including app-level URLs

path('coder/', include('coder.urls')),

path('tester/', include('tester.urls')),

# Project-level views

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

path('login/', login, name='login'),

]

**🧠 URL Resolution Behavior**

* Visiting /coder/jnrcoder/ → Calls jnrcoder view from coder/urls.py
* Visiting /tester/jnrtester/ → Calls jnrtester view from tester/urls.py

**✅ Summary**

| **Concept** | **Reason** |
| --- | --- |
| Use urls.py in each app | Keeps apps modular and clean |
| Use include() in main URLs | To connect app-level routing |
| Avoid putting everything in project/urls.py | Leads to poor scalability and clutter |

**short and clear notes** on the Django modules:

**🔹 django.urls**

**Purpose:** URL routing system in Django

**Key Functions:**

* path() – Define simple URL patterns
* re\_path() – Define regex-based URL patterns
* include() – Include other app-level URL files
* reverse() – Generate URLs using view names
* resolve() – Match a URL path to a view

**🔹 django.contrib**

**Purpose:** Built-in Django apps for common features

**Important Modules:**

* admin – Admin dashboard
* auth – Authentication (login, logout, register)
* sessions – Manage user sessions
* messages – Flash messages
* staticfiles – Handle static assets

**🔹 django.http**

**Purpose:** Handle low-level HTTP requests and responses

**Key Classes/Functions:**

* HttpResponse – Return plain HTML or text
* JsonResponse – Return JSON data
* Http404 – Raise "Page Not Found" error
* HttpRequest – Request object passed to views

**🔹 django.shortcuts**

**Purpose:** Provides helper functions to simplify views

**Key Functions:**

* render() – Render a template with context
* get\_object\_or\_404() – Fetch model object or return 404
* redirect() – Redirect to another URL or view

**🧾 Notes: How to Configure and Use HTML Templates in Django**

**✅ 1. Create a templates Folder in Your Project Directory**

* Folder structure:

project/

├── manage.py

├── sample/

│ ├── views.py

│ ├── ...

├── templates/

│ ├── jr.html

│ ├── sr.html

💡 You can name the folder anything, but templates is the convention.

**✅ 2. Add HTML Files**

**📄 jr.html**

<html>

<body>

<h1>Junior developer from HTML</h1>

</body>

</html>

**📄 sr.html**

<html>

<body>

<h1>Senior developer from HTML</h1>

</body>

</html>

**✅ 3. Use the render() Method in Views**

**📄 views.py**

from django.shortcuts import render

def jnrcoder(request):

return render(request, "jr.html")

def snrcoder(request):

return render(request, "sr.html")

**📝 Short Notes: render() in Django**

**✅ Purpose:**

The render() function is used to:

* Combine a **request**, an **HTML template**, and a **context dictionary**
* Return an **HttpResponse** with the final rendered HTML content

**✅ Syntax:**

render(request, 'template.html', context)

**✅ Parameters:**

* **request**:  
  Required. The incoming HTTP request object. It holds info like headers, method, and submitted data.
* **'template.html'**:  
  The name of the HTML file to render. Django searches for it in the templates directory.
* **context**:  
  A dictionary of data (key: value) passed to the template. It allows dynamic content rendering using template variables like {{ key }}.

**✅ 4. Configure Templates in settings.py**

**📄 settings.py**

import os

TEMPLATE\_DIR = os.path.join(BASE\_DIR, "templates")

TEMPLATES = [

{

...

'DIRS': [TEMPLATE\_DIR], # Add this line

...

},

]

🔔 This tells Django where to look for HTML files when using render().

**✅ Summary Table**

| **Step** | **Description** |
| --- | --- |
| Create Folder | Create templates/ folder in your project |
| Create HTML | Add jr.html, sr.html, etc. |
| Use render() | Return HTML from views using render(request, "filename.html") |
| Update Settings | Add template path to DIRS in settings.py |

**🌟 Full Guide to Django Template Language (DTL)**

**🧠 What is DTL?**

**Django Template Language (DTL)** is a special syntax used in Django HTML templates to dynamically:

* Display variables
* Control flow (if/else, loops)
* Include other templates
* Extend base layouts
* Load static files

It allows **separation of logic and presentation**, adhering to the **MVT (Model-View-Template)** architecture.

**🔷 1. Variable Interpolation – {{ variable }}**

Displays a variable passed from the view.

<p>Welcome, {{ username }}</p>

You can also access:

* **Object attributes**: {{ user.name }}
* **Dictionary keys**: {{ data.key }}
* **List items**: {{ items.0 }}

**🔷 2. Filters – {{ variable|filter }}**

Filters transform the value of a variable before displaying it.

**🔹 Common Filters:**

| **Filter** | **Purpose** |
| --- | --- |
| upper | Converts text to uppercase |
| lower | Converts text to lowercase |
| length | Returns length of a list/string |
| date:"D d M Y" | Formats a date |
| default:"N/A" | Uses default if variable is empty |
| truncatechars:5 | Shortens text to N characters |
| safe | Marks HTML as safe (no auto-escaping) |
| join:", " | Joins list items |

**🔹 Example:**

<p>{{ name|upper }}</p> <!-- TEJA -->

<p>{{ items|length }}</p> <!-- 5 -->

<p>{{ bio|truncatechars:20 }}</p> <!-- Shortened text -->

**🔷 3. Tags – {% tag %}**

Used for logic like loops, conditions, and template inheritance.

**🔹 a. if Statements**

{% if user.is\_authenticated %}

<p>Hello {{ user.username }}</p>

{% elif user.is\_staff %}

<p>Welcome Staff</p>

{% else %}

<p>Please log in</p>

{% endif %}

**🔹 b. for Loops**

<ul>

{% for item in product\_list %}

<li>{{ item.name }} - ₹{{ item.price }}</li>

{% endfor %}

</ul>

**Special loop variables:**

| **Variable** | **Description** |
| --- | --- |
| forloop.counter | 1-based loop index |
| forloop.first | True for first iteration |
| forloop.last | True for last iteration |

{% for item in items %}

{{ forloop.counter }}. {{ item }}

{% endfor %}

**🔹 c. with Tag (Alias)**

Assigns a value to a temporary variable:

{% with full\_name=person.first\_name|add:" "|add:person.last\_name %}

<p>{{ full\_name }}</p>

{% endwith %}

**🔹 d. include Tag**

Includes another template.

