



Django v5

Documentación oficial: <https://docs.djangoproject.com/es/5.0/>

Elabora: Matías Santos
APP CRUD de Personas
Marzo/Abril 2024

Django es un framework (marco de trabajo) web de alto nivel que permite el desarrollo de sitios web seguros, escalables y mantenibles. Es gratuito de y de código abierto, está desarrollado con el lenguaje Python, y consta de una gran comunidad y documentación.

Django nace en un entorno periodístico, en la redacción de World Online fue diseñado para crear aplicaciones web fácil y rápidamente.

Django sigue el principio DRY, Don't Repeat Yourself, significa no te repitas, o también conocido como “una vez, y sólo una vez”, es una filosofía de definición de procesos que promueve la reducción de la duplicación especialmente en computación. Según este principio toda pieza de información nunca debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias.

Seguir el principio de programación DRY permite lograr una alta capacidad de mantenimiento del proyecto, facilidad para realizar cambios y pruebas de alta calidad.

Este principio es clave para crear buenas aplicaciones, con la unificación y reutilización se mejora la calidad del código base y a su vez se mejora la capacidad de mantenimiento.

El uso de Django en el desarrollo web ofrece numerosas ventajas, entre ellas:

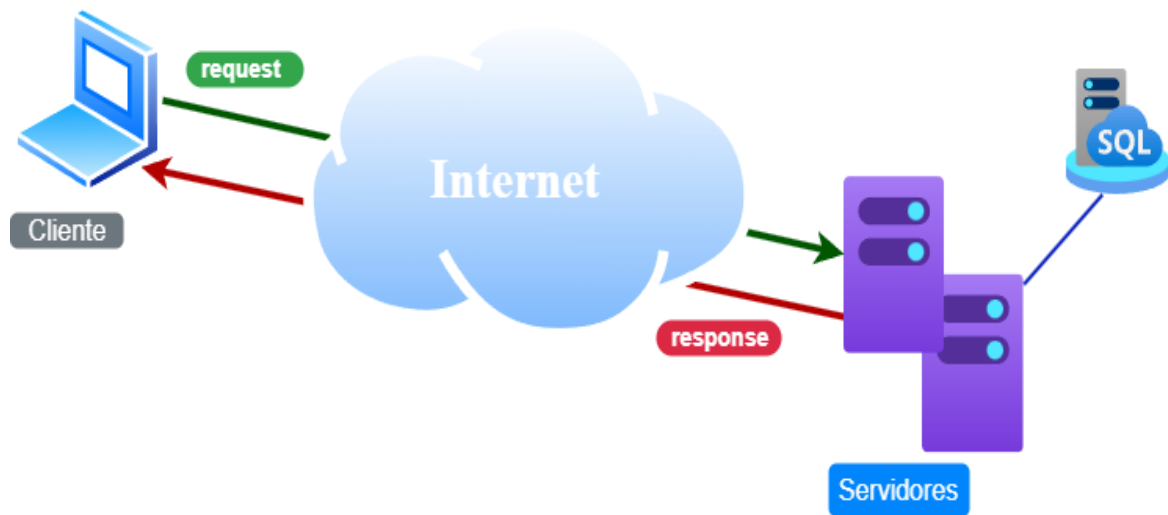
- Productividad: Django acelera el proceso de desarrollo web gracias a sus componentes reutilizables y su capacidad para automatizar tareas comunes.
- Seguridad: Django se enfoca en la seguridad, lo que lo convierte en una elección sólida para aplicaciones web sensibles que requieren protección contra amenazas comunes.
- Escalabilidad: Django es altamente escalable, lo que permite que las aplicaciones crezcan y se adapten a las necesidades cambiantes a medida que evolucionan.

En un sitio web tradicional basado en datos una aplicación web espera peticiones http del explorador web (o de otro cliente). Cuando se recibe una petición web la aplicación elabora lo que se necesita basándose en la URL y posiblemente en la información incluida en los datos POST o GET que le llegan desde el cliente.

Dependiendo de que se necesita quizás pueda entonces leer o escribir información desde una base de datos o realizar tareas requeridas para satisfacer la petición recibida.

La aplicación devolverá a continuación una respuesta al explorador web, con frecuencia creando dinámicamente una página html para que el explorador la presente insertando los datos recuperados o procesados en marcadores de posición dentro de una plantilla html.

Arquitectura Cliente – Servidor



Django sigue el patrón de diseño **MVT** (modelo, vista, template), que es una redefinición del conocido patrón MVC (modelo, vista, controlador).

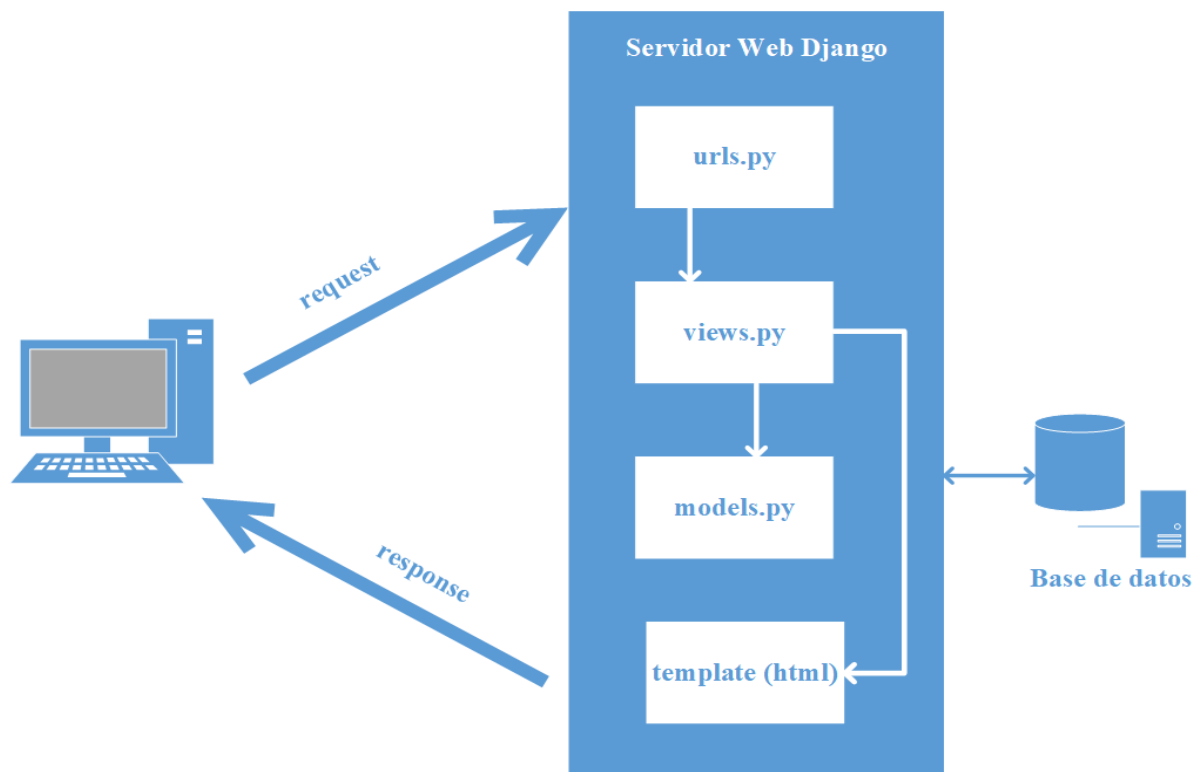
El **modelo** es la parte de la aplicación que maneja los datos, contiene todo lo relacionado a los datos, como acceder, como validarlos, cual es su comportamiento y las relaciones entre ellos.

La **vista** describe los datos que se envían al cliente (usuario), pero no describe su presentación, es la capa lógica del negocio, contiene la lógica que accede al modelo y lo lleva al template apropiado.

El **template** (plantilla) es la parte que se encarga de mostrar la información con html, css, javascript, etc. Es la capa de presentación, contiene todas las decisiones de cómo debe mostrarse la información en la página web.

La configuración de Django posee un archivo de URLs que permite asociar las vistas con las diferentes URLs que los usuarios pueden visitar para acceder a las diferentes funciones de la aplicación.

Patrón MVT (esquema de una aplicación)



Siguiendo ese patrón Django recibirá una petición desde un cliente, si la URL que se accede desde el cliente matchea con una función o clase en la vista, se pasará la petición a la vista correspondiente, en la vista se procesarán y/o recuperaran los datos desde el modelo (base de datos) que corresponda y finalmente se renderizará un template integrando los datos dinámicamente para luego enviar el html al cliente como respuesta a su petición.

urls.py: es un archivo Python que se utiliza para mapear una URL con su correspondiente función o clase de Python implementada en la vista (**views.py**), se usa un mapeador para redirigir las peticiones http a la vista apropiada basándose en la url de la petición. El mapeador llamado **urlpatterns**, puede también emparejar patrones de cadenas o dígitos específicos que aparecen en la URL y los pasan a la función o clase en la vista correspondiente como datos para ser procesados.

views.py: una vista es una función o clase de Python que gestiona las peticiones http que recibe y retorna una respuesta.

Las vistas acceden a los datos que necesitan para satisfacer las peticiones por medio de modelos, y delegan el formateo de la respuesta a los templates.

models.py: los modelos son objetos de Python que definen la estructura de los datos de una aplicación y proporcionan mecanismos para gestionar (agregar, modificar, eliminar, listar) y consultar registros en la base de datos.

templates: un template o plantilla es un archivo de texto que define la estructura o diseño de otro fichero (como una página html), con marcadores de posición que se utilizan para representar el contenido real. Django automáticamente buscara plantillas en un directorio llamado 'templates' de su aplicación. Una vista puede crear dinámicamente una página usando un template y rellenándolo con datos de un modelo.

Organización de un proyecto Django

Un desarrollo es un proyecto, y un proyecto consta de una o varias aplicaciones. Cada aplicación realiza algo en concreto y puede ser utilizada por distintos proyectos a la vez, así como también un proyecto puede hacer funcionar varios sitios web.

La estructura básica de un proyecto en Django es:

```
/proyecto/  
  /proyecto/  
    __init__.py  
    urls.py  
    manage.py  
    settings.py  
  aplicación_1/  
    __init__.py  
    models.py  
    views.py  
    urls.py  
  aplicación_2/  
    __init__.py  
    models.py  
    views.py  
    urls.py  
  templates/  
    aplicación_1/  
    aplicación_2/  
    base.html  
    header.html  
    footer.html  
  static/
```

Archivos del proyecto

`__init__.py`: indica a Python que el directorio sea interpretado como un paquete de Python

`settings.py`: contiene toda la configuración del proyecto, aplicaciones creadas, conexión a la base de datos, aplicaciones de terceros, etc.

`urls.py`: contiene los patrones de URLs del proyecto, generalmente se incluyen los archivos `urls.py` de cada aplicación mediante la función ‘include’

Archivos de una aplicación

`models.py`: contiene los modelos de datos de la aplicación

`views.py`: contiene las vistas de la aplicación

`tests.py`: permite que incluyamos tests para las aplicaciones

`urls.py`: usualmente este archivo se añade con las urls de la aplicación, para luego importarlas en el `urls.py` del proyecto, es una forma de organizar el proyecto y las aplicaciones con las urls.

Entornos virtuales

Cuando desarrollamos software, creamos aplicaciones solemos utilizar funciones y métodos de módulos y paquetes para hacer la mayor parte posible del desarrollo.

Esto tiene el inconveniente de que podemos ir acumulando paquetes innecesarios instalados, que en caso de eliminar un proyecto los paquetes quedarían instalados, o si eliminamos un paquete por eliminar un proyecto, ese paquete podría ser necesario en otro proyecto, o quizás dos proyectos distintos necesiten dos versiones diferentes de un mismo paquete, en general pueden darse diversos problemas debido a los distintos requisitos de unos u otros proyectos.

Para evitar esto se inventaron los entornos virtuales.

Un entorno virtual es por decirlo de algún modo un contexto seguro donde instalar paquetes de Python. Consiste en un directorio con el ejecutable de Python y todo lo que se necesita para trabajar, acompañado de una estructura de directorios para instalar paquetes y algunas utilidades para facilitar su gestión.

En la práctica, es como si fueran instalaciones de Python independientes.

Podemos tener varios entornos virtuales en nuestro sistema, cada uno con sus ejecutables de Python y sus propios módulos instalados, y activar uno u otro según nos convenga en cada momento, además cuando no necesitemos más uno de estos entornos podemos eliminarlo sin afectar a los demás.

Para crear un entorno virtual se utiliza el comando:

```
python -m venv nombre_del_entorno
```

Lo normal es tener un entorno virtual independiente para cada proyecto, y lo más usual es hacerlo creando en el propio directorio raíz con nombre del proyecto y llamarlo 'env'.

