Django Class Notes

Clarusway

Pizza App

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets

Needs

- Python
- pip
- virtualenv

Summary

- Introduction
- Spin up the project
- Create Models
- Create Forms (ModelForm)
    - Form fields
    - Widgets
- Create Views
    - home view
    - order view
        - Dealing with post requests
        - csrf
- Creating templates
- Working with static files

- Create url patterns
- Adding messages
- Adding base.html
- Adding bootstrap
- Adding navbar
    - Include navbar inside base.html
- Put messages to base.html
- Create update_order view
    - Add an edit button to the order page
- Adding authentication
    - using Django auth views
    - login
    - register
    - logout
- Adding Profile (Extending the existing User model)
- Adding price to the model (and earn some money)

Introduction

In this session, we will create a Django app named "Nadia's Garden" which will let users to order pizza.

Most websites require the use of forms to receive data from users, so it is crucial to know how to safely collect and handle data while maintaining a user-friendly experience on your website. In this course, learn how to use Django to create forms, templates and much more!

Spin up the project

- Create a working directory, with a meaningful name, and cd to the new directory.

- Create a virtual environment as a best practice.

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv env -p python3 # for Mac/Linux
```

- Activate the virtual environment.

```
.\env\Scripts\activate  # for Windows, may need to switch powershell on this
operation.
source env/bin/activate  # for MAC/Linux
```

See the (env) sign before your command prompt.

- Install Django.

```
pip install django  # or
py -m pip install django
```

- (Optional) See installed python packages:

```
pip freeze

# you will see like:

# asgiref==3.5.0
# Django==4.0.4
# sqlparse==0.4.2
# tzdata==2022.1

# If you see lots of things here, that means there is a problem with your virtual
env activation. Activate scripts again!
```

- Create the `requirements.txt` on your working directory, and send your installed packages to this file, requirements file must be up to date.

```
pip freeze > requirements.txt
# In this project we will not use this file. But this is a standard procedure to
learn.
```

- Install python-decouple, make necessary settings, and put the secret key to a seperate file named `.env`. Put a `.gitignore` file on the working directory.
- Publish your project to Github.
- Create the project.

```
django-admin startproject main .
# With . it creates a single project folder.
# Avoiding nested folders.
# Naming depends on the company, team and the project.
```

- (Optional) If you created your project without "." at the end, change the name of the project main directory to src to distinguish it from a subfolder with the same name.

```
# Be careful to use your own folder names.
mv main src
```

- (Optional) Go to the same level with the `manage.py` file if you create your project with nested folders at step 7:

```
cd main  # or, if you changed;
cd src
```

- Create the app.

```
# Using app name as "pizza"
python manage.py startapp pizzas  # or
py -m manage.py startapp pizzas
```

- Go to the `settings.py` and add another line to the INSTALLED_APPS:

```
'pizzas',
```

- Run the server and check if your project up and running.

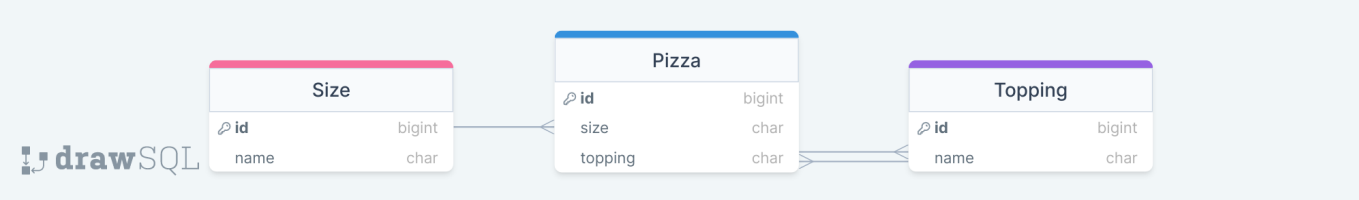
```
python manage.py runserver  # or
py -m manage.py runserver
```

Go to http://localhost:8000/ in your browser. You should see Django rocket, which means you successfully created the project.

The installation worked successfully! Congratulations!

Create Models

In this project, you need to create a Size, a Topping and a Pizza model.



- Go to models.py and create Size, Topping and Pizza models like below:

