Programacion web II

Tercer trabajo práctico

“ Blog ”

Alumno : Gonzalo sampini

Prof. y Lic. Julio César Casco

INTRODUCCIÓN

Este proyecto tiene como objetivo desarrollar un blog utilizando Django y el patron MTV (Modelo - View - Tempate). El proyecto cuenta con una aplicacion la cual permite utilizar todas las funcionalidades basicas permitiendo la creacion, edicion y eliminacion de publicaciones. En este proyecto se utiliza la autentificacion de usuarios restringiendo las funciones segun sea su estado (anonimo - autentificado).

Se integra el proyecto a Git Hub y se emplea un entorno virtual para manejar correctamente las dependencias.

Tambien se le extiende un fucnoinalidad atrvez de un middleware personalizado para registrar las solicitudes y las muestr por consola.

DESARROLLO

Requerimientos previos:

Entorno virtual(venv)

La creacion de un entorno virtual permite aislar un espacio dentro de nuestro sistema, en el cual podemos instalar paquetes y dependencias especificas para un proyecto ,en este caso nuestro blog, sin afectar configuraciones globales. Con esta herrmanieta nos aseguramos de que el gestionamiento de dependendencias fue echo de forma eficiente y no interferira con futuros proyectos.

Instalacion: (linux y windows)

Para la instalacion se utilizó venv, una herramienta integrada de python.

1 - ir al directorio del proyecto en la terminal:cd path/del/proyecto/blog.
“cd path/del/proyecto/blog”
2- Crear el entorno virtual usando:
“python -m venv nombre_entorno_virtual”

Una vez creado el entorno se debe activar

Linux: “env\Scripts\activate”
Windows: “source env/bin/activate”

Después de activar el entorno virtual, en la terminal aparece el nombre del entorno antes del prompt, indicando que el entorno está activo, por ejemplo: `(env)`.

Instalacion de dependencias:

Una vez instalado nuestro entorno virtual tenemos que proceder a instalar las dependencias. La principal sería Django para poder empezar a crear nuestro proyecto.

“pip install django”

Creación de requerimientos (requirements.txt)

Para facilitar la instalacion de dependencias en otro proyecto o por otros desarrolladores que quieran replicar el proyecto correctamente se genero un archivo requirements.txt el cual contiene la lista de dependencias necesarias y su version exacta

“pip freeze > requirements.txt”

Esto permite que la persona que clone el repositorio pueda instalar las mismas dependencias ejecutando

“pip install -r requirements.txt”

Por ultimo solo queda desocupar el entorno al finalizar a través de:

“deactivate”

Control de versiones
(Git Hub)

Para comenzar, se creó un repositorio remoto en GitHub, que servirá como el almacén central del proyecto.

Pasos a seguir

1- inicia sesion en GitHub.

2- Hacer clic en el botón **New** o + y seleccionar "New repository".

3- Llenar los detalles del repositorio

4- hacer clic en Create repository.

- inicializarlo en el Proyecto Local

1- Navegar hasta la carpeta del proyecto local en la terminal:

cd path/del/proyecto/"nombre-del proyecto"
en mi caso: cd home/gonzalo/djproyect

2- Inicializar git con el comando

init git
Esto crea un repositorio Git en el directorio local

- Vincularlo con repositorio Remoto

1- Copiar la URL del repositorio GitHub.
https://github.com/tu-usuario/blog-project.git

2- Vincular repositorio remoto
git remote add origin https://github.com/tu-usuario/blog-project.git

3- Verificar que todo este correcto

git remote -v

- Añadir, comentar y subir proyecto

1- añadir los archivos al area de preparacion

git add .

2- verificar si todo esta correcto

git status

3- Realizar el primer commit

git commit -m “primer comentario”

4. Realizar el primer push

git push origin main

MODELO

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

class Post(models.Model):
   titulo = models.TextField()
   contenido = models.TextField()
   fecha = models.DateTimeField(auto_now_add=True)
   autor = models.ForeignKey(User, on_delete=models.CASCADE)
```

- Autor: Este campo es una relación de tipo `ForeignKey` con el modelo `User`, lo que indica que cada publicación está asociada con un usuario (autor) registrado. Se utiliza `on_delete=models.CASCADE` para que, si se elimina un usuario, también se eliminen sus publicaciones.

- Título: Un campo de texto corto **(`CharField`)** para el título del post

- Contenido: Un campo de texto largo **(`TextField`)** donde se almacena el contenido principal de la publicación.

- Fecha: Este campo almacena la fecha y hora en que se creó la publicación, utilizando

auto_now_add=True para que se registre automáticamente cuando se crea el post.

FORMULARIO (FORM)

