Anexo V - Manual del programador

Trabajo de Fin de Grado

Ingeniería Informática

VNiVERSiDAD D SALAMANCA

julio de 2023

Autor:

Lidia Alaejos Herrero

Tutores:

Alicia García Holgado

Andrea Vázquez Ingelmo

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

1.-INTRODUCCIÓN

En el presente anexo, se presenta el Manual del Programador para el desarrollo de una plataforma web destinada a la gestión de centros de Plena Inclusión. Este manual ha sido elaborado con el objetivo de proporcionar una guía exhaustiva y detallada para el desarrollo y mantenimiento de la mencionada plataforma web.

El desarrollo de esta plataforma se ha llevado a cabo utilizando Django como framework de backend, PostgreSQL como sistema de gestión de bases de datos y Bootstrap como librería de frontend. Estas tecnologías han sido seleccionadas por su capacidad para ofrecer una solución robusta, escalable y segura, necesaria para el ámbito de gestión de centros de Plena Inclusión.

El contenido del manual será el siguiente:

- Configuración del entorno de desarrollo
    - Instalación y configuración de Django, PostgreSQL y Bootstrap
    - Configuración del proyecto y la aplicación principal
    - Configuración de las dependencias y entorno virtual
- Estructura del proyecto Django
    - Organización de directorios y archivos relevantes
- Desarrollo de modelos con PostgreSQL
    - Diseño y creación de modelos de datos relacionales
    - Utilización de migraciones para mantener la consistencia de la base de datos
- Implementación de vistas y plantillas
    - Creación de vistas para gestionar la lógica de la plataforma
    - Utilización de plantillas HTML con Bootstrap para la presentación de la interfaz
- Configuración de rutas y URLs
    - Configuración de las URL para el enrutamiento adecuado de la plataforma
    - Implementación de rutas dinámicas y parámetros
- Seguridad y autenticación
    - Gestión de usuarios y roles dentro de la plataforma
    - Implementación de medidas de seguridad para proteger la información sensible

- Despliegue
    - Preparación para el despliegue en un entorno de producción

Este manual del programador ha sido creado con el propósito de brindar una referencia sólida para el desarrollo y mantenimiento de la plataforma web. Con el objetivo de complementar este manual, se recomienda consultar las guías con la documentación oficial de Django [1], PostgreSQL [2] y Bootstrap [3].

2.-CONFIGURACIÓN DEL ENTORNO DE DESARROLLO

Para poder comenzar a desarrollar la web es necesario configurar correctamente el entorno de desarrollo. A continuación, se detallan los pasos para la instalación y configuración de Django, PostgreSQL y la integración de Bootstrap en el código para el diseño del frontend, así como la configuración de las dependencias y el entorno virtual. Puntualizar que se ha realizado el proyecto en el IDE PyCharm [4].

2.1.-INSTALACIÓN Y CONFIGURACIÓN DE DJANGO

2.1.1.-INSTALACIÓN DE PYTHON

Utilizamos Django como *framework*, el cual, está escrito en Python, hay que instalar Python en el sistema. Para ello, recurrimos al sitio oficial y seguimos las correspondientes instrucciones de instalación [5].

2.1.2.-CREACIÓN ENTORNO VIRTUAL

Para el desarrollo de nuestro proyecto decidimos utilizar un entorno virtual, el cual nos permitirá aislar las dependencias y configuraciones. Mediante la herramienta **'venv'** de Python lo creamos en la ubicación deseada, ejecutamos como comando 'python3 -m venv ubicación'.

2.1.3.-ACTIVACIÓN DEL ENTORNO VIRTUAL

Tras crear el entorno virtual, debemos activarlo, ejecutando el comando '**.\activate**' en la ruta 'ubicacionEV\Scripts'.

2.1.4.-INSTALACIÓN DJANGO

Una vez que tenemos el entorno virtual activado, instalamos Django con el gestor de paquetes de Python '**pip**', ejecutando el comando: 'pip install django'.

2.2.-CREACIÓN PROYECTO DJANGO

Tras instalar Django, creamos el proyecto con el comando 'django-admin startproject nombreProyecto'. Tras la creación de este, observamos un fichero llamado 'manage.py', que actúa como interfaz de línea de comandos para ejecutar diversos comandos relacionados con la administración y gestión del proyecto Django y establece la configuración del entorno para el proyecto. Importa y configura el módulo settings.py, que contiene la configuración global de la aplicación, como la base de datos, las rutas URL, las claves secretas, etc.

A continuación, creamos la aplicación mediante 'python manage.py startapp nombreAplicacion'. La añadimos al fichero settings.py en INSTALLED_APPS

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'webApp',
    'webApp.templatetags'
```

Ahora ya tenemos todo preparado para continuar con la integración de la base de datos y desarrollar la web.

2.3.-CONFIGURACIÓN DE POSTGRESQL

2.3.1.-INSTALACIÓN DE POSTGRESQL

Descargamos e instalamos PostgreSQL desde el sitio oficial [7] y seguimos las instrucciones que nos especifica paso a paso.

2.3.2.-CONFIGURACIÓN BASE DE DATOS

Tras descargar e instalar Postgres, accedemos a la aplicación PgAdmin 4 para crear la base de datos con el nombre que posteriormente integramos en nuestra aplicación.

Para trabajar en la base de datos creada, abrimos el archivo models.py y definimos las clases que tendrá nuestra base de datos, posteriormente realizamos la configuración necesaria en el archivo settings.py en DATABASES, e instalamos la librería para conectar el proyecto con la BDD con 'pip install psycopg2'

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'madrigal',
        'USER': 'postgres',
        'PASSWORD': 'lidiaalaejos',
        'HOST': 'localhost',
        'DATABASE_PORT': '5432',
    }
}
```

FIGURA 1.CÓDIGO SMTP

Para guardar todo y configurar nuestros modelos en la BDD realizamos 'python manage.py makemigrations' para crear las migraciones, y posteriormente, para pasarlas a la BDD de Postgres, ejecutamos 'python manage.py migrate'.

