**Timeline of Architectural Pattern Evolution**

**1960s – Monolithic Architecture**

* All logic, UI, and data access in a **single program**.
* No clear separation of concerns.
* Suitable for mainframes and simple desktop apps.

**Example**: Early COBOL or FORTRAN programs

**1970s – Modular Programming**

* Code organized into **modules or procedures**.
* Encouraged reusability and separation of logic.
* Still tightly coupled, but a big improvement over monolithic code.

**Languages**: Pascal, C

**1980s – Layered Architecture**

* Also called **N-tier architecture**.
* Divides applications into logical layers:
  + UI → Business Logic → Data Access
* Supports separation of concerns, better testing, and scaling.
* Still widely used today in enterprise and web development.

**1990s – MVC (Model-View-Controller)**

* First introduced in 1979 (Smalltalk), but **popularized in the 1990s** in GUI frameworks and web apps.
* Became the foundation for many web frameworks (like Django, Rails, ASP.NET)
* Encouraged **separation of UI and logic**, improving maintainability.

**2000s – SOA (Service-Oriented Architecture)**

* Systems started being **decomposed into services** that communicate over a network (usually via XML/SOAP).
* Allowed **reusability and interoperability** across platforms.
* Precursor to microservices, but heavier and less agile.

**2010s – Microservices**

* Evolved from SOA, but with **lighter, independently deployable services**.
* Each service handles a specific business function.
* Uses REST, gRPC(Google Remote Procedure Call), message queues, etc.

Enabled massive scaling (e.g., Netflix, Amazon)

**2020s – Event-Driven & Serverless Architectures**

* Focus on **asynchronous communication** and **on-demand compute**.
* Serverless (like AWS Lambda) abstracts infrastructure.
* Event-driven architecture supports real-time responsiveness and loose coupling.

Popular in modern cloud-native and IoT applications.

| **Decade** | **Pattern** | **Key Feature** |
| --- | --- | --- |
| 1960s | Monolithic | All-in-one app |
| 1970s | Modular | Separated into modules |
| 1980s | Layered | UI, logic, DB layers |
| 1990s | MVC | Separated model, view, controller |
| 2000s | SOA | Distributed services via contracts |
| 2010s | Microservices | Independent RESTful services |
| 2020s | Event-Driven, Serverless | Reactive, scalable, cloud-native |

**Key Drivers of Evolution:**

* **Scaling needs** (user traffic, services)
* **Maintainability** (ease of updates)
* **Technology advancements** (containers, cloud, async I/O)
* **Business agility** (faster deployment, CI/CD)

**Popular Architectural Patterns in Python**

These patterns describe how to organize your entire **application's structure** — including how components interact.

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

* **Framework**: Django
* **Why it's popular**: Clean separation, batteries-included
* **Structure**:
  + **Model**: Handles database/data layer
  + **View**: Business logic / controller
  + **Template**: Renders HTML for the user (presentation)

# views.py

from django.shortcuts import render

from .models import Product

def product\_list(request):

products = Product.objects.all()

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

**Used for**: Web apps, admin dashboards, CMSs  
Django powers sites like Instagram, Disqus, Pinterest.

**2. Model-View-Controller (MVC)**

* **Frameworks**: Flask, PyQt, Kivy
* **Why it's popular**: Simple, flexible
* **Structure**:
  + **Model**: Data layer
  + **View**: Presentation/UI
  + **Controller**: Handles input, updates model/view

# Flask MVC example

@app.route("/books")

def get\_books():

books = Book.query.all() # Model

return render\_template("books.html", books=books) # View

**Used for**: GUIs, web servers, minimal apps  
Flask is a go-to micro-framework for APIs and microservices.

**3. Layered Architecture (N-tier)**

* **Frameworks**: Any (Flask, FastAPI, Django)
* **Why it's popular**: Clean separation of concerns, easy to test/scale
* **Structure**:
  + **Presentation Layer** (UI/API)
  + **Service Layer** (business logic)
  + **Repository Layer** (data access)

UI (FastAPI Route)

↓

Service (Business Logic)

↓

Repository (DB)

**Used for**: Scalable web apps, APIs, enterprise systems

**4. Microservices Architecture**

* **Tools**: FastAPI, Flask, Celery, RabbitMQ, Docker
* **Why it's popular**: Scalable, deployable, decoupled
* **Structure**:
  + Each service is a self-contained Python app (e.g., user-service, order-service)
  + Communicate via HTTP/REST or message queues