```
from django import forms
from .models import Post



class PostForm(forms.ModelForm):
   class Meta:
       model = Post
       fields = ['titulo', 'contenido']
```

Se personalizó un formulario basado en el Post utilizando ModelForm. El formulario incluye los campos de título y contenido. Si bien utilizo las funciones de django no es necesario definirlo pero para fines de practicar sirvió ya que lo aplique en el crear post basado en un formulario personalizado.

FUNCIONALIDADES

Crear publicacion:

Esta funcionalidad permite crear publicaciones a los usuarios autentificados . Esto se logra mediante un formulario que se renderiza a través de un metodo get el cual lo maneja django en el que los usuarios pueden ingresar el título y el contenido del post. Una vez enviado a través de un metodo post se guarda en la base de datos. Se renstringio la creación de post a sólo los usuarios autenticados, a través de la clase LoginRequiredMix.

urls.py

Los usuarios pueden entrar en esta funcionalidad a través de **/crear/** que está mapeada en urls.py,

path('crear/', CrearPost.as_view(), name='crear_post')

View

Se utilizó la vista basada en clases **CrearPost** la cual hereda de **CreateViews** simplificando la creacion de objetos en la base de datos.

```
class CrearPost(LoginRequiredMixin,CreateView):

   form_class = PostForm
   template_name = 'crear_post.html'
   success_url= reverse_lazy ('home')


   def form_valid(self, form):
       post = form.save(commit=False)
       post.autor = self.request.user  # Asigna el usuario autenticado
omo autor del post
       post.save()
       return super().form_valid(form)
```

Se utilizó un PostForm para capturar los datos del usuario, ademas, se sobrescribio el metodo form_valid para asignar al post el usuario con el que se ingresó al blog

Plantilla

Se creó la plantilla crear_post.html para mostrar el formulario al usuario

```
{% extends 'base.html' %}
{% block content %}

 <h2>Crear Nuevo Post</h2>
 <form method="POST">
```

```
    {% csrf_token %}

    {{ form.as_p }}  <!-- Muestra el formulario con los campos de título y
contenido -->

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

 </form>
 <a href="{% url 'home' %}">Volver a la lista de posts</a>  <!-- Enlace
opcional →


{% endblock %}
```

Listado de publicaciones

Esta funcionalidad permite listar todas las publicaciones del usuario autenticado atravez del login . Dentro del listado se puede editar y eliminar cualquier publicacion.

Urls.py

Los usuarios pueden entrar en esta funcionalidad a través de **/listar_post/** que está mapeada en urls.py,

path('listar/', ListarPost.as_view(), name='listar_post')

Views

Se utilizó una vista basada en clases que hereda de **(ListView)** la cual maneja la logica de acceso a la base de datos y brinda un contexto para mostrar los datos a los usuario

```
class ListarPost(ListView):

    model = Post
    context_object_name = 'posts'  # Nombre del contexto en la plantilla
```

```
    template_name = 'listar_post.html'  # Nombre de la plantilla
```

- Metodos

GET: La vista maneja automáticamente las solicitudes GET para listar las publicacion

para mostrar la lista al usuario se definio una plantill llamada listar_post.html

```
{% extends 'base.html' %}  <!-- Asegúrate de estar heredando de la
plantilla base -->

{% block header %}

<h1>Listado de Posts</h1>

{% endblock %}

{% block content %}

    <ul>
        {% for post in posts %}

            <li>
                <h2>{{ post.titulo }}</h2>
                <p><small>Publicado el {{ post.fecha }}</small></p>
                <a href="{% url 'eliminar_post' post.pk %}">Eliminar</a>
<               <a href="{% url 'detalle_post' post.pk %}">Detalle</a>
                <a href="{% url 'editar_post' post.pk %}">Editar</a>

            </li>

        {% empty %}

            <p>No hay posts disponibles.</p>

        {% endfor %}

    </ul>

{% endblock %}
```

Eliminar post

Esta funcionalidad permite eliminar los post del usuario autenticado . Esto se logra mediante un boton “eliminar” el cual envia la solicitud para su eliminación. Para ingresar a esta pantalla se deben listar primero los post los cuales tendran cada uno el boton “eliminar”.

Urls

Esta URL permite acceder a la vista para eliminar una publicación específica. El parámetro `<int:pk>` captura el identificador único de la publicación que se desea eliminar, y este valor se pasa a la vista `EliminarPost` para procesar la eliminación de la publicación.

path('listar/<int:pk>', ListarPost.as_view(), name='listar_post')

Views

La clase **`EliminarPost`** es una vista basada en clases que permite a los usuarios autenticados eliminar una publicación existente. Está basada en la clase genérica `DeleteView` de Django, que proporciona la funcionalidad necesaria para eliminar objetos de la base de datos.