{% include "navbar.html" %}

Useful for reusable UI parts like headers, footers.

**🔹 e. comment Tag**

Template-level comments that are not rendered.

{% comment %}

This section won't be shown in HTML

{% endcomment %}

**🔹 f. csrf\_token**

Adds CSRF protection in forms.

<form method="post">

{% csrf\_token %}

...

</form>

**🔷 4. Template Inheritance**

Lets you create a base layout and extend it in child templates.

**🔹 base.html**

<html>

<head><title>{% block title %}My Site{% endblock %}</title></head>

<body>

{% include "navbar.html" %}

<div class="content">

{% block content %}{% endblock %}

</div>

</body>

</html>

**🔹 home.html**

{% extends "base.html" %}

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

{% block content %}

<h1>Welcome to Home Page</h1>

{% endblock %}

**🔷 5. Working with Static Files**

1. In settings.py:

STATIC\_URL = '/static/'

STATICFILES\_DIRS = [ BASE\_DIR / "static" ]

1. Folder structure:

project/

├── static/

│ ├── css/style.css

│ └── images/logo.png

1. In template:

{% load static %}

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

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

**✅ Final Thoughts**

* DTL is powerful and integrates smoothly with Django views.
* It ensures logic stays in views/models and UI stays clean.
* For more power, you can also use **Jinja2**, but DTL is default and preferred in most cases.

**🧠 Django Template Variables and Template Inheritance**

**✅ 1. Passing Variables from Views to Templates**

In Django, you can pass data (like strings, numbers, lists, or even objects) from the view to the template using a **context dictionary**.

**📌 Example View (views.py):**

from django.shortcuts import render

from django.http import HttpResponse

def home(request):

return render(request, "home.html", {"name": "Teja"})

* "home.html" is the template being rendered.
* {"name": "Teja"} is the context dictionary. The key name is accessible in the HTML template.

**✅ 2. Using Variables in Templates (home.html)**

You can use the passed variable in the template using double curly braces {{ }}.

**📌 Example Template (home.html):**

{% extends "base.html" %}

{% block content %}

<h1>Welcome to the page, {{ name }}! (HTML)</h1>

{% endblock %}

* {{ name }} is replaced with "Teja" when the page is rendered.

**✅ 3. Combining Pages using Template Inheritance**

This is called **template inheritance**, where one base HTML page acts as a layout and other pages (like home.html) can extend it.

**📌 Base Template: base.html**

<!DOCTYPE html>

<html>

<head>

<title>Page Title</title>

</head>

<body bgcolor="cyan">

<h1>My First Heading</h1>

<p>My first paragraph.</p>

<!-- Content block that other pages can override -->

{% block content %}

{% endblock %}

</body>

</html>

* This is the **master layout** used by other templates.
* It defines a block called content that other templates can **fill in**.

**📌 Child Template: home.html**

{% extends "base.html" %}

{% block content %}

<h1>Welcome to the page, {{ name }}!</h1>

{% endblock %}

* {% extends "base.html" %} means this file uses base.html as its base layout.
* The content block replaces the block in base.html.

**🧠 Django: Add Two Numbers Dynamically with a Form**

**✅ Goal:**

Build a Django page that:

* Takes two numbers from the user using a form.
* Submits them to the backend.
* Adds them and shows the result dynamically.

**✅ 1. View Functions (views.py)**

from django.shortcuts import render

from django.http import HttpResponse

# Home page view to display the form

def home(request):

return render(request, "home.html", {"name": "Teja"})

# Addition logic

def add(request):

val1 = request.GET["num1"]

val2 = request.GET["num2"]

res = int(val1) + int(val2)

return render(request, "result.html", {"result": res})

* home() shows the form.
* add() handles the form data and returns the sum.

**✅ 2. URLs Setup (urls.py)**

from django.urls import path

from . import views

urlpatterns = [

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

path('add', views.add, name='add'), # add result page

]

Make sure urls.py is correctly included in the project's main urls.py.

**✅ 3. Template 1: home.html**

{% extends "base.html" %}

{% block content %}

<h1>Welcome to the page, {{ name }}!</h1>

<form action="{% url 'add' %}">

Enter first number: <input type="text" name="num1"><br><br>

Enter second number: <input type="text" name="num2"><br><br>

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

</form>

{% endblock %}

* The form uses the GET method (default in HTML forms).
* It sends the two inputs (num1, num2) to the /add route.

**✅ 4. Template 2: result.html**

{% extends "base.html" %}

{% block content %}

<h2>Result: {{ result }}</h2>

{% endblock %}

* Shows the result passed from the add() view.

**✅ 5. Base Template: base.html (Optional Layout File)**

<!DOCTYPE html>

<html>

<head>

<title>Django App</title>

</head>

<body bgcolor="lightblue">

<h1>Simple Django Calculator</h1>

{% block content %}

{% endblock %}

</body>

</html>

**🔄 How It Works:**

1. User visits / (home) → sees a form.
2. Fills in numbers and submits the form.
3. Browser sends GET request to /add?num1=5&num2=10.
4. add() view calculates the sum and sends it to result.html.
5. result.html displays 15.

**📁 Folder Structure**

app/

├── views.py

├── urls.py

└── templates/

├── base.html

├── home.html

└── result.html

**🧠 Django: Handling Forms Using POST Method**

**✅ Why Use POST?**

* **POST is more secure** than GET.
* **Data is not shown in the URL** (better for sensitive data).
* It's the standard method for submitting forms in real applications.

**✅ 1. Update views.py to Use POST**

from django.shortcuts import render

from django.http import HttpResponse

def home(request):

return render(request, "home.html", {"name": "Teja"})

def add(request):

if request.method == "POST":

val1 = request.POST["num1"]

val2 = request.POST["num2"]

res = int(val1) + int(val2)

return render(request, "result.html", {"result": res})

else:

return HttpResponse("Invalid request method")

✅ **Changes:**

* Check if request.method == "POST" to prevent errors on direct access.