**Used for**: Large-scale apps, cloud-based apps  
Popular with FastAPI + Docker + Kubernetes setups

**5. Event-Driven Architecture**

* **Tools**: Celery, Redis, RabbitMQ, Kafka
* **Why it's popular**: Async, decoupled, real-time handling
* **Structure**:
  + Producer emits events
  + Consumer listens and reacts (e.g., background tasks)

# Celery task

@app.task

def send\_email(to, subject):

# send email here

**Used for**: Background jobs, real-time systems, ETL pipelines

**6. RESTful Architecture**

* **Frameworks**: FastAPI, Flask-RESTful, Django REST Framework
* **Why it's popular**: Standard for APIs
* **Structure**:
  + Resources are accessed using HTTP verbs: GET, POST, PUT, DELETE

@app.get("/users/{id}")

def get\_user(id: int):

return {"id": id, "name": "Alice"}

**Used for**: Web APIs, mobile backends, headless services

**7. Clean Architecture**

* **Conceptual** (not tied to a framework)
* **Why it's popular**: Testable, scalable, framework-agnostic
* **Structure**:
  + **Entities** (core business logic)
  + **Use Cases / Interactors**
  + **Interfaces / Adapters**
  + **Frameworks & Drivers** (e.g., FastAPI)

**Used for**: Enterprise systems, projects requiring long-term maintainability

| **Architectural Pattern** | **Main Use** | **Popular Tools/Frameworks** |
| --- | --- | --- |
| MVT (Model-View-Template) | Web apps | Django |
| MVC | Web & GUI apps | Flask, PyQt, Kivy |
| Layered (N-Tier) | Business apps, scalable APIs | Flask, FastAPI, Django |
| Microservices | Cloud-native, distributed apps | FastAPI, Docker, Celery |
| Event-Driven | Async/background jobs | Celery, RabbitMQ, Kafka |
| RESTful | APIs | FastAPI, Flask, DRF |
| Clean Architecture | Maintainable, testable apps | Framework-agnostic |

**What is MVC?**

**MVC** stands for **Model–View–Controller** — a **software architectural pattern** used to separate concerns in application development. It organizes code into three interconnected components:

**MVC Components:**

| **Component** | **Responsibility** | **Analogy** |
| --- | --- | --- |
| **Model** | Handles the **data** and **business logic** | Your database or backend |
| **View** | Handles the **UI** / what the user sees | HTML page or UI layer |
| **Controller** | Handles **user input** and updates model/view | The logic in between |

**Common Frameworks Using MVC:**

| **Framework** | **Language** | **Architecture** |
| --- | --- | --- |
| Django | Python | MVT (a variant of MVC) |
| Flask | Python | You can implement MVC manually |
| Ruby on Rails | Ruby | Strict MVC |
| ASP.NET MVC | C# | Classic MVC |

**Django – MVT (Model-View-Template)**

**What is MVT?**

| **MVT Component** | **Role** |
| --- | --- |
| **Model** | Manages data (DB ORM) |
| **View** | Business logic (controller) |
| **Template** | HTML rendering (UI) |

## What Are HTTP Response Codes?

HTTP status codes are **three-digit numbers** sent by a server in response to a client (browser or API) request.  
They indicate **whether the request was successful**, failed, redirected, or had some issue.

## Categories of HTTP Status Codes

| **Category** | **Code Range** | **Meaning** |
| --- | --- | --- |
| **1xx** | 100–199 | Informational responses |
| **2xx** | 200–299 | Success |
| **3xx** | 300–399 | Redirection |
| **4xx** | 400–499 | Client errors (your request) |
| **5xx** | 500–599 | Server errors (their problem) |

## Common HTTP Response Codes (Cheat Sheet)

### 1xx – Informational

| **Code** | **Meaning** |
| --- | --- |
| 100 | Continue |
| 101 | Switching Protocols |

### 2xx – Success

| **Code** | **Meaning** |
| --- | --- |
| 200 | OK – The request succeeded |
| 201 | Created – Resource was created |
| 202 | Accepted – Processing started |
| 204 | No Content – Success, no body |

### 3xx – Redirection

| **Code** | **Meaning** |
| --- | --- |
| 301 | Moved Permanently |
| 302 | Found (Temporary redirect) |
| 304 | Not Modified (for caching) |

