Django

What is Django?

Django is a **powerful web framework** built with Python that helps developers create web applications quickly and efficiently.
It follows the **MVT (Model-View-Template)** architecture and comes with many built-in features like authentication, admin panel, and ORM support.

Why We Use Django:

- Fast and clean development process
- Built-in security features
- Scalable and maintainable
- "Don't Repeat Yourself" (DRY) principle
- Great for both beginners and professionals

1. Create a virtual environment

Isolates project dependencies so different projects don't conflict. Recommended for every Python project**.**

Using the built-in `venv` :

python -m venv venv

This creates a directory called venv containing a private Python interpreter and pip.

2. Activate the virtual environment

Makes the python and pip commands point to the environment's interpreter and package space**.**

.\venv\Scripts\Activate.ps1

If PowerShell blocks execution, run (as Admin) **Set-ExecutionPolicy -Scope CurrentUser ExecutionPolicy RemoteSigned**.

3. Install Django

Get Django package into your venv.
Install the latest stable release:

pip install Django

To install a specific version:

pip install "django==4.2.5"

4. Check Django version

Verify the version:

python -m django --version

5. Create requirements.txt

Lists exact packages to reproduce the environment (useful for sharing, deployment).

Create a new file named `requirements.txt` and add this content **pip**

freeze > requirements.txt

6. Start a Django project — create project structure startproject

bootstraps the directory and minimal settings.

django-admin startproject project_name

This creates:

project/

manage.py

project/

__init__.py

settings.py

urls.py

asgi.py

wsgi.py

7. Move into the project folder

CMD

cd project_name

8. Creating an App

Each app in Django manages a specific function.

python manage.py startapp app_name

9. Running Migrations

These ensure the database is properly set up before using Django's models.

python manage.py makemigrations python

manage.py migrate

10. Run the development server

Quickly serve the site for development

python manage.py runserver

This runs at http://127.0.0.1:8000/.

GitIgnore:

- Contains the information of files which should be ignore while pushing the code to Github.

- Itis a **special file used in Git** that tells Git **which files or folders to ignore** — meaning Git will **not track changes** to those files.
- It helps keep the project **clean, secure, and efficient** by ignoring unnecessary or sensitive files.

requirements.txt:

- requirements.txt is a simple text file that lists all the Python packages and dependencies the project needs to run.
- It helps others install the exact same versions of libraries — ensuring the project runs the same way on any system **site-packages:**

- site-packages is the folder where Python stores all third-party libraries and modules.

Query Parameters:

- Used to pass extra information through URL while making http request.
- Query Parameters are key-value pairs that are added to the end of a URL to send small pieces of data to a web server**.**
- They come **after a question mark (`?`)** in the URL and are separated by **ampersands (`&`)**.
- In Django, we can access them using `request.GET.get('key').`

Syntax:

URL/?key_1=value_1&key_2=value_2&……key_n=value_n……

Example:

```
from Django.http import JsonResponse

    def add(request):

        name=request.GET.get(“name”)        city=request.GET.get(“city”) return

    JsonResponse({“status”:”success”,”data”:{“name”:name,”city”:city},status=200)
```

DEPLOYMENT PROCESS ON RENDER (STEP-BY-STEP)

Deployment is the process of publishing our project so others can use it online.

Step 1: Create Project Repository

1. Go to GitHub
2. Click **New Repository**.
3. Give a name (example: my-django-app)
4. Click **Create Repository**

Step 2: .gitignore

Inside .gitignore, add:

Venv

venv → to ignore virtual environment (big + not needed in git).

Step 3: Commit and Push Code to GitHub

- git init • git add .
- git commit -m "Initial commit"
- git branch -M main
- git remote add origin <repository-link>
- git push -u origin main

Step 4: Login to Render

1. Go to render.com in browser.
2. Sign up or log in .
3. We can see the Dashboard.

Step 5: Create a New Web Service

1. On the dashboard, click "New" → "Web Service".
2. Choose "Build and deploy from a Git repository".
3. Click "Public Git Repository".
4. Paste your GitHub repository link.
5. Click Continue or Connect.

