**Steps to Start a Django Project and Commit to GitHub**

1. **Create a Virtual Environment**:
   * Navigate to your project directory:
   * cd E:/Coding and Other Learning/Brototype/Tasks/Week4
   * Create a virtual environment:
   * python -m venv projects # Or use your preferred name for the virtual environment
2. **Create a .gitignore File**:
   * Add the following content to .gitignore to ignore unnecessary files:
   * # Python files
   * \*.pyc
   * \_\_pycache\_\_/
   * # Environment variables
   * .env
   * # Django stuff
   * db.sqlite3
   * media/
   * staticfiles/
   * # IDE files
   * .vscode/
   * .idea/
3. **Activate the Virtual Environment**:
   * For **Windows**:
   * .\projects\Scripts\activate
   * For **macOS/Linux**:
   * source projects/bin/activate
4. **Install Django**:
5. pip install django
6. **Create a Django Project**:
7. django-admin startproject myproject # Replace "myproject" with your desired project name
8. **Initialize Git**:
   * Navigate to your project directory:
   * cd myproject
   * Initialize the Git repository:
   * git init
9. **Add Files to Git**:
10. git add .
11. **Commit Your Changes**:
12. git commit -m "Initial commit: Start Django project"
13. git remote add origin <https://github.com/Tijo-11/week4>
14. git push -u origin master

With these steps, your Django project is now initialized and committed to Git. You can now push it to GitHub as described earlier.

As you continue working on your project and make changes, here’s the process to upload (commit and push) your changes to GitHub:

**Step-by-Step Process:**

1. **Check the Status of Your Changes**: First, you can check which files have changed by running:
2. git status

This will show you the modified or new files that are not yet staged for commit.

1. **Stage Your Changes**: To add your changes to the staging area, use:
2. git add .

This stages all the modified and new files. If you want to stage specific files, you can replace . with the file names:

git add filename.py

1. **Commit Your Changes**: After staging, you need to commit the changes with a meaningful message:
2. git commit -m "Your commit message describing the change"

Example:

git commit -m "Fix bug in user authentication"

1. **Push Your Changes to GitHub**: Finally, push the changes to your remote repository:
2. git push

Since you already set up the upstream with -u, you can just use git push in the future.

**Summary of Commands:**

1. git status — To see what’s changed.
2. git add . — To stage all changes (or git add filename for specific files).
3. git commit -m "message" — To commit your changes.
4. git push — To push your changes to GitHub.

The command django-admin startproject myproject can work without explicitly installing Django **because django-admin is often bundled with Python when you install Django**, and it can be accessed directly. However, if Django is not properly installed or the virtual environment is not activated, you will likely encounter an error when running commands like python manage.py startapp home.

Here’s why:

1. **django-admin startproject works because**:
   * When you install Django via pip, the django-admin command is available system-wide or in your virtual environment (if activated).
   * Running django-admin startproject myproject uses the Django management tool to generate the project folder structure, and it's likely you have Django installed globally or within your virtual environment.
2. **python manage.py startapp home fails because**:
   * The python manage.py command depends on the django package being installed in the active Python environment (your virtual environment in this case).
   * If you haven't activated the virtual environment or haven't installed Django properly in the environment, it will raise an error like:
   * ModuleNotFoundError: No module named 'django'

**Steps to Resolve:**

* **Ensure Django is installed in your virtual environment**:
  + Run pip show django to check if Django is installed.
  + If it’s not installed, run pip install django to install it.
* **Activate the Virtual Environment**:
  + If you're not in your virtual environment, activate it:
    - For Windows:
    - .\projects\Scripts\activate
    - For macOS/Linux:
    - source projects/bin/activate

Once you have activated the virtual environment and installed Django, you should be able to run python manage.py startapp home without any issues.

* init.py file is to read other files in the folder
* Models.py is used to access database
* Tests.py to test functionalities
* Whenever a create app add it to settings.py(of app level project directory(here project1)) as external\_apps and redefine installed\_apps=installed\_apps+external\_apps
* All the logical part is written in views.py
* Urls.py in app level is used for routing
  + From home.views import \*
  + Path(‘’, home, name=home)##blank suggests calling default port, that is 8000
  + Importance of name will be explained later
  + We have to register each function defined in the views.py file in urls.py routes
* Function based views:
  + Def home(request):
  + Return(“html file”)
  + from django.http import HttpResponse

**Class Notes: Triple Quotes in Python**

**What are Triple Quotes?**

Triple quotes in Python are denoted by three consecutive single quotes (''') or double quotes ("""). They are primarily used for **multi-line strings** and **docstrings**.