### 4xx – Client Errors

| **Code** | **Meaning** |
| --- | --- |
| 400 | Bad Request – Invalid input |
| 401 | Unauthorized – Login required |
| 403 | Forbidden – No permission |
| 404 | Not Found – Resource missing |
| 405 | Method Not Allowed |
| 409 | Conflict – Duplicate or mismatch |
| 422 | Unprocessable Entity – Bad format in correct request |

### 5xx – Server Errors

| **Code** | **Meaning** |
| --- | --- |
| 500 | Internal Server Error |
| 502 | Bad Gateway |
| 503 | Service Unavailable |
| 504 | Gateway Timeout |

**How HTTP Responses Work in Django**

In Django, every view must return an **HTTP response** — typically using:

* HttpResponse (basic)
* JsonResponse (for APIs)
* Class-based views (HttpResponse under the hood)
* HttpResponseNotFound, HttpResponseForbidden, etc. for common status codes
* raise Http404 for 404 errors

**Basic HTTP Response Example**

from django.http import HttpResponse

def hello\_view(request):

return HttpResponse("Hello!", status=200) # 200 OK

**Returning JSON with a Status Code**

from django.http import JsonResponse

def data\_view(request):

data = {"message": "Success"}

return JsonResponse(data, status=200) # 200 OK

**Common Status Code Responses in Django**

| **Code Status** | **Use Case** | **Django Response Class / Method** |
| --- | --- | --- |
| 200 OK | Successful request | HttpResponse() or JsonResponse() |
| 201 Created | Created resource | JsonResponse(data, status=201) |
| 204 No Content | No body returned | HttpResponse(status=204) |
| 400 Bad Request | Invalid input | HttpResponseBadRequest() |
| 401 Unauthorized | Not built-in; use middleware | Custom response with status=401 |
| 403 Forbidden | No permission | HttpResponseForbidden() |
| 404 Not Found | Resource not found | raise Http404("message") |
| 500 Internal Server Error | Server issue | Let Django handle it, or custom middleware |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Framework Comparison:-**

**1. Django**

* **Best for**: Large-scale, full-featured web applications (e.g., content management systems, marketplaces).
* **Features**:
  + “Batteries included” – comes with admin panel, ORM, authentication, etc.
  + Follows **MVC/MVT** pattern.
  + Great for rapid development.
* **Use cases**: Instagram, Pinterest, Disqus.
* **Pros**:
  + Secure, scalable.
  + Large community and ecosystem.
  + Reduces boilerplate code.
* **Cons**:
  + Heavy for simple projects.
  + Less flexibility due to conventions.

**2. Flask**

* **Best for**: Small to medium-sized projects, or when you want more control over components.
* **Features**:
  + Lightweight and minimalist.
  + Microframework – you choose your own libraries (ORM, forms, etc.).
* **Use cases**: Netflix APIs, Reddit microservices.
* **Pros**:
  + Very flexible.
  + Easy to learn.
  + Perfect for building APIs.
* **Cons**:
  + Not as many built-in tools.
  + You manage more infrastructure manually.

**3. FastAPI**

* **Best for**: Building high-performance APIs and modern microservices.
* **Features**:
  + Based on Starlette and Pydantic.
  + Async support (great with WebSockets, background tasks).
  + Automatic OpenAPI docs (Swagger).
* **Use cases**: Machine learning APIs, microservices, real-time apps.
* **Pros**:
  + Blazing fast (thanks to async).
  + Easy validation and typing.
  + Modern syntax (Python 3.7+).
* **Cons**:
  + Newer, so fewer plugins than Django/Flask.
  + Learning curve with async and Pydantic.

<https://code.visualstudio.com/download>

**Module 1: Introduction to Django**

**1. Overview of Django Framework**

**Django** is a high-level Python web framework that encourages rapid development and clean, pragmatic design.

**Why Django?**

* Built-in tools to handle common web dev tasks (auth, forms, admin panel)
* Emphasizes security, scalability, and reusability
* Follows the **“batteries-included”** philosophy — many tools are already included

**2. Key Features**

**MVC vs. MTV Architecture**

| **MVC** | **MTV (Django's equivalent)** |
| --- | --- |
| Model | Model |
| View | Template |
| Controller | View |

**Model** – Handles data (database)

**Template** – Presentation layer (HTML)

**View** – Business logic (takes request, returns response)



**Typical Django Project Folder Structure**

myproject/

│

├── manage.py

├── requirements.txt # Optional: for dependencies

├── .gitignore # Optional: for Git version control

│

├── myproject/ # Main project package

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ ├── asgi.py

│ └── wsgi.py

│

├── app1/ # A Django app

│ ├── migrations/

│ │ └── \_\_init\_\_.py

│ ├── \_\_init\_\_.py

│ ├── admin.py

│ ├── apps.py

│ ├── models.py

│ ├── tests.py

│ ├── views.py

│ ├── urls.py # Optional: if app has routes

│ └── forms.py # Optional: for Django forms

│

├── app2/ # Another Django app

│ └── ...