**✅ 2. Update home.html Form to Use POST**

{% extends "base.html" %}

{% block content %}

<h1>Welcome to the page, {{ name }}!</h1>

<form method="POST" action="{% url 'add' %}">

**{% csrf\_token %}**

Enter first number: <input type="text" name="num1"><br><br>

Enter second number: <input type="text" name="num2"><br><br>

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

</form>

{% endblock %}

✅ **Important:**

* Use method="POST" in the form.
* Add {% csrf\_token %} for **security** (Cross Site Request Forgery protection).

**✅ 3. URLs (urls.py) – No Change Needed**

from django.urls import path

from . import views

urlpatterns = [

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

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

]

**✅ 4. Result Template (result.html)**

{% extends "base.html" %}

{% block content %}

<h2>Result: {{ result }}</h2>

{% endblock %}

**🔄 How It Works with POST:**

1. User visits / → sees the form.
2. Enters numbers → form sends data using **POST** to /add.
3. The add() view receives the data via request.POST.
4. It calculates the result and sends it to result.html.

**🧾 Steps to Integrate a Downloaded HTML Template into a Django Project**

**✅ 1. Download Template & Copy Index File**

* Download a Bootstrap or HTML website template from any free template provider (e.g., Google search, TemplateMo, HTML5 UP, etc.).
* Open the downloaded folder, find the index.html file.
* **Copy index.html** into your Django app’s templates folder (we’ll create this in step 2–4).

**✅ 2. Create a Django App**

If you don’t already have a Django project, start one:

django-admin startproject myproject

cd myproject

python manage.py startapp myapp

✅ Your structure now looks like:

myproject/

│

├── manage.py

├── myproject/

│ └── settings.py

├── myapp/

│ ├── views.py

│ └── templates/

│ └── index.html ← paste your HTML file here

**✅ 3. Configure URLs**

**In myapp/views.py:**

from django.shortcuts import render

def index(request):

return render(request, 'index.html')

**In myapp/urls.py (create if not exists):**

from django.urls import path

from . import views

urlpatterns = [

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

]

**In myproject/urls.py:**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path('', include('myapp.urls')), # include app URLs

]

**✅ 4. Create Static Folder and Place Files**

HTML templates usually come with folders like:

css/

js/

images/

fonts/

You need to place these inside a static folder that sits **next to manage.py**:

myproject/

│

├── static/

│ ├── css/

│ ├── js/

│ ├── images/

│ └── fonts/

├── manage.py

**✅ 5. Configure settings.py**

Add this to your settings.py:

import os

STATICFILES\_DIRS = [

os.path.join(BASE\_DIR, 'static')

]

STATIC\_URL = '/static/'

STATIC\_ROOT = os.path.join(BASE\_DIR, 'assets')

**✅ 6. Collect Static Files**

When ready for deployment or organizing, run:

python manage.py collectstatic

This will copy all static files from static/ to the assets/ directory (as per STATIC\_ROOT).

Now your assets/ folder will be auto-filled with static content like:

assets/

├── css/

├── js/

├── images/

**✅ 7. Link Static Files in HTML**

At the **top of index.html**, add:

{% load static %}

Replace all static paths like:

<link rel="stylesheet" href="css/fontawesome.css">

With:

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

✅ Example updated block:

{% load static %}

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

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

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

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

<img src="{% static 'images/banner.jpg' %}" alt="Banner">

**✅ Including Dynamic Data Using Models in Django**

**✅ models.py**

from django.db import models

# Define the Destination model

class Destination(models.Model): # Must inherit from models.Model

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

desc = models.TextField()

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

rating = models.IntegerField()

**✅ views.py**

from django.shortcuts import render

from .models import Destination

# View to pass a single Destination object to template

def index(request):

user1 = Destination() # Create instance without saving

user1.name = "John"

user1.desc = "iam john from us"

user1.rating = 25

user1.job = "trainer"

return render(request, "index.html", {"user1": user1})

⚠️ Even though not saved to DB, you can pass temporary object to template.

**✅ index.html**

<div class="text-center mt-3">

<span class="text-white">{{ user1.name }}</span>

<strong class="d-block text-white">{{ user1.job }}</strong>

<p class="text-white">{{ user1.desc }}</p>

<p class="text-white">Rating: {{ user1.rating }}</p>

</div>

**✅ Summary Steps**

1. **Create model** using models.Model and define fields.
2. **Create instance** in views and assign values.
3. **Pass to template** using context ({"user1": user1}).
4. **Use Django template tags** to access fields.

**📘 Displaying Multiple Objects in Django Template Using for Loop**

**✅ Objective:**

To display a list of objects dynamically in a Django HTML template using a for loop.

**🧠 Concept:**

* In Django, you can create multiple objects (e.g., user profiles) and pass them to a template as a list.
* In the HTML template, use a for loop to iterate over the list and display each object’s data.

**🧩 Step-by-Step Example**

**📁 1. views.py**

Create multiple Destination objects (without saving to DB) and pass them in a list.

from django.shortcuts import render

from .models import Destination # Import your model

def index(request):

# Create object 1

user1 = Destination()

user1.id = 1

user1.name = "John"

user1.desc = "I am John from US"

user1.rating = 25

user1.type = "trainer"

# Create object 2

user2 = Destination()

user2.id = 2

user2.name = "Jane"

user2.desc = "I am Jane from Dubai"

user2.rating = 22

user2.type = "tester"

# Create object 3

user3 = Destination()

user3.id = 3

user3.name = "Jessy"

user3.desc = "I am Jessy from Thailand"

user3.rating = 23

user3.type = "coder"

# Create a list of users

users = [user1, user2, user3]

# Pass the list to template

return render(request, "index.html", {"users": users})

**📁 2. index.html (template)**

Use {% for user in users %} to loop and display details.

<!DOCTYPE html>

<html>

<head>

<title>Destination Users</title>

</head>

<body>

<h1>Destination Users List</h1>

<ul>

{% for user in users %}

<li>

<h2>{{ user.name }}</h2>

<p>{{ user.desc }}</p>

<p>Rating: {{ user.rating }}</p>

<p>Type: {{ user.type }}</p>

<hr>

</li>

{% endfor %}

</ul>

</body>

</html>

**📝 Summary**

| **Step** | **Description** |
| --- | --- |
| 1️ | Create multiple instances of a model in views.py |
| 2️ | Add them to a list |
| 3️ | Pass the list as context to the template |
| 4️ | Use {% for item in list %} loop in the template |
| 5️ | Access individual attributes using {{ item.attribute }} |

**🧠 What is Django ORM?**

**Django ORM** is a powerful abstraction layer that allows developers to interact with relational databases (like PostgreSQL, MySQL, SQLite) using **Python objects instead of raw SQL**.