```
class EliminarPost(LoginRequiredMixin, DeleteView):

   model = Post
   template_name ='eliminar_post.html'
  success_url = reverse_lazy('listar_post')
```

Plantilla

Se creó la plantilla de confirmacion de eliminación donde se muestra el titulo y el contenido del mensaje.

```
{% extends 'base.html' %}
{% block content %}

 <h2>Confirmar Eliminación</h2>
 <p>¿Estás seguro de que quieres eliminar el siguiente post?</p>
```

```
<div>

  <h3>{{ object.titulo }}</h3>
  <p>{{ object.contenido }}</p>

  <!-- Puedes añadir más detalles del objeto aquí si lo deseas →
</div>

<form method="post">

  {% csrf_token %}

  <button type="submit">Confirmar Eliminación</button>
  <a href="{% url 'listar_post' %}">Cancelar</a>

</form>

{% endblock %}
```

Detalle post

Esta funcionalidad permite ver el detalle del mensaje listado. Esto se logra dentro de la lista de mensajes mediante un boton “detalle” el cual envía la solicitud para recuperar el post de la lista permitiendo ver todo el mensaje creado por el usuario.

Urls

Para acceder a la vista de la lista de posts, se define una ruta en el archivo **`urls.py`** de la aplicación. Esto permite asociar una URL a la vista **`PostListView`.**

path('detalle/<int:pk>', DetallePost.as_view(), name='detalle_post')

Views

La clase **`DetallePost`** extiende de la vista genérica basada en clases **`DetailView`.** Esta clase permite obtener un objeto específico del modelo **`Post`** y renderizarlo en una plantilla.

```
class DetallePost(DetailView):
```

```
    model = Post
    template_name = 'detalle_post.html'
```

Plantilla

La plantilla **`detalle_post.html`** es responsable de mostrar los detalles de una publicación en el navegador. Presenta información relevante como el título, el contenido completo, el autor y la fecha de publicación.

```
{% extends 'base.html' %}  <!-- Asegúrate de estar heredando de la
plantilla base -->



{% block header %}

<h1>Listado de Posts</h1>

{% endblock %}
{% block content %}

                <h2>{{ object.titulo}}</h2>
                <p><small>{{ object.contenido }}</small></p>
                <p>{{ object.fecha }} </p>

{% endblock %}
```

Editar post

Esta sección describe la implementación de la funcionalidad que permite editar un post existente dentro del proyecto Django. Utilizamos la vista genérica **`UpdateView`** para permitir a los usuarios modificar los detalles de un post.

Urls

Para acceder a la vista de la lista de posts, se define una ruta en el archivo **`urls.py`** de la aplicación. Esto permite asociar una URL a la vista **`PostListView`.**

path('editar/<int:pk>', EditarPost.as_view(), name='editar _post')

Views

```
class EditarPost(LoginRequiredMixin,UpdateView):

   model = Post
   fields = ['contenido']
   template_name = 'editar_post.html'
   success_url = reverse_lazy('home')
```

Plantilla

La plantilla `editar_post.html` es responsable de mostrar un formulario prellenado con los detalles del post que se está editando. Los usuarios pueden modificar la información y guardar los cambios.

```
{% extends 'base.html' %}
{% block content %}
 <h1>Editar Post: {{ form.instance.titulo }}</h1>
```

```
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}  <!-- Genera los campos del formulario →
         <button type="submit">Guardar cambios</button>
    </form>

{% endblock %}
```

AUTENTIFICACION

login/logout

La autenticación de usuarios es una parte fundamental de cualquier aplicación web, ya que permite controlar el acceso a ciertas funcionalidades y proteger la información sensible. En este proyecto Django, se implementó un sistema de autenticación que permite a los usuarios registrarse, iniciar sesión y cerrar sesión.

Setting

Django proporciona un sistema de autenticación robusto que incluye modelos y vistas listas para usar. Para implementar la autenticación, se realizó la configuración básica en el archivo `settings.py` y se añadieron las URLs correspondientes.