│

├── templates/ # Global templates (optional)

│ └── base.html

│

├── static/ # Global static files (CSS, JS, images)

│ └── css/

│ └── style.css

│

└── media/ # User-uploaded files (optional)

**Key Components**

**manage.py**

* Command-line utility for managing the Django project (running server, migrations, etc.)

**myproject/ (Inner folder)**

* **settings.py**: Main configuration file (database, installed apps, middleware, etc.)
* **urls.py**: URL routing for the entire project
* **wsgi.py** & **asgi.py**: Entry points for WSGI/ASGI servers (used in deployment)

**app1/, app2/ (Apps)**

* Self-contained modules with:
  + **models.py**: Database models
  + **views.py**: Logic to handle requests/responses
  + **urls.py**: (optional) App-specific routes
  + **admin.py**: Admin panel registration
  + **forms.py**: (optional) Custom forms
  + **tests.py**: Tests for the app
  + **migrations/**: Auto-generated files for database schema changes

**templates/**

* Stores HTML templates
* Can be app-specific or global (configure TEMPLATES in settings.py)

**static/**

* CSS, JavaScript, images, etc.
* Can be global or per app (e.g., app1/static/app1/)

**media/**

* Used to store uploaded files (like user profile images)

**Optional But Common Additions**

* .env: For environment variables (using python-decouple or django-environ)
* README.md: Documentation
* docker/ or Dockerfile: For containerization

**Built-in Admin Panel**

Django has a powerful auto-generated admin interface to manage your models (e.g., users, posts).

**ORM (Object-Relational Mapping)**

You use Python classes to define and interact with your database — no raw SQL needed.

**3. Setting Up the Development Environment**

**Prerequisites:**

* Python 3.8+
* pip (Python package manager)

**Step-by-Step:**

# 1. Create project folder

mkdir­ my\_django\_project

cd my\_django\_project

# 2. Set up a virtual environment

python -m venv venv

venv\Scripts\activate

# 3. Install Django

pip install django

# 4. Check version

django-admin --version

**4. Creating a Django Project and App**

**📁 Project vs. App:**

* **Project** = overall website (can have multiple apps)
* **App** = a module (like blog, shop, accounts)

**Create Project:**

django-admin startproject mysite

This creates:

mysite/

├── manage.py

├── mysite/

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ ├── asgi.py

│ └── wsgi.py

**Create an App:**

Cd mysite

python manage.py startapp main

This creates:

main/

├── admin.py

├── apps.py

├── models.py

├── tests.py

├── views.py

├── migrations/

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

# mysite/settings.py

INSTALLED\_APPS = [

...

'main',

]

**5. Running the Development Server**

python manage.py runserver

You’ll see:

Starting development server at http://127.0.0.1:8000/

Visit that URL in your browser — Django’s welcome page confirms it's working!

**Checklist:**

✔ Installed Django  
✔ Created virtual environment  
✔ Started a project and app  
✔ Run the development server

Microsoft Windows [Version 10.0.19045.6216]

(c) Microsoft Corporation. All rights reserved.

C:\Users\piyush>mkdir my\_django\_project

C:\Users\piyush>cd my\_django\_project

C:\Users\piyush\my\_django\_project>python -m venv venv

C:\Users\piyush\my\_django\_project>venv\Scripts\activate

(venv) C:\Users\piyush\my\_django\_project>pip install django

Collecting django

Using cached django-5.2.5-py3-none-any.whl.metadata (4.1 kB)

Collecting asgiref>=3.8.1 (from django)

Using cached asgiref-3.9.1-py3-none-any.whl.metadata (9.3 kB)

Collecting sqlparse>=0.3.1 (from django)

Using cached sqlparse-0.5.3-py3-none-any.whl.metadata (3.9 kB)

Collecting tzdata (from django)

Using cached tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)

Using cached django-5.2.5-py3-none-any.whl (8.3 MB)

Using cached asgiref-3.9.1-py3-none-any.whl (23 kB)

Using cached sqlparse-0.5.3-py3-none-any.whl (44 kB)