```
from django.db import models


class Size(models.Model):
    name = models.CharField(max_length=6)
```

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


class Topping(models.Model):
    name = models.CharField(max_length=15)

    def __str__(self):
        return self.name


class Pizza(models.Model):
    size = models.ForeignKey(Size, on_delete=models.CASCADE)
    topping = models.ManyToManyField(Topping)

    # To handle str method while using many-to-many relationship
    # is a little tricky; export toppings from the list and show
    # them with a comma seperation:
    def __str__(self):
        toppings = ', '.join(str(v) for v in self.topping.all())
        return f'{self.size.name} - {toppings}'
```

- Register your page to admin.py.
- Manage migrations and create a superuser:

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
```

Create Forms

- Forms can be created using html. With the backend perspective, we can create django forms from scratch. But the most elagant way is using ModelForm. Create forms.py under pizza app and create your forms like below:

```
from django import forms
from .models import Pizza


class PizzaForm(forms.ModelForm):
    class Meta:
        model = Pizza
        # form fields must be the same with the model.
        # We don't have to use all of them though!
        fields = (
            'size',
            'topping',
```

```
        )
        # widgets are changing the behaviour of the form field
        widgets = {
            'size':forms.RadioSelect,
            'topping':forms.CheckboxSelectMultiple,
        }
```

Create Views

- Go to views.py under "pizzas" directory, and create your views like below:

```
from django.shortcuts import render
from .forms import PizzaForm
from django.contrib import messages


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


def order(request):

    # Always we will begin with an empty pizza form.
    form = PizzaForm()

    # Here we need to handle post request
    # if a user fills out the form and push
    # order button, this action will create a pizza
    # object. That will change the state of our db.
    # That is a post request!
    if request.method == 'POST':
        # User filled out the form and we can catch the submission by request.POST
        form = PizzaForm(request.POST)
        # if the form passes all the validations
        if form.is_valid():
            # we will save the object to the db
            form.save()
    # a context like data in REST, enables us to send some
    # info to the frontend
    context = {
        'form':form,
    }
    # at the end, we are rendering the request, the template,
    # and the context including pizza form
    return render(request, 'pizzas/order.html', context)
```

Creating Templates

- We have views and we need to create home.html and order.html templates. Create a new folder under `pizzas` named `templates/pizzas`. There may be multiple apps in your project and that means

multiple templates folder. Django can serve wrong template if you do not specify the app name under templates folder.

- Create home.html and order.html under `templates/pizzas` folder.
- order.html will be like:

```
<h1>Order Page</h1>

<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Order">
</form>
```

- home.html will be like:

```
<h1>Home Page</h1>

<p>{{ request.user | upper }}</p>

{% load static %}

<img style='background-size: contain;' src="{% static 'pizzas/nadiasgarden.jpg'
%}" alt="Nadia's Garden" width="100%" height="auto">
```

- Here we see a request.user variable which dinamically gets the current user of the session. And an image for background using static files.

Working with static files

- To work with static files first we need to define `STATIC_URL`, which by default defined by django. Static files finder engine will search every app and look for folders named `static`. So we need to create a folder named `static/pizzas` under our app and put the image we want to use inside this folder.

Create url patterns

- Include URL path of the new app to the project url list, go to urls.py and add:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pizzas.urls'))
]
```

- Add urls.py file under pizzas directory and add:

```
from django.urls import path
from .views import home, order

urlpatterns = [
    path('', home, name='home'),
    path('order/', order, name='order'),
]
```

- Now we are ready to check the endpoints.

Adding messages

- After the customers order a pizza, they need to see some message that shows the order status. Adding a message to our view is a must to show some feedback to users.

- Go to order view and add:

```
from django.contrib import messages


def order(request):
    form = PizzaForm()
    if request.method == 'POST':
        form = PizzaForm(request.POST)
        if form.is_valid():
            form.save()
            # Add a success message after the order
            messages.success(request, 'Order is successful!!!')
    context = {
        'form':form,
    }
    return render(request, 'pizzas/order.html', context)
```

- Thanks to Django messages framework, the message will be sent to frontend automatically. Just we need to catch them in our template.