2.4.-CONFIGURACIÓN DE BOOTSTRAP Y DEPENDENCIAS

En nuestro proyecto, para la integración de Bootstrap para el diseño de la web, hemos aprovechado los archivos CSS y JavaScript proporcionados por un CDN (*Content Delivery Network*) en lugar de tener que descargarlos e incluirlos localmente, lo cual es mucho más cómodo y nos permite acceder a las últimas versiones y actualizaciones de Bootstrap. Esto se hace mediante el enlazado a los archivos CSS y JS en la sección <head> de nuestras plantillas HTML. Algún ejemplo de nuestra plantilla base.html donde guardamos todos los links y scripts que utilizamos:

```
{% load static %}
<!-- Bootstrap CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css" rel="stylesheet"
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
<link rel="stylesheet" href="https://cdn.datatables.net/1.10.25/css/jquery.dataTables.min.css">
<link rel="stylesheet" href="https://unpkg.com/swiper/swiper-bundle.min.css">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.2/css/bootstrap.min
<link  href="{% static 'webApp/css/styleBlog.css' %}" rel="stylesheet" >
<!-- Calendario links-->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet">
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/fullcalendar/3.9.0/fullcalendar.css" />

<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/
<script src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/popper.min.js" integrity="sha384-ApNbgh9B
<script src="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/js/bootstrap.min.js" integrity="sha384-JZR6Spe
<script src="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.9.0/js/bootstrap-datepicker.min.js
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js" integrity="
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.2/js/bootstrap.min.js"></script>
<script src="https://unpkg.com/swiper/swiper-bundle.min.js"></script>

<!-- jQuery -->
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

<!-- DataTables JS -->
<script src="https://cdn.datatables.net/1.10.25/js/jquery.dataTables.min.js"></script>

<!-- Calendario scripts-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.1/moment.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/fullcalendar/3.10.2/fullcalendar.min.js"></script>

</head>
```

FIGURA 2. CÓDIGO LINKS Y SCRIPTS

Además, para la configuración de nuestras plantillas, en el archivo 'settings.py' del proyecto, agregamos las rutas correspondientes al directorio donde se encuentran nuestros archivos CSS y donde almacenamos los archivos multimedia.

```
STATIC_URL = '/webApp/static/'
STATIC_ROOT = os.path.join(BASE_DIR, '/webApp/static/')

MEDIA_URL = '/webApp/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'webApp/media/')
```

FIGURA 3.CÓDIGO CONFIG PLANTILLAS

3.-ESTRUCTURA DEL PROYECTO DJANGO

En esta sección, se describe la estructura organizativa del proyecto, la cual nos proporciona una base para la organización del código fuente y los recursos del proyecto. A continuación, se detallará la organización de directorios y ficheros, así como la configuración del proyecto y aplicación principal (la cual vimos en el apartado anterior, que llamamos 'webApp').

3.1.-ORGANIZACIÓN DE DIRECTORIOS Y FICHEROS

El proyecto sigue una estructura de directorios bien definida que nos ayuda a dar una buena modularidad y en la reutilización del código. A continuación, se describen los directorios y archivos más relevantes en esta estructura:

- Directorio del proyecto (proyectoTFG). Contiene los archivos y configuraciones principales del proyecto.
    - 'manage.py': archivo de gestión de comandos de Django. Se utiliza para ejecutar diversas tareas de administración, como iniciar el servidor de desarrollo y ejecutar migraciones de base de datos.

```python
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys


lidiaAH9
def main():
    """Run administrative tasks."""
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'proyectoTFG.settings')
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)


if __name__ == '__main__':
    main()
```

FIGURA 4. CÓDIGO 'MANAGE.PY'

- ‘settings.py’: archivo de configuración principal del proyecto. Se definen las configuraciones de la base de datos (como vimos en el apartado anterior 2.3.2.-CONFIGURACIÓN BASE DE DATOS) el enrutamiento de URL, las configuraciones de seguridad y otras configuraciones relacionadas con el proyecto, como la configuración SMTP para mandar correos.

```
#SMTP Config
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_USE_SSL = False
EMAIL_HOST_USER = 'lidia.alaejos@gmail.com'
EMAIL_HOST_PASSWORD = 'envjsmygezezirxw'
```

FIGURA 5.CÓDIGO CONFIG BDD

- ‘urls.py’: archivo de enrutamiento principal del proyecto. Aquí se definen las URL del proyecto, en nuestro caso lo hemos asociado a archivo ‘urls.py’ de la aplicación creada ‘webApp’.

```
urlpatterns = [
    path('', include('webApp.urls')),
    path('admin/', admin.site.urls),
    path('', include('django.contrib.auth.urls')),
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

FIGURA 6.CÓDIGO ‘URLS.PY’

- Aplicaciones. Django utiliza una estructura modular basada en aplicaciones. En nuestro caso creímos oportuno definir todas las funcionalidades en una misma aplicación, la cual llamamos ‘webApp’. Alguno de los directorios y archivos asociados a las aplicaciones son:
  - ‘models.py’: define los modelos de datos utilizados por la aplicación. Aquí se especifica la estructura y las relaciones de las tablas de la base de datos.

```
5 usages  lidiaAH9
class Protocolo(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False, unique=True)
    archivo_multimedia = models.FileField(upload_to='media/%Y/%m/%d/%H%M%S/')
    descripcion = models.TextField()
    fecha = models.DateField()
    version = models.CharField(max_length=20)
    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE)
```

FIGURA 7.CÓDIGO 'MODELS.PY'

- 'views.py': contiene las vistas (funciones o clases) que definen la lógica de procesamiento de las solicitudes del cliente.

```
@login_required(login_url='/login/')
def protocolos(request):
    if request.method == 'POST':
        form = CrearProtocoloForm(request.POST, request.FILES)
        print(request.FILES)

        if form.is_valid():
            protocolo = form.save(commit=False)
            protocolo.usuario = request.user
            archivo = form.cleaned_data['archivo_multimedia']
            protocolo.nombre_archivo = os.path.basename(archivo.name)
            protocolo.save()
            return redirect('/protocolos/protocolos/')
        else:
            print(form.errors)
    else:

        form = CrearProtocoloForm()

    context = {}

    protocolos = Protocolo.objects.all()
    protocolos = sorted(protocolos, key=lambda c: (c.fecha))

    context['protocolos'] = protocolos

    paginator = Paginator(protocolos, 8)

    page_number = request.GET.get('page')