```
LOGIN_REDIRECT_URL = '/'  # Redirige al home después de iniciar sesión
LOGIN_URL = 'login'  # URL para la vista de inicio de sesión
LOGOUT_REDIRECT_URL = '/'  # Redirige a la página principal o a cualquier
otra URL que prefieras
```

Plantillas

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>

   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>Login</title>
   {% load static %}

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


<body>

   <div class="login-container">
       <h2>Login</h2>
       <form method="post">
           {% csrf_token %}
           <div class="form-group">
               <label for="username">Username</label>
               <input type="text" id="username" name="username" required>

           </div>
           <div class="form-group">
               <label for="password">Password</label>
               <input type="password" id="password" name="password"
required>

           </div>
           <button type="submit">Login</button>
           <div class="registro" >
               <a href="{% url 'registro' %}" class="btn-home" >No tienes
cuenta? Registrate</a>
           </div>
           <div class="back-to-home" >
               <a href="{% url 'home' %}" class="btn-home" >Volver al
Inicio</a>
           </div>
```

```
        </form>
    </div>

</body>

</html>
```

MIDDLEWARE

Los middleware en Django son componentes que se ejecutan durante el procesamiento de las solicitudes y respuestas. Permiten agregar funcionalidades y modificar el comportamiento de la aplicación en varias etapas del ciclo de vida de una solicitud. En este proyecto, se implementaron middlewares personalizados y se utilizó el middleware de autenticación proporcionado por Django.

Setting

```
MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',AUTENTIFICAION
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'post.middleware.LogginMiddleware',PERSONALIZADO
```

`AuthenticationMiddleware`:

Este middleware se encarga de asociar cada solicitud a un usuario autenticado.

Proporciona el atributo `request.user`, que permite verificar si un usuario está autenticado y acceder a sus datos.

`LogginMiddleware`:

El middleware de logging es un componente que permite registrar información sobre las solicitudes HTTP que maneja la aplicación. Este tipo de middleware es útil para depurar y monitorear el comportamiento de la aplicación, así como para realizar un seguimiento de la actividad del usuario. A continuación, se presenta la implementación de un middleware de logging en el proyecto.

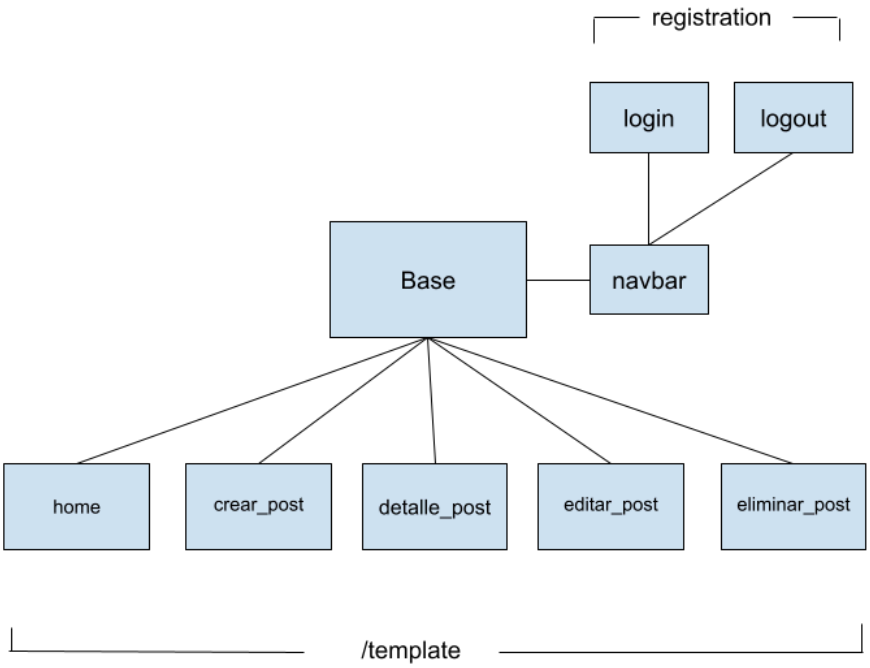
```
from typing import Any


class LogginMiddleware:


   def __init__ (self,get_response):
       self.get_response = get_response


   def __call__    (self, request):
       print (f"Request URL: {request.path} - User {request.user}
      - Method: {request.method} - Estos es middelware")
       response = self.get_response(request)
       print(f"response: {response.status_code}")
       return (response)
```

HERENCIA Y REUTILIZACION DE PLANTIL



DIFICULTADES:

Las dificultades que se presentaron están relacionadas con la sintaxis rigurosa que tiene python ya que por algun espacio te puede causar error en la ejecucion lo que cuesta encontrar el error. Siguiendo los tutoriales creados en la asignatura logre completar la etapa del login y los middleware. luego con la parte de creacion, eliminación, edicion y detalle se tuvo que recurrir a otras fuentes como por ejemplo youtube y chat gtp.

CONCLUSION:

REFERENCIAS:

Casco, Julio. (2024). *Plantillas django* [Apunte de clase]. Universidad del chubut.

Casco, Julio. (2024). *Concepto y uso de middleware en Django* [Apunte de clase]. Universidad del chubut.

Casco, Julio. (2024).Actividad integradora III [encuentro virtual]. Universidad del chubut.

https://chatgpt.com/ (dudas y sintaxis ademas)