Pero solo con crear un entorno no es suficiente, una vez creado el entorno, para usarlo debemos activarlo con el siguiente comando:

(Windows)

Estando en el directorio raíz del proyecto

```
cd env/Scripts <<enter>>
```

```
activate <<enter>>
```

Observación:

```
cmd.exe: env/Scripts/activate.bat
```

(linux)

Estando en el directorio raíz del proyecto

```
cd env/bin <<enter>>
```

```
source activate <<enter>>
```

Al activarlo normalmente se nos indicará en la consola el cambio de entorno añadiendo el nombre del entorno al principio del prompt:

```
(env) C:\Users\usuario\proyecto> _
```

```
(env) usuario@lenovo:~/proyecto/$ _
```

Lo que estamos haciendo al activar el entorno virtual es indicarle al sistema que a partir de ahora el ejecutable de Python que vamos a usar es el que se encuentra en el entorno virtual y que la ruta a los módulos instalados también es la que corresponde al entorno virtual.

Ahora si instalamos algún modulo utilizando el comando pip (pip install modulo) se instalará solo en ese entorno virtual y será accesible a nuestro desarrollo sólo si tenemos el entorno activado.

Para desactivar el entorno virtual, solo hay que ir a la ruta de activación y ejecutar el comando 'deactivate'.

Si queremos replicar el entorno y el proyecto en otro sistema, debemos crear el entorno virtual nuevamente y luego instalar los módulos que contenía.

Afortunadamente, el comando pip nos permite gestionar esto de forma fácil, con nuestro entorno virtual activado, ejecutamos pip freeze que nos mostrará una lista de paquetes instalados y sus versiones, por ejemplo:

```
django==5.0
Pillow==8.4.0
tzdata==1.0.1
```

Lo interesante de este formato es que se puede usar en un archivo de texto para que sea interpretado por pip a la hora de instalar los paquetes necesarios para el proyecto.

Para hacer eso podemos crear un archivo de texto llamado requirements.txt y copiar la salida del comando pip freeze, o redirigir la salida del comando de la siguiente manera:

```
pip freeze > requirements.txt
```

Ahora podemos utilizar esta lista para instalar exactamente los mismos paquetes en sus mismas versiones, para ello solo tenemos que pasar como argumento el archivo requirements.txt al comando pip install con la opción -r del siguiente modo:

```
pip install -r requirements.txt
```

No olvidar que previamente debemos tener el entorno virtual activado.

Deberíamos utilizar un entorno virtual en todos nuestros proyectos, para tener el control de los paquetes que estamos utilizando, además, cualquier proyecto medianamente complejo debería ir siempre acompañado de un archivo requirements.txt para facilitar su despliegue.

Primer proyecto Django

Un proyecto es una instancia de varias aplicaciones Django, más las configuraciones de esas aplicaciones.

Una aplicación es un conjunto de funcionalidades de Django, usualmente incluyen modelos, vistas, urls.

Para empezar nuestro proyecto en Django, lo primero es lo mencionado anteriormente, crear un entorno virtual, activar el entorno virtual y tener el archivo requirements.txt listo para guardar los paquetes, una estructura como la siguiente:

(Windows)

```
proyecto/  
  env/  
    Include/  
    Lib/  
    Scripts/  
    requirements.txt
```

Ahora que tenemos todo pronto entramos desde la consola al directorio 'env' y procedemos a instalar Django con el siguiente comando:

```
pip install django
```

Al finalizar la instalación para crear el proyecto ejecutamos el siguiente comando:

```
django-admin startproject nombre_del_proyecto
```

La estructura quedaría así:

(Windows)

```
proyecto/  
  env/  
    Include/  
    Lib/  
    Scripts/  
    nombre_del_proyecto/  
    requirements.txt
```

Ahora para crear una aplicación en nuestro proyecto accedemos al directorio del proyecto con el comando “cd nombre_del_proyecto” y tenemos dos comandos que podemos ejecutar para crear nuestra primer aplicación:

```
python manage.py startapp nombre_de_la_aplicacion
```

```
django-admin startapp nombre_de_la_aplicacion
```

El archivo manage.py es el principal archivo del proyecto, dado que se encarga de ejecutar el proyecto, crear migraciones, crear aplicaciones y levantar el servidor Django.

El archivo settings.py en el directorio del proyecto es quién contiene toda la configuración asociada al mismo, y es allí donde debemos definir cada aplicación que vayamos creando, para ello alcanza con ingresar su nombre en la lista que se llama

INSTALLED_APPS:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
  
    'nombre_de_la_aplicacion',  
]
```

Ahora si todo ha salido bien, podemos levantar el servidor con el comando:

```
python manage.py runserver
```

```
python manage.py runserver ip:puerto
```

El primer comando lanzara nuestro servicio web en la IP local de la máquina en que tenemos el proyecto, con acceso mediante el puerto 8000.

En caso de utilizar el segundo comando la IP que estamos indicando para que responda nuestro servicio debe estar incluida en el archivo settings.py en la variable (lista) llamada ALLOWED_HOSTS, de la siguiente manera:

Por ejemplo:

```
ALLOWED_HOSTS = [ '192.168.1.1' ]
```

La salida al ejecutar el comando sería similar a la siguiente:

```
(env) C:\Users\Matias Santos\Desktop\django_personas\env\personas>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
March 02, 2024 - 13:10:50
Django version 5.0.2, using settings 'personas.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Ahora podemos acceder a nuestra aplicación web mediante un navegador introduciendo la dirección `http://127.0.0.1:8000` o `http://localhost:8000` y veremos la página de bienvenida que nos indica que todo está funcionando correctamente:

django

[View release notes](#) for Django



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



Django Documentation
Topics, references, & how-to's



Tutorial: A Polling App
Get started with Django

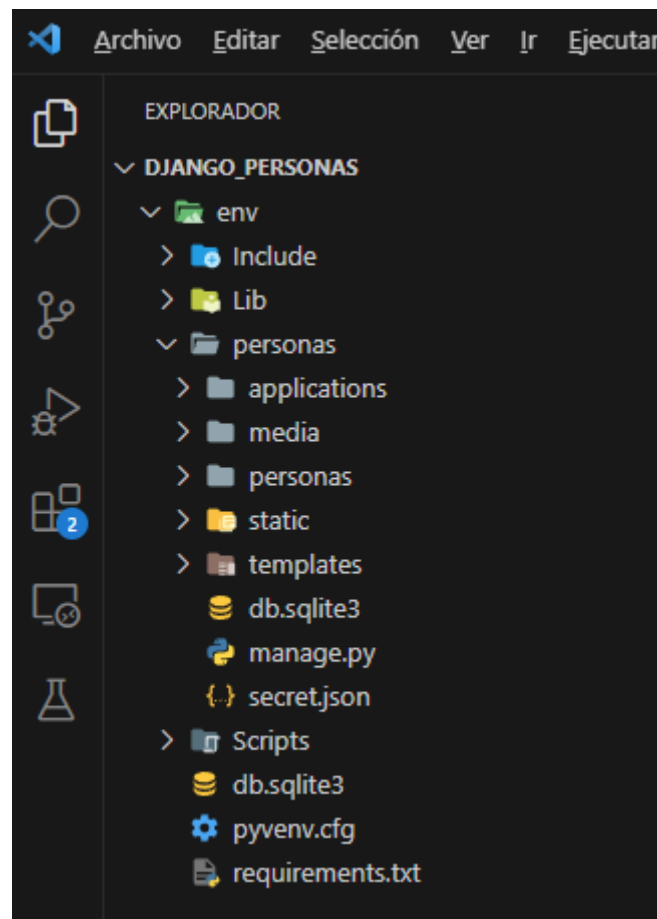


Django Community
Connect, get help, or contribute

Ubicación de los templates

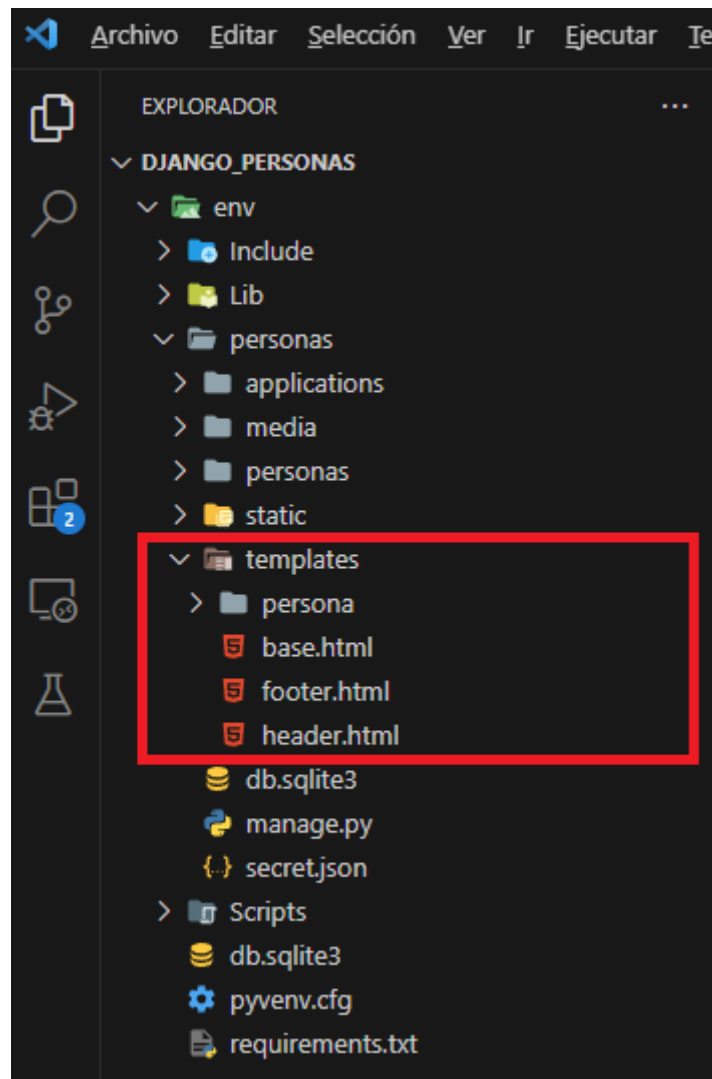
Django busca los templates (páginas html) en una carpeta que debemos crear con el nombre de “templates” y que usualmente se ubica en el mismo directorio que el proyecto que estemos creando, por ejemplo:

Tenemos el proyecto “django_personas”:



Dentro de la carpeta templates usualmente se crea un directorio con el nombre de cada aplicación que contenga el proyecto para alojar los templates correspondientes a cada aplicación.

En la ruta raíz del directorio “templates” usualmente se crean los templates base.html, header.html y footer.html, esto es una recomendación, pero no necesariamente tiene que ser así, eso dependerá de cada proyecto como desee implementarse.



Luego de creado estos directorios debemos indicarle a Django mediante el archivo `settings.py` del proyecto la ubicación de este directorio templates:

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [BASE_DIR.child('templates')],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
            ],  
        },  
    ],  
]
```

Directorio “static”

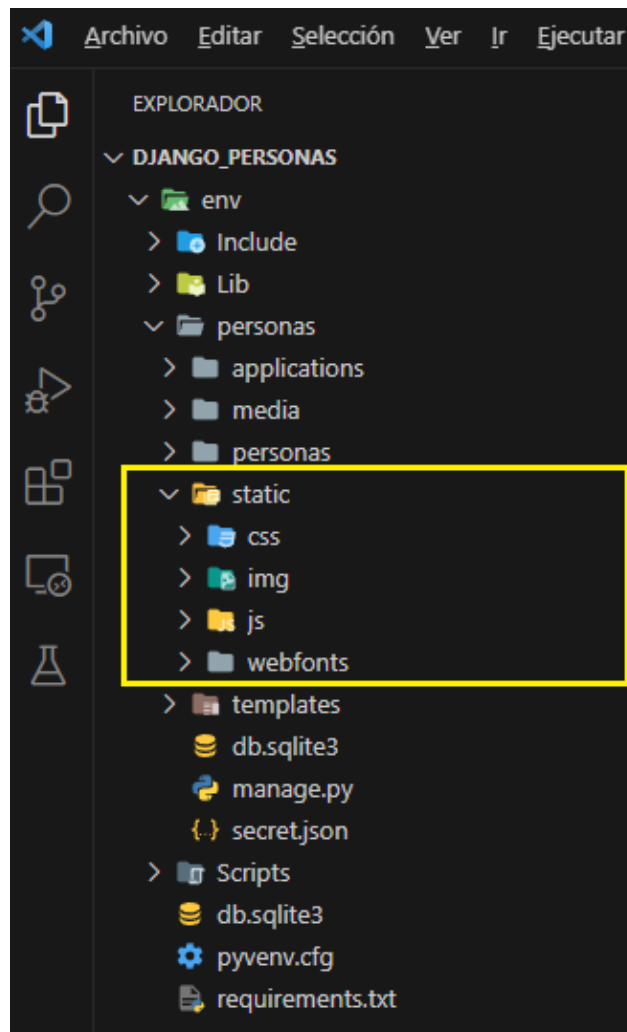
Otro directorio importante es el directorio “static” del proyecto, el cuál también debemos crear, usualmente se ubica en el mismo directorio del proyecto.

Su función es almacenar todos los archivos estáticos necesarios como ser hojas de estilos CSS, scripts de Javascript, imágenes, fuentes, y otros recursos que se utilizan en el frontend de las aplicaciones web. Estos archivos son servidos directamente por el servidor web en lugar de ser generados dinámicamente por Django.

Habitualmente este directorio “static” suele tener una estructura de subdirectorios para organizar mejor los diferentes tipos de archivos estáticos, por ejemplo, conteniendo los subdirectorios CSS, IMG, JS, entre otros.

Es importante mencionar que Django sirve estos archivos estáticos durante el desarrollo, cuando tenemos la variable del archivo settings.py `DEBUG = True`, en un entorno de producción se suele utilizar un servidor web separado, como Apache o Nginx, para servir estos archivos estáticos de manera más eficiente.

En tales casos, Django puede configurarse para recopilar los archivos estáticos en un solo lugar durante el despliegue, facilitando su servido por el servidor web estático, esto se hace con el comando “collectstatic”.

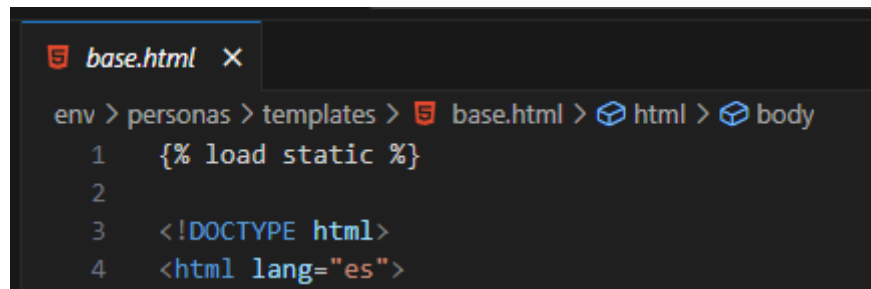


La creación de este directorio también lleva su configuración en el archivo `settings.py` del proyecto, usualmente sería la siguiente:

```
settings.py X
env > personas > personas > settings.py > ...
126
127
128 # Static files (CSS, JavaScript, Images)
129 # https://docs.djangoproject.com/en/5.0/howto/static-files/
130
131 STATIC_URL = '/static/'
132 STATICFILES_DIRS = [BASE_DIR, "static"]
133
```

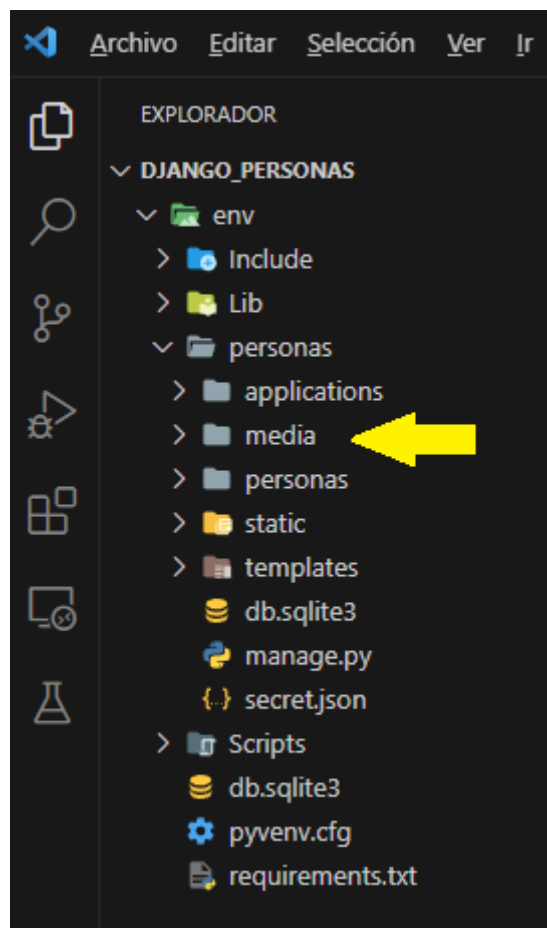
Como podemos ver en cada sección del archivo `settings.py` tenemos la URL de la documentación oficial de Django dónde poder ver las configuraciones de cada tipo de variable.

En los template html para que se carguen los archivos estáticos necesarios debemos indicarlo con la etiqueta `{% load static %}`, por ejemplo:



```
base.html x
env > personas > templates > base.html > html > body
1  {% load static %}
2
3  <!DOCTYPE html>
4  <html lang="es">
```

Directorio media

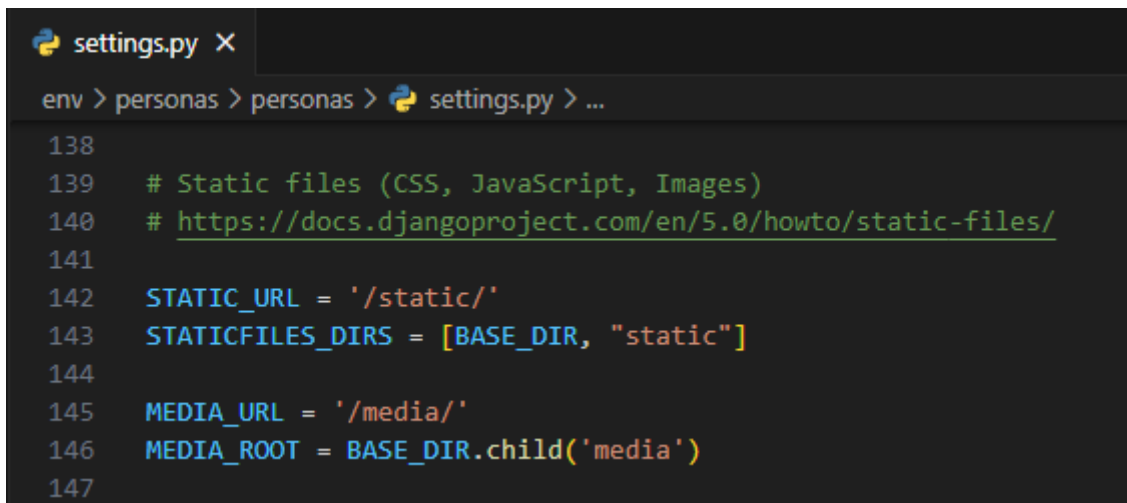


El directorio “media” en un proyecto Django, directorio que debemos crear en el mismo directorio que nuestro proyecto, tiene una función específica relacionada con el almacenamiento y manejo de archivos multimedia, como imágenes, videos, archivos PDF, etc., que serán cargados por los usuarios de las aplicaciones.

Cuando los usuarios de una aplicación Django suben archivos multimedia, estos archivos se almacenan típicamente en un directorio con el mismo nombre de la aplicación dentro del directorio “media”. Django proporciona una forma conveniente de manejar estos archivos mediante el uso del modelo “FileField” o “ImageField” en los modelos de la base de datos.

Estos campos permiten a los usuarios cargar archivos a través de formularios y almacenar las ubicaciones correspondientes a estos archivos en la base de datos.

Este directorio “media” debe ser correctamente configurado en el archivo settings.py del proyecto Django, especificando la ruta donde se guardarán los archivos multimedia que se suban.



```
138
139 # Static files (CSS, JavaScript, Images)
140 # https://docs.djangoproject.com/en/5.0/howto/static-files/
141
142 STATIC_URL = '/static/'
143 STATICFILES_DIRS = [BASE_DIR, "static"]
144
145 MEDIA_URL = '/media/'
146 MEDIA_ROOT = BASE_DIR.child('media')
147
```

Es importante recordar, al igual que con los archivos estáticos, en un entorno de producción se suele utilizar un servidor web separado para servir los archivos multimedia de manera eficiente, Django también proporciona la utilidad “collectstatic” para recopilar todos los archivos estáticos y multimedia en un solo lugar durante el despliegue.

Otra observación importante es que para trabajar con archivos multimedia en los modelos es necesario instalar el módulo Pillow, utilizando el comando pip install.

`pip install Pillow`

También es necesario en los modelos que vayamos a definir un campo por ejemplo de tipo `ImageField` indicar la ruta a dónde debe subir Django los archivos multimedia, utilizando el parámetro “`upload_to`”, por ejemplo:

```
imagen = models.ImageField(upload_to='persona', null=True, blank=True)
```

Hay que recordar también que en el formulario HTML para que permita subir archivos debemos indicárselo en el atributo “`enctype`”, por ejemplo:

```
<form method="POST" enctype="multipart/form-data">
```

Herencia de templates

La herencia de templates es una característica poderosa en los frameworks de desarrollo web como Django. Permite la reutilización de código y la organización eficiente de la estructura de las páginas web.

En Django, la herencia de templates se basa en la idea de que muchas páginas web tienen una estructura común, pero con contenido específico que varía entre ellas.

En lugar de duplicar el código HTML común en cada una de estas páginas, podemos definir un “template base” que contenga la estructura general del sitio web y luego extender este template base en cada página específica que se necesite.

En el siguiente template, `base.html`, tenemos cargados los archivos estáticos necesarios para su correcta visualización mediante la etiqueta `{% load static %}` y la forma de indicar la ruta ya sea para las hojas de estilo CSS, como para archivos Javascript es la siguiente (manteniendo la estructura mencionada anteriormente):

```
<!-- Bootstrap 5.1.3 -->
<link rel="stylesheet" type="text/css" href="{% static 'css/bootstrap.min.css' %}">
<!-- Fontawesome 6 -->
<link rel="stylesheet" type="text/css" href="{% static 'css/fontawesome6.min.css' %}">
<!-- Estilos Propios -->
<link rel="stylesheet" type="text/css" href="{% static 'css/estilos.css' %}">
```

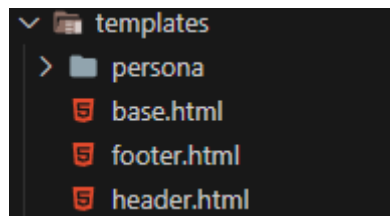
```

<script type="text/javascript" src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
<script type="text/javascript" src="{% static 'js/popper.min.js' %}"></script>
<script type="text/javascript" src="{% static 'js/bootstrap.min.js' %}"></script>
<script type="text/javascript" src="{% static 'js/fontawesome6.min.js' %}"></script>
<!-- JS Propio -->
<script type="text/javascript" src="{% static 'js/main.js' %}"></script>

```

También tenemos las etiquetas “include” de Django, que nos permiten incluir ciertos fragmentos de código en nuestro template base HTML, implementado de esta forma lo que hacemos es dividir el código por secciones para que sea más mantenible y fácil de implementar.

Lo que estamos incluyendo en este caso en el template base.html son los template header.html y footer.html que están ubicados en el mismo directorio dónde se encuentra el template base.html



```

<body>
  <header>
    {% include 'header.html' %}
  </header>

  <main class="container-fluid">
    <section class="row">

    </section>
  </main>

  <footer id="footer" class="bg-dark p-5">
    {% include 'footer.html' %}
  </footer>

  <script type="text/javascript" src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
  <script type="text/javascript" src="{% static 'js/popper.min.js' %}"></script>
  <script type="text/javascript" src="{% static 'js/bootstrap.min.js' %}"></script>
  <script type="text/javascript" src="{% static 'js/fontawesome6.min.js' %}"></script>
  <!-- JS Propio -->
  <script type="text/javascript" src="{% static 'js/main.js' %}"></script>