    page_obj = paginator.get_page(page_number)
    context['page_obj'] = page_obj

    return render(request, "protocolos/protocolos.html", context)
```

FIGURA 8.CÓDIGO 'VIEWS.PY'

- ‘templates’: directorio que contiene las plantillas HTML utilizadas por la aplicación para generar las páginas web dinámicas. En nuestro caso a su vez recogimos en otros directorios cada plantilla según el apartado correspondiente de la web.

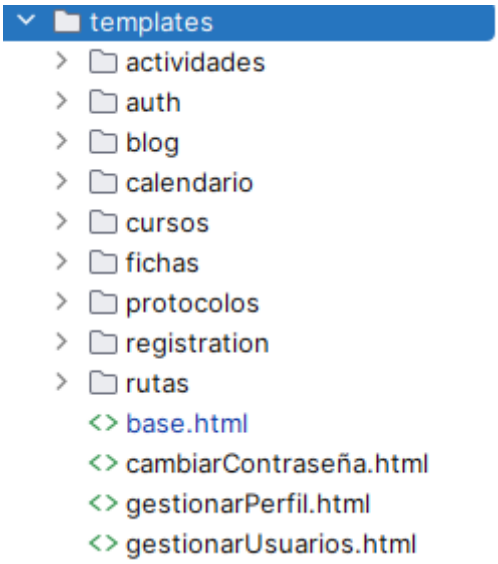

FIGURA 9.ESTRUCTURA DIRECTORIO TEMPLATES

- ‘static’: directorio que almacena los archivos estáticos (CSS, JavaScript, imágenes, etc.) utilizados por la aplicación.

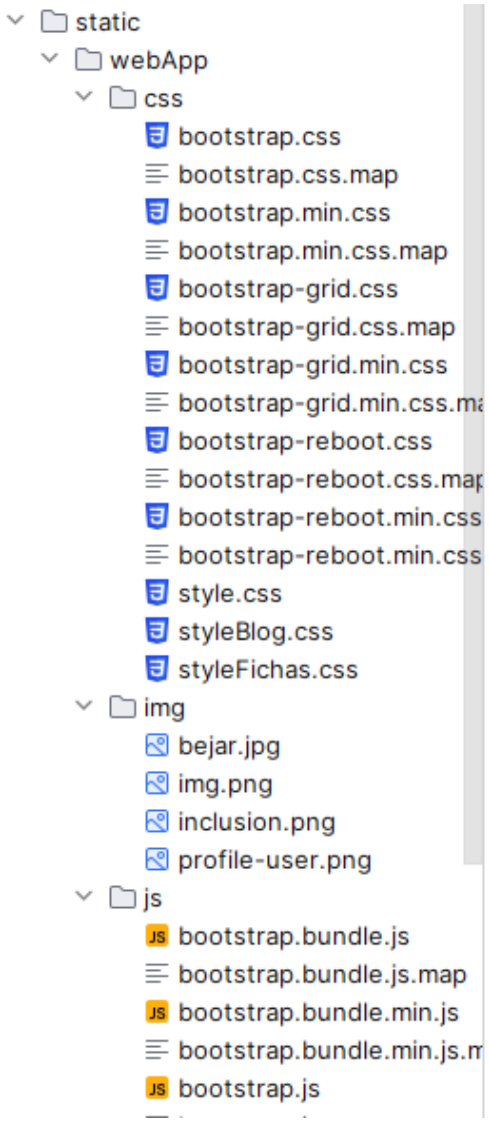

FIGURA 10.ESTRUCTURA DIRECTORIO STATIC

- ‘templatetags’: en nuestro caso definimos este directorio para poder utilizar el método ‘starstwith’ creando y configurando el archivo ‘startswith.py’ [8]

views.py | startswith.py | base.html

```
from django import template

register = template.Library()


1 usage (1 dynamic)  lidiaAH9
@register.filter('startswith')
def startswith(text, starts):
    if isinstance(text, str):
        return text.startswith(starts)
    return False
```

FIGURA 11.CÓDIGO STARTSWITH

- 'forms.py': archivo que se utiliza para definir los formularios personalizados, que nos permiten recolectar los datos ingresados por el usuario y validarlos antes de ser procesados.