**Uses of Triple Quotes**

1. **Creating Multi-line Strings**
   * Triple quotes allow you to define strings that span multiple lines without needing to escape newline characters (\n).
   * Example:
   * multi\_line\_string = """This is a string
   * that spans multiple lines.
   * It's easy to include single or double quotes inside!"""
2. **Defining Docstrings**
   * Triple quotes are widely used for **docstrings**—a special kind of comment describing modules, classes, or functions.
   * Docstrings are **actual strings** (not comments) and can be accessed using the \_\_doc\_\_ attribute.
   * Example:
   * def my\_function():
   * """This is a docstring.
   * It explains what the function does."""
   * pass
   * print(my\_function.\_\_doc\_\_) # Outputs the docstring
3. **Embedding Large Text Blocks**
   * Triple quotes are useful for embedding large text blocks, such as HTML, JSON, or SQL, directly in your code.
   * Example:
   * html\_content = """<h1>Welcome</h1>
   * <p>This is a multi-line HTML content.</p>"""
4. **Used as Block Comments (Unofficial Use)**
   * Triple quotes are often used as block comments, but **this is not their intended purpose**.
   * Example:
   * '''
   * This is a block comment.
   * It's just an unused string and has no effect on the program.
   * '''
   * For proper comments, use #:
   * # This is a proper comment

**Common Misconception**

* Many people think triple quotes are solely for commenting. While they **can** act as comments if unused, their primary purpose is **creating multi-line strings** or **docstrings**.

**Why Use Triple Quotes in Django's HttpResponse?**

When generating HTML responses, triple quotes are helpful for embedding multi-line HTML content:

* They allow you to include HTML without escaping internal quotes (") or handling line breaks.
* Example:
* def home(request):
* return HttpResponse("""<h1>Hello, I am a Django server!</h1>
* <p>This is an HTML response.</p>
* <hr>
* <h3>Hope you are doing well!</h3>""")

**Summary**

* **Triple quotes are for multi-line strings and docstrings.**
* **They are not true comments.** Use # for comments instead.
* They simplify embedding large text blocks, making them highly readable and convenient.

Let me know if you need further additions or edits! 😊

* Using function for each page is not feasible. We need to return html page from backend using Django.
* Hence we create a folder called templates for each app(eg. home, accounts, login etc)
  + Now we don’t use Httpresponse, but we will return render
  + Def home(request)
    - Return render(request, “index.html”)
  + def home(request):
  + return render(request , "home/index.html")
    - #You can add html files in other folders, but it needs to be in templates folder. Here it is in template>>home>>index.html

**DJANGO COMPLETE TEMPLATE ENGINE**

* The purpose of template engine is to produce dynamic data in the page
* The data from Django server reach html page through context
  + def home(request):
  + people=[{'name':'Abhijeet','age':26},
  + {'name':'Tijo','age':29},
  + {'name':'Vicky','age':24},
  + {'name':'Jithin','age':20},
  + {'name':'Sandeep','age':17},
  + {'name':'Abhishek','age':13},
  + ]
  + return render(request , "home/index.html", context={'people':people})
  + In html add {{people}} ##directly adds dictionary
  + ##Now this data will be returned on the page
  + In the table you can use pythons for loop using
    - {% expression%}
    - {%endfor%}

EG.

* <table class="container">
* <tr>
* <th>S.No.</th>
* <th>Name</th>
* <th>Age</th>
* <th>Can Vote</th>
* </tr>
* {%for person in people%}
* <tr>
* <td>{{forloop.counter}}</td>
* <td>{{person.name}}</td>
* <td>{{person.age}}</td>
* <td> {% if person.age >= 18 %}👍
* {% else %}👎
* {% endif %}
* </td>
* </tr>
* {% endfor %}
* </table>
* Windows+period for emoji’s
* Ctrl + / to comment out in VS code(after selecting the lines to comment out) for html, python, css
* Built in tag templates and filters for Django
* <a href="/">Home</a>
* <a href="/about/">About </a></h2>
* Accordingly define urls.py and views.py
* DRY-
  + Base.html
  + <body>
  + {% block start %}
  + {% endblock %}
  + </body>
  + </html>
* Things to add in other html files
  + {% extends "home\base.html" %}
  + {% block start %}
  + Content
  + Content
  + {% endblock %}

When using template inheritance in Django, if you want to dynamically change the title of the HTML page, you can use a dedicated block for the title in the base template. Here's how you can do it:

**1. Update the Base Template (home/base.html)**

Add a {% block title %} in the <title> tag of your base template:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>{% block title %}Default Title{% endblock %}</title>

<link rel="stylesheet" href="path/to/your/css">

</head>

<body>

{% block start %}

{% endblock %}

</body>

</html>

**2. Update the Child Template**

In your child template, override the title block and specify your custom title:

{% extends "home/base.html" %}

{% block title %}

Project 1 About

{% endblock %}

{% block start %}

<nav class="navbar navbar-expand-lg navbar-light bg-light">

<h2>

<a href="/">Home</a>

<a href="/contact/">Contact </a>

</h2>

</nav>

<h1>This is About page of Project1</h1>

{% endblock %}

**Explanation**

1. **Base Template**:
   * The {% block title %} in the <title> tag allows child templates to replace its content dynamically.
2. **Child Template**:
   * You override the title block in the child template to set a specific title for that page.
   * The start block is used for other content on the page.

This approach ensures that your title is dynamic while keeping your base template clean and reusable.