It maps Python **classes to database tables** and Python **objects to rows** in those tables.

**🔧 Basic Terminology**

| **Concept** | **Django Term** | **SQL Equivalent** |
| --- | --- | --- |
| Table | Model | Table |
| Row | Model Instance | Record |
| Column | Model Field | Column |
| Query | ORM Query | SQL Query |

**⚙️ Defining Models**

You define models in models.py inside your Django app:

from django.db import models

class Product(models.Model):

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

price = models.FloatField()

is\_available = models.BooleanField(default=True)

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

This will create a table called yourapp\_product with 4 fields in the database.

**🛠️ Field Types**

| **Field Type** | **Description** |
| --- | --- |
| CharField | String (varchar) with max length |
| TextField | Long text |
| IntegerField | Integer number |
| FloatField | Decimal number |
| BooleanField | True/False |
| DateTimeField | Date and time |
| ForeignKey | One-to-many relation |
| ManyToManyField | Many-to-many relation |
| OneToOneField | One-to-one relation |

**🔄 ORM Operations**

**📥 Create (Insert)**

product = Product(name="Milk", price=25.0)

product.save()

**📤 Read (Select)**

Product.objects.all() # Get all rows

Product.objects.filter(price\_\_gt=20) # WHERE price > 20

Product.objects.get(id=1) # Get one row (raises error if not found)

**✏️ Update**

product = Product.objects.get(id=1)

product.name = "Updated Name"

product.save()

**❌ Delete**

product = Product.objects.get(id=1)

product.delete()

**🔍 QuerySet Methods**

| **Method** | **Description** |
| --- | --- |
| all() | Returns all records |
| filter(\*\*kwargs) | Returns records matching condition |
| exclude(\*\*kwargs) | Opposite of filter |
| get(\*\*kwargs) | Returns single record |
| order\_by() | Sort results |
| count() | Number of records |
| exists() | Check if any record exists |
| first() / last() | First or last record |

**🔗 Filtering with Lookups**

| **Lookup** | **Example** |
| --- | --- |
| exact | price\_\_exact=100 |
| iexact | name\_\_iexact="milk" |
| contains | name\_\_contains="il" |
| icontains | name\_\_icontains="il" |
| gt, lt, gte, lte | price\_\_gt=50 |
| in | id\_\_in=[1, 2, 3] |
| isnull | description\_\_isnull=True |
| startswith | name\_\_startswith="M" |

**🔗 Relationships**

**1. One-to-Many (ForeignKey)**

class Category(models.Model):

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

class Product(models.Model):

category = models.ForeignKey(Category, on\_delete=models.CASCADE)

**2. One-to-One**

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

**3. Many-to-Many**

class Student(models.Model):

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

class Course(models.Model):

students = models.ManyToManyField(Student)

**🧪 Aggregation & Annotation**

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

Product.objects.aggregate(Avg('price'))

Product.objects.aggregate(Sum('price'))

**📋 Raw SQL (if needed)**

Product.objects.raw('SELECT \* FROM app\_product WHERE price > %s', [100])

**📎 Model Meta Options**

Inside your model:

class Product(models.Model):

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

class Meta:

ordering = ['-id']

verbose\_name = 'My Product'

**✅ Migration Commands Summary**

python manage.py makemigrations # Create migration files

python manage.py migrate # Apply migrations to DB

python manage.py sqlmigrate appname 0001 # See SQL query

**🧠 Best Practices**

* Always run makemigrations after changing models.
* Use get\_or\_create() to avoid duplicates.
* Use select\_related and prefetch\_related to optimize joins.
* Keep models.py clean and modular.

**🐘 Django with PostgreSQL & pgAdmin – Full Setup Notes**

**📦 What is PostgreSQL?**

* PostgreSQL is a powerful open-source **relational database system**.
* Highly robust and preferred for **production-level Django projects**.

**🌐 What is pgAdmin?**

* pgAdmin is a **browser-based GUI tool** to interact with your PostgreSQL databases easily.

**✅ Step 1: Install PostgreSQL & pgAdmin**

1. Go to: <https://www.postgresql.org/download/>
2. Choose your OS (Windows/macOS/Linux).
3. Download the **EDB installer**.
4. During installation:
   * Install **PostgreSQL + pgAdmin**
   * Set a **master password** (remember it)
   * Use default port: 5432

**🔓 Step 2: Launch pgAdmin in Browser**

* Open **pgAdmin** from Start Menu or Applications.
* It opens in your browser at:
* http://127.0.0.1:5050/browser/
* Enter the master password when prompted.

**🧱 Step 3: Create a Database in pgAdmin**

1. In pgAdmin: Right-click **Databases** → Create → **Database**
2. Name it something like: mydb
3. Leave owner as postgres
4. Save.

**👤 Step 4 (Optional): Create a PostgreSQL User**

1. Go to **Login/Group Roles** → Right-click → Create → **Login/Group Role**
2. Fill:
   * **Name**: myuser
   * **Password**: Set under "Definition" tab
   * **Privileges**: Check **Can login?**
3. Save.

**🐍 Step 5: Connect PostgreSQL to Django**

**🧩 1. Install PostgreSQL adapter**

pip install psycopg2-binary

**🛠️ 2. Update settings.py in your Django project**

DATABASES = {

'default': {

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

'NAME': 'mydb', # DB name from pgAdmin

'USER': 'postgres', # PostgreSQL username

'PASSWORD': 'your\_password', # PostgreSQL password

'HOST': 'localhost',

'PORT': '5432',

}

}

**🧩 3. Register your app in INSTALLED\_APPS**

INSTALLED\_APPS = [

'lugx.apps.LugxConfig',

'django.contrib.admin',

'django.contrib.auth',

...