```
class CrearActividadForm(forms.ModelForm):
    lidiaAH9
    class Meta:
        model = Actividad
        fields = ('imagen', 'nombre', 'descripcion', 'fecha', 'lugar', 'lugar_salida', 'hora_salida',
                  'hora_llegada',   'duracion', 'aforo', 'plazas_disponibles')
        widgets = {
            'fecha': forms.DateInput(attrs={'type': 'date'}),
        }
```

FIGURA 12.CÓDIGO 'FORMS.PY'

Observamos el esqueleto completo del sitio web:

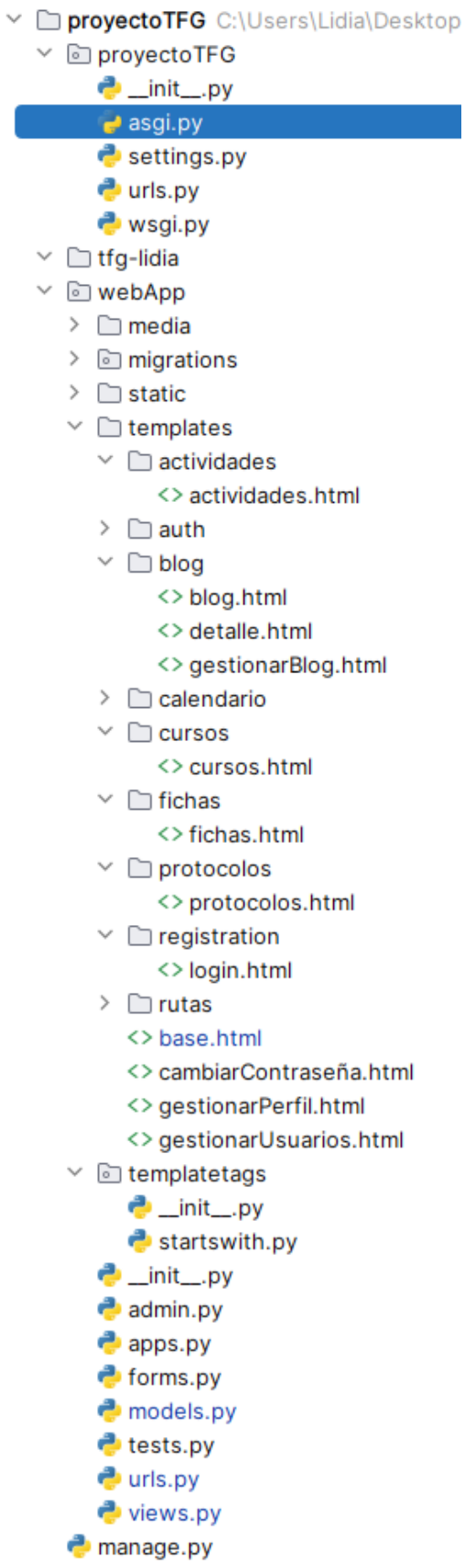

FIGURA 13.ESQUELETO PROYECTO

4.-DESARROLLO DE MODELOS CON POSTGRESQL

El desarrollo de modelos de datos es una parte fundamental del diseño de una aplicación web. En Django, los modelos de datos se definen utilizando clases Python que heredan de la clase 'django.db.models.Model'. Estas clases representan las tablas de la base de datos y definen los campos y las relaciones entre los datos. Para utilizar PostgreSQL como base de datos, se requiere configurar Django para que se conecte a la base de datos PostgreSQL y utilice sus características avanzadas. A continuación, se describen los pasos clave para el desarrollo de modelos con PostgreSQL:

4.1.-CONFIGURACIÓN BASE DE DATOS Y CREACIÓN DE MODELOS

4.1.1.-CONFIGURACIÓN DE LA BASE DE DATOS

Abrimos el archivo 'settings.py' del proyecto y en la sección 'DATABASES' configuramos los parámetros necesarios para conectarnos a la base de datos que creamos en la aplicación de PgAdmin 4.

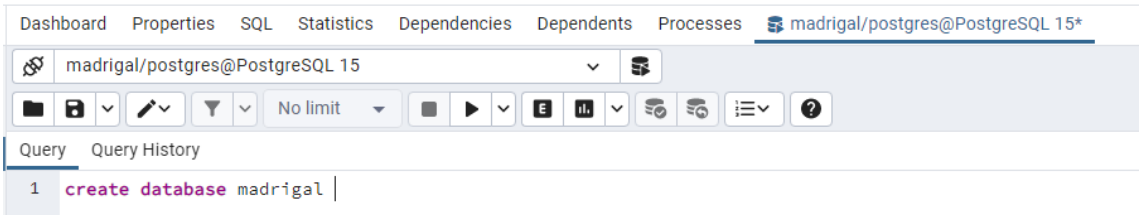

FIGURA 14.CONFIG PGAMDIN4

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'madrigal',
        'USER': 'postgres',
        'PASSWORD': 'lidiaalaejos',
        'HOST': 'localhost',
        'DATABASE_PORT': '5432',
    }
}
```

FIGURA 15.CÓDIGO BDD

Instalamos mediante el comando ‘pip install psycopg2’ la librería para poder aplicar la configuración añadida en ‘DATABASES’.

4.1.2.-DISEÑO Y CREACIÓN DE MODELOS

En el archivo ‘models.py’ de nuestra aplicación ‘webApp’, definimos los modelos de datos que utilizaremos para guardar todos los datos y archivos que usemos a lo largo del funcionamiento de nuestra página web. Para asignar el tipo de dato de cada clase podemos utilizar las clases proporcionadas por Django como ‘models.CharField’, ‘models.IntegerField’, ‘models.ForeignKey’ y así establecer también relaciones foráneas o primarias entre ellas.

```
9 usages  lidiaAH9
class Ruta(models.Model):
    nombre = models.CharField(max_length=20)
    fecha = models.DateField()
    conductor = models.ForeignKey(Usuario, on_delete=models.CASCADE)



4 usages  lidiaAH9
class Parada(models.Model):
    ruta = models.ForeignKey(Ruta, on_delete=models.CASCADE, related_name='paradas')
    nombre = models.CharField(max_length=20)
    direccion = models.CharField(max_length=40)
    hora_recogida = models.TimeField()
    personas = models.JSONField(default=list)
```

FIGURA 16.CÓDIGO MODELOS

En este ejemplo de nuestro proyecto vemos dos modelos: ‘Ruta’ y ‘Parada’, el modelo ‘Parada’ establece una relación de muchos a uno con el modelo ‘Ruta’ utilizando la clase ‘ForeignKey’, ya que en nuestro caso cuando creemos una ruta, esta puede tener muchas paradas asociadas.

4.2.-UTILIZACIÓN DE MIGRACIONES PARA MANTENER LA CONSISTENCIA DE LA BASE DE DATOS

4.2.1.-REALIZACIÓN DE MIGRACIONES

Tras definir los modelos, debemos generar sus correspondientes migraciones. . Las migraciones son archivos generados automáticamente por Django que describen los cambios en la estructura de la base de datos. Para generar las migraciones, utiliza el comando ‘python manage.py makemigrations’.

4.2.2.-APLICACIÓN DE MIGRACIONES

Después de generar las migraciones, las aplicamos a la base de datos para que los cambios en los modelos se reflejen en la estructura de la base de datos. Esto lo hacemos mediante el comando ‘python manage.py migrate’. Podemos comprobar en la aplicación PgAdmin 4 si se han creado las tablas tras ejecutar este comando.

5.-IMPLEMENTACIÓN DE VISTAS Y PLANTILLAS

Las vistas y las plantillas desempeñan un papel fundamental en el desarrollo de una plataforma web. Las vistas se encargan de gestionar la lógica de la aplicación y proporcionar respuestas a las solicitudes de los usuarios, mientras que las plantillas se utilizan para presentar la interfaz de usuario de manera visualmente atractiva y estructurada. A continuación, se describen los pasos y algunos ejemplos de nuestra implementación de vistas y plantillas en nuestro proyecto.