Using cached tzdata-2025.2-py2.py3-none-any.whl (347 kB)

Installing collected packages: tzdata, sqlparse, asgiref, django

Successfully installed asgiref-3.9.1 django-5.2.5 sqlparse-0.5.3 tzdata-2025.2

[notice] A new release of pip is available: 25.1.1 -> 25.2

[notice] To update, run: python.exe -m pip install --upgrade pip

(venv) C:\Users\piyush\my\_django\_project>django-admin --version

5.2.5

(venv) C:\Users\piyush\my\_django\_project>django-admin startproject mysite

(venv) C:\Users\piyush\my\_django\_project>cd mysite

(venv) C:\Users\piyush\my\_django\_project\mysite>python manage.py startapp main

(venv) C:\Users\piyush\my\_django\_project\mysite>python manage.py runserver

Watching for file changes with StatReloader

Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

August 24, 2025 - 10:23:05

Django version 5.2.5, using settings 'mysite.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.

For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/

[24/Aug/2025 10:23:33] "GET / HTTP/1.1" 200 12068

**Gihub**

Microsoft Windows [Version 10.0.19045.6216]

(c) Microsoft Corporation. All rights reserved.

C:\Users\piyush>cd my\_django\_project

C:\Users\piyush\my\_django\_project>git init

Initialized empty Git repository in C:/Users/piyush/my\_django\_project/.git/

C:\Users\piyush\my\_django\_project>git add .

C:\Users\piyush\my\_django\_project>git commit -m "Initial commit"

[master (root-commit) 3d454cf] Initial commit

23 files changed, 217 insertions(+)

create mode 100644 mysite/db.sqlite3

create mode 100644 mysite/main/\_\_init\_\_.py

create mode 100644 mysite/main/\_\_pycache\_\_/\_\_init\_\_.cpython-313.pyc

create mode 100644 mysite/main/\_\_pycache\_\_/admin.cpython-313.pyc

create mode 100644 mysite/main/\_\_pycache\_\_/apps.cpython-313.pyc

create mode 100644 mysite/main/\_\_pycache\_\_/models.cpython-313.pyc

create mode 100644 mysite/main/admin.py

create mode 100644 mysite/main/apps.py

create mode 100644 mysite/main/migrations/\_\_init\_\_.py

create mode 100644 mysite/main/migrations/\_\_pycache\_\_/\_\_init\_\_.cpython-313.pyc

create mode 100644 mysite/main/models.py

create mode 100644 mysite/main/tests.py

create mode 100644 mysite/main/views.py

create mode 100644 mysite/manage.py

create mode 100644 mysite/mysite/\_\_init\_\_.py

create mode 100644 mysite/mysite/\_\_pycache\_\_/\_\_init\_\_.cpython-313.pyc

create mode 100644 mysite/mysite/\_\_pycache\_\_/settings.cpython-313.pyc

create mode 100644 mysite/mysite/\_\_pycache\_\_/urls.cpython-313.pyc

create mode 100644 mysite/mysite/\_\_pycache\_\_/wsgi.cpython-313.pyc

create mode 100644 mysite/mysite/asgi.py

create mode 100644 mysite/mysite/settings.py

create mode 100644 mysite/mysite/urls.py

create mode 100644 mysite/mysite/wsgi.py

C:\Users\piyush\my\_django\_project>git remote add origin https://github.com/ModernAIisthefuture/django.git

C:\Users\piyush\my\_django\_project>git branch -M main

C:\Users\piyush\my\_django\_project>git push -u origin main

Enumerating objects: 29, done.

Counting objects: 100% (29/29), done.

Delta compression using up to 8 threads

Compressing objects: 100% (27/27), done.

Writing objects: 100% (29/29), 6.98 KiB | 649.00 KiB/s, done.

Total 29 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (4/4), done.

To https://github.com/ModernAIisthefuture/django.git

\* [new branch] main -> main

branch 'main' set up to track 'origin/main'.

C:\Users\piyush\my\_django\_project>

## Module 2: Django Views and URL Routing

### 1. ****Understanding Views in Django****

A **view** is a Python function (or class) that takes a web request and returns a web response.

**Basic example:**

# main/views.py

from django.http import HttpResponse

def home(request):

return HttpResponse("Hello, Django!")

### 2. ****Creating Function-Based Views (FBVs)****

Function-Based Views are the most common starting point in Django.