</body>

```

También tenemos las etiquetas de `{% block content %}`, estas etiquetas en los templates tienen la función de definir áreas donde el contenido específico de una página web será insertado desde otra página web, hace que el contenido sea dinámico de acuerdo con la página web que estemos visualizando.

Estos bloques permiten a los templates que se extiendan y que el contenido de la página pueda ser personalizado sin tener que modificar el template base.

`{% block content %}` define el bloque de contenido llamado “content” (podemos llamarle de otra manera si quisieramos), este bloque actúa como un marcador de posición dónde se debe insertar el contenido específico de cada página, y finalizaría cuando encuentre la etiqueta `{% endblock %}`

Cuando un template extiende este template base y sobrescribe este bloque “content”, el contenido definido en el nuevo template se insertará en el lugar del bloque de contenido predeterminado, esto permite la flexibilidad considerable en la construcción de páginas web en Django, ya que te permite definir la estructura general de los templates y luego personalizar el contenido de cada página según sea necesario.

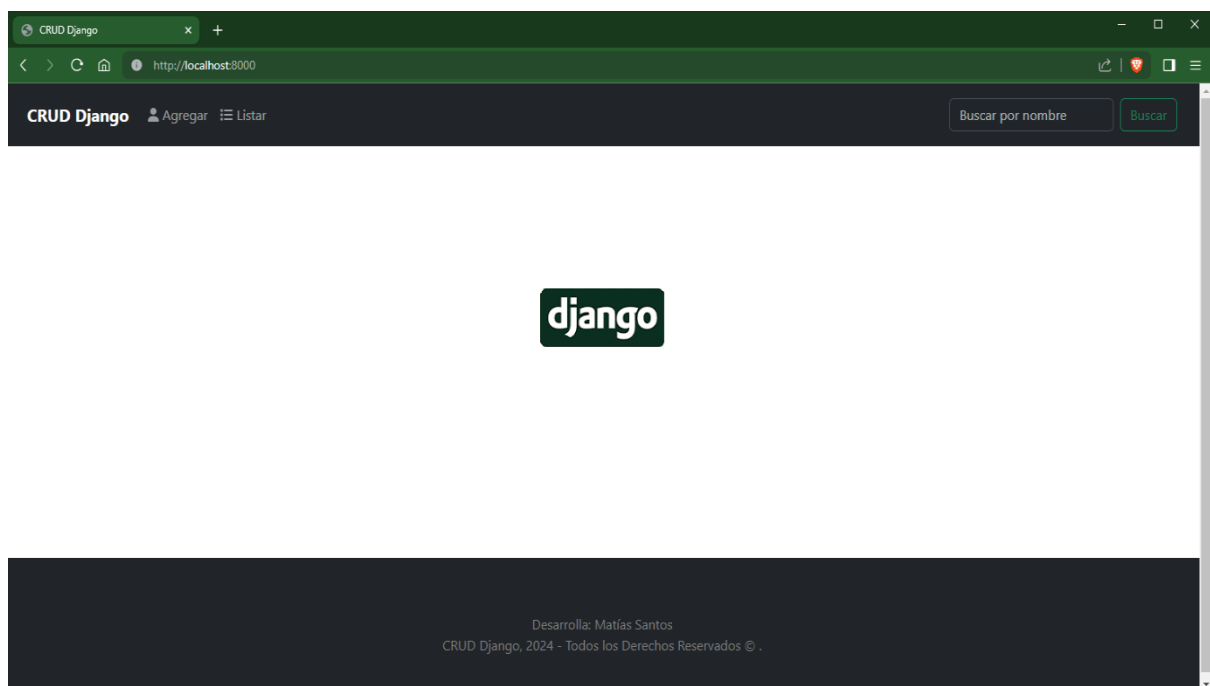
Finalmente, el archivo `base.html` quedaría de la siguiente forma:

```
base.html X
env > personas > templates > base.html > ...
1  {% load static %}
2
3  <!DOCTYPE html>
4  <html lang="es">
5  <head>
6      <meta charset="utf-8">
7      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
8      <meta http-equiv="X-UA-Compatible" content="ie=edge">
9      <link rel="shortcut icon" href="#" rel="noopener noreferrer">
10     <!-- Bootstrap 5.1.3 -->
11     <link rel="stylesheet" type="text/css" href="{% static 'css/bootstrap.min.css' %}">
12     <!-- Fontawesome 6 -->
13     <link rel="stylesheet" type="text/css" href="{% static 'css/fontawesome6.min.css' %}">
14     <!-- Estilos Propios -->
15     <link rel="stylesheet" type="text/css" href="{% static 'css/estilos.css' %}">
16
17     {% block css %}{% endblock %}
18     <title>{% block title %}{% endblock %}</title>
19 </head>
20 <body>
21     <header>
22         {% include 'header.html' %}
23     </header>
24
25     <main class="container-fluid">
26         <section class="row">
27             {% block content %}
28             {% endblock %}
29         </section>
30     </main>
31
32     <footer id="footer" class="bg-dark p-5">
33         {% include 'footer.html' %}
34     </footer>
35
36     <script type="text/javascript" src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
37     <script type="text/javascript" src="{% static 'js/popper.min.js' %}"></script>
38     <script type="text/javascript" src="{% static 'js/bootstrap.min.js' %}"></script>
39     <script type="text/javascript" src="{% static 'js/fontawesome6.min.js' %}"></script>
40     <!-- JS Propio -->
41     <script type="text/javascript" src="{% static 'js/main.js' %}"></script>
42     {% block js %}{% endblock %}
43 </body>
44 </html>
```

Para extender desde el archivo `base.html` que además este ya incluye un `header.html` y un `footer.html`, lo haríamos de la siguiente forma:

```
index.html x
env > personas > templates > persona > index.html > ...
1 {% extends 'base.html' %}
2
3 {% load static %}
4
5 {% block title %}CRUD Django{% endblock %}
6
7 {% block content %}
8     <figure class="d-flex align-items-center justify-content-center" id="imagen-principal">
9         
10     </figure>
11 {% endblock %}
```

Y si accedemos a la aplicación en su página principal `index.html` el resultado sería:



Modelos de datos

Los modelos de datos en Django, son clases de Python que se crean a partir del módulo `django.db.models` y que definen la estructura y el comportamiento de los datos que se almacenarán en la base de datos.

Definen las tablas, índices, restricciones, etc., de la base de datos que se generara automáticamente a partir de estos modelos.

Cada modelo de datos representa una tabla en la base de datos y define los campos que tendrá la tabla, así como también métodos para realizar operaciones entre ellos.

Los modelos de datos en Django se definen heredando de la clase `models.Model`, que se encuentra en el módulo mencionado anteriormente.

Estas clases modelan los datos de manera orientada a objetos, lo que facilita la interacción con ellos en las aplicaciones.

Cada vez que creamos un modelo de datos, o actualizamos algo en ellos debemos crear y aplicar las migraciones en Django para que surjan efecto, esto lo hacemos mediante dos comandos:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

De esta forma Django comprobara errores y en caso de no existir reflejara los cambios en la base de datos.

Siguiendo con el proyecto `django_personas` a continuación se crean dos modelos de datos, Departamento y Persona.

En este ejemplo cada persona tendrá sus campos como ser nombre, apellido, cédula, email, teléfono, imagen y dos campos que son “`created`” y “`updated`” que son gestionados por Django de forma automática, uno para cuando se cree un objeto Persona y otro para cuando se actualice el objeto Persona, estos campos guardaran la fecha y hora de dichas acciones, la Persona también tendrá un departamento en el cuál trabaja, generando de esta forma una relación de “muchos a 1” entre la clase Persona y la clase Departamento, lo cuál significa que en un departamento pueden trabajar muchas personas, o sea que cada instancia de Persona está asociada con una instancia de Departamento.

```
models.py X
env > personas > applications > persona > models.py > ...
1  from django.db import models
2
3
4  class Departamento(models.Model):
5      nombre = models.CharField("Departamento", max_length=100)
6      created = models.DateTimeField(auto_now_add=True)
7      updated = models.DateTimeField(auto_now=True)
8
9      class Meta:
10         verbose_name = "Departamento"
11         verbose_name_plural = "Departamentos"
12         ordering = ['nombre']
13
14     def __str__(self):
15         return self.nombre
16
17
18 class Persona(models.Model):
19     nombre = models.CharField("Nombre", max_length=20)
20     apellido = models.CharField("Apellido", max_length=20)
21     cedula = models.CharField("Cédula", max_length=12, unique=True)
22     email = models.EmailField("Email", unique=True)
23     telefono = models.CharField("Teléfono", max_length=15)
24     imagen = models.ImageField(upload_to='persona', null=True, blank=True)
25
26     departamento = models.ForeignKey(Departamento, related_name="departamento", on_delete=models.CASCADE)
27
28     created = models.DateTimeField(auto_now_add=True)
29     updated = models.DateTimeField(auto_now=True)
30
31     class Meta:
32         verbose_name = "Persona"
33         verbose_name_plural = "Personas"
34         ordering = ['nombre', 'apellido']
35
36     def __str__(self):
37         return self.nombre + " " + self.apellido
38
```

La clase Meta en los modelos de Django se utiliza para proporcionar metadatos adicionales sobre el modelo en sí. Estos metadatos no son campos de datos, sino que son configuraciones adicionales que afectan a como Django trata y procesa el modelo.

Algunos usos comunes son:

Especificar el nombre de la tabla de la base de datos, con el atributo “db_table”, por defecto Django utiliza el nombre del modelo en minúscula y con guiones bajos como nombre de la tabla, pero podemos anular este comportamiento utilizando “db_table”.

Especificar el orden de los resultados de las consultas, para eso podemos utilizar el atributo “ordering”, como por ejemplo en la clase Departamento los resultados estarán ordenados por nombre.

Especificar restricciones en caso de ser necesarias, para esto podemos utilizar el atributo “constraint” para especificar restricciones adicionales a nivel de la base de datos para el modelo.

En este ejemplo no se usan restricciones, pero podrían definirse de la siguiente manera:

```
class MyModel(models.Model):
    # campos del modelo

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=['campo_unico'], name='nombre_de_restriccion_unico'),
            models.CheckConstraint(check=models.Q(campo_check=True), name='nombre_de_restriccion_check'),
        ]
```

En sí la clase Meta proporciona una forma conveniente de agregar configuraciones y metadatos, en la documentación oficial tenemos todas las opciones que podemos utilizar:

<https://docs.djangoproject.com/en/5.0/ref/models/options/#django.db.models.Options>

Algunos de los campos utilizados en estos modelos son:

CharField para campos de texto y con una longitud máxima especificada por el atributo `max_length` (el máximo puede ser 255).

DateTimeField para campos de texto de tipo fecha y hora

EmailField para campos de texto de tipo email, con el atributo `unique=True` que significa que dicho campo no puede repetirse en toda la tabla, o sea no pueden existir dos registros con el mismo email.

ImageField para un campo de tipo imagen

ForeignKey que indica que el campo departamento está asociado con una instancia de la clase Departamento, y el atributo `related_name="departamento"` significa que desde el modelo Departamento podemos acceder a las instancias relacionadas de Persona utilizando el nombre “departamento”. El modelo secundario esta asociado con muchas instancias del modelo primario, relación uno a muchos.

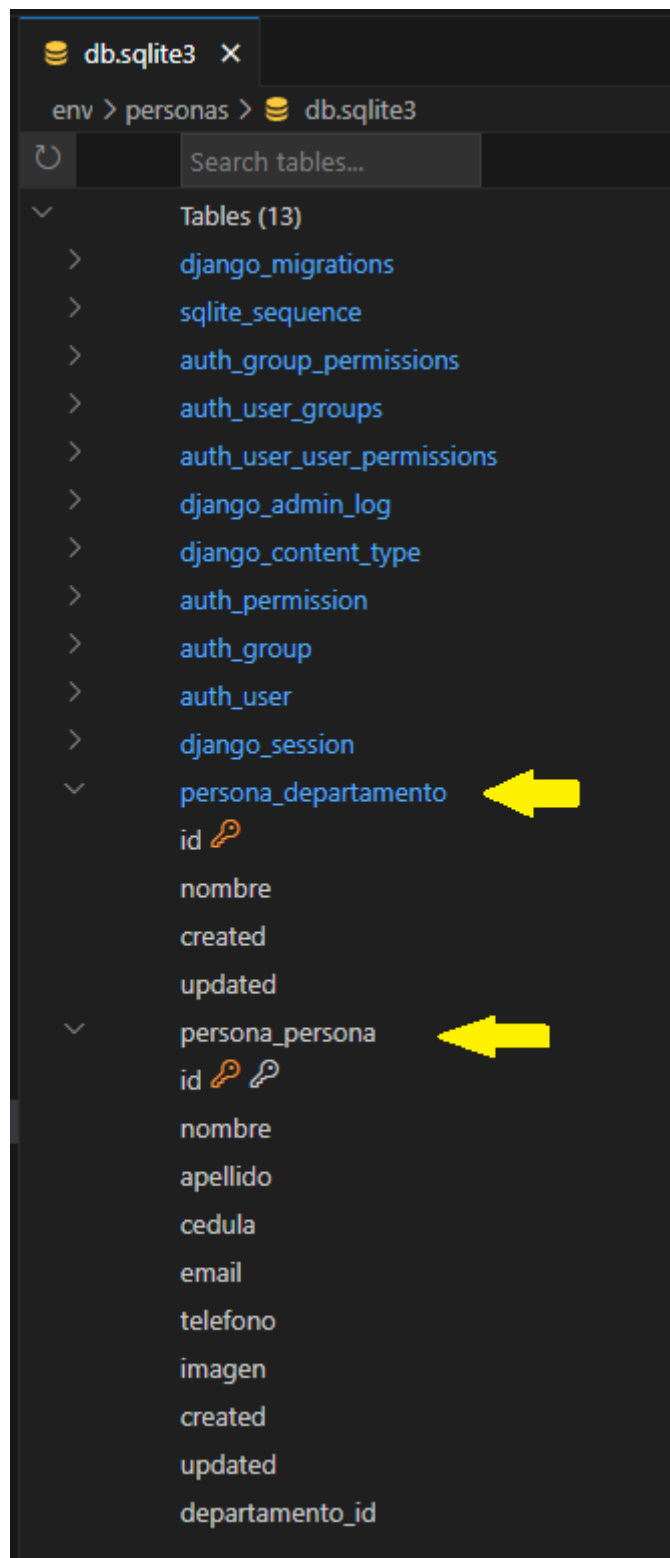
Si fuese una relación **OneToOneField**, sería una relación uno a uno entre dos modelos.

Existen otros como `BooleanField`, `TextField`, `DateField`, `TimeField`, `IntegerField`, `FloatField`, `PositiveIntegerField`, `DecimalField`, `FileField`, `SlugField`, `URLField`, `ForeignKeyField`, `ManyToManyField`, `OneToOneField`, etc.

Se pueden ver en la documentación oficial:

<https://docs.djangoproject.com/en/5.0/ref/models/fields/>

Luego de crear los modelos de datos y ejecutar las migraciones, Django creara las tablas en la base de datos, cabe mencionar que Django automáticamente le crea a cada modelo un atributo ID único y autoincremental.



Django de esta manera crea las tablas:

`personas_departamento` y `persona_persona`

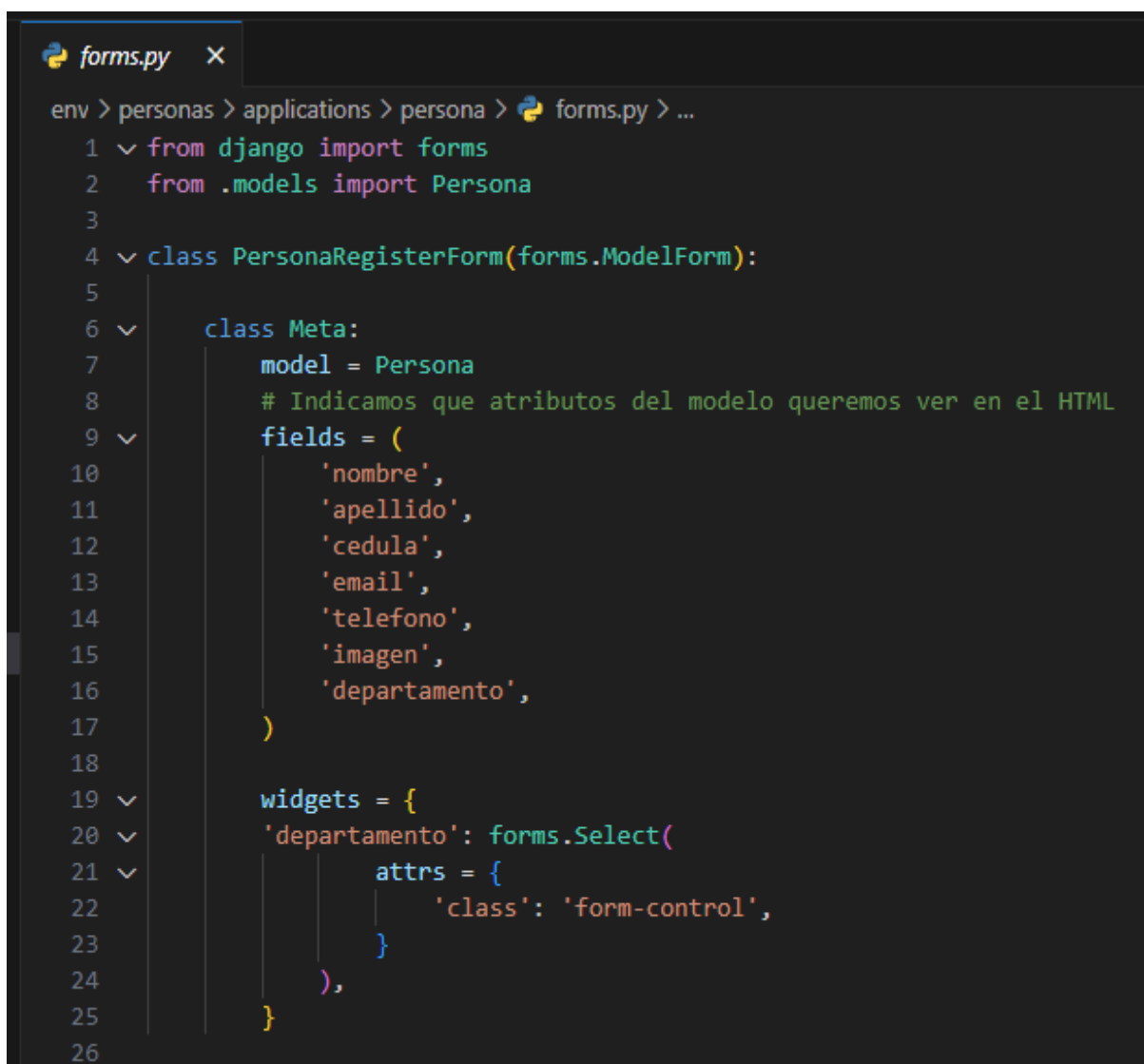
Formularios personalizados forms.py

El archivo `forms.py` en un proyecto Django tiene la función de definir formularios personalizados que se utilizan para recopilar y validar datos del usuario en las vistas necesarias.

Los formularios en Django se utilizan para manejar la entrada de datos de parte del usuario, validar estos datos y procesarlos de acuerdo con la lógica que se defina en la aplicación.

En general los formularios en Django se definen mediante la creación de clases que heredan de la clase “`forms.Form`” o “`forms.ModelForm`” dependiendo de si estás creando un formulario que se basa en un modelo de base de datos o no.

En nuestro ejemplo tenemos el siguiente archivo `forms.py`



```
forms.py X
env > personas > applications > persona > forms.py > ...
1  from django import forms
2  from .models import Persona
3
4  class PersonaRegisterForm(forms.ModelForm):
5
6      class Meta:
7          model = Persona
8          # Indicamos que atributos del modelo queremos ver en el HTML
9          fields = (
10             'nombre',
11             'apellido',
12             'cedula',
13             'email',
14             'telefono',
15             'imagen',
16             'departamento',
17         )
18
19         widgets = {
20             'departamento': forms.Select(
21                 attrs = {
22                     'class': 'form-control',
23                 }
24             ),
25         }
26
```

En este ejemplo definimos un formulario llamado `PersonaRegisterForm` que se utilizará para crear y actualizar instancias del modelo `Persona`.

A la clase Meta de este formulario le debemos indicar con que modelo debe trabajar, esto lo hacemos mediante el atributo `model = Persona`, le estamos indicando que trabajara sobre el modelo `Persona`.

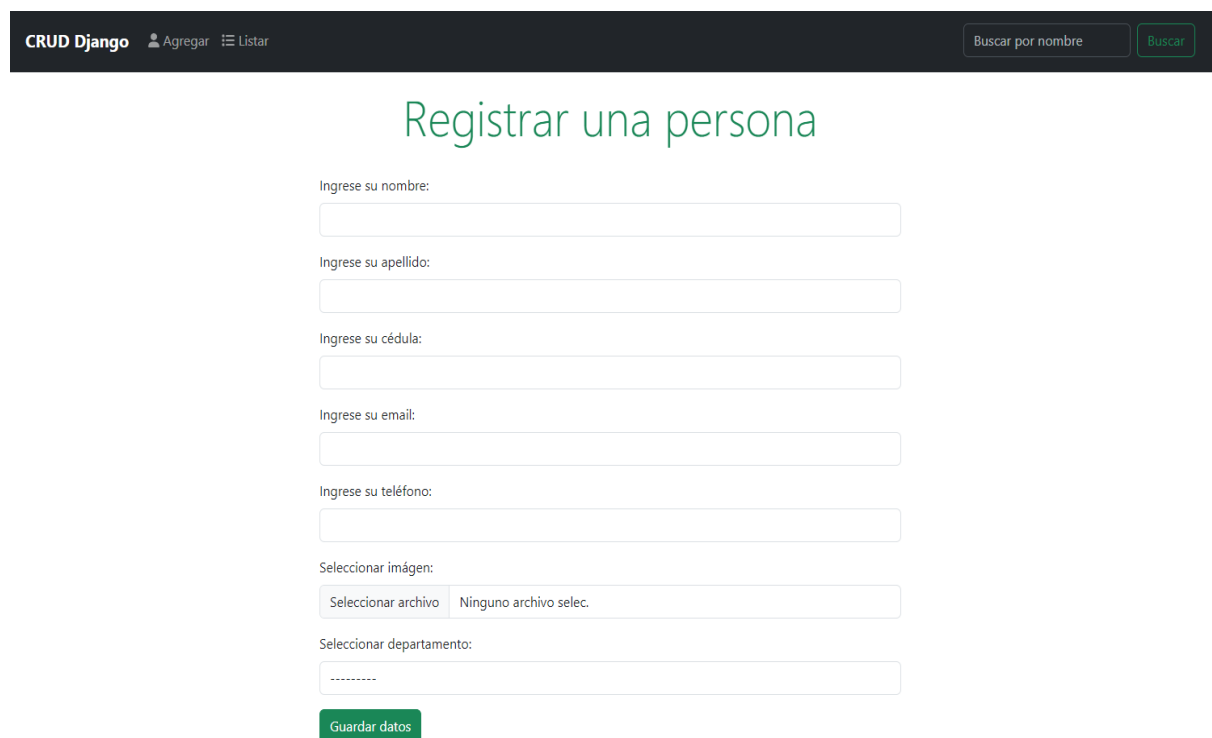
El atributo “`fields`” es una tupla que especifica los campos del modelo que se incluirán en el formulario, estos campos son los que el usuario podrá ver y completar en el formulario HTML generado.

Los “`widgets`” son un diccionario que contiene las configuraciones adicionales para los campos del formulario, en este caso se especifica que el campo “`departamento`” utilizara un campo de selección (etiqueta `<select>` de HTML), y se le especifica la clase CSS que debe aplicar a dicha etiqueta, en este caso la clase “`form-control`”.

En la documentación oficial tenemos muchas más opciones para trabajar con formularios:

<https://docs.djangoproject.com/es/5.0/topics/forms/>

Si accedemos a la URL de nuestra aplicación para visualizar el formulario que se generará a partir de esta configuración veremos lo siguiente:



The screenshot shows a web application interface with a dark header bar. On the left, it says 'CRUD Django' followed by 'Agregar' and 'Listar' with a hamburger menu icon. On the right, there are two buttons: 'Buscar por nombre' and 'Buscar'. Below the header, the main content area has a title 'Registrar una persona' in green. The form consists of several input fields with labels: 'Ingrese su nombre:', 'Ingrese su apellido:', 'Ingrese su cédula:', 'Ingrese su email:', 'Ingrese su teléfono:', 'Seleccionar imagen:', and 'Seleccionar departamento:'. The 'Seleccionar imagen:' field has two buttons: 'Seleccionar archivo' and 'Ninguno archivo selec.'. The 'Seleccionar departamento:' field has a dropdown menu showing '.....'. At the bottom of the form is a green button labeled 'Guardar datos'.

Y el template HTML correspondiente a este caso sería (create.html):

```
create.html •
env > personas > templates > persona > create.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block css %}
6  {% endblock %}
7
8  {% block content %}
9
10     <h1 class="display-4 mt-3 text-center text-success mb-3">Registrar una persona</h1>
11
12     <form method="POST" class="p-3 w-50 mx-auto enctype="multipart/form-data">
13         {% csrf_token %}
14
15         <section class="mb-3">
16             <label for="id_nombre" class="form-label">Ingrese su nombre:</label>
17             <input type="text" name="nombre" class="form-control" required id="id_nombre">
18         </section>
19
20         <section class="mb-3">
21             <label for="id_apellido" class="form-label">Ingrese su apellido:</label>
22             <input type="text" name="apellido" maxlength="20" class="form-control" required id="id_apellido">
23         </section>
24
25         <section class="mb-3">
26             <label for="id_cedula" class="form-label">Ingrese su cédula:</label>
27             <input type="text" name="cedula" maxlength="12" class="form-control" required id="id_cedula">
28         </section>
29
30         <section class="mb-3">
31             <label for="id_email" class="form-label">Ingrese su email:</label>
32             <input type="email" name="email" maxlength="254" class="form-control" required id="id_email">
33         </section>
34
35         <section class="mb-3">
36             <label for="id_telefono" class="form-label">Ingrese su teléfono:</label>
37             <input type="text" name="telefono" maxlength="15" class="form-control" required id="id_telefono">
38         </section>
39
40         <section class="mb-3">
41             <label for="id_imagen" class="form-label">Seleccionar imagen:</label>
42             <input type="file" name="imagen" class="form-control" accept="image/*" id="id_imagen">
43         </section>
44
45         <section class="mb-3">
46             <label for="id_departamento" class="form-label">Seleccionar departamento:</label>
47             {{ form.departamento }}
48         </section>
49
50         <input type="submit" value="Guardar datos" class="btn btn-success">
51     </form>
52
53 {% endblock %}
54
```

Notar que este template ya hace uso de la herencia de templates heredando el template base.html, así como también utiliza la etiqueta `{% load static %}` la cual posibilita utilizar las clases de Bootstrap en la definición del formulario html.

Archivo `urls.py` del proyecto

El archivo `urls.py` del proyecto de Django sirve como el enrutador principal para dirigir las solicitudes de URLs entrantes por parte del cliente, a las vistas correspondientes en las aplicaciones.

En este archivo se definen las rutas URL que nuestro proyecto va a manejar, esto se hace utilizando la función `urlpatterns` y el módulo `path()` ó `re_path()` para definir las rutas URL junto con las vistas que manejarán esas URL, sería como la asociación de “url – vista”. Hay que recordar que las vistas estarán definidas en el archivo `views.py` de cada aplicación creada en el proyecto.

Por lo tanto, cada ruta URL definida en el archivo `urls.py` se mapea a una vista específica del proyecto, esto se logra especificando una función de vista o una vista basada en clases junto con la ruta URL correspondiente.

Cuando se accede a una URL específica el enrutador busca la URL correspondiente en el archivo `urls.py` y dirige la solicitud a la vista asociada.

El archivo `urls.py` del proyecto sirve como punto centralizado para definir todas las rutas URL del proyecto, esto facilita la organización y la gestión de las rutas URL del proyecto.

Al definir las rutas URL en el archivo `urls.py` estamos separando la lógica de enrutamiento de la lógica de las vistas, esto promueve una arquitectura limpia y modular en la que las rutas URL y las vistas están claramente separadas y se pueden gestionar de forma independiente.

En resumen, el archivo `urls.py` del proyecto sirve como enrutador principal para dirigir las solicitudes de URL entrantes a las vistas correspondientes en la aplicación y proporciona un medio centralizado y organizado para definir y gestionar dichas URL en el proyecto.

En nuestro proyecto de ejemplo, CRUD de Personas tenemos el siguiente archivo `urls.py` del proyecto:

```
urls.py X
env > personas > personas > urls.py > ...
1  """
2  URL configuration for personas project.
3
4  The `urlpatterns` list routes URLs to views. For more information please see:
5  |   https://docs.djangoproject.com/en/5.0/topics/http/urls/
6  | Examples:
7  | Function views
8  |     1. Add an import: from my_app import views
9  |     2. Add a URL to urlpatterns: path('', views.home, name='home')
10 | Class-based views
11 |     1. Add an import: from other_app.views import Home
12 |     2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
13 | Including another URLconf
14 |     1. Import the include() function: from django.urls import include, path
15 |     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
16 | """
17 from django.contrib import admin
18 from django.urls import path, re_path, include
19 from django.conf.urls.static import static
20 from django.conf import settings
21
22 urlpatterns = [
23     path('admin/', admin.site.urls),
24     re_path('', include('applications.persona.urls')),
25 ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
26
```

En nuestro archivo `urls.py` del proyecto tenemos la importación de los módulos necesarios para definir las URL necesarias de forma correcta de acuerdo con el proyecto realizado.

La variable “`urlpatterns`” es la lista que define las URLs que nuestro proyecto Django podrá manejar.

La primera ruta definida “`path('admin', admin.site.urls)`” es una ruta definida por Django, y lo que está definiendo es la ruta URL para acceder a la interfaz de administración de Django cuando accedemos a la URL `http://dominio/admin` (tema que veremos más adelante).

Luego tenemos una segunda definición de una ruta URL, `“re_path(‘ ‘, include(applications.persona.urls)’”`, acá utilizamos el módulo `“re_path()”` para incluir las rutas URL de nuestra aplicación `“persona”` del proyecto, indicando que existe un archivo `urls.py` de dicha aplicación que se encuentra dentro del directorio `“applications”`, o sea, lo que le estamos diciendo a Django es que incluya un archivo `urls.py` de una aplicación llamada `“persona”` en las rutas del proyecto, y para ello le estamos pasando la ubicación a dicho archivo.

La cadena vacía `‘ ‘` indica que estas rutas serán manejadas por las rutas definidas en el archivo `urls.py` de la aplicación `“persona”`. Por lo tanto, cuando se acceda a una URL que coincida con las rutas URL definidas en `applications.persona.urls`, se dirigirá a las vistas correspondientes de esa aplicación.

Al final de la lista `“urlpatterns”` tenemos una ruta para servir archivos de medios durante el desarrollo.

La ruta `“+ static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)”` especifica la URL base para los archivos de medios cargados por el usuario, como imágenes, pdfs, etc., mediante el uso de la función `“static()”` en conjunto con la configuración `“MEDIA_URL”` y `“MEDIA_ROOT”`, variables definidas en el archivo `settings.py` del proyecto.

`MEDIA_URL` especifica la URL base para los archivos de medios, por ejemplo, si `“MEDIA_URL”` se establece en `“/media/”`, los archivos de medios se servirán desde la URL `http://dominio/media/`

`MEDIA_ROOT` especifica la ruta del sistema de archivos donde se almacenan los archivos de medios, este es el directorio donde se guardarán físicamente los archivos cargados por el usuario.

Al utilizar la función `static()`, se especifica que las URL que coincidan con `MEDIA_URL` se manejarán y servirán los archivos almacenados en `MEDIA_ROOT`.

Algo importante recordar es que esta configuración es adecuada para entornos de desarrollo, en un entorno de producción, es recomendable configurar un servidor web adecuado para servir archivos de medios de manera eficiente y segura.

En resumen, el archivo `urls.py` del proyecto define las rutas URL que el proyecto Django puede manejar y especifica como se deben dirigir las solicitudes entrantes a las vistas correspondientes en el proyecto y en las aplicaciones incluidas, también configura el manejo de archivos de medios para el desarrollo.

Analicemos ahora el archivo `urls.py` de la aplicación Persona.

```
urls.py x
env > personas > applications > persona > urls.py > ...
1  from django.urls import path
2  from . import views
3
4  app_name = 'persona_app'
5
6  urlpatterns = [
7      path('', views.IndexView.as_view(), name='index'),
8      path('listar/', views.Listar.as_view(), name='listar'),
9      path('detalle/<pk>/', views.Detalle.as_view(), name='detalle'),
10     path('create/', views.Create.as_view(), name='create'),
11     path('update/<pk>/', views.Update.as_view(), name='update'),
12     path('delete/<pk>/', views.Delete.as_view(), name='delete'),
13     path('buscar/', views.Buscar.as_view(), name='buscar'),
14 ]
15
```

Este archivo `urls.py` de la aplicación Persona, es creado por nosotros para gestionar las URL de dicha aplicación. Empezamos importando los módulos necesarios para la definición de las URLs, entre ellos tenemos la importación de todas las clases que estén en el archivo `views.py`

La variable “`app_name`” especifica el nombre de la aplicación, se utiliza para evitar conflictos de nombres de URL cuando se incluyen las URL de la aplicación en el proyecto principal.

Luego tenemos la definición de la lista “`urlpatterns`”, lista utilizada para definir las rutas de las URL de nuestra aplicación Persona. Tenemos varias URLs definidas y a cada una de ellas le asociamos una clase definida en el archivo `views.py`, para esto utilizamos el nombre de la clase que tenemos definida, y también a cada URL le definimos un nombre mediante el parámetro “`name`”.

En algunas de las URLs definidas vemos que la URL espera recibir un parámetro junto a su ruta, por ejemplo en

“`path('detalle/<pk>/', views.Detalle.as_view(), name='detalle')`”

esto significa que la clase a la cual la estamos asociando en el archivo `views.py` necesita recibir un identificador único para su funcionamiento.

En este ejemplo la URL para acceder desde el navegador para ver el detalle de una persona sería <http://localhost:8000/detalle/3>

Según estas URLs definidas en este archivo la forma de acceder a ellas mediante un navegador web serian, por ejemplo:

```
http://localhost:8000/  
http://localhost:8000/listar  
http://localhost:8000/detalle/3  
http://localhost:8000/create  
http://localhost:8000/update/2  
http://localhost:8000/delete/1  
http://localhost:8000/buscar
```

A su vez cada una de ellas está asociada a una clase definida en nuestro archivo `views.py` por lo tanto las solicitudes recibidas por dichas URLs serán derivadas a la clase correspondiente para su procesamiento, a continuación, detallaremos el funcionamiento de las clases definidas en el archivo `views.py` de la aplicación `Persona`.

En resumen, el archivo `urls.py` de la aplicación “`Persona`” define las rutas URL para diversas funcionalidades de la aplicación, como mostrar listas, detalles, creación, actualización, eliminación y búsqueda de objetos de la base de datos, y las asigna a las vistas correspondientes definidas en el archivo `views.py`. Cada ruta URL está asociada a un nombre para facilitar su referencia en el código y en las plantillas html.

Vistas (views.py)

El archivo views.py cumple la función de definir las funciones o clases de vista que manejan las solicitudes entrantes del cliente en nuestra aplicación, y retornan las respuestas adecuadas.

Las vistas son el componente de una aplicación web que procesa la lógica de negocio y determina que respuesta se envía de vuelta al navegador del usuario.

En Django existen dos enfoques para manejar las vistas, vistas basadas en funciones (VBF) y vistas basadas en clases (VBC).

Dependiendo de los requerimientos a implementar en nuestro proyecto podremos optar por una u otra forma de definir nuestras vistas, en el ejemplo que venimos manejando utilizaremos Vistas Basadas en Clases.

Las VBC generalmente heredan de `django.views.generic` los templates necesarios según sea el caso, por ejemplo si queremos listar datos utilizaremos un `ListView`, si queremos crear una instancia de un objeto utilizaremos `CreateView`, si queremos eliminar un registro utilizaremos un `DeleteView`, si queremos actualizar un registro utilizaremos un `UpdateView`, si queremos los detalles de un objeto utilizaremos un `DetailView`, etc.

Estos templates ya están definidos en Django y simplemente tenemos que importarlos para poder utilizarlos:

```
from django.views.generic import TemplateView, ListView, DetailView, CreateView, UpdateView, DeleteView
```

Para ver en detalle las vistas que podemos importar y utilizar, así como los métodos que contiene cada una podemos ir a la documentación en la siguiente url:

<https://ccbv.co.uk/>

ccbv.co.uk Django 5.0 Auth Mixins Views Generic Base Dates Detail Edit List

Classy Class-Based Views.

Detailed descriptions, with full methods and attributes, for each of Django's class-based generic views.

Did you know?
ccbv.co.uk/ClassName/ will take you straight to the class you're looking for.

Start Here for Django 5.0.

AUTH VIEWS
[LoginView](#)
[LogoutView](#)
[PasswordChangeDoneView](#)
[PasswordChangeView](#)
[PasswordResetCompleteView](#)
[PasswordResetConfirmView](#)
[PasswordResetDoneView](#)
[PasswordResetView](#)

GENERIC BASE
[RedirectView](#)
[TemplateView](#)
[View](#)

GENERIC DATES
[ArchiveIndexView](#)
[DateDetailView](#)
[DayArchiveView](#)
[MonthArchiveView](#)
[TodayArchiveView](#)
[WeekArchiveView](#)
[YearArchiveView](#)

GENERIC DETAIL
[DetailView](#)

GENERIC EDIT
[CreateView](#)
[DeleteView](#)
[FormView](#)
[UpdateView](#)

GENERIC LIST
[ListView](#)

What are class-based views anyway?

Django's class-based generic views provide abstract classes implementing common web development tasks. These are very powerful, and heavily-utilise Python's object orientation and multiple inheritance in order to be extensible. This means they're more than just a couple of generic shortcuts — they provide utilities which can be mixed into the much more complex views that you write yourself.

Great! So what's the problem?

All of this power comes at the expense of simplicity. For example, trying to work out exactly which method you need to customise, and what its keyword arguments are, on your `UpdateView` can feel a little like wading through spaghetti — it has 10 separate ancestors (plus `object`), spread across 3 different python files. This site shows you exactly what you need to know.

How does this site help?

To make things easier, we've taken all the attributes and methods that every view defines or inherits, and flattened all that information onto one comprehensive page per view. Check out [UpdateView](#), for example.

En nuestro proyecto la primera vista que tenemos es la siguiente:

```
# Página principal
class IndexView(TemplateView):
    template_name = 'persona/index.html'
```

Esta vista hace referencia a la página principal y retorna un template llamado “index.html” que se encuentra dentro del directorio “persona” que a su vez está dentro del directorio “templates”.

Definimos una clase llamada `IndexView` que hereda de `TemplateView` que es una de las vistas basadas en clases proporcionadas por Django y esta diseñada para renderizar una plantilla específica.

Esto significa que `IndexView` obtiene todas las funcionalidades de `TemplateView` y puede agregar o modificar comportamientos según sea necesario.

La variable “`template_name`” indica la ruta de la plantilla html que se utilizará para renderizar la respuesta y su nombre de archivo es `index.html`.

Entonces cuando un usuario acceda a la URL asociada a esta vista, Django renderizará la plantilla html `index.html` y devolverá el resultado al navegador del cliente.

Es una de las vistas más simples de implementar.

Vista para listar:

```
class Listar(ListView):
    template_name = 'persona/listar.html'
    paginate_by = 7
    model = Persona
    context_object_name = 'personas'
    ordering = ('-id')
```

Esta clase llamada Listar hereda de la clase `ListView`, que es una vista proporcionada por Django y esta diseñada para mostrar una lista de objetos de un modelo particular en una plantilla html.

El atributo “`template_name`” establece la ruta de la plantilla que se utilizará para renderizar la respuesta a la petición del cliente, en este caso la plantilla se encuentra en el directorio “`persona`” que a su vez está dentro del directorio “`templates`” y el nombre del archivo es `listar.html`

El atributo “`paginate_by`” es un paginador, establece una forma de paginar los resultados, por ejemplo, si la respuesta obtenida de la base de datos trae 1000 registros, este paginador establece que se mostraran de a 7 registros por página. Para utilizarlo también se debe crear el código html que defina el paginador html.

Cuando utilizamos el `ListView` debemos indicarle cual es el modelo con el cual queremos trabajar, esto lo hacemos con el atributo “`model`”, y en este caso le indicamos que el modelo es `Persona`, por lo tanto, obtendrá todos los registros del modelo `Persona`.

La variable “`context_object_name`” indica el nombre con el que los objetos del modelo `Persona` se pasarán al contexto de la plantilla html, dicho de otra forma, es el nombre de la variable a la cual debemos acceder desde la plantilla html para obtener los datos de los registros.

El atributo “`ordering`” establece el orden de los registros, le indicamos un campo del modelo `Persona` por el cual debe ordenar los registros para listarlos posteriormente.

En este caso los objetos `Persona` se ordenarán por ID en orden descendente, del más reciente al más antiguo.

En resumen, la vista renderiza la plantilla “listar.html” y mostrará una lista paginada de a 7 registros de objetos del modelo Persona, ordenados por el campo ID en orden descendente, y los objetos serán accesibles desde el template html mediante la variable llamada “personas”

Para este caso el template `listar.html` es el siguiente:

```
listar.html X
env > personas > templates > persona > listar.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block title %}
6  CRUD Django - Listado
7  {% endblock %}
8
9  {% block content %}
10
11  <h1 class="text-center display-4 text-success mt-3"><i class="fa-solid fa-users"></i> Listado de personas</h1>
12
13  <section class="p-3">
14    <table class="table mt-3">
15      <thead class="text-center table-dark">
16        <tr>
17          <td><i class="fa-solid fa-hashtag"></i> ID</td>
18          <td><i class="fa-solid fa-file-signature"></i> Nombre y Apellido</td>
19          <td><i class="fa-solid fa-envelope"></i> Email</td>
20          <td><i class="fa-solid fa-phone"></i> Teléfono</td>
21          <td><i class="fa-solid fa-gear"></i> Acciones</td>
22        </tr>
23      </thead>
24      <tbody class="text-center">
25        {% for p in personas %}
26          <tr>
27            <td>{{p.id}}</td>
28            <td>{{p.nombre}} {{p.apellido}}</td>
29            <td>{{p.email}}</td>
30            <td>{{p.telefono}}</td>
31            <td>
32              <a href="{% url 'persona_app:dDetalle' p.id %}"><span class="badge bg-primary p-2">VER</span></a>&nbsp;
33              <a href="{% url 'persona_app:update' p.id %}"><span class="badge bg-warning p-2">EDITAR</span></a>&nbsp;
34              <a href="{% url 'persona_app:delete' p.id %}"><span class="badge bg-danger p-2">ELIMINAR</span></a>&nbsp;
35            </td>
36          </tr>
37        {% endfor %}
38      </tbody>
39    </table>
40  </section>
41
42  {% include 'persona/paginator.html' %}
43
44  {% endblock %}
```

Al acceder a la URL en el navegador obtenemos lo siguiente:


CRUD Django

Agregar

Listar

Buscar por nombre

Buscar

 Listado de personas

# ID	Nombre y Apellido	Email	Teléfono	Acciones
13	Pierina Rodríguez	pierina@gmail.com	099121212	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
11	Ximena Silvera	ximena@gmail.com	099112233	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
10	Facundo Gonzalez	facundo@gmail.com	099999999	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
9	Gonzalo Silva	gonzalo@gmail.com	099888888	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
8	Maria Gonzalez	maria@gmail.com	099777777	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
7	Mikaela Rodriguez	mikaela@gmail.com	099666666	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>
6	Pablo Perez	pablo@gmail.com	099555555	<div>VER</div> <div>EDITAR</div> <div>ELIMINAR</div>

<

1

2

>

Vista para ver los detalles de una Persona:

```
class Detalle(DetailView):  
    model = Persona  
    template_name = 'persona/detalle.html'
```

En este caso para acceder al detalle de un objeto Persona, se crea la clase Detalle que hereda de `DetailView`, que es una clase proporcionada por Django y esta diseñada para mostrar los detalles de un objeto específico de un modelo en una plantilla html.

Se establece mediante el atributo “`model`” cuál es el modelo con el cual se va a trabajar para recuperar el objeto específico que se mostrara en detalle, en este caso el modelo Persona.

Algo importante es que en la URL que se matchea con esta vista se deberá pasar el ID del objeto al cual se desea acceder, así como también enviar ese ID desde el template HTML para que sea procesado posteriormente y se pueda acceder al objeto, todo la búsqueda y retorno de ese objeto se encarga automáticamente la clase `DetailView` de la cual estamos heredando.

El atributo “`template_name`” al igual que lo mencionado anteriormente indica la ruta al template que se deberá renderizar y el nombre de la plantilla html.

De esta forma cuando un usuario acceda a la URL que vendrá acompañada del ID de un objeto del modelo Persona, Django recuperará el objeto correspondiente de la base de datos y lo pasará a la plantilla “detalle.html”, desde la plantilla se podrá acceder al objeto Persona y mostrar sus campos.

Como en este caso no estamos definiendo la variable de contexto para la plantilla html “context_object_name”, el nombre del contexto para acceder al objeto es “object”.

La plantilla detalle.html es la siguiente:

```
views.py  detalle.html x
env > personas > templates > persona > detalle.html > ...
4
5 {% block css %}
6   <style>
7     img{
8       max-width: 50%;
9     }
10  </style>
11 {% endblock %}
12
13 {% block content %}
14
15   <h4 class="display-4 fw-bold mt-5 text-center mb-3">Datos de <span class="text-success">{{object.nombre}} {{object.apellido}}</span></h4>
16
17   <section class="p-5 mx-auto w-50">
18
19     <section class="mb-5 text-center">
20       
25     </section>
26
27     <section class="mb-3">
28       <label class="form-label">Fecha de registro: </label>
29       <input type="text" value="{{object.created}}" class="form-control" disabled>
30     </section>
31
32     <section class="mb-3">
33       <label class="form-label">Cédula: </label>
34       <input type="text" value="{{object.cedula}}" class="form-control" disabled>
35     </section>
36
37     <section class="mb-3">
38       <label class="form-label">Email: </label>
39       <input type="text" value="{{object.email}}" class="form-control" disabled>
40     </section>
41
42     <section class="mb-3">
43       <label class="form-label">Teléfono de contacto: </label>
44       <input type="text" value="{{object.telefono}}" class="form-control" disabled>
45     </section>
46
47     <section class="mb-3">
48       <label class="form-label">Trabaja en: </label>
49       <input type="text" value="{{object.departamento}}" class="form-control" disabled>
50     </section>
51
52     <section class="text-center">
53       <!-- Retorna a la pagina anterior, utiliza JS para volver atrás -->
54       <a id="btnVolver" class="btn btn-outline-primary mt-5"><i class="fa-solid fa-backward"></i> Volver</a>
55     </section>
56   </section>
57 {% endblock %}
```

Si accedemos a la URL de detalle pasándole un ID de un objeto Persona, obtenemos la siguiente pantalla:

CRUD Django


Agregar

Listar

Buscar por nombre

Buscar

Datos de Maria Gonzalez



Fecha de registro:

26 de febrero de 2024 a las 20:10

Cédula:

7777777

Email:

maria@gmail.com

Teléfono de contacto:

099777777

Trabaja en:

Gerencia Gral.

Vista para el registro de Personas

```
class Create(CreateView):
    template_name = 'persona/create.html'
    model = Persona

    #Si deseo utilizar la personalizacion del formulario en forms.py
    form_class = PersonaRegisterForm
    success_url = reverse_lazy('persona_app:listar')
```

Esta vista que hereda de la clase “CreateView” que también es proporcionada por Django, y está diseñada para manejar la creación de nuevos objetos de un modelo en la base de datos.

Se le indica a la clase definida “Create” cuál es el modelo con el cual se desea trabajar, en este caso el modelo Persona, mediante la variable “model”, también le indicamos cual será la ruta a la plantilla que se utilizará para renderizar el formulario de creación del objeto Persona.

La variable “form_class” le indica a la vista que utilizará la clase de formulario personalizado llamada “PersonaRegisterForm”, clase que hemos implementado en el archivo forms.py y que importamos en las vistas para poder utilizar, esta clase será quien procese los datos del formulario de creación del objeto Persona.

Utilizar la variable “`form_class`” es una forma de personalizar el formulario utilizado en la vista.

Esta vista debe indicar cual será la página a la cual redireccionar luego de la creación correcta del objeto Persona, dicho de otra forma, luego de completar el formulario html con los datos correctos, y enviarlo al servidor para su procesamiento, se puede redirigir al usuario a una URL del sistema, en caso de error en los datos, comúnmente se mantiene al usuario en la URL del formulario y se le muestran los errores.

Para redirigir al usuario a una URL luego de la correcta creación del objeto Persona, utilizamos la variable “`success_url`”.

En este ejemplo se utiliza dicha variable con la función “`reverse_lazy`”, la cual es utilizada para resolver la URL inversa de manera diferida, lo que garantiza que la resolución de la URL se realice después de que se cargue todo el módulo de URLs de la aplicación. Cuando definimos las URLs en el archivo `urls.py`, le asignamos un nombre a cada URL con el parámetro “`name`”, esto permite referenciar esas URLs por su nombre en lugar de por su ruta, “`reverse_lazy`”, utiliza el nombre de la aplicación definido en el archivo `urls.py` y el nombre de la URL para hacer la redirección.

Cabe mencionar que “`reverse_lazy`” es una función de Django que debe importarse desde `django.urls`

En resumen, esta vista “`Create`” renderiza la plantilla “`create.html`” que contiene un formulario html de creación para el modelo Persona, cuando el usuario completa correctamente el formulario y lo envía, los datos se procesarán utilizando el formulario personalizado definido en `forms.py` “`PersonaRegisterForm`”, y si la creación del objeto es exitosa, el usuario será redirigido a la lista de personas a través de la función `reverse_lazy`.

La plantilla create.html seria la siguiente:

```
create.html X
env > personas > templates > persona > create.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block css %}
6  {% endblock %}
7
8  {% block content %}
9
10     <h1 class="display-4 mt-3 text-center text-success mb-3">Registrar una persona</h1>
11
12     <form method="POST" class="p-3 w-50 mx-auto" enctype="multipart/form-data">
13         {% csrf_token %}
14
15         <section class="mb-3">
16             <label for="id_nombre" class="form-label">Ingrese su nombre:</label>
17             <input type="text" name="nombre" class="form-control" required id="id_nombre">
18         </section>
19
20         <section class="mb-3">
21             <label for="id_apellido" class="form-label">Ingrese su apellido:</label>
22             <input type="text" name="apellido" maxlength="20" class="form-control" required id="id_apellido">
23         </section>
24
25         <section class="mb-3">
26             <label for="id_cedula" class="form-label">Ingrese su cédula:</label>
27             <input type="text" name="cedula" maxlength="12" class="form-control" required id="id_cedula">
28         </section>
29
30         <section class="mb-3">
31             <label for="id_email" class="form-label">Ingrese su email:</label>
32             <input type="email" name="email" maxlength="254" class="form-control" required id="id_email">
33         </section>
34
35         <section class="mb-3">
36             <label for="id_telefono" class="form-label">Ingrese su teléfono:</label>
37             <input type="text" name="telefono" maxlength="15" class="form-control" required id="id_telefono">
38         </section>
39
40         <section class="mb-3">
41             <label for="id_imagen" class="form-label">Seleccionar imagen:</label>
42             <input type="file" name="imagen" class="form-control" accept="image/*" id="id_imagen">
43         </section>
44
45         <section class="mb-3">
46             <label for="id_departamento" class="form-label">Seleccionar departamento:</label>
47             {{ form.departamento }}
48         </section>
49
50         <input type="submit" value="Guardar datos" class="btn btn-success">
51     </form>
52
53 {% endblock %}
54
```

Si accedemos mediante un navegador web, el formulario sería el siguiente:

The screenshot shows a web application interface with a dark header. On the left, it says 'CRUD Django' followed by 'Agregar' and 'Listar' with icons. On the right, there are two buttons: 'Buscar por nombre' and 'Buscar'. The main content area has a title 'Registrar una persona' in green. Below the title are several form fields: 'Ingrese su nombre:', 'Ingrese su apellido:', 'Ingrese su cédula:', 'Ingrese su email:', 'Ingrese su teléfono:', 'Seleccionar imagen:' (with a file selection button and text 'Ninguno archivo selec.'), and 'Seleccionar departamento:' (with a dropdown menu). At the bottom is a green button labeled 'Guardar datos'.

Si prestamos atención al código HTML que define este formulario, y también cada vez que tengamos un formulario en Django que envíe datos al servidor, veremos que se define un token dentro del formulario html mediante el tag `{% csrf_token %}`

¿Qué es este token, que función cumple?

El token **CSRF** (Cross Site Request Forgery), es una medida de seguridad para proteger las aplicaciones web contra ataques CSRF. El objetivo del token es asegurarse de que las solicitudes HTTP provengan de fuentes legítimas y no de sitios maliciosos que intenten realizar acciones no autorizadas en nombre del usuario.

El tag `{% csrf_token %}` en los formularios Django se utiliza para incluir un campo oculto que contiene un token CSRF único en cada formulario generado, este token se utiliza para validar que la solicitud HTTP proviene de un formulario generado por la propia aplicación y no ha sido manipulado por un atacante.

Cuando el usuario envía el formulario, el navegador incluirá este campo oculto junto con los datos del formulario en la solicitud HTTP, luego Django verificará que el valor del token CSRF en la solicitud coincida con el valor esperado, que se almacena en la sesión del usuario. Si los valores coinciden, la solicitud se considera válida y se procesa normalmente, si no coinciden, Django rechazará la solicitud y mostrará un error de CSRF.

Eliminar un objeto Persona