]

**🛠️ Step 6: Migrations**

**🔧 Make & Apply Migrations**

python manage.py makemigrations

python manage.py migrate

* If you see: No changes detected, it means models are already synced.
* Your first migration creates a file: migrations/0001\_initial.py
  + Inside: model definitions and table creation commands

**🔁 To Reapply Migrations**

python manage.py makemigrations

python manage.py migrate

to create table

python manage.py sqlmigrate appname 0001

**🧪 Step 7: Verify Table Creation**

1. Open **pgAdmin**
2. Navigate: Databases → mydb → Schemas → public → Tables
3. Run query:
4. SELECT \* FROM your\_table\_name;

**🧾 Step 8: How to Add Data**

**✅ Option 1: Using Custom Page**

1. **In views.py:**

from .models import Destination

def index(request):

user = Destination.objects.all()

return render(request, "index.html", {"users": user})

1. **In urls.py:**

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

1. **In index.html:**

{% for u in users %}

<p>{{ u.name }}</p>

{% endfor %}

**✅ Option 2: Using Admin Panel**

1. **Register model in admin.py:**

from django.contrib import admin

from .models import Destination

admin.site.register(Destination)

1. **Create Superuser:**

python manage.py createsuperuser

* **Name**: teja
* **Email**: saitejast211.ys@gmail.com
* **Password**: tej1234

1. Run server and login:
2. python manage.py runserver

Visit: http://127.0.0.1:8000/admin

1. Add data manually through the admin panel.

**🧠 Extra Tips**

**🛠 Helpful Commands**

python manage.py help

python manage.py makemigrations

python manage.py migrate

python manage.py runserver

python manage.py createsuperuser

**🔁 Migration Troubleshooting**

If changes are not detected:

* Ensure models are modified correctly
* Confirm app is listed in INSTALLED\_APPS
* Delete migrations/ folder and re-run if needed (use with caution)

**✅ Final Checklist**

| **Task** | **Done** |
| --- | --- |
| PostgreSQL & pgAdmin installed | ✅ |
| Database created | ✅ |
| psycopg2-binary installed | ✅ |
| Django connected to PostgreSQL | ✅ |
| Migrations applied | ✅ |
| Tables verified in pgAdmin | ✅ |
| Admin user created | ✅ |
| Data viewed using ORM | ✅ |

**✅ Django User Registration System – Complete Notes**

**📁 1. Create a New Django App**

python manage.py startapp accounts

**📄 2. accounts/urls.py**

Create a urls.py inside the accounts app and define the registration path:

from django.urls import path

from . import views

urlpatterns = [

path('register', views.register, name='register'), # Avoid extra spaces here!

]

**📄 3. Include accounts URLs in the Main Project urls.py**

In your main urls.py file (e.g., teja/urls.py), include the accounts app URLs:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path("accounts/", include('accounts.urls')), # This prefixes your URL

]

So the final URL becomes: http://127.0.0.1:8000/accounts/register

**🧭 4. Add Registration Link to index.html**

In your navigation bar:

<li class="nav-item bg-primary">

<a class="nav-link click-scroll" href="/accounts/register">Register</a>

</li>

✅ Note: Always use the correct path without spaces.

**🧾 5. register.html – Registration Form**

Use a clean form with POST method and csrf\_token:

<form action="register" method="POST" style="max-width: 400px; margin: auto; padding: 20px; border: 1px solid #ccc; border-radius: 8px;">

{% csrf\_token %}

<!-- Messages -->

{% if messages %}

{% for message in messages %}

<p style="color: red;">{{ message }}</p>

{% endfor %}

{% endif %}

<label>First Name</label>

<input type="text" name="first\_name" placeholder="Enter name" required>

<label>Last Name</label>

<input type="text" name="last\_name" placeholder="Enter last name" required>

<label>Username</label>

<input type="text" name="username" placeholder="Enter username" required>

<label>Email</label>

<input type="email" name="email" placeholder="Enter email" required>

<label>Password</label>

<input type="password" name="password1" placeholder="Enter password" required>

<label>Confirm Password</label>

<input type="password" name="password2" placeholder="Confirm password" required>

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

</form>

**⚙️ 6. views.py – User Registration Logic with Messages**

from django.shortcuts import render, redirect

from django.contrib import messages

from django.contrib.auth.models import User

def register(request):

if request.method == "POST":

first\_name = request.POST["first\_name"]

last\_name = request.POST["last\_name"]

username = request.POST["username"]

password1 = request.POST["password1"]

password2 = request.POST["password2"]

email = request.POST["email"]

if password1 == password2:

if User.objects.filter(username=username).exists():

messages.info(request, 'Username already exists')

return redirect('register')

elif User.objects.filter(email=email).exists():

messages.info(request, 'Email already exists')

return redirect('register')

else:

user = User.objects.create\_user(

first\_name=first\_name,

last\_name=last\_name,

username=username,

email=email,

password=password1

)

user.save()

messages.success(request, 'User registered successfully')

return redirect("/")

else:

messages.error(request, "Passwords do not match")

return redirect('register')

return render(request, "register.html")

**✅ 7. Django Setup Requirements for Messages to Work**

Ensure the following are correctly configured in settings.py:

**✅ INSTALLED\_APPS**

INSTALLED\_APPS = [

...

'django.contrib.messages',

]

**✅ MIDDLEWARE**

MIDDLEWARE = [

...

'django.contrib.sessions.middleware.SessionMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

]

**✅ TEMPLATES Context Processor**

'OPTIONS': {

'context\_processors': [

...

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

],

},

**📦 8. Use Bootstrap Alerts for Better Styling (Optional)**

Instead of plain <p> tags:

{% if messages %}

{% for message in messages %}

<div class="alert alert-warning">{{ message }}</div>

{% endfor %}

{% endif %}

Add Bootstrap CSS in your HTML head if not already:

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">

**🎉 Final Result**

* Validates passwords
* Checks for existing users
* Displays friendly messages
* Handles success and error states gracefully

**✅ Django Login System – Step-by-Step Notes**

**🔹 1. Create Login Page – login.html**

This HTML form lets users input their username and password and submit it securely using POST.