- Go to order.html and add:

```
<h1>Order Page</h1>

{% if messages %}
    {% for message in messages %}
        {{ message }}
    {% endfor %}
{% endif %}
```

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Order">
</form>
```

- This code will get the messages and display them on the page.

Adding base.html

- If you do the same thing on your pages, don't repeat yourself. Add a base.html and put some common html code inside as a best practice. Create this base template including bootstrap like below:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
crossorigin="anonymous">

    <title>Nadia's Garden</title>
  </head>
  <body>


    {% block body %}

    {% endblock body %}


    <!-- Optional JavaScript; choose one of the two! -->

    <!-- Option 1: Bootstrap Bundle with Popper -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-
MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtIaxVXM"
crossorigin="anonymous"></script>

    <!-- Option 2: Separate Popper and Bootstrap JS -->
    <!--
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"
integrity="sha384-
```

```
IQsoLXl5PILFhosVNubq5LC7Qb9DXgDA9i+tQ8Zj3iwWAwPtgFTxbJ8NT4GN1R8p"
crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.min.js"
integrity="sha384-
cVKIPhGWiC2Al4u+LWgxfKTRIcfu0JTxR+EQDz/bgldoEyl4H0zUF0QKbrJ0EcQF"
crossorigin="anonymous"></script>
    -->
  </body>
</html>
```

- We have got this starter template from bootstrap webpage. Just change the title to show our page name. And add a body block inside. Other pages will be here inside this body block!
- To inherit base.html to other pages just add `extends` and `block` tags.

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

{% block body %}

    {# Put the html code here!!! #}

{% endblock body %}
```

- Ther result for home.html will be like:

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


{% block body %}

    <h1>Home Page</h1>
    <p>{{ request.user | upper }}</p>
    {% load static %}
    <img style='background-size: contain;' src="{% static
'pizzas/nadiasgarden.jpg' %}" alt="Nadia's Garden" width="100%" height="auto">

{% endblock body %}
```

- And for order page: <

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


{% block body %}

<h1>Order Page</h1>
```

```
{% if messages %}
    {% for message in messages %}
        {{ message }}
    {% endfor %}
{% endif %}

<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Order">
</form>

{% endblock body %}
```

Second Session

Adding navbar

- Lets find a navbar from bootstrap page and use it in our project.

```
<nav class="navbar navbar-expand-lg bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-
bs-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav">
        <a class="nav-link active" aria-current="page" href="#">Home</a>
        <a class="nav-link" href="#">Features</a>
        <a class="nav-link" href="#">Pricing</a>
        <a class="nav-link disabled">Disabled</a>
      </div>
    </div>
  </div>
</nav>
```

- I want to seperate base.html and navbar.html to have a cleaner code. Create a navbar.html put the code inside.

- Change the color on the very first line. Change the first link to home page. Change home link with order page. Delete last three links. The result will be like:

```
<nav class="navbar navbar-expand-lg navbar-dark" style="background-color:
#238a44;">
```

```
  <div class="container-fluid">
    <a class="navbar-brand" href="{% url 'home' %}">Nadia's Garden</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-
bs-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav">
        <a class="nav-link active" aria-current="page" href="{% url 'order'
%}">Order</a>
      </div>
    </div>
  </div>
</nav>
```

- It is possible to use endpoint path like '/' but the best practice is to use name values from urls.py. So when you change the path for this endpoint it will not effect the link. So go to pizzas/urls.py and add name variables to the endpoints.

```
from django.urls import path
from .views import home, order


urlpatterns = [
    path('', home, name='home'),
    path('order/', order, name='order'),
]
```

- To use navbar.html inside base.html use `include` tag just before body block.

```
<body>

    {% include 'pizzas/navbar.html' %}

    {% block body %}

    {% endblock body %}
```

Put messages to base.html

- We will have messages almost on every page. So let's put that to the base.html. Cut messages lines from order.html and paste it to base.html:

```
<body>
```

```
    {% include 'pizzas/navbar.html' %}

    {% if messages %}
      {% for message in messages %}
        {{ message }}
      {% endfor %}
    {% endif %}

    {% block body %}

    {% endblock body %}
```

- By the way, there is a better if block for messages, working well with the bootstrap:

```
<div class="messages">
    {% if messages %}
        <div>
            {% for message in messages %}
                <p {% if message.tags %} class="text-center alert alert-{{
message.tags }}" {% endif %}>
                {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}Important:{%
endif %}
                {{ message }}
                </p>
            {% endfor %}
        </div>
    {% endif %}
</div>
```

- Replace this code with existing one and try!
- With this bootstrap changes, you can add a div with container class to order page to make things a little shiny:

```
<div class="container">

    <h1>Order Page</h1>

    <form action="" method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Order">
    </form>

</div>
```

Create update_order view

- Users can change their minds and edit their orders. So, we need to create a new view:

```
from .models import Pizza


# To update an object, we need the id of the object.
# So, put id next to request.
def update_pizza(request, id):
    # Define a pizza object with the id comming with request
    pizza = Pizza.objects.get(id=id)
    # To update the pizza, we will fill the form with the
    # pizza object informations coming from the database
    form = PizzaForm(instance=pizza)
    # If the user changes anything, or keeps existing data and hit update
    # button this will be a post request which will change our database
    if request.method == 'POST':
        form = PizzaForm(request.POST, instance=pizza)
        if form.is_valid():
            form.save()
            messages.success(request, 'Updated the pizza!')
        else:
            messages.warning(request, 'Something wrong!')
    context = {
        'form': form,
    }
    return render(request, 'pizzas/update_pizza.html', context)
```

- We need to create `pizzas/update_pizza.html` template:

```
{% extends "pizzas/base.html" %}

{% block body %}

<div class="container">
    <h1>Edit Order</h1>
    <div>
        <form action="" method="POST">
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Update">
        </form>
    </div>
</div>

{% endblock body %}
```

- Lets create the correspanding url path:

```
from .views import edit_pizza


urlpatterns = [
    # ...
    path('order/<int:id>/', update_pizza, name='update'),
]
```

Add an edit button to the order page

- After a successful order, there may be a button to update the order. We need to add this button to order page. But we need a trigger to show this button only after a successful order. So, we can do that on order view:

```
def order(request):
    form = PizzaForm()
    if request.method == 'POST':
        form = PizzaForm(request.POST)
        if form.is_valid():

            # Here we will catch the pizza object and get the id
            # so we may edit the pizza object having this id
            pizza = form.save()
            pizza_pk = pizza.id

            messages.success(request, 'Your order is on the way!')
            form = PizzaForm()
        else:
            messages.warning(request, 'Something wrong!')
            # if there is something wrong and no pizza object created
            # there will be no id
            pizza_pk = None

        context = {
            'form': form,
            # sending pizza_pk to frontend to show/no-show update button
            'pizza_pk': pizza_pk,
        }
        return render(request, 'pizzas/order.html', context)

    context = {
        'form': form,
    }
    return render(request, 'pizzas/order.html', context)
```

- Now we have a trigger including the id of the pizza object. Go to order template and add the button:

```
{% if pizza_pk %}
    <button><a href="{% url 'edit' pizza_pk %}">Edit Order</a></button>
{% endif %}
```

Adding authentication

- For this project we can use Django built-in views for login, logout etc.
- Create a new app named `users`.

```
python manage.py startapp users  # or
py -m manage.py startapp users
```

- Go to the `settings.py` and add another line to the INSTALLED_APPS:

```
'users',
```

- Create `users/urls.py`.

```
from django.urls import path, include


urlpatterns = [
    # using some urls which Django give us about authentication:
    path('', include('django.contrib.auth.urls')),
]
```

- Include it to main url patterns.

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pizzas.urls')),
    path('user/', include('users.urls')),
]
```

- Visit http://127.0.0.1:8000/user/ and see the available patterns. We need a couple of things here. First start with login.html. This must be under `users/templates/registration` folder.

```
{% extends "pizzas/base.html" %}

{% block body %}

<div class="container">
    <h1>Login Page</h1>

    <form action="" method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Login">
    </form>
</div>

{% endblock body %}
```

- We can use base template here also!
- Login action will send us to a profile page which does not exists. If you want you can send users to home page after successful login operation. Just create a variable on settings.py.

```
# Redirect to home URL after login (Default redirects to /accounts/profile/)
LOGIN_REDIRECT_URL = "/"  # or more elegant way
LOGIN_REDIRECT_URL = "home"
```

Register

- Django views enables us to login, logout, etc. But we must create our register view.

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm
from django.contrib import messages
from django.shortcuts import redirect
from django.contrib.auth import login, authenticate


def register(request):
    form = UserCreationForm()
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()

            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password1')
            user = authenticate(username=username, password=password)
            login(request, user)
```

```
            messages.success(request, 'Registered Successfully!!')
            return redirect('home')
    context = {
        'form': form,
    }
    return render(request, 'registration/register.html', context)
```

- Create register.html under templates/registration folder.