Step 6: Configure Service Settings

Here is the set how Render will run app.

1. **Name & Region**
    - **Name**: Any name (example: `my-django-app`).
    - **Region**: Choose **Southeast Asia** (closer and faster for you).
2. **Root Directory/Source Path**

Step 7: Build Command

Render needs to install thedependencies.

- In the Build Command field, write:

pip install -r requirements.txt

Make sure the project has a requirements.txt file.

Example to create it locally: **pip**

freeze > requirements.txt

Step 8: Start Command (Important!)

Render must know how to run the app.

python manage.py runserver 0.0.0.0:8000

Step 9: Choose Plan

- Select Free plan (for learning/testing).

Step 10: In settings.py

Set:

ALLOWED_HOSTS = ["your-render-url", "localhost"]

Step 11: Deploy the Service

1. Click Deploy 2.
Render will:

- Clone your repo
- Install packages (`pip install -r requirements.txt`) o Run your start command

3.If everything is correct, it will show **Live**/ **Healthy**.

Render will give you a **public URL**, like:

`https://my-django-app.onrender.com`

Open that link → app is live

DATABASE CONNECTION PROCESS (Django + MySQL + PyMySQL)

- We connect a database **to store, manage, and retrieve data** for our application.
- Without a database, data will not be saved permanently.

Main Reasons to Connect a Database

1. To Store Data Permanently

   - When users register → save details
   - When users login → check stored data
   - When we add products, posts, files → store them

Without a DB, data disappears when you stop the server.

2. To Retrieve Data Anytime

   - Display user profile
   - Show product list
   - Show messages, comments, posts

A database allows us to **fetch data quickly**.

3. To Update Data

   - Edit profile
   - Change password

- Update product price

Databases help modify data safely.

4. To Delete Data

    - Remove users
    - Delete old records
    - Clear logs

All CRUD operations need a database. 5.

For Large Applications

Apps like:

- Instagram
- Facebook
- Amazon
- Banking systems

They all require strong database systems to store millions of records.

6. Security and Backup **Databases**

provide:

- Data protection
- User access control
- Regular backups

So the data stays safe.

We create the database in MySQL shell so that Django can connect to it and store tables, models, and application data.

STEP 1: Install PyMySQL (MySQL Connector)

This installs the library that Django uses to connect to MySQL

- Outside Virtual Environment: **pip install pymysql**
- Inside Virtual Environment:

  python -m pip install pymysql

Without this, Django cannot communicate with MySQL.

STEP 2: Configure PyMySQL in __init__.py

Inside your project folder (where settings.py exists), open: __init__.py

Add this**:**

import pymysql pymysql.install_as_MySQLdb()

- This tells Django to use PyMySQL instead of MySQLdb
- Avoids MySQLdb installation errors.

STEP 3: Configure DATABASES in settings.py

Delete any existing DATABASES configuration and add this:

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'database_name',
      'USER':'root',
      'PASSWORD':'password',
      'HOST':'localhost',
      'PORT':'3306',
      "OPTIONS":{
         'charset':'utf8mb4',
         'init_command':"SET sql_mode='STRICT_TRANS_TABLES'"
      }
   }
}
```

WHERE:

- ENGINE → tells Django to use MySQL
- NAME → your MySQL database name
- USER → MySQL username
- PASSWORD → MySQL password
- HOST → use localhost for local db
- PORT → default MySQL port (3306)
- OPTIONS → improves performance & avoids unicode errors

STEP 4: Check MySQL Version From Django

Run this command in the terminal:

python manage.py shell -c "from django.db import connection; c=connection.cursor(); c.execute('SELECT DATABASE(), VERSION()'); print(c.fetchone())"

This checks:

- Whether Django is connected to MySQL
- MySQL server version
- Database name currently being used

STEP 5: Create Health Check API (Optional but Useful)

In **views.py**:

```
from django.http import JsonResponse from

django.db import connection


def health(request):

  try:      with

connection.cursor() as c:
```

```
        c.execute("SELECT 1")   # Executes test query
return JsonResponse({"status":"ok","db":"connected"})
except Exception as e:
    return JsonResponse({"status":"error","db":str(e)})
```

Why this?

✓ Helps check if database is connected
✓ Useful for debugging and deployment
✓ Shows clear error message if DB is not reachable

STEP 6: Add URL for Health Check

In urls.py:

```
from django.urls import path from
.views import health urlpatterns =
[   path('health/', health),
]
```

✓ Now open in browser: http://localhost:8000/health/
You will see database status.

STEP 7: Run Django Commands

Run database migration commands: **python**

manage.py makemigrations python

manage.py migrate

✓ This will create required tables inside your MySQL database.

Finally, run the server:

python manage.py runserver

DONE! You have successfully connected Django with MySQL

What is MVT in Django?

MVT = Model – View – Template

- Django follows the **MVT – Model View Template** architecture.
- It is similar to MVC, but Django itself handles the controller automatically.

Model

- The Model is the data layer of the application.
- It defines the structure of the database (tables, columns, constraints).
- Each model is a Python class and each class variable represents a field in the database.
- Django automatically converts the model into SQL queries using the ORM.
- Models handle:
    - Database creation ○ Data validation
    - Data manipulation (insert, update, delete) ○ Relationships (One-to-One, One-to-Many, Many-to-Many)

Examples name =

models.CharField(max_length=100) age =

models.IntegerField() email =

models.EmailField(unique=True) date =

models.DateField()

View

- The View is the business logic layer.
- It receives HTTP requests and returns HTTP responses.
- Views interact with the models (database) and pass data to templates.

- Views decide what the user should see**.**

Example:

```
def home(request):
    data = Student.objects.all()     return render(request,
'home.html', {'students': data})
```

Template

- Templates handle the presentation layer (UI).
- They are usually HTML files with Django Template Language (DTL).
- They show dynamic data passed from views.

Example:

```
{% for s in students %}
    <p>{{ s.name }}</p>
{% endfor %}
```

MODELS in Django

- Models are used to create database schemas (tables).
- We define models inside models.py of a Django app.

✓ Steps to create Models:

1. Create/open **models.py**
2. Write a Python class for each table
3. Add fields using Django model fields
4. Run migrations to create tables in DB

Common Model Fields

Data Type	Field

Short Text	CharField
Long Text	TextField
Number	IntegerField, FloatField
Email	EmailField
Date	DateField, DateTimeField
Boolean	BooleanField
File	FileField, ImageField

Example Model

```
from django.db import models


class Student(models.Model):     name =
models.CharField(max_length=100)     age =
models.IntegerField()
   email = models.EmailField(unique=True)
join_date = models.DateField()
```

Migrations

After writing models, run:

python manage.py makemigrations python manage.py migrate

✓ Migrations generate SQL queries automatically
✓ They create tables in the database

What is ORM?

- ORM → Object Relational Mapping
- Django ORM allows us to interact with the database using Python code instead of SQL.
- Helps in CRUD operations

✓ Why ORM is Important?

- No need to write SQL queries manually
- More secure (prevents SQL injection)
- Cross-database support (e.g., MySQL, PostgreSQL, SQLite)
- Developer-friendly
- Automatically updates queries based on model changes

How ORM Works?

Student.objects.all() **ORM**

converts it into:

SELECT * FROM student;

ORM Methods in Django

ORM provides various methods to perform CRUD operations.

- **Create / Insert**

Create(): Creates a new object and saves it to the database in a single step.

Student.objects.create(name="Sharanya", age=21)

Django will insert a new row into the *student* table with name and age.

Another way:

First make an object in memory, then call **save()** to store it in the database.

s = Student(name="Sharanya", age=21) s.save()

Useful when we want to set values step-by-step before saving.

- **Read / Retrieve**

The process of fetching or getting data from the database.

1. Get all records:

all(): returns a queryset containing all rows in the Student table.