<form style="max-width: 400px; margin: 50px auto; padding: 20px; border: 1px solid #ccc; border-radius: 8px; font-family: Arial, sans-serif;"

action="{% url 'login' %}" method="POST">

{% csrf\_token %}

<div style="margin-bottom: 15px;">

<label for="username">Username</label>

<input type="text" name="user" id="username" placeholder="Enter user name"

style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 4px;">

</div>

<div style="margin-bottom: 15px;">

<label for="password">Password</label>

<input type="password" name="password" id="password" placeholder="Enter password"

style="width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 4px;">

</div>

<button type="submit"

style="background-color: #007bff; color: white; padding: 10px 20px; border: none; border-radius: 4px; cursor: pointer;">

Submit

</button>

</form>

<!-- Display error messages -->

{% if messages %}

{% for message in messages %}

<div class="alert alert-warning">{{ message }}</div>

{% endfor %}

{% endif %}

**🔹 2. Create URL Path for Login – urls.py**

Add a URL route for the login view:

from django.urls import path

from . import views

urlpatterns = [

path('login/', views.login, name="login"),

]

**🔹 3. Update Navigation – Example from index.html**

Create a login link for users to access the login page:

<li class="nav-item bg-primary">

<a class="nav-link click-scroll" href="/accounts/login">Login</a>

</li>

**🔹 4. Backend View for Login – views.py**

Handle the form data and authenticate the user using Django’s built-in auth module.

from django.shortcuts import render, redirect

from django.contrib import messages

from django.contrib.auth.models import User, auth # auth contains login, logout, authenticate

def login(request):

if request.method == "POST":

username = request.POST['user']

password = request.POST['password']

# Check credentials

user = auth.authenticate(username=username, password=password)

if user is not None:

# Successful login

auth.login(request, user)

return redirect('/')

else:

# Failed login

messages.info(request, 'Invalid credentials')

return redirect('login')

return render(request, 'login.html')

**✅ Summary of Login Flow**

| **Step** | **Action** |
| --- | --- |
| 1 | User fills login form and submits it (POST) |
| 2 | Django receives POST request and extracts username and password |
| 3 | auth.authenticate() checks if credentials are valid |
| 4 | If valid, user is logged in using auth.login() and redirected |
| 5 | If invalid, error message is shown using messages |

**🔒 Tips for Secure Login**

* Always use POST method for login.
* Always include {% csrf\_token %} inside forms.
* Never log sensitive data like passwords.
* You can add login limits or CAPTCHA to prevent brute force attacks.

**✅ Django Logout Functionality – Step-by-Step Notes**

**🔹 1. Navigation Menu Using user.is\_authenticated**

Show different navigation items depending on whether the user is logged in:

{% if user.is\_authenticated %}

<!-- Show username and logout -->

{{ user.first\_name }}

<li class="nav-item bg-primary">

<a class="nav-link click-scroll" href="/accounts/logout">Logout</a>

</li>

{% else %}

<!-- Show login and register links -->

<li class="nav-item bg-primary">

<a class="nav-link click-scroll" href="/accounts/register">Register</a>

</li>

<li class="nav-item bg-primary">

<a class="nav-link click-scroll" href="/accounts/login">Login</a>

</li>

{% endif %}

✅ user.is\_authenticated is a built-in Django property to check if the user is logged in.

**🔹 2. Add Logout URL – urls.py**

Define the logout route and link it to the logout view:

from django.urls import path

from . import views

urlpatterns = [

path('logout/', views.logout, name="logout"),

]

**🔹 3. Create Logout View – views.py**

Use Django's auth.logout() function to log the user out and redirect to the login page:

from django.contrib.auth import logout as django\_logout

from django.shortcuts import redirect

def logout(request):

django\_logout(request)

return redirect('login') # You can also redirect to homepage if desired

📌 auth.logout(request) clears the session and logs the user out.

**✅ How Logout Works**

| **Step** | **Action** |
| --- | --- |
| 1 | User clicks the "Logout" link. |
| 2 | A GET request is sent to /accounts/logout. |
| 3 | The logout() view calls auth.logout(request). |
| 4 | The user session is cleared (logged out). |
| 5 | User is redirected to the login page. |

**🧾 Django + MySQL CRUD Project (With CSS Styling)**

**✅ 1. Install Dependencies**

**1.1. Install Django & MySQL Client**

**pip install django mysqlclient**

⚠️ Make sure MySQL is installed and running. Also, you need the mysql\_config tool available (e.g., via mysql-devel or libmysqlclient-dev).

**✅ 2. Create Django Project and App**

**django-admin startproject myproject**

**cd myproject**

**python manage.py startapp myapp**

**✅ 3. Connect to MySQL Database**

**Edit myproject/settings.py:**

**DATABASES = {**

**'default': {**

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

**'NAME': 'yourdbname',**

**'USER': 'yourusername',**

**'PASSWORD': 'yourpassword',**

**'HOST': 'localhost',**

**'PORT': '3306',**

**}**

**}**

Then create the DB in MySQL:

**CREATE DATABASE yourdbname;**

**✅ 4. Register App and Static Files**

In settings.py:

**INSTALLED\_APPS = [**

**...**

**'myapp',**

**]**

# Static files setup

**STATIC\_URL = '/static/'**

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

**✅ 5. Create Student Model**

In myapp/models.py:

**from django.db import models**

**class Student(models.Model):**

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

**age = models.IntegerField()**

**email = models.EmailField()**

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

**return self.name**

**✅ 6. Make Migrations**

**python manage.py makemigrations**

**python manage.py migrate**

**✅ 7. Create Superuser (Optional for Admin Panel)**

**python manage.py createsuperuser**

**✅ 8. Register Model in Admin**

In myapp/admin.py:

**from django.contrib import admin**

**from .models import Student**

**admin.site.register(Student)**

**✅ 9. Create Forms**

In myapp/forms.py:

**from django import forms**

**from .models import Student**

**class StudentForm(forms.ModelForm):**

**class Meta:**

**model = Student**

**fields = ['name', 'age', 'email']**

**✅ 10. CRUD Views in views.py**

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

**from .models import Student**

**from .forms import StudentForm**

**def student\_list(request):**

**students = Student.objects.all()**

**return render(request, 'student\_list.html', {'students': students})**

**def student\_create(request):**

**form = StudentForm(request.POST or None)**

**if form.is\_valid():**

**form.save()**

**return redirect('student\_list')**

**return render(request, 'student\_form.html', {'form': form})**

**def student\_update(request, pk):**

**student = get\_object\_or\_404(Student, pk=pk)**

**form = StudentForm(request.POST or None, instance=student)**

**if form.is\_valid():**

**form.save()**

**return redirect('student\_list')**

**return render(request, 'student\_form.html', {'form': form})**

**def student\_delete(request, pk):**

**student = get\_object\_or\_404(Student, pk=pk)**

**if request.method == 'POST':**

**student.delete()**

**return redirect('student\_list')**

**return render(request, 'student\_confirm\_delete.html', {'student': student})**

**✅ 11. URLs Setup**

**myapp/urls.py**

**from django.urls import path**

**from . import views**

**urlpatterns = [**

**path('', views.student\_list, name='student\_list'),**

**path('create/', views.student\_create, name='student\_create'),**

**path('update/<int:pk>/', views.student\_update, name='student\_update'),**

**path('delete/<int:pk>/', views.student\_delete, name='student\_delete'),**

**]**

**Include app urls in myproject/urls.py**

**from django.contrib import admin**

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

**urlpatterns = [**

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

**path('', include('myapp.urls')),**

**]**

**✅ 12. Create Templates**

**12.1 File Structure**

myapp/

│

├── templates/

│ └── student\_list.html

│ └── student\_form.html

│ └── student\_confirm\_delete.html

│

├── static/

└── myapp/

└── style.css

**✅ 13. style.css**

Located at myapp/static/myapp/style.css:

**body {**

**font-family: Arial, sans-serif;**

**background-color: #f4f6f8;**

**margin: 0;**

**padding: 30px;**

**}**

**.container {**

**max-width: 800px;**

**background: white;**

**margin: auto;**

**padding: 2rem;**

**border-radius: 10px;**

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

**}**

**h2 {**

**text-align: center;**

**margin-bottom: 1.5rem;**

**}**

**a {**

**color: #2980b9;**

**text-decoration: none;**

**font-weight: bold;**

**}**

**a:hover {**

**text-decoration: underline;**

**}**

**ul {**

**list-style: none;**

**padding: 0;**

**}**

**li {**

**padding: 10px;**

**background: #ecf0f1;**

**margin-bottom: 10px;**

**border-radius: 5px;**

**display: flex;**

**justify-content: space-between;**

**align-items: center;**

**}**

**button {**

**padding: 6px 12px;**

**background-color: #2980b9;**

**color: white;**

**border: none;**

**border-radius: 4px;**

**cursor: pointer;**

**}**

**button:hover {**

**background-color: #1c5980;**

**}**

**✅ 14. Templates**

**student\_list.html**

**{% load static %}**

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

**<div class="container">**

**<h2>Student List</h2>**

**<a href="{% url 'student\_create' %}">➕ Add Student</a>**

**<ul>**

**{% for student in students %}**

**<li>**

**{{ student.name }} ({{ student.age }}) | {{ student.email }}**

**<span>**

**<a href="{% url 'student\_update' student.id %}">✏️ Edit</a>**

**<a href="{% url 'student\_delete' student.id %}">❌ Delete</a>**

**</span>**

**</li>**

**{% empty %}**

**<li>No students found.</li>**

**{% endfor %}**

**</ul>**

**</div>**

**student\_form.html**

**{% load static %}**

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

**<div class="container">**

**<h2>Student Form</h2>**

**<form method="post">**

**{% csrf\_token %}**

**{{ form.as\_p }}**

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

**</form>**

**</div>**

**student\_confirm\_delete.html**

**{% load static %}**

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

**<div class="container">**

**<h2>Delete Confirmation</h2>**

**<p>Are you sure you want to delete <strong>{{ student.name }}</strong>?</p>**

**<form method="post">**

**{% csrf\_token %}**

**<button type="submit">Yes, Delete</button>**

**</form>**

**<a href="{% url 'student\_list' %}">Cancel</a>**

**</div>**

**✅ 15. Run Server**

**python manage.py runserver**

Visit: <http://127.0.0.1:8000/>

**🏁 Final Features You Now Have**

* ✅ Django connected to MySQL
* ✅ CRUD operations
* ✅ Admin panel
* ✅ Styled HTML templates using CSS
* ✅ Clean file organization

**How to deploy MySQL**

for your Django project.

Since **Render does not provide MySQL hosting**, we will use a third-party service like:

* [🔵 Railway](https://railway.app/)
* [🟣 PlanetScale](https://planetscale.com/)
* [🟡 ClearDB (Heroku)](https://www.cleardb.com/) (less common now)

**✅ We’ll use: Railway (Easy and free tier available)**

**🚀 Step-by-Step: Deploy MySQL on Railway**

**🧱 Step 1: Sign in to** [**Railway**](https://railway.app/)

1. Go to [https://railway.app](https://railway.app/)
2. Click **Login** → Sign in with **GitHub** or **email**

**🛠 Step 2: Create MySQL Database**

1. Click **“New Project”**
2. Select **“Provision MySQL”**
3. Choose **“MySQL”**

✅ Railway will automatically create your MySQL database and show a **connection URL**.

**🔑 Step 3: Get Your Database Credentials**

Once the database is created:

1. Click the **“Connect”** button
2. Select **MySQL**
3. Copy this URL:

mysql://username:password@host:port/database

Or break it into:

| **Key** | **Value (example)** |
| --- | --- |
| DB\_NAME | railway |
| DB\_USER | root |
| DB\_PASS | yourpassword |
| DB\_HOST | containers-us-west-12.railway.app |
| DB\_PORT | 1234 (typically 3306 or Railway port) |

**🧩 Step 4: Update Your Django settings.py**

In your Django project:

Import os

DATABASES = {

'default': {

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

'NAME': os.environ.get('DB\_NAME'),

'USER': os.environ.get('DB\_USER'),

'PASSWORD': os.environ.get('DB\_PASSWORD'),

'HOST': os.environ.get('DB\_HOST'),

'PORT': os.environ.get('DB\_PORT', '3306'),

}

}

**🌐 Step 5: Set Environment Variables on Render**

When deploying your Django app on **Render**, go to:

* **Dashboard → Environment**
* Add the credentials as environment variables:

DB\_NAME=railway

DB\_USER=root

DB\_PASSWORD=yourpassword

DB\_HOST=containers-xyz.railway.app

DB\_PORT=1234

**🧪 Step 6: Test MySQL Connection Locally**

Install mysqlclient if not already:

pip install mysqlclient

Then run:

python manage.py migrate

python manage.py runserver

If it connects to the Railway-hosted MySQL database successfully, you’re all set to deploy.