* Content between block start and blockend willnot be common and other content will be common

**MODELS AND MIGRATIONS**

* In each app there will be models.py file
* Schema – structure of database
* Search for Django models and migrations in google
* Django add id=models.Autofield() automatically to the classes created in models, it will be the primary key.
* As you run makemigration, 0001\_initial.py file is created
  + In that there is dependency which depends on app
  + As you make more migrations the new one will depend on old one. If you delete old one the new one will not work, the database will collapse.
  + Eg 0002 will have dependency as 0001.
* Error Lens extension in VS code
* Django compares the last file with second last migration applied file and then update details. Django generates states of models once you run migrate. Then compare the states in database to check whether to make any changes.
* If you delete migration either in Django or in database from the end it will not cause error. But if you delete any migration file which lies in the middle, database will collapse

**DJANGO SHELL**

* Python manage.py shell
* Exit()
* From home.models import \*
* Student1=Student(name="Abhijeet", age=26, email="abhi23@gmail.com, address="Luckknow")
* Student1.save()## no need to give name as student1, just student is fine
* Call student1 to display it
* Student=Student.objects.create()# if you use this you don’t have to use student.save
* Objects is a model manager
* Student(class\_name).objects.all()## gets all the objects
* Student.objects.all()[0].name
* You can’t create a file with function and run it terminal while you are working in Django. It will cause ImportError. It is because the function needs to use some resources from Django
* But you can import this function to Django shell and run it
* Suppose the function is in util.py then  
  from home.util import \*  
  function()
* To check id of a data  
  Student.objects.all()[1].id

**Advanced CRUD Operations in Django**

* Create, Read, Update, Delete
* If you find error in migrating( for example you made spelling mistake error, corrected it but forgot to migrate) then you can delete the previous migrations and corresponding database tables and then create models again.
* <https://www.scaler.com/topics/django/resetting-db-in-django/>
* Database locked error:-  
  <https://stackoverflow.com/questions/3172929/operationalerror-database-is-locked>
* Close DB browser to get rid of database locked error
* Operational error- No such tables  
  python manage.py migrate --run-syncdb   
  Use it to create tables for unmigrated apps
* It looks like there was an issue with my migration.
  + I ran ./manage.py schemamigration research --auto and found that many of the fields didn't have a default specified.
  + So, I ran ./manage.py schemamigration research --init followed by ./manage.py migrate research
  + Rerunning the server from there did the trick!
* python manage.py makemigrations appName
* django.db.utils.OperationalError: no such table
  + Most likely, the problem is that you didn’t do a makemigrations / migrate when deleting the model.If you look at your database, you’ll see that there’s a “django\_contenttype” table which tracks models and tables.Django still thinks that the table exists.
  + In your settings.py file change the name of your database. For example, You have to change name mydatabase below.
  + DATABASES = {
    - 'default': {
  + 'ENGINE': 'django.db.backends.sqlite3',
  + 'NAME': 'mydatabase',
  + }
  + }
  + I found another solution:  
    Delete *pycache* and db.sqlite3
* Car.create.objects.(\*\*car\_dict)# define dictionary with values and then pass it

Read

* Cars.objects.all()
* Cars.objects.all()[1].speed
* While defining class in model add  
  def \_\_str\_\_(self)->str:

Return self.name, self.car\_name

* For car in cars:  
  print(f””The car name is {car.car\_name} and speed is {car.speed}”)
* To copy and paste something in terminal use ctrl+shif+c and ctrl+shift+v
* car=Cars.objects.get(id=1)#if you pass number greater than the number of objects you will get error. Catch it as exception

Update

* After getting the object update value
* car=Cars.objects.get(id=1)  
  car.car\_name=”creta”  
  car.speed=200  
  car.save()
* Car.objects.filter(id=1).update(car\_name=”Creata limited Edition”K)

Delete

Car.objects.get(id=1).delete()

Now the previous second one will not occupy first position. The data corresponding to the id is only getting delted.

**PROJECT RECEIPE**

* ImageField(upload\_to=receipe)# not applied, receipe is not defined
* Go to bootstrap website search for form there and copy the form from there
* Add method=post in form in order to submit data from front end to backend, also add enctype=multipart/formdata for submitting images
* To bring the data to view  
  data=request.POST
* We need to give {% csrf\_token % } below form to
* Run python shell in cmd.
* Name should be added in input/textarea tag, otherwise data will not be sent

VIEWS

from django.shortcuts import render,redirect

from .models import \*

# Create your views here.

def receipes(request):

    if request.method=="POST":

        data=request.POST

        receipe\_name=data.get('receipe\_name')

        receipe\_description=data.get('receipe\_description')

        receipe\_image=request.FILES.get('receipe\_image')

        print(receipe\_name)

        print(receipe\_description)

        print(receipe\_image)

        Receipe.objects.create(receipe\_name=receipe\_name,

            receipe\_description=receipe\_description,

            receipe\_image=receipe\_image)

    return redirect('/receipes')#inside function

    return render(request, 'receipes.html')

* Return redirect is added so that each time you submit data it will not give rise to dialogue box

**ADMIN.PY FILE**

* App level in Vege app for this project
* from django.contrib import admin
* from .models import \*
* # Register your models here.
* admin.site.register(Receipe)

**RECEIPE PART -2**

from django.shortcuts import render,redirect

from .models import \*

# Create your views here.

def receipes(request):

    if request.method=="POST":

        data=request.POST

        receipe\_name=data.get('receipe\_name')

        receipe\_description=data.get('receipe\_description')

        receipe\_image=request.FILES.get('receipe\_image')

        print(receipe\_name)

        print(receipe\_description)

        print(receipe\_image)