```
{% extends "pizzas/base.html" %}

{% block body %}

<div class="container">
    <h1>Registration page</h1>
    <form action="{% url 'register' %}" method="post">
        {% csrf_token %}

        {% if form.errors %}
            <p>There is something wrong what you entered!</p>
        {% endif %}

        {{ form.as_p }}

        <input type="submit" value="Register">
    </form>
</div>

{% endblock body %}
```

- Create the url path:

```
from .views import register


urlpatterns = [
    path('register/', register, name='register'),
]
```

- Test the endpoint.

Logout

- If we try to logout we see that we are on a some kind of django admin page. lets change this behaviour, simply adding url setting for logout.

```
LOGOUT_REDIRECT_URL = "home"
```

- Now it is time to update links on navbar, by adding login, logout, and register.

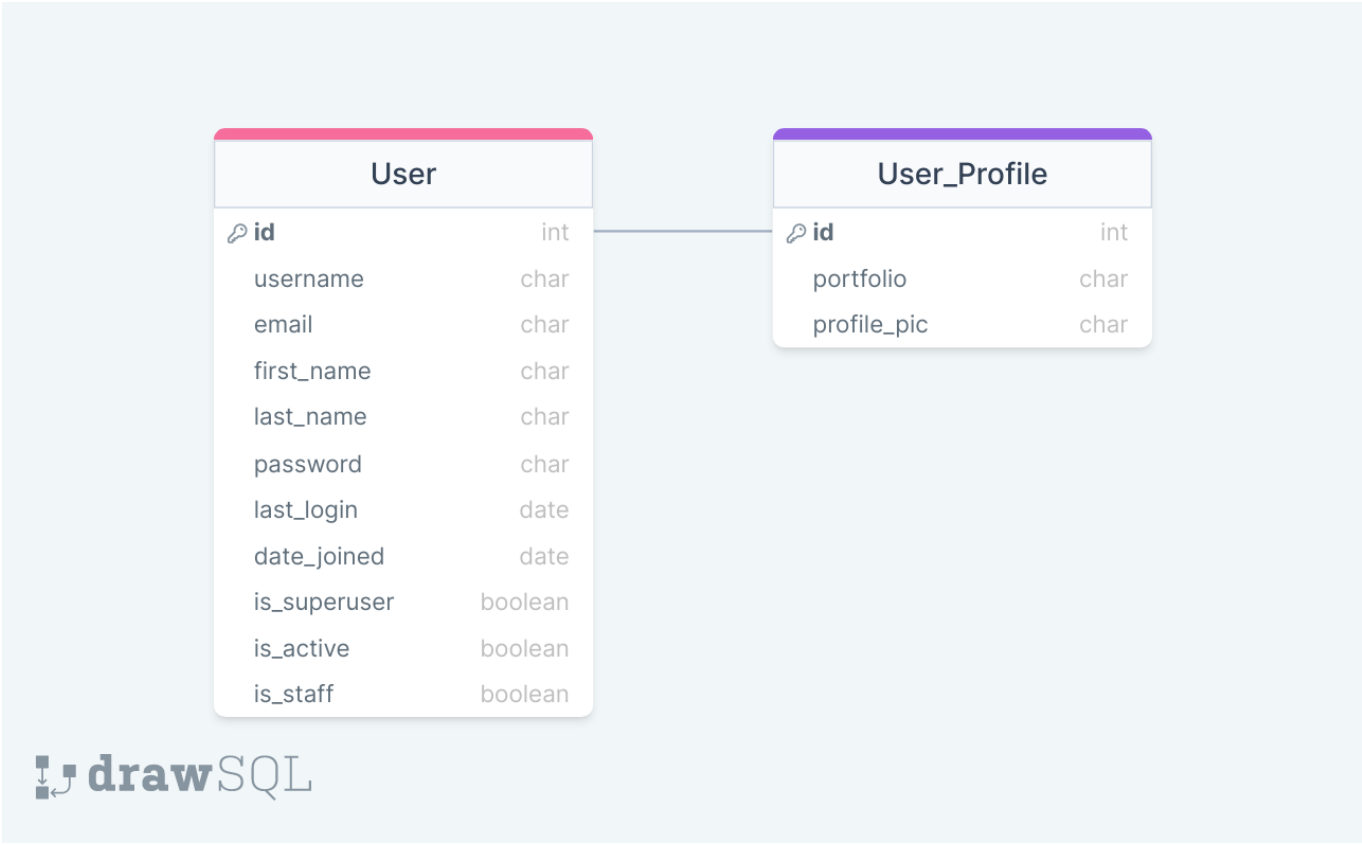
```
<nav class="navbar navbar-expand-lg navbar-dark" style="background-color:
#238a44;">
    <div class="container-fluid">
        <a class="navbar-brand" href="{% url 'home' %}">Nadia's Garden</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup" aria-
expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
            {% if user.is_authenticated %}
            <div class="navbar-nav">
               <a class="nav-link active" aria-current="page" href="{% url 'order'
%}">Order</a>
            </div>
            <div class="navbar-nav position-fixed end-0">
               <a class="nav-link active" aria-current="page" href="{% url
'logout' %}">Logout</a>
            </div>
            {% else %}
            <div class="navbar-nav">
               <a class="nav-link active" aria-current="page" href="{% url 'login'
%}">Login</a>
               <a class="nav-link active" aria-current="page" href="{% url
'register' %}">Register</a>
            </div>
            {% endif %}
        </div>
    </div>
</nav>
```

- And we can get rid of home page h1 tag and user.

Adding Profile (Extending the existing User model)

- If you wish to store information related to User, you can use a `OneToOneField` to a model containing the fields for additional information. This one-to-one model is often called a profile model, as it might store non-auth related information about a site user.

- For example:



- Lets create Profile model under users/models.py:

```
from django.db import models
from django.contrib.auth.models import User


class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    # file will be uploaded to MEDIA_ROOT/profile_pics
    profile_pic = models.ImageField(upload_to = 'profile_pics', blank=True)
    address = models.CharField(max_length=50)
    phone_number = models.CharField(max_length=15)
    address = models.CharField(max_length=200)

    def __str__(self):
        return self.user.username
```

- If you try to run server, you will get an error about pillow package. This is a python image package to work with images on your application.
- Install pillow and put dependencies to requirements.txt. Create new table and apply changes to the database:

```
python -m pip install Pillow
pip freeze > requirements.txt
py manage.py makemigrations
py manage.py migrate
```

- Register new model to users/admin.py and create some profiles.
- To see images on the web page, we need MEDIA_ROOT and MEDIA_URL, and adding media path to our url patterns. See the documentation.
- We can associate our user with the pizza order and we can now where to send the order, the number of the customer.

Register with new forms

- To create a register page, we need a form to display. We can use UserCreationForm for User model, and ModelForm for UserProfile.
- Create forms.py under users app.

```
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

from django.forms import ModelForm
from .models import UserProfile


class UserForm(UserCreationForm):
    class Meta:
        model = User
        fields = ('username', 'email')


class UserProfileForm(ModelForm):
    class Meta:
        model = UserProfile
        # fields = ('profile_pic', 'portfolio')
        exclude = ('user',)
```

- Write corresponding view:

```
from .forms import UserForm, UserProfileForm

def register(request):
    form_user = UserForm()
    form_profile = UserProfileForm()

    context = {
        'form_user': form_user,
        'form_profile': form_profile,
    }
    return render(request, 'users/register.html', context)
```

- Create users/register.html:

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

{% block body %}

{% load crispy_forms_tags %}

<h2>Registration Form</h2>

{% if request.user.is_authenticated %}

<h3>Thanks for registering</h3>

{% else %}

<h3>Fill out the form please!</h3>

<form action="" method="post" enctype="multipart/form-data">
  {% csrf_token %}
  {{ form_user | crispy }}
  {{ form_profile | crispy }}
  <button type="submit" class="btn btn-danger">Register</button>
</form>
{% endif %}

{% endblock body %}
```

- Install django-crispy-forms with the help of documentation.
- Create url:

```
from .views import register

path('register/', register, name='register'),
```

- Activate the link for register on navbar.
- Complete the view to save user!

```
def register(request):
    form_user = UserForm()
    form_profile = UserProfileForm()

    if request.method == 'POST':
        form_user = UserForm(request.POST)
        # uploaded files coming from request.FILES
        form_profile = UserProfileForm(request.POST, request.FILES)
```

```
        if form_user.is_valid() and form_profile.is_valid():
            # form_user.save()
            # form_profile.save()

            # we need to define user for profile before save
            # how to get the user?
            user = form_user.save()
            profile = form_profile.save(commit=False)
            # get the profile info without saving to db

            profile.user = user
            # now we know the user

            profile.save()

            return redirect('home')

    context = {
        'form_user': form_user,
        'form_profile': form_profile,
    }
    return render(request, 'users/register.html', context)
```

- Do not forget to login and add a message to the register view!

```
from django.contrib.auth import login

login(request, user)
messages.success(request, 'User Created!')
```

☺ **Thanks for Attending** ✍

Clarusway