```
Student.objects.all()
```

2. Filter records:

filter(): retrieves multiple records that satisfy the given condition.

```
Student.objects.filter(age=20)
```

Returns all students whose age = 20.

3. Get single record:

get(): returns exactly one record that matches the condition.

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

Note:

If no record found → error

If more than one record → error

4. Get records as dictionary: values(): returns data as dictionaries instead of model objects.

```
Student.objects.values()
```

Example:

{'id': 1, 'name': 'Sharanya', 'age': 21}

5. Count: count(): returns the number of records in the table.

```
Student.objects.count()
```

• Update

update(): modifies existing records in the database without fetching them.

```
Student.objects.filter(id=1).update(age=25)
```

Updates age to 25 for the student whose id = 1.

- **Delete delete()**: removes the matching row from the database.

```
Student.objects.get(id=1).delete()
```

- **Sorting**

  order_by(): sorts results based on the field given.

  ✓ **Ascending**

```
Student.objects.order_by('name')
```

  ✓ **Descending**

```
Student.objects.order_by('-age')
```

  Using - in front sorts in descending order.

- **Check existence**

  exists(): returns **True** if at least one matching object is found, otherwise **False**.

```
Student.objects.filter(email="abc@gmail.com").exists()
```

  Useful for validation like checking duplicate emails.

HTTP Methods

- HTTP (HyperText Transfer Protocol) defines how a client (browser/app) communicates with a server.
- These methods specify *what action* should be performed on the server resources.

GET Method

- Used to retrieve or fetch data from the server.
- Commonly used for loading webpages, fetching lists, or displaying user profiles.

POST Method

- Used to send data to the server.
- Creates new records**.**

PUT Method

- To update/replace the entire existing resource.

PATCH Method

- Used to update partial data of an existing record.

DELETE Method

- Used to delete a resource.

HEAD Method

- Same as GET, but returns only headers — no response body.

OPTIONS Method

- Used to check which HTTP methods are allowed on a server or URL.

✓ **Usage**

 Very important in CORS (Cross-Origin Resource Sharing).

✓ **Example**

OPTIONS /users

Server response:

Allow: GET, POST, PUT, DELETE, PATCH

MIDDLEWARE

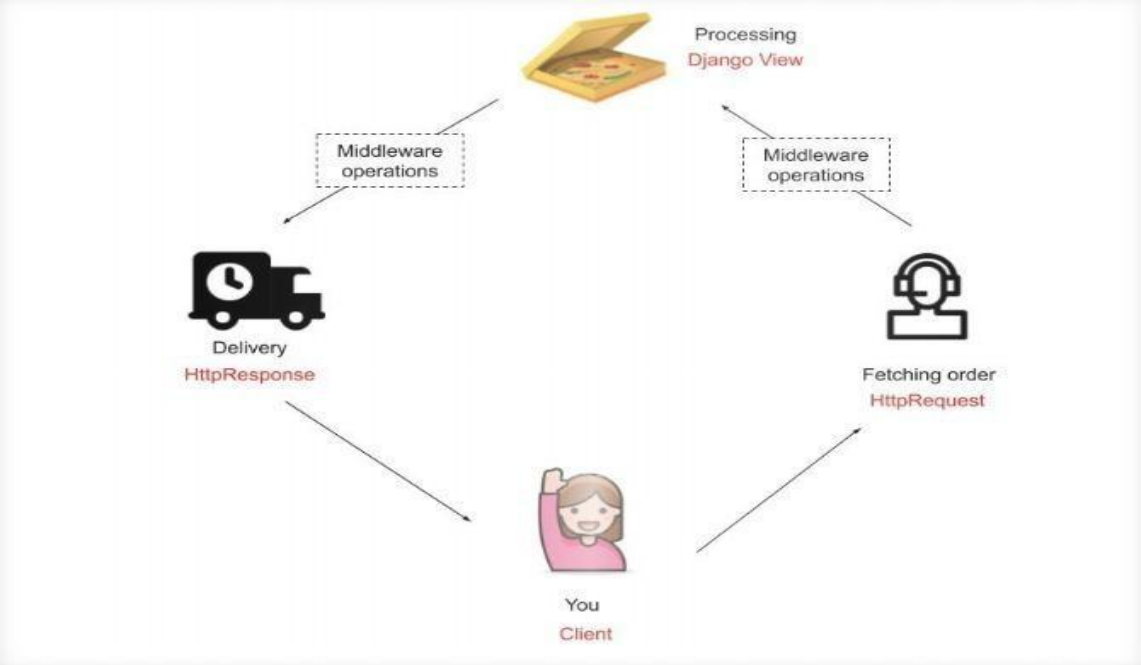
- Middleware in Django is a layer between the request and response.
- It processes the **HTTP request** before it reaches the view and processes the **HTTP response** before it goes back to the browser.
- Think of Middleware as a **filter or pipeline** that can modify requests/responses.

What is Middleware?

Middleware is a layer of code that processes the request before it reaches the view and processes the response before it reaches the browser.

In simple words:

- Middleware sits between the user request and Django's view.
- It can modify the request, response, headers, authentication, security, etc.



Why Do We Use Middleware?

Middleware is used for:

- Authentication checking
- Logging user activity
- Blocking requests
- Validating data (username, email, password)
- Security checks
- Session and cookies handling
- Request/response modification **How to Create a Middleware?**

STEP-1: Create a file called middleware.py

Inside Django app, create:

middleware.py STEP-2:

Write Middleware Class

A middleware class must contain:

- __init__(self, get_response) → runs one time when server starts
- __call__(self, request) → runs for every request

Example Middleware