        Receipe.objects.create(receipe\_name=receipe\_name,

            receipe\_description=receipe\_description,

            receipe\_image=receipe\_image)

        return redirect('/receipes')

    queryset=Receipe.objects.all()

    context={'receipes':queryset}

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

* To show receipes, we can update html file as follows

<table class="table">

        <thead>

          <tr>

            <th scope="col">#</th>

            <th scope="col">Receipe Name</th>

            <th scope="col">Receipe Description</th>

            <th scope="col">Image</th>

          </tr>

        </thead>

        <tbody>

            {% for receipe in receipes %}

          <tr>

            <th scope="row">{{forloop.counter}}</th>

            <td>{{receipe.receipe\_name}}</td>

            <td>{{receipe.receipe\_description}}</td>

            <td>{{receipe.receipe\_image}}</td>

          </tr>

          {% endfor %}

        </tbody>

      </table>

* Search for Django media files. Django static files. Go settings.py and update static files.

Settings.py static

import os

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/5.1/howto/static-files/

STATIC\_URL = '/static/'

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

STATICFILES\_DIR={os.path.join(BASE\_DIR,"public/static")}

MEDIA\_ROOT=os.path.join(BASE\_DIR,'media')

MEDIA\_URL='/media/'

# Default primary key field type

# https://docs.djangoproject.com/en/5.1/ref/settings/#default-auto-field

DEFAULT\_AUTO\_FIELD = 'django.db.models.BigAutoField'

Urls.py code

from django.contrib import admin

from django.urls import path

from home.views import \*

from vege.views import \*

from django.conf.urls.static import static

from django.conf import settings

from django.contrib.staticfiles.urls import staticfiles\_urlpatterns

urlpatterns = [

    path('admin/', admin.site.urls),

    path('', home, name="home"),

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

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

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

] +static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)#blank suggests calling default port, that is 8000

# if settings.DEBUG:

#     urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

# urlpatterns += staticfiles\_urlpatterns

if settings.DEBUG:

    urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

HTML

<table class="table">

        <thead>

          <tr>

            <th scope="col">#</th>

            <th scope="col">Receipe Name</th>

            <th scope="col">Receipe Description</th>

            <th scope="col">Image</th>

          </tr>

        </thead>

        <tbody>

            {% for receipe in receipes %}

          <tr>

            <th scope="row">{{forloop.counter}}</th>

            <td>{{receipe.receipe\_name}}</td>

            <td>{{receipe.receipe\_description}}</td>

            <td>

                <img src="/media/{{receipe.receipe\_image}}" style="height: 200px;">

            </td>

          </tr>

          {% endfor %}

        </tbody>

      </table>

**Making links dynamic**

def delete\_receipe(request, id):

    queryset= Receipe.objects.get(id = id)

    queryset.delete()

    return redirect('/receipes')

path('delete\_receipe/<id>', delete\_receipe, name="delete\_receipe"),

Html

<td>

                <a href="/delete\_receipe/{{receipe.id}}" class="btn btn-danger">Delete>

</td>

UPDATE

def update\_receipe(request, id):

    queryset= Receipe.objects.get(id = id)

    if request.method=="POST":

        data=request.POST

        receipe\_name=data.get('receipe\_name')

        receipe\_description=data.get('receipe\_description')

        receipe\_image=request.FILES.get('receipe\_image')

        queryset.receipe\_name=receipe\_name

        queryset.receipe\_description=receipe\_description

        queryset.receipe\_image=receipe\_image

        queryset.save()

        return redirect('/receipes')

    context={'receipe':queryset}

    return render(request, 'update\_receipes.html',context)

Html

{% extends "home/base.html" %}

{% block title %}

Project 1 Receipes

{% endblock %}

{% block start %}

<ul class="nav nav-tabs">

    <li class="nav-item">

        <a class="nav-link" href="/">Home</a>

    </li>

    <li class="nav-item">

        <a class="nav-link" href="/about/">About</a>

    </li>

    <li class="nav-item">

        <a class="nav-link active" href="/receipes/">Receipes</a>

    </li>

    <li class="nav-item">

        <a class="nav-link" href="/contact/">Contact</a>

    </li>

</ul>

<h1>This is Receipe Page</h1>

<div class="container mt-5">

    <form class="col-6 mx-auto card p-3" method="post" enctype="multipart/form-data">

        {% csrf\_token %}

        <h3>Update Receipes</h3>

        <hr>

        <div class="mb-3">

            <label for="receipeName" class="form-label">Receipe Name</label>

            <input name="receipe\_name" value="{{receipe.receipe\_name}}" required class="form-control">

        </div>

        <div class="mb-3">

            <label for="receipe\_description" class="form-label">Receipe Description</label>

            <textarea class="form-control" value="" required name="receipe\_description">{{receipe.receipe\_description}} </textarea>

        </div>

        <div class="mb-3">

            <label for="receipeImage" class="form-label">Receipe Image</label>

            <input name="receipe\_image" type="file" value="{{receipe\_image}}" class="form-control">

        </div>

        <button type="submit" class="btn btn-success" id="myButton">Update</button>

    </form>

</div>

<!--

<script>

    document.getElementById('myButton').addEventListener('click', function(event) {

        event.preventDefault(); // Prevent the form from submitting

        alert("Recipe has been added!"); // Display the alert

        // If you want to submit the form after the alert, you can manually submit it here:

        // document.querySelector('form').submit();

    });

</script>

-->

{% endblock %}

**SEARCH FUNCTIONALITY**

**Views**

queryset=Receipe.objects.all()

    if request.GET.get('search'):

        queryset=queryset.filter(receipe\_name\_\_icontains = request.GET.get('search'))

html

<hr>

    <div class="mt-5">

        <div class="mx-auto col-6 ">

            <form >

            <div class="mb-3">

                <label for="search\_receipe" class="form-label">Search Receipe</label>

                <input name="search" type="text" required class="form-control">

            </div>

            <button type="submit" class="btn btn-primary">Search </button>

            </form>

        </div>

    </div>

**Advanced Authentication System**

* ALLOWED\_HOSTS = []

Is empty initially

* Choose Debug =True,
* Let’s add authentication for receipes page(so edit models of that app)

from django.contrib.auth.models import User

* This imports the User model from Django's built-in authentication framework, allowing you to manage user accounts, authentication, and permissions in your application. Use it to create, retrieve, update, or delete user instances.

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

* This creates a foreign key relationship to the User model. It links the current model to a specific user, and when the user is deleted, related records will also be deleted (on\_delete=models.CASCADE).

user=models.ForeignKey(User,on\_delete=models.SET\_NULL)

* This creates a foreign key relationship to the User model. If the related user is deleted, the field is set to NULL (on\_delete=models.SET\_NULL). Ensure the field allows null=True for this to work.

user=models.ForeignKey(User,on\_delete=models.SET\_DEFAULT

VIEWS

from django.contrib.auth.models import User

from django.contrib import messages

def register\_page(request):

    if request.method == "POST":

        first\_name = request.POST.get('first\_name')

        last\_name = request.POST.get('last\_name')

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

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

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

        # Validate username and other required fields

        user=User.objects.filter(username=username)

        if user.exists():

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

            return redirect('/register/')

        user=User.objects.filter(email=email)

        if user.exists():

            messages.info(request,'This email is already registered')

            return redirect('/register/')

        # Create the user

        user = User.objects.create\_user(

            username=username,

            first\_name=first\_name,

            last\_name=last\_name,

            email=email,

            password=password  # Automatically hashes the password

        )

        # Redirect to login or home page

        messages.info(request,"Account created Successfully")

        return redirect('/register/')  # Redirect to the login page after registration

    return render(request, 'register.html')

register.html

<h1>This is Sign Up Page</h1>

<div class="container mt-5">

    <form class="col-6 mx-auto card p-3" method="post" enctype="multipart/form-data">

        {% csrf\_token %}

        <h3>Register</h3>

        <hr>

        {% if messages %}

        <div class="alert alert-primary" role="alert">

          {% for message in messages %}

          {{ message }}

          {% endfor %}

        </div>

        {% endif %}

        <div class="mb-3">

            <label for="exampleInputEmail1" class="form-label">First Name</label>

            <input type="text" class="form-control" name="first\_name" id="exampleInputEmail1" aria-describedby="emailHelp">

          </div>

          <div class="mb-3">

            <label for="exampleInputEmail1" class="form-label">Last Name</label>

            <input type="text" class="form-control" name="last\_name" id="exampleInputEmail1" aria-describedby="emailHelp">

          </div>

          <div class="mb-3">

            <label for="exampleInputEmail1" class="form-label">Email</label>

            <input type="email" class="form-control" name="email"  id="exampleInputEmail1" aria-describedby="emailHelp">

          </div>

            <div class="mb-3">

              <label for="exampleInputEmail1" class="form-label">Username</label>

              <input type="text" class="form-control"  name="username"   id="exampleInputEmail1" aria-describedby="emailHelp">

            </div>

            <div class="mb-3">

              <label for="exampleInputPassword1" class="form-label">Password</label>

              <input type="password" class="form-control" name="password" id="exampleInputPassword1">

            </div>

            <p>Already have account?  <a href="/login/"> Login Here</a> </p>

            <button type="submit" class="btn btn-primary">Submit</button>

    </form>

</div>

{% endblock %}

Login, logout/register

1.html

<li class="nav-item">

        {% if request.user.is\_authenticated %}

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

        {% else %}

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

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

        {% endif %}

    </li>

</ul>

**Views**

If you're able to access the "Recipes" page without logging in, there’s likely an issue with the @login\_required decorator or your Django setup. Let's troubleshoot and fix this step.

**Checklist to Ensure @login\_required Works:**

**1. Check @login\_required Decorator Placement**

The @login\_required decorator must be applied directly to the view function. Ensure your receipes view starts like this:

from django.contrib.auth.decorators import login\_required

@login\_required

def receipes(request):

# Your code here...

return render(request, 'receipes.html')

**2. Confirm Middleware Setup**

The @login\_required decorator relies on Django's AuthenticationMiddleware. Ensure that it’s included in the MIDDLEWARE list in settings.py:

MIDDLEWARE = [

...,

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

'django.contrib.auth.middleware.AuthenticationMiddleware', # Required for authentication

...,

]

**3. Check the Login URL Setting**

The @login\_required decorator redirects unauthenticated users to the login page. By default, it redirects to /accounts/login/. If your login page is at /login/, update the LOGIN\_URL setting in settings.py:

LOGIN\_URL = '/login/'

**4. Test the @login\_required Behavior**

To verify that @login\_required is working:

1. Log out of the application.
2. Manually visit /receipes/ in the browser.
3. You should be redirected to the login page (/login/).