**🟢 Final Step: Run Migrations on Render**

After deploying your Django project on Render:

1. Open **Shell** or **Deploy logs terminal**
2. Run:

python manage.py migrate

python manage.py collectstatic --noinput

**🌍 How to Deploy a Django Project on Render**

**✅ 1. Prepare Your Django Project**

**1.1 Create and activate a virtual environment**

python -m venv venv

# Activate it:

# On Windows:

venv\Scripts\activate

# On Mac/Linux:

source venv/bin/activate

**1.2 Install necessary packages**

**pip install django gunicorn**

If you're using static files:

**pip install whitenoise**

**1.3 Generate requirements.txt**

**pip freeze > requirements.txt**

This file tells Render what dependencies your project needs.

**1.4 Configure settings.py**

Open yourproject/settings.py and update:

**a. Allowed Hosts:**

**ALLOWED\_HOSTS = ['\*']**

**b. Static files:**

At the bottom of settings.py:

**STATIC\_URL = '/static/'**

**STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')**

**c. If using WhiteNoise (optional but recommended):**

**MIDDLEWARE = [**

**'whitenoise.middleware.WhiteNoiseMiddleware',**

**...**

**]**

**STATICFILES\_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'**

**1.5 Run collectstatic (if using static files)**

**python manage.py collectstatic**

**✅ 2. Create a Procfile**

In the root directory (same as manage.py), create a file named Procfile (no extension). Add:

**web: gunicorn yourproject.wsgi:application**

Replace yourproject with the folder name that contains settings.py.

**✅ 3. Push Code to GitHub**

git init

git add .

git commit -m "Ready for Render"

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

git push -u origin main

**✅ 4. Deploy on Render**

1. Go to [https://render.com](https://render.com/)
2. Sign in with GitHub
3. Click **"New" → "Web Service"**
4. Select your repository

**Fill Render Deployment Settings:**

| **Setting** | **Value** |
| --- | --- |
| Name | your-app-name |
| Root Directory | . (dot) or leave it blank |
| Runtime | Python |
| Build Command | **pip install -r requirements.txt** |
| Start Command | **gunicorn yourproject.wsgi:application** |

Again, replace yourproject with your actual Django project name.

1. Click **"Create Web Service"**

Render will now build and deploy your Django project! 🎉

**✅ 5. (Optional) Handle Environment Variables (.env)**

If you use secrets like SECRET\_KEY, DEBUG, etc., do **not** hardcode them.

1. Create .env file:

**SECRET\_KEY=your-secret-key**

**DEBUG=False**

1. In settings.py, access them using:

**import os**

**SECRET\_KEY = os.environ.get('SECRET\_KEY', 'fallback-secret')**

**DEBUG = os.environ.get('DEBUG') == 'True'**

1. In Render dashboard → your web service → **Environment → Add Environment Variables**

**✅ 6. (Optional) Connect to PostgreSQL**

1. On Render → create a new PostgreSQL database.
2. Add the credentials as environment variables.
3. Update settings.py to use those credentials via os.environ.

**🧪 7. Test the Live Site**

* Open your Render app URL
* Check if all pages, forms, and static files work
* You're live on the internet!

**📌 Final Checklist**

* requirements.txt
* Procfile
* gunicorn installed
* ALLOWED\_HOSTS set
* Static files configured
* GitHub repo pushed
* Render app created

**🏁 Done!**

You’ve successfully deployed your Django project to Render 🎉

<https://django-crud-1-deqj.onrender.com>

**🌤️ Simple Weather App in Django – Step-by-Step Notes**

**✅ 1. Setup Django Project**

**🔹 Create project & app**

django-admin startproject weatherproject

cd weatherproject

python manage.py startapp weatherapp

**✅ 2. Register the App**

**🔹 In weatherproject/settings.py, add:**

INSTALLED\_APPS = [

...

'weatherapp',

]

**✅ 3. Set Up URLs**

**🔹 In weatherproject/urls.py:**

from django.contrib import admin

from django.urls import path

from weatherapp import views

urlpatterns = [

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

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

]

**✅ 4. Create View**

**🔹 In weatherapp/views.py:**

from django.shortcuts import render

import requests

def home(request):

weather\_data = {}

if request.method == "POST":

city = request.POST.get('city')

api\_key = 'YOUR\_API\_KEY' # 🔑 Replace with your OpenWeatherMap API Key

url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api\_key}&units=metric"

response = requests.get(url)

if response.status\_code == 200:

data = response.json()

weather\_data = {

'city': city,

'temperature': data['main']['temp'],

'description': data['weather'][0]['description'],

'icon': data['weather'][0]['icon'],

}

else:

weather\_data['error'] = "City not found!"

return render(request, 'weatherapp/home.html', {'weather': weather\_data})

**✅ 5. Install Requests Library**

pip install requests

**✅ 6. Create Template Directory**

**🔹 In weatherapp/, create:**

weatherapp/

├── templates/

│ └── weatherapp/

│ └── home.html

**✅ 7. Create Template home.html**

<!DOCTYPE html>

<html>

<head>

<title>Weather App</title>

</head>

<body>

<h1>Simple Weather App 🌤️</h1>

<form method="POST">

{% csrf\_token %}

<input type="text" name="city" placeholder="Enter city name">

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

</form>

{% if weather %}

{% if weather.error %}

<p>{{ weather.error }}</p>

{% else %}

<h2>Weather in {{ weather.city }}</h2>

<p>Temperature: {{ weather.temperature }}°C</p>

<p>Description: {{ weather.description }}</p>

<img src="http://openweathermap.org/img/wn/{{ weather.icon }}@2x.png" alt="weather icon">

{% endif %}

{% endif %}

</body>

</html>

**✅ 8. Run Server**

python manage.py runserver

Open in browser: [http://127.0.0.1:8000](http://127.0.0.1:8000/)

**✅ 9. Get API Key from OpenWeather**

* Go to: <https://openweathermap.org/api>
* Sign up → Get your API Key
* Replace in views: api\_key = 'YOUR\_API\_KEY'