```
# middleware.py

        from django.http import JsonResponse import

        re


        class UsernameMiddleware:    def

        __init__(self, get_response):

            self.get_response = get_response   # one-time configuration    def

        __call__(self, request):        # This runs on every request

        username = request.GET.get("username")        if username and not

        re.match(r'^[a-zA-Z0-9._]{3,20}$', username):

                return JsonResponse({"error": "Invalid username"})

        response = self.get_response(request)

            return response
```

STEP-3:

Add Middleware to settings.py Open

settings.py → find:

```
        MIDDLEWARE = [ ……..  ]
```

Add middleware class path:

```
        MIDDLEWARE = [

          'django.middleware.security.SecurityMiddleware',

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

          # your custom middleware

          'myapp.middleware.UsernameMiddleware',

        ]
```

✓ Django will now run the middleware for every request.

How Middleware Works (Flow)

User Request

↓

Middleware 1 → Middleware 2 → Middleware 3 → View

↓ ↓

Middleware 3 ← Middleware 2 ← Middleware 1

↑

Response to Browser

Each middleware can:

- Stop the request
- Modify the request
- Modify the response
- Allow request to pass to next middleware

Example 1: Block a request if user not logged in

```
class BlockUserMiddleware:     def
__init__(self, get_response):
self.get_response = get_response
def __call__(self, request):
      if not request.user.is_authenticated:
         return JsonResponse({"error": "Login required"})
      return self.get_response(request)
```

Example 2: Check for valid email

```
class EmailMiddleware:     def
__init__(self, get_response):
self.get_response = get_response
def __call__(self, request):
```

```
        email = request.GET.get("email")
    if email and "@" not in email:
            return JsonResponse({"error": "Invalid Email"})
    return self.get_response(request)
```

Key Points on Django Middleware

- Middleware is a request/response processing layer in Django.
- Runs before the view function and after the view returns response.
- Created using __init__ and __call__ methods.
- Must be added in MIDDLEWARE list in settings.py.
- Used for authentication, validation, logging, and security.