5.1.-CREACIÓN DE VISTAS

Las vistas en Django son funciones o clases que procesan las solicitudes HTTP y devuelven las respuestas correspondientes. Estas las implementamos en el archivo views.py de la aplicación ‘webApp’. En estas vistas realizamos diversas tareas, como mostrar información, procesar formularios o interactuar con los modelos de datos. A continuación, se muestra un ejemplo de unas vistas que crean cursos con los datos recogidos de un formulario de nuestra plantilla ‘curso.html’ y también de cómo inscribirse en él:

```python
@login_required(login_url='/login/')
def cursos(request):
    if request.method == 'POST':
            form = CrearCursoFormNoEvento(request.POST, request.FILES)

            if form.is_valid():
                curso = form.save(commit=False)
                curso.usuario = request.user
                curso.save()
                return redirect('/cursos/cursos/')
            else:
                print(form.errors)
    else:

            form = CrearCursoFormNoEvento()

    context = {}
    cursos = Curso.objects.all()

    paginator = Paginator(cursos, 6)
    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)
    context['page_obj'] = page_obj

    cursos = sorted(cursos, key=lambda c: (c.inscritos.all().filter(pk=request.user.pk).exists(), c.hora))
    context['cursos'] = cursos
    return render(request, "cursos/cursos.html", context)

1 usage  new *
@login_required(login_url='/login/')
def inscribirse_curso(request, curso_id):
  curso = Curso.objects.get(id=curso_id)
  EventoCalendario.objects.create( nombre=curso.nombre, fecha=curso.hora, curso=curso, usuario=request.user)
  curso.inscribir_usuario(request.user)
  return redirect('/cursos/cursos/')
```

FIGURA 17.CÓDIGO VISTAS

- ‘def cursos’: esta vista maneja tanto las solicitudes GET como POST. En una solicitud POST, se verifica si el formulario CrearCursoFormNoEvento es válido y se guarda el nuevo curso en la base de datos. A continuación, se redirige al usuario a la página de cursos. Si el formulario no es válido, se imprimen los errores del formulario en la consola. En una solicitud GET, se crea una instancia del formulario vacío CrearCursoFormNoEvento para mostrar el formulario en la página. Además, se obtienen todos los cursos existentes y se realiza una paginación de los resultados utilizando la clase Paginator. Luego, se ordenan los cursos primero por si el usuario está inscrito en ellos y luego por la hora del curso. Finalmente, se pasa el contexto con el formulario, la paginación y los cursos a la plantilla cursos/cursos.html.
- ‘def inscribirse_curso’: esta vista se encarga de inscribir al usuario actual a un curso existente. Se obtiene el curso con el ID proporcionado y se crea un nuevo evento en EventoCalendario con los detalles del curso y el usuario

actual. Luego, se invoca el método inscribir_usuario del curso para registrar al usuario como inscrito. Finalmente, se redirige al usuario de vuelta a la página de cursos.

Estas vistas hacen uso de decoradores como @login_required para asegurar que el usuario esté autenticado antes de acceder a ellas.

5.2.-UTILIZACIÓN DE PLANTILLAS HTML CON BOOTSTRAP

Las plantillas HTML se utilizan para definir la estructura y la presentación visual de las páginas web. Como explicamos previamente, en nuestro caso los archivos HTML los hemos guardado en el directorio templates. Además, puedes mediante la integración de Bootstrap hemos aplicado estilos y componentes predefinidos. A continuación, se muestra un ejemplo de la plantilla 'cursos.html' que comentábamos en el apartado inmediatamente anterior:

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

{% block content %}
{% if user.is_superuser or user.user_role == 1 %}
      <div class="d-flex justify-content-end px-4 ">
        <div class=" mb-3 ml-auto">
              <button type="button" class="btn btn-primary mb-3" data-toggle="modal" data-target="#uploadModal" style="background-color: darkblue;">AÑADIR CURSO</button>
          </div>
      </div>
{% endif %}

  <div class="container mt-0">
      <h1>Cursos ofertados</h1>

      <div class="row p-4">
          {% for curso in cursos %}
          <div class="col-lg-4 col-md-6 col-sm-12 mb-4">
              <div class="card h-100">
                  <img class="card-img-top" src="{{ curso.imagen.url }}" style=" height: 350px; width:100%; object-fit: cover;"  alt="">
                  <div class="card-body">
                      <h5 class="card-title">{{ curso.nombre }}</h5>
                      <p class="card-text"><strong>Lugar de realización:</strong> {{ curso.lugar }}</p>
                      <p class="card-text"><strong>Fecha de realización:</strong> {{ curso.hora }}</p>
                      <p class="card-text"><strong>Duración:</strong> {{ curso.duracion }}</p>
                      <p class="card-text"><strong>Aforo:</strong> {{ curso.aforo }}</p>
                      <p class="card-text"><strong>Plazas disponibles:</strong> {{ curso.plazas_disponibles }}</p>

                      {% if curso.plazas_disponibles == 0 %}
                                  <script>
                                          document.getElementById("btnInscribirse{{ curso.id }}").style.display = "none";
                                  </script>
                                  <div class="mt-3 border-top">
                                      <p class="mt-3"><strong><span style="...">No hay plazas disponibles para este curso</span></strong></p>
                                  </div>
                          {% endif %}
```

FIGURA 18.CÓDIGO HTML BOOTSTRAP I

En este ejemplo de plantilla, se extiende de ‘base.htm’l, lo que significa que hereda la estructura y el contenido de esa plantilla, en la cual tenemos los links y scripts referentes a Bootstrap y la cabecera predeterminado que utilizamos en toda la web. Dentro del bloque de contenido específico ({% block content %}), se muestra un botón "AÑADIR CURSO" solo si el usuario actual es un superusuario o tiene un rol específico.

Luego, se muestra una sección con el título "Cursos ofertados" y se itera sobre la lista de cursos utilizando el bucle {% for curso in cursos %}. Dentro de cada iteración, se muestra la información del curso, como el nombre, el lugar de realización, la fecha, la duración, el aforo y las plazas disponibles, los cuales recogemos de la base de datos tras haberlos guardados mediante el form que mandamos a la vista.