# main/views.py

from django.shortcuts import render

def home(request):

return render(request, 'home.html') # renders an HTML template

### 3. ****Mapping URLs to Views using**** urls.py

#### Step-by-step:

##### 1. Create urls.py in your app (main/):

# main/urls.py

from django.urls import path

from . import views

urlpatterns = [

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

]

##### 2. Include app URLs in the project-level urls.py (mysite/urls.py):

# mysite/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path('', include('main.urls')), # root URL goes to the main app

]

### 4. ****Rendering Dynamic Content in HTML Templates****

#### Step-by-step:

##### 1. Create a folder called templates in your main/ app:

main/

└── templates/

└── home.html

##### 2. Add a simple HTML file:

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

<!DOCTYPE html>

<html>

<head>

<title>Django Home</title>

</head>

<body>

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

</body>

</html>

##### 3. Update your view to send dynamic content:

# main/views.py

def home(request):

context = {'name': 'Django Developer'}

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

Now visiting http://127.0.0.1:8000/ should show:

Welcome, Django Developer!

### 5. ****Static Files (CSS, JS, Images) in Django****

#### Step-by-step:

##### 1. Create a folder called static inside your app:

main/

└── static/

└── css/

└── style.css

##### 2. Add CSS (example):

/\* main/static/css/style.css \*/

body {

background-color: #f0f0f0;

font-family: Arial;

}

##### 3. Load static files in your template:

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

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Django Home</title>

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

</head>

<body>

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

</body>

</html>

##### 4. Make sure Django knows to find static files. In settings.py:

# mysite/settings.py

STATIC\_URL = '/static/'

Now Django will serve CSS files in development mode automatically!

## Checklist:

✔ Defined function-based views  
✔ Mapped URLs using urls.py  
✔ Rendered dynamic content with templates  
✔ Linked static CSS files to your HTML

**Django App – "formapp"**

Task List :-

* Displays an HTML form on one page
* Collects the form input
* Sends it to another page and displays it

**1. Create Django Project and App**

C:\Users\piyush>mkdir django\_form

C:\Users\piyush>cd django\_form

C:\Users\piyush\django\_form>python -m venv venv

C:\Users\piyush\django\_form>venv\Scripts\activate

(venv) C:\Users\piyush\django\_form>pip install django

(venv) C:\Users\piyush\django\_form>django-admin startproject myproject

(venv) C:\Users\piyush\django\_form>cd myproject

(venv) C:\Users\piyush\django\_form\myproject>python manage.py startapp formapp

**2. Register the App**

Edit myproject/settings.py:

INSTALLED\_APPS = [

...

'formapp', # Add this line

]

**3. Create URLs**

**myproject/urls.py (Project-level)**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path('', include('formapp.urls')), # Point to app URLs

]

**formapp/urls.py (App-level)**

Create this file inside the formapp/ folder:

# formapp/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('', views.form\_page, name='form\_page'),

path('submit/', views.submit\_form, name='submit\_form'),

]

**4. Create Views**

**formapp/views.py**

from django.shortcuts import render, redirect

def form\_page(request):

return render(request, 'formapp/form.html')

def submit\_form(request):

if request.method == 'POST':

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

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

return render(request, 'formapp/result.html', {

'name': name,

'email': email

})

return redirect('form\_page')

**5. Create Templates**

Create this folder structure:

formapp/

└── templates/

└── formapp/

├── form.html

└── result.html

**formapp/templates/formapp/form.html**

<!DOCTYPE html>

<html>

<head>

<title>Form Page</title>

</head>

<body>

<h2>Submit Your Info</h2>

<form method="POST" action="{% url 'submit\_form' %}">

{% csrf\_token %}

<label>Name:</label>

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

<label>Email:</label>

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

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

</form>

</body>

</html>

**formapp/templates/formapp/result.html**

<!DOCTYPE html>

<html>

<head>

<title>Result Page</title>

</head>

<body>

<h2>Form Submitted Successfully!</h2>

<p><strong>Name:</strong> {{ name }}</p>

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

<a href="{% url 'form\_page' %}">Back to Form</a>

</body>

</html>

**6. Run the Server**

python manage.py runserver

Now visit:

* http://127.0.0.1:8000/ → The form page
* After submission → You’ll see the result page with the entered values

| **File / Folder** | **Purpose** |
| --- | --- |
| formapp/views.py | Handle form and result display |
| formapp/urls.py | URL routing for the app |
| formapp/templates/ | HTML templates for the form & result |
| myproject/urls.py | Includes app URLs |
| settings.py | Registers the app |