**If It’s Still Not Working:**

* Double-check that your Django server is running with the latest code (restart the server if needed).
* Ensure you're accessing the correct endpoint (e.g., /receipes/ and not another similar URL).

Let me know if these steps resolve the issue. If not, share any specific errors or unexpected behavior, and we’ll debug further!

**DON’T GO BACK TO HOME PAGE**

To prevent users from accessing the "Recipes" page after logging out and clicking the back button, you need to ensure that the browser doesn’t cache the protected page. This involves setting the appropriate cache control headers.

Let’s implement this step-by-step.

**Step 1: Disable Caching for the Recipes Page**

Add cache control headers to the "Recipes" page to ensure it isn't stored in the browser cache. You can use Django's cache\_control decorator.

**Updated Code for Recipes View:**

from django.views.decorators.cache import cache\_control

from django.contrib.auth.decorators import login\_required

@login\_required

@cache\_control(no\_cache=True, must\_revalidate=True, no\_store=True)

def receipes(request):

if request.method == "POST":

data = request.POST

receipe\_name = data.get('receipe\_name')

receipe\_description = data.get('receipe\_description')

receipe\_image = request.FILES.get('receipe\_image')

Receipe.objects.create(

receipe\_name=receipe\_name,

receipe\_description=receipe\_description,

receipe\_image=receipe\_image

)

return redirect('/receipes')

queryset = Receipe.objects.all()

if request.GET.get('search'):

queryset = queryset.filter(receipe\_name\_\_icontains=request.GET.get('search'))

context = {'receipes': queryset}

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

**What This Does:**

The @cache\_control decorator adds the following headers to the response:

1. **no\_cache=True:** Prevents caching of the page.
2. **must\_revalidate=True:** Forces the browser to validate with the server before reloading the page.
3. **no\_store=True:** Ensures the page is not stored in the browser or proxy caches.

**Step 2: Clear Browser Cache After Logout**

Modify the logout\_page view to add cache control headers as well, ensuring that cached pages are cleared on logout.

**Updated Code for Logout View:**

from django.contrib.auth import logout

from django.views.decorators.cache import cache\_control

@cache\_control(no\_cache=True, must\_revalidate=True, no\_store=True)

def logout\_page(request):

logout(request) # Logs out the user

return redirect('/login/')

**Step 3: Test the Behavior**

1. **Log In:**
   * Access the "Recipes" page as a logged-in user.
2. **Log Out:**
   * Click the logout button to log out of the application.
3. **Back Button:**
   * Click the browser's back button. You should be redirected to the login page, and the browser should not show the cached "Recipes" page.

**Step 4: Verify Across Browsers**

Different browsers handle caching differently. Test this functionality on:

* Google Chrome
* Firefox
* Safari
* Edge

If everything works, you're all set. Let me know if there's any issue!

**RENAMING APP BY RIGHT CLICKING**

Yes, renaming an app folder in a Django project directly (e.g., by right-clicking) can cause issues if you don't update all related references. Django uses the app name in various places, and renaming it requires careful adjustments.

Here are the potential issues and required steps to handle them:

**1. App Name in INSTALLED\_APPS**

* The app's name must match its folder name as registered in the INSTALLED\_APPS list in settings.py. Update INSTALLED\_APPS with the new name.

**2. Migrations**

* Django's migrations are tied to the app name. Renaming an app might confuse Django about which app the existing migrations belong to.
* You might need to manually edit the migration files in the migrations/ folder to reflect the new app name, especially in the apps.py and dependencies.

**3. References in Code**

* Search for any imports or references to the old app name in your codebase and update them to the new name.
* Examples:
* from old\_app.models import MyModel # Update to new\_app.models

**4. App Configuration (apps.py)**

* Update the name attribute in the app's apps.py file to reflect the new folder name. For example:
* class MyAppConfig(AppConfig):
* name = 'new\_app'

**5. Database Table Names**

* Django generates database table names using the app name by default. Renaming the app can lead to table mismatches.
* To avoid issues, you might need to use the db\_table attribute in your models to explicitly define the table names or use a migration to rename tables.

**6. URLs and Static Files**

* If your app folder contains static files or templates, ensure paths referencing the old app name are updated.

**7. Admin Panel**

* If your app is registered in the Django admin, you might need to update admin-related configurations.

**Recommendation:**

If renaming is unavoidable, follow these steps systematically:

1. Rename the folder.
2. Update all references in settings.py, apps.py, and codebase imports.
3. Run makemigrations and migrate to ensure database integrity.
4. Test your project thoroughly to ensure everything works as expected.

If possible, avoid renaming apps in production projects, as it can lead to significant maintenance overhead.

**Session And Cookie Management**

**Concepts We’ll Cover:**

1. **Django Session Basics**
   * What sessions are and how to store/retrieve data in them.
2. **Session Expiration**
   * Setting expiration times for sessions.
3. **Using Cookies in Django**
   * How to read and set cookies in views.
4. **Session Storage Backends**
   * Understanding where session data is stored (default is the database).
5. **Customizing Session Behavior**
   * Customizing session settings in settings.py.
6. **Session Authentication**
   * How to use sessions to manage user authentication.