```
<div class= modal-body >
    <form method='POST' id="uploadForm" action="/cursos/cursos/"  enctype="multipart/form-data">
        {% csrf_token %}
        <div class="form-group">
            <label for="fileUpload">Imagen:</label>
            <input type="file" class="form-control-file" id="fileUpload" name="imagen" accept=".jpg,.jpeg,.png,.gif"  required>
        </div>
        <div class="form-group">
            <label for="fileTitulo">Título:</label>
            <input type="text" class="form-control" id="fileTitulo" name="nombre" required>
        </div>
        <div class="form-group">
            <label for="fileDescripcion">Descripción:</label>
            <input type="text" class="form-control" id="fileDescripcion" name="descripcion" required>
        </div>
        <div class="form-group">
            <label for="uploadDateTime">Fecha:</label>
            <input type="datetime-local" class="form-control" id="uploadDateTime" name="hora"  required>
        </div>
        <div class="form-group">
            <label for="fileLugar">Lugar:</label>
            <input type="text" class="form-control" id="fileLugar" name="lugar" required>
        </div>
        <div class="form-group">
            <label for="fileDuracion">Duración:</label>
            <input type="text" class="form-control" id="fileDuracion" name="duracion" required>
        </div>
        <div class="form-group">
            <label for="fileAforo">Aforo:</label>
            <input type="number" class="form-control" id="fileAforo" name="aforo" required>
        </div>
        <div class="form-group">
            <label for="filePlazas">Plazas disponibles:</label>
            <input type="number" class="form-control" id="filePlazas" name="plazas_disponibles" required>
            <p><span  style="...">Recuerde que al registrar un curso las plazas disponibles deben ser iguales al aforo.</span></p>
        </div>
        <div class="modal-footer">
            <button type="submit" class="btn btn-primary" id="uploadFileBtn">Guardar</button>
        </div>
    </form>
```

FIGURA 19.CÓDIGO HTML BOOTSTRAP II

Además, se incluye una condición {% if curso.plazas_disponibles == 0 %} para verificar si no hay plazas disponibles para el curso. En ese caso, se oculta el botón de

inscripción mediante el uso de un script JavaScript y se muestra un mensaje indicando que no hay plazas disponibles.

```
<script>
    var form = document.getElementById('uploadForm');
    form.addEventListener('submit', function(event) {
        var aforo = parseInt(document.getElementById('fileAforo').value);
        var plazas = parseInt(document.getElementById('filePlazas').value);
        if (plazas > aforo) {
            event.preventDefault();
            alert('El número de plazas disponibles no puede ser mayor que el aforo.');
        }
    });
</script>
```

FIGURA 20.CÓDIGO JAVASCRIPT

6.-CONFIGURACIÓN DE RUTAS Y URLS

Django utiliza el archivo 'urls.py' para gestionar las rutas y redirigir las solicitudes a las vistas correspondientes. En este, debemos importar la función 'path' para definir las rutas de URL y asociarlas a las vistas y definimos una lista 'urlspatterns' donde definiremos todas las rutas de URL.

6.1.-CONFIGURACIÓN URL PARA EL ENRUTAMIENTO

En el archivo mencionado, cada ruta mediante la función path contendrá tres elementos clave: la ruta relativa, la vista asociada y el nombre de la ruta para hacer referencia a la ruta en cualquier parte del código. Lo vemos con las URL correspondientes de cursos por seguir el mismo ejemplo visto de los apartados anteriores.

```python
from django.urls import path
from django.contrib.auth.views import LogoutView
from .views import CustomLoginView
from webApp import views
from django.contrib.auth import views as auth_views

urlpatterns = [

    path('cursos/cursos/', views.cursos, name='cursos'),
    path('inscribirse_curso/<uuid:curso_id>/', views.inscribirse_curso, name='inscribirse_curso'),
```

FIGURA 21.CÓDIGO URL ENRUTAMIENTO

6.2.-IMPLEMENTACIÓN DE RUTAS DINÁMICAS Y PARÁMETROS

Las rutas dinámicas en Django, permiten manejar URLs con parámetros variables. Estos parámetros se definen dentro de los patrones de URL mediante la sintaxis '<tipo:nombre>'. [9]

En la imagen anterior vemos un ejemplo de este tipo de rutas.

```python
path('inscribirse_curso/<uuid:curso_id>/', views.inscribirse_curso, name='inscribirse_curso'),
```

FIGURA 22.CÓDIGO RUTAS DINÁMICAS

En este caso ‘<int:curso_id>’ captura un entero que será el utilizado como parámetro en la vista y se lo pasa como argumento, de forma que nos ayuda a identificar cada curso creado para poder inscribirse en cada uno.

7.-SEGURIDAD Y AUTENTICACIÓN

La seguridad y autenticación son aspectos fundamentales en cualquier plataforma web. En este apartado, abordaremos las medidas implementadas para proteger la información sensible y garantizar la identidad de los usuarios. Exploraremos la gestión de usuarios y roles, así como la implementación de medidas de seguridad para prevenir posibles amenazas. A continuación, veremos en detalle cómo se realiza la gestión de usuarios y roles, y cómo se han implementado medidas de seguridad para proteger la información en nuestra plataforma web.

7.1.-GESTIÓN DE USUARIOS Y ROLES

Hemos definido un modelo personalizado para los usuarios, extendiendo la clase 'AbstractUser' de Django, este modelo nos ha permitido agregar campos adicionales y funcionalidades personalizadas [10]. Hemos definido 5 roles de usuario mediante el método choices definido por Django.

```
25 usages   lidiaAH9
class Usuario(AbstractUser):
    USER_ROLE_CHOICES = (
        (1, 'Técnico'),
        (2, 'Cuidador'),
        (3, 'Conductor'),
        (4, 'Familiar'),
        (5, 'Usuario'),
    )
    user_role = models.IntegerField(choices=USER_ROLE_CHOICES, null=True)
    centro = models.CharField(max_length=40)
    imagen_perfil = models.ImageField(upload_to='profile_images', blank=True, null=True)
```

FIGURA 23.DISEÑO CENTRADO EN EL USUARIO

Para utilizar cada rol en la funcionalidad, mediante la condición 'if' en las plantillas HTML elegimos el usuario que va a poder acceder a ciertas de estas funcionalidades.

Por ejemplo, en nuestro archivo 'rutas.html', los usuarios de tipo conductor (3), verán sólo el listado de rutas, mientras que si es superusuario o técnico (1) además accederá a crear las rutas.

```
<div class="container ">
  {% if user.user_role == 3 %}
  <div class="d-flex justify-content-center align-items-center" >
      <div class="card card-outline-secondary " style="...">
        <div class="card-header p-2">
          <h3 class="mb-0">Rutas disponibles</h3>
        </div>
```

```
{% if user.user_role == 1 or request.user.is_superuser %}
    <div class="col-md-6">
      <h3>RUTA</h3>
      <hr style="...">
      <form method="POST" action="{% url 'rutas' %}">
        {% csrf_token %}
        <div class="py-1">
          <p style="...">Introduzca el nombre de la ruta</p>
          <input type="text" class="form-control" placeholder="Nombre ruta" name="nombre">
        </div>
        <hr style="...">

        <p style="...">Fecha de ruta</p>
        <input type="date" class="form-control" style="..." name="fecha">
```

Figura 24.Diseño Centrado en el Usuario

Además, hemos configurado el backend de autenticación en el archivo 'settings.py' para que utilice nuestro modelo de usuario personalizado. Esto nos permite aprovechar todas las características y métodos implementados en el modelo personalizado al manejar la autenticación de los usuarios.