A **view** in Django is a Python function that handles requests and returns a response. Here's an example:

from django.http import HttpResponse

def my\_view(request):

return HttpResponse("Hello, World!")

This view simply returns "Hello, World!" as the response.

**What is a Session in Django?**

In Django, a **session** allows you to temporarily store information about a user across multiple requests.

* To **set** a session value, you use:  
  request.session['key'] = value
* To **retrieve** a session value, you use:  
  value = request.session.get('key')

**Session Expiration:**

**1. Session Expiration**

* Sessions are stored on the server, and Django assigns each session a unique ID.
* By default, Django’s sessions expire when the browser is closed.
* You can set a custom expiration time, so the session lasts longer or expires after a set period of time.

**Task for You:**

1. Set the session to expire after 5 minutes.
2. Update your view (e.g., set\_favorite\_movie) to use this new expiration time.

Here's how you can set the session expiration time:

python

CopyEdit

# In your views.py or inside the set\_favorite\_movie view

request.session.set\_expiry(300) # Time in seconds (300 seconds = 5 minutes)

Once you've done this, test by setting the favorite movie and checking if the session expires after 5 minutes. Let me know how it goes!

The reason to use {% url 'set\_favorite\_movie' %} instead of directly adding the URL (/set-favorite-movie/) is for **Django's URL reversing system**. Here's why it's beneficial:

**Advantages of Using {% url 'set\_favorite\_movie' %}:**

1. **Maintainability**:
   * If the URL pattern for set-favorite-movie changes (e.g., if you rename it to /movie/set-favorite-movie/), you only need to update the URL in one place: your urls.py. The link in the templates using {% url %} will automatically adjust.
   * Without {% url %}, you'd have to update all hardcoded URLs across your templates manually.
2. **Refactoring and Organization**:
   * Django gives each URL pattern a name (like 'set\_favorite\_movie'). This allows you to reference the URL by name rather than hardcoding the path, making your code cleaner and easier to refactor in the future.
3. **Cross-App Linking**:
   * If your project grows and you use the same URL structure across multiple apps, {% url %} helps Django correctly link across different apps without needing to manage full paths manually.

**Example:**

If you define the URL in urls.py like this:

path('set-favorite-movie/', views.set\_favorite\_movie, name='set\_favorite\_movie'),

Then, in your template, this will be linked properly using {% url %}:

<a href="{% url 'set\_favorite\_movie' %}">Set Favorite Movie</a>

**Conclusion:**

If you hardcode the URL (/set-favorite-movie/), it will work in the short term. But if you ever decide to change the URL structure (or move the app around), using {% url %} ensures that your links continue to work without needing to change each one.

You can continue using /set-favorite-movie/ if you're okay with manually managing the paths, but for long-term flexibility and maintainability, it's a good practice to use {% url 'set\_favorite\_movie' %}.

Perfect! Setting the session expiration time to 5 seconds will let you quickly test the session expiry behavior.

**Steps to Set the Session Expiry to 5 Seconds:**

1. In your view (e.g., set\_favorite\_movie), add the following line to set the session expiration:

# In your views.py or inside the set\_favorite\_movie view

request.session.set\_expiry(5) # 5 seconds

1. After setting the session, you can test the following:
   * Set the favorite movie by visiting /set-favorite-movie/.
   * Then, wait 5 seconds and try visiting /get-favorite-movie/ again.
   * The session should expire after 5 seconds, and you'll see the message saying **"No favorite movie set."**

Let me know when you've tried it, and we can discuss the next concept!

Awesome! I'm glad it worked.

Now that you’ve seen how session expiration works, let's move on to the next concept:

**Concept 3: Using Cookies in Django**

In Django, cookies are small pieces of data sent from the server to the client’s browser and stored there. You can access and modify cookies in Django views just like session data.

**Key Points:**

* **Setting Cookies**: You can set cookies in the response using response.set\_cookie().
* **Retrieving Cookies**: You can retrieve cookies from request.COOKIES.

**Task for You:**

1. **Set a Cookie**: After setting the session value, set a cookie called movie\_cookie with the movie name ('Inception').
2. **Retrieve the Cookie**: In the get\_favorite\_movie view, retrieve the movie\_cookie value and display it alongside the session value.

**Code Example:**

**1. Set Cookie in the View:**

In your set\_favorite\_movie view:

def set\_favorite\_movie(request):

# Set session expiration and value

request.session.set\_expiry(5) # 5 seconds

request.session['favorite\_movie'] = 'Inception'

# Set a cookie

response = HttpResponse("Your favorite movie has been set!")

response.set\_cookie('movie\_cookie', 'Inception', max\_age=5) # Cookie expires in 5 seconds

return response

**2. Retrieve Cookie in the View:**

In the get\_favorite\_movie view, retrieve the cookie:

def get\_favorite\_movie(request):

# Get session value

favorite\_movie = request.session.get('favorite\_movie', 'No favorite movie set.')

# Get cookie value

movie\_cookie = request.COOKIES.get('movie\_cookie', 'No movie cookie set.')

return HttpResponse(f"Session movie: {favorite\_movie}<br>Cookie movie: {movie\_cookie}")

The **session timeout** and **cookie timeout** are independent mechanisms:

1. **Session Timeout**:
   * The session data is stored on the server (e.g., database or file system).
   * When you set request.session.set\_expiry(), you’re telling Django to delete the session data after the specified time.
   * Once the session expires, the data is no longer available—even if the session cookie still exists in the browser.
2. **Cookie Timeout (via max\_age)**:
   * Cookies are stored on the client (browser).
   * When you set a cookie with response.set\_cookie('key', 'value', max\_age=seconds), the cookie will be deleted from the client after the specified time (max\_age).
   * The cookie can still exist even after the session expires, depending on its max\_age.

Great! Now that you're comfortable with session expiration and cookies, let's move on to the next concept.

**Concept 4: Session Storage Backends**

By default, Django stores session data in the **database**. However, Django supports several session storage backends, and you can choose one based on your needs.

**Common Session Backends in Django:**

1. **Database-backed sessions (Default)**:
   * Session data is stored in the database.
   * Django automatically creates a table (django\_session) for storing session data.
2. **File-based sessions**:
   * Session data is stored in files on the server.
   * Useful for simple setups or development.
3. **Cache-backed sessions**:
   * Session data is stored in the cache (e.g., Redis, Memcached).
   * Ideal for high-performance applications.
4. **Cookie-based sessions**:
   * All session data is stored in the user's browser as a signed cookie.
   * No server-side storage is required, but limited to small amounts of data.

**Task for You:**

1. Check the default session backend your Django project is using.
2. Change the backend to **file-based sessions**.
3. Test if the session still works after the change.

**Step-by-Step Instructions:**

**1. Check the Current Session Backend:**

By default, Django uses the database backend. Check your settings.py file to confirm:

SESSION\_ENGINE = 'django.contrib.sessions.backends.db' # Default

**2. Switch to File-Based Sessions:**

In your settings.py, change the session backend:

SESSION\_ENGINE = 'django.contrib.sessions.backends.file'

SESSION\_FILE\_PATH = None # Default (uses the system's temp directory)

This tells Django to store session data in temporary files on the server.

**3. Restart the Server:**

After modifying settings.py, restart your Django server to apply the changes.

**4. Test:**

* Visit /set-favorite-movie/ to set the session.
* Visit /get-favorite-movie/ to retrieve it.
* Check the server's temporary directory for session files (e.g., /tmp on Linux).

Great! Now that you’ve learned about session storage backends, let’s move on to the next concept:

**Concept 5: Customizing Session Behavior**

Django allows you to customize various aspects of session behavior through settings in settings.py. This includes settings for:

1. **Session Cookie Settings** (e.g., secure cookies, cookie name, etc.).
2. **Persistent vs. Non-Persistent Sessions** (whether to clear sessions when the browser closes).
3. **Session Timeout Defaults** (global timeout for all sessions).

**Key Settings to Know:**

1. **SESSION\_COOKIE\_NAME**:
   * Default: 'sessionid'
   * The name of the cookie used for the session ID. You can change it to something custom.
2. **SESSION\_COOKIE\_AGE**:
   * Default: 1209600 (2 weeks in seconds).
   * The age of the session cookie in seconds. Controls how long sessions last by default.
3. **SESSION\_COOKIE\_SECURE**:
   * Default: False
   * Ensures the session cookie is sent only over HTTPS.
4. **SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE**:
   * Default: False
   * If True, the session will expire when the browser is closed.

**Task for You:**

1. Customize your session settings in settings.py:
   * Change the session cookie name to "my\_session\_cookie".
   * Set the session cookie age to 60 seconds.
   * Enable the session to expire when the browser closes.
2. Test the behavior:
   * Verify the cookie name in your browser's developer tools.
   * Confirm the session expires as expected after 60 seconds or when the browser is closed.

**Example Configuration:**

Add the following to your settings.py:

# Change session cookie name

SESSION\_COOKIE\_NAME = 'my\_session\_cookie'

# Set session cookie age to 60 seconds

SESSION\_COOKIE\_AGE = 60

# Make session expire when the browser is closed

SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE = True

Fantastic! You're doing great. Now let's move on to the **final concept**.

**Concept 6: Clearing Sessions**

There are times when you might need to clear session data, either:

1. **Clearing specific session keys** (e.g., removing only one piece of session data).
2. **Clearing all session data for a user**.

**1. Clearing Specific Session Keys**

You can delete a single session key using the del keyword:

# Delete a specific session key

del request.session['favorite\_movie']

If the key does not exist, it will raise a KeyError. To avoid this, use pop with a default value:

# Safely remove a session key

request.session.pop('favorite\_movie', None) # None prevents KeyError if the key doesn't exist

**2. Clearing All Session Data for a User**

You can clear all session data for the current user using:

# Clear all session data for the user

request.session.flush()

This will:

* Clear all session data for the user.
* Remove the session cookie from the browser.
* Create a new session for future requests.

**Task for You:**

1. **Update get\_favorite\_movie**:
   * Add a button/link to **clear the favorite movie session key** (specific clearing).
   * Add another button/link to **clear all session data** (flush).
2. **Test**:
   * Set the favorite movie using /set-favorite-movie/.
   * Clear the favorite movie using the specific key clearing.
   * Set it again and then flush all session data.
   * Verify that the session behaves as expected in each case.