```
AUTH_USER_MODEL = 'webApp.Usuario'
```

Figura 25.Diseño Centrado en el Usuario

Para manejar las operaciones de inicio de sesión y cierre de sesión, hemos definido las URL correspondientes en el archivo 'urls.py'. Utilizando las vistas predeterminadas de Django, como 'CustomLoginView' y 'LogoutView', hemos asegurado un proceso de autenticación fluido y seguro. Estas vistas nos brindan funcionalidades estándar, como la validación de credenciales y la redirección adecuada después del inicio de sesión o cierre de sesión.

```
urlpatterns = [

    path('login/', CustomLoginView.as_view(), name='login'),
    path('logout/', LogoutView.as_view(), name='logout'),
```

```
LOGIN_URL = '/login/'
LOGIN_REDIRECT_URL = '/blog/blog/'
LOGOUT_REDIRECT_URL = '/login/'
```

Figura 26.Diseño Centrado en el Usuario

Destacar la recuperación de contraseñas, utilizando un enlace de restablecimiento de contraseña enviado por correo electrónico. Para ello, lo hemos hecho con los formularios y confirmaciones predeterminados de Django [11]. Explicaremos el proceso de recuperación de contraseña:

- Formulario de restablecimiento de contraseña: hemos definido una URL ('password_reset/') que está asociada a la vista PasswordResetView proporcionada por Django. Cuando un usuario desea restablecer su contraseña, puede acceder a esta URL y completar el formulario correspondiente. El formulario de restablecimiento de contraseña permite al usuario proporcionar su dirección de correo electrónico asociada a la cuenta.
- Confirmación de restablecimiento de contraseña enviada: después de enviar el formulario de restablecimiento de contraseña, el usuario recibe un correo electrónico que contiene un enlace único para restablecer su contraseña. Hemos definido una URL ('password_reset/done/') asociada a la vista PasswordResetDoneView de Django, que muestra una página de confirmación de que el correo electrónico de restablecimiento de contraseña ha sido enviado correctamente.
- Formulario de confirmación de restablecimiento de contraseña: cuando el usuario accede al enlace de restablecimiento de contraseña recibido en su correo electrónico, se le redirige a una URL dinámica ('reset/<uidb64>/<token>/') asociada a la vista PasswordResetConfirmView de Django. Esta vista muestra un formulario que permite al usuario ingresar una nueva contraseña y confirmarla.

- Confirmación de restablecimiento de contraseña completada: Después de enviar el formulario de confirmación de restablecimiento de contraseña, si todo es válido, la contraseña del usuario se restablece y se muestra una página de confirmación ('reset/done/') asociada a la vista PasswordResetCompleteView de Django. En esta página, se informa al usuario que el restablecimiento de contraseña se ha completado exitosamente.

```
# URL para el formulario de restablecimiento de contraseña
path('password_reset/', auth_views.PasswordResetView.as_view(template_name='auth/password_reset_form.html'),
     name='password_reset'),
# URL para la confirmación de restablecimiento de contraseña enviada
path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(template_name='auth/password_reset_done.html'),
     name='password_reset_done'),
# URL para el formulario de confirmación de restablecimiento de contraseña
path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(template_name='auth/password_reset_confirm.html'),
     name='password_reset_confirm'),
# URL para la confirmación de restablecimiento de contraseña completada
path('reset/done/', auth_views.PasswordResetCompleteView.as_view(template_name='auth/password_reset_complete.html'),
     name='password_reset_complete'),
```

Figura 27.Diseño Centrado en el Usuario

Hemos recogido todos estos ficheros predeterminados en el directorio ‘auth’ dentro de nuestro templates.

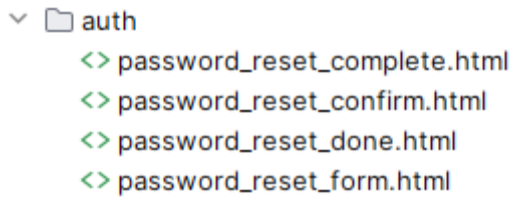

Figura 28.Diseño Centrado en el Usuario

Con esta implementación, nuestros usuarios pueden recuperar y restablecer su contraseña de manera segura y conveniente. Al proporcionar un flujo de trabajo claro y sencillo, garantizamos que los usuarios tengan acceso continuo a sus cuentas, incluso si han olvidado su contraseña.

Con todas estas configuraciones, nuestra plataforma web garantiza un sistema de autenticación sólido y confiable, proporcionando a los usuarios la capacidad de acceder a sus cuentas de forma segura y realizar operaciones relacionadas con su perfil y funcionalidades específicas de su rol.

7.2.-IMPLEMENTACIÓN DE MEDIDAS DE SEGURIDAD

De cara a realizar los formularios para guardar los datos, protegemos las vistas utilizando el decorador @login_required que requieren autenticación para asegurar de que solo los usuarios autenticados pueden acceder a ellas, esto ayuda a prevenir el acceso no autorizado a áreas restringidas de la plataforma y protege la información confidencial.

```
@login_required(login_url='/login/')
```

Figura 29.Código medidas de seguridad I

También hemos implementado una validación de formularios, realizando una validación exhaustiva de los datos ingresados por los usuarios en estos. Esto incluye verificar la integridad de los datos, asegurarse de que los campos obligatorios estén completos y validar el formato correcto de las entradas, ayudando así a prevenir ataques de inyección de código malicioso o inserción de datos incorrectos que podrían comprometer la seguridad de la aplicación. Para esto hay que poner '{% csrf_token %}' en cada *form*.

```
<form method="POST" action="{% url 'rutas' %}">
  {% csrf_token %}
  <div class="py-1">
    <p style="...">Introduzca el nombre de la ruta</p>
    <input type="text" class="form-control" placeholder="Nombre ruta" name="nombre">
  </div>
  <hr style="...">

  <p style="...">Fecha de ruta</p>
  <input type="date" class="form-control" style="..." name="fecha">
```

Figura 30. Código medidas de seguridad II

8.-DESPLIEGUE

Por último, crucial es llevar a cabo el despliegue de la aplicación en un entorno de producción. Además, es importante establecer un proceso de mantenimiento continuo para garantizar el funcionamiento óptimo y la disponibilidad constante de la plataforma. A continuación, se describen los pasos y consideraciones clave para el despliegue y mantenimiento de la página web.

- Subida del proyecto a un repositorio Git: para facilitar el despliegue y el seguimiento de cambios, es recomendable utilizar un sistema de control de versiones como Git. En este caso, se creó un proyecto en GitLab y se subió el proyecto de Django utilizando los comandos de Git correspondientes. Esto permite tener un historial de versiones y facilita la colaboración entre el equipo de desarrollo. Podemos acceder a este repositorio en el siguiente enlace: https://gitlab.grial.eu/lidiaah/tfg-lidia
- Configuración del entorno de producción: el despliegue de la página web se realizó en un servidor Ubuntu. Para ello, se configuraron los requisitos necesarios en el servidor, como la instalación de Python, Django, PostgreSQL y otras dependencias específicas del proyecto. Además, se estableció un entorno virtual para mantener las dependencias aisladas y evitar conflictos con otras aplicaciones o proyectos en el mismo servidor.
- Uso de Gunicorn y Nginx: para servir la aplicación Django en el entorno de producción, se emplearon Gunicorn y Nginx. Gunicorn se utilizó como servidor de aplicaciones para gestionar las solicitudes y respuestas de la página web. Por su parte, Nginx actuó como un proxy inverso, dirigiendo las solicitudes del cliente a Gunicorn y mejorando el rendimiento y la seguridad de la aplicación. Esta configuración asegura una alta disponibilidad y un mejor manejo del tráfico.
- Configuración del dominio: para acceder a la página web, se asignó un dominio específico, en este caso, el dominio de GRIAL. Esto implica configurar los registros DNS adecuados para apuntar el dominio al servidor donde se encuentra desplegada la aplicación. También se pueden configurar certificados SSL/TLS para habilitar conexiones seguras a través del protocolo HTTPS, lo cual es altamente recomendado para proteger la comunicación entre el cliente y el servidor.

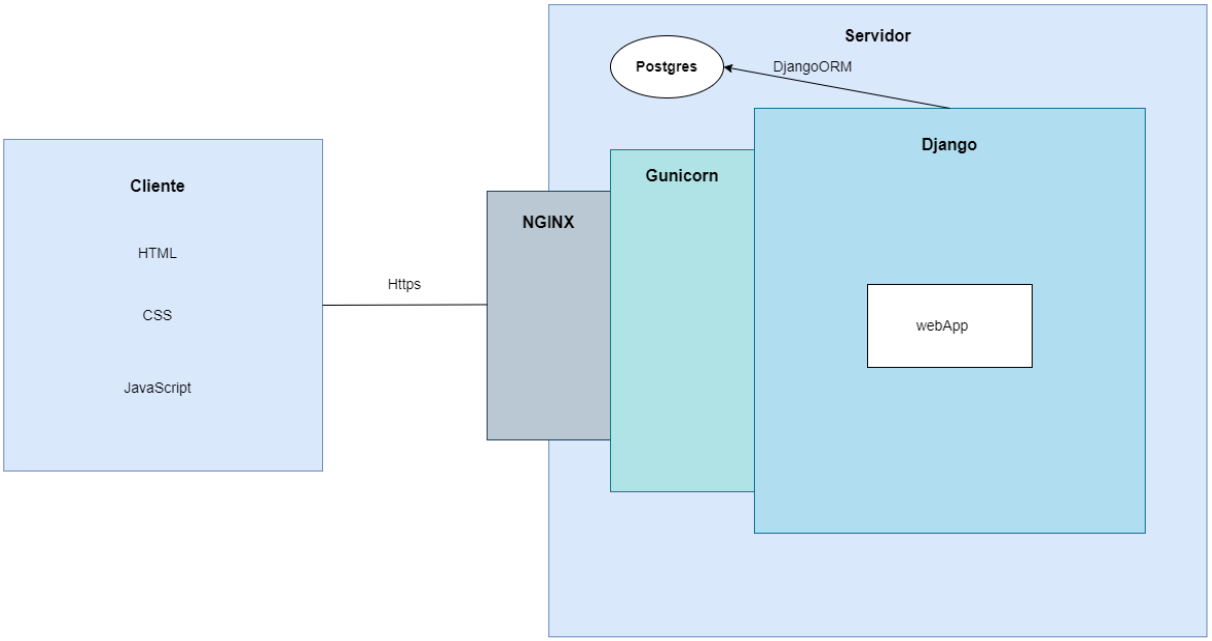

FIGURA 31.DIAGRAMA DESPLIEGUE

REFERENCIAS

[1] Django Software Foundation. (s.f.). Documentation. Recuperado de https://docs.djangoproject.com

[2] PostgreSQL Global Development Group. (s.f.). PostgreSQL: The world's most advanced open source relational database. Recuperado de https://www.postgresql.org

[3] Bootstrap. (s.f.). Bootstrap: The world's most popular framework for building responsive, mobile-first websites. Recuperado de https://getbootstrap.com

[4] JetBrains. (s.f.). PyCharm: Python IDE for Professional Developers. Recuperado de https://www.jetbrains.com/pycharm/

[5] Python Software Foundation. (s.f.). Welcome to Python.org. Recuperado de https://www.python.org

[6] FreeCodeCamp.org. (2018, 29 de junio). Django Web Development with Python Tutorial. Recuperado de https://www.youtube.com/watch?v=7XO1AzwkPPE&list=PLU8oAlHdN5BmfvwxFO7HdPciOCmmYneAB

[7] PostgreSQL Global Development Group. (s.f.). Download PostgreSQL for Windows. Recuperado de https://www.postgresql.org/download/windows/

[8] vitorfs. (2011, 9 de marzo). Django + jQuery file upload. Gist. Recuperado de https://gist.github.com/vitorfs/7d9e2d2c48fab9d6f432717814e6b762

[9] MDN Web Docs. (s.f.). Server-side website programming. Recuperado de https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Generic_views

[10] Django Software Foundation. (s.f.). Customizing authentication in Django. Recuperado de https://docs.djangoproject.com/en/4.2/topics/auth/customizing/

[11] Learn Django. (s.f.). Django Password Reset Tutorial. Recuperado de https://learndjango.com/tutorials/django-password-reset-tutorial