```
class Delete(DeleteView):  
    template_name = 'persona/eliminar.html'  
    model = Persona  
    success_url = reverse_lazy('persona_app:listar')
```

La clase definida “Delete” que hereda de la clase “DeleteView” proporcionada por Django, esta diseñada para manejar la eliminación de un objeto de la base de datos.

La clase `DeleteView` de Django proporciona una implementación simple para mostrar un formulario de confirmación de eliminación y realizar la posterior eliminación del objeto de la base de datos si el usuario confirma la acción.

La clase comienza definiendo la variable “`template_name`” que indica la ruta a la plantilla que se utilizará para renderizar el formulario de confirmación para la eliminación del objeto Persona en la base de datos.

La variable “`model`” es quién establece el modelo con el cual se va a trabajar en esta vista, en este caso el modelo es `Persona`, por lo cual estaremos eliminando un objeto Persona.

En este caso también tenemos la definición de la variable “`success_url`” la cuál indicara a que URL redirigir al usuario en caso de que la eliminación del objeto Persona de la base de datos sea correcta.

También utiliza la función “`reverse_lazy`” que se explicó en la vista de creación de objetos mencionada anteriormente, la clase `Create`. De forma resumida, esta función utiliza el nombre de la aplicación definida en el archivo `urls.py` y el nombre de la URL definido también en el archivo `urls.py` para hacer la redirección al usuario.

Al igual que la clase `DetailView` de Django, esta clase `DeleteView`, necesita recibir un identificador único del objeto que se desea eliminar. Por lo general, esto se hace a través de la URL al incluir el parámetro de la primary key (PK) en la ruta de la URL, la cuál esta definida en la plantilla html que mostrará el enlace para acceder a la eliminación del objeto Persona.

Esto hace necesario que cuando se defina en el archivo `urls.py` el matcheo entre la URL y la clase de la vista en `views.py` se tenga que definir que la URL espera recibir también el parámetro de identificador único para el objeto Persona.

En resumen, la clase “Delete” es una vista que hereda de la clase “DeleteView” de Django para manejar la eliminación de objetos del modelo Persona, debe recibir como parámetro desde la URL el identificador único del objeto Persona para procesar su eliminación, renderizará un formulario de confirmación de eliminación utilizando la plantilla “eliminar.html” y redirige si la eliminación del objeto es correcta utilizando la función “reverse_lazy” a la URL de la lista de personas.

La plantilla eliminar.html es la siguiente:



```
eliminar.html X
env > personas > templates > persona > eliminar.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block css %}
6  {% endblock %}
7
8  {% block content %}
9
10     <h1 class="display-4 mt-3 text-center text-danger mb-3">Eliminar los datos de:
11     <span class="fw-bold">{{object.nombre}} {{object.apellido}}</span>
12     </h1>
13
14     <form method="POST" class="p-5 mx-auto text-center mb-5">
15         {% csrf_token %}
16         <p>¿Está seguro que desea eliminar el registro?</p>
17         <input type="submit" value="Sí, Confirmar" class="btn btn-danger mt-3">
18     </form>
19
20
21 {% endblock %}
22
```

El enlace para la URL de eliminación del objeto Persona está definido en el listado de personas, notar como se indica que debe pasar el identificador único para el objeto Persona en la URL:

```
<td>
  <a href="{% url 'persona_app:detalle' p.id %}"><span class="badge bg-primary p-2">VER</span></a>&nbsp;
  <a href="{% url 'persona_app:update' p.id %}"><span class="badge bg-warning p-2">EDITAR</span></a>&nbsp;
  <a href="{% url 'persona_app:delete' p.id %}"><span class="badge bg-danger p-2">ELIMINAR</span></a>&nbsp;
</td>
```

p.id = es la primary key del objeto persona

Si accedemos a la URL de eliminación el formulario de confirmación de eliminación podría verse de la siguiente manera:

CRUD Django  Agregar  Listar

Eliminar los datos de: **Pierina Rodriguez**

¿Está seguro que desea eliminar el registro?

Sí, Confirmar

Actualizar un objeto Persona

```
class Update(UpdateView):
    template_name = 'persona/update.html'
    model = Persona
    fields = [
        'nombre',
        'apellido',
        'cedula',
        'email',
        'telefono',
        'imagen',
        'departamento'
    ]
    success_url = reverse_lazy('persona_app:listar')

    def post(self, request, *args, **kwargs):
        self.object = self.get_object()
        return super().post(request, *args, **kwargs)

    def form_valid(self, form):
        return super(Update, self).form_valid(form)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        # Agregar context['form'] para agregar cualquier clase u otro atributo a los campos
        context['form'].fields['departamento'].widget.attrs['class'] = 'form-control'
        return context
```

La clase “Update” que hereda de la clase “UpdateView” proporcionada por Django esta diseñada para manejar la actualización de un objeto específico de un modelo de la base de datos.

La definición de la variable “`model`” indica cual es el modelo con el que se desea trabajar, en este caso se espera que la vista maneje objetos del modelo `Persona`.

La variable “`template_name`” especifica la plantilla que se utilizará para renderizar el formulario de actualización del objeto `Persona`, en este caso la plantilla se llama “`update.html`”

Luego tenemos la definición de la variable “`fields`”, que está definida como una lista, esta lista especifica que campos del modelo `Persona` estarán disponibles en el formulario html para su actualización, sólo los campos del modelo que figuren en dicha lista serán los campos que el usuario podrá actualizar.

En caso de que se desee que el usuario pueda actualizar todos los campos del modelo, la variable “`fields`” quedara definida de la siguiente forma: `fields = '__all__'`

La variable “`success_url`” indica la URL a la que se redirigirá al usuario después de que la actualización del objeto `Persona` sea exitosa, en este caso esta definida para redirigir a la lista de `Personas`.

El método “`post`” se utiliza para manejar las solicitudes POST, más específicamente las solicitudes POST que lleguen a esta clase de actualización, recordar que el formulario html en el cual se actualizarán los campos del objeto `Persona`, tiene que tener definido el atributo `method="POST"`, el cual indicara la forma de enviar los datos al servidor, de allí que la clase defina este método para manejar estas solicitudes.

Este método “`post`” lo primero que hace es obtener el objeto específico que se esta actualizando, esto lo hace a través de la variable

```
self.object = self.get_object()
```

Luego llama al método “`post`” de la clase padre para procesar la solicitud de actualización, eso lo vemos en la siguiente línea:

```
return super().post(request, *args, **kwargs)
```

Los parámetros `request`, `*args`, `**kwargs`, son parámetros que deben estar definidos dado que la implementación que define Django así lo requiere.

Cabe recordar que, en este caso de actualización de un objeto de la base de datos, cuando se llame a esta vista hay que indicarle cual es el identificador único del objeto que se desea actualizar, esto se hace al definir el enlace en el formulario html y luego en el archivo `urls.py`, de esta manera la clase `Update` sabrá sobre qué modelo y objeto se está trabajando.

El método “`form_valid`”, es un método utilizado para implementar cierta lógica cuando los datos del formulario recibido son válidos, en este caso, simplemente se llama al método “`form_valid`” de la clase padre y se le pasan los parámetros que Django requiere.

El método en la clase padre (`UpdateView`) en Django se llama cuando el formulario enviado es válido después de las validaciones, se encarga de guardar los datos del formulario en la base de datos y de manejar cualquier acción adicional que se desee realizar después que el formulario haya sido validado correctamente. Generalmente, es la clase padre quien internamente utiliza el método “`save()`” para guardar los datos en la base de datos.

Método “`get_context_data`”, este método se utiliza para agregar datos adicionales al contexto de la plantilla html antes que la misma sea renderizada y se muestre al usuario.

Su propósito es proporcionar datos adicionales al contexto que pueden ser útiles para la presentación o el funcionamiento de la interfaz de usuario. En este caso, se utiliza para agregar una clase de CSS a un campo específico del formulario html.

La variable “`context`” es un diccionario que se utiliza para pasar datos adicionales al contexto de la plantilla html, es la clase padre (`UpdateView`) quien tiene la implementación y sabe como acceder al contexto de la plantilla que ya se ha generado en la clase, luego de obtenerlo podemos agregar nuestros propios datos adicionales a dicho contexto.

En este ejemplo, se agrega la clase “`form-control`” de Bootstrap al widget del campo “departamento” del formulario, en el atributo “`class`” que tienen las etiquetas html.

En resumen, la clase `Update` es una vista que hereda de `UpdateView` para manejar la actualización de objetos del modelo `Persona`. Proporciona funcionalidad para mostrar un formulario de actualización, con los campos que hayamos definido, procesa la solicitud POST proveniente del formulario html, valida sus campos y agrega datos adicionales al contexto de la plantilla en caso de ser necesario.

En este caso la plantilla `update.html` es la siguiente:

```
update.html X
env > personas > templates > persona > update.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block css %}
6  {% endblock %}
7
8  {% block content %}
9      <h1 class="display-4 mt-3 text-center text-success mb-3">Actualizar datos de:
10      <span class="fw-bold">{{object.nombre}} {{object.apellido}}</span>
11      </h1>
12
13      <form method="POST" enctype="multipart/form-data" class="w-50 mx-auto p-3">
14          {% csrf_token %}
15
16          <section class="mb-3">
17              <label for="id_nombre" class="form-label">Nombre:</label>
18              <input type="text" name="nombre" value="{{object.nombre}}" class="form-control" maxlength="20" required id="id_nombre">
19          </section>
20
21          <section class="mb-3">
22              <label for="id_apellido" class="form-label">Apellido:</label>
23              <input type="text" name="apellido" value="{{object.apellido}}" class="form-control" maxlength="20" required id="id_apellido">
24          </section>
25
26          <section class="mb-3">
27              <label for="id_cedula" class="form-label">Cédula:</label>
28              <input type="text" name="cedula" value="{{object.cedula}}" class="form-control" maxlength="12" required id="id_cedula">
29          </section>
30
31          <section class="mb-3">
32              <label for="id_email" class="form-label">Email:</label>
33              <input class="form-control" type="email" name="email" value="{{object.email}}" maxlength="254" required id="id_email">
34          </section>
35
36          <section class="mb-3">
37              <label for="id_telefono" class="form-label">Teléfono:</label>
38              <input type="text" name="telefono" class="form-control" value="{{object.telefono}}" maxlength="15" required id="id_telefono">
39          </section>
40
41          <section class="mb-3">
42              <label for="id_imagen" class="form-label d-block">Seleccionar imagen:</label>
43              <input class="form-control" type="file" name="imagen" accept="image/*" id="id_imagen">
44              {% if object.imagen %}
45                  <p></p>
46              {% endif %}
47          </section>
48
49          <section class="mb-3">
50              <label for="id_departamento" class="form-label">Departamento:</label>
51              {{ form.departamento }}
52          </section>
53
54          <input type="submit" value="Actualizar Datos" class="btn btn-primary">
55      </form>
56  {% endblock %}
```

Para acceder a la actualización de un objeto Persona, lo hacemos a través de la lista de Personas, y el enlace para acceder, el cual también envía el identificador único del objeto Persona es el siguiente:

```
<td>
  <a href="{% url 'persona_app:detalle' p.id %}"><span class="badge bg-primary p-2">VER</span></a>&nbsp;
  <a href="{% url 'persona_app:update' p.id %}"><span class="badge bg-warning p-2">EDITAR</span></a>&nbsp;
  <a href="{% url 'persona_app:delete' p.id %}"><span class="badge bg-danger p-2">ELIMINAR</span></a>&nbsp;
</td>
```

Si accedemos a la URL para actualizar los datos de una Persona veremos lo siguiente:

Actualizar datos de: **Gonzalo Silva**

Nombre:

Apellido:

Cédula:

Email:

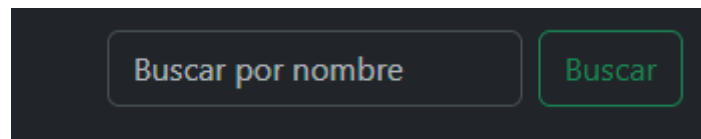
Teléfono:

Seleccionar imagen:

 Ninguno archivo selec.

Departamento:

Si prestamos atención en la página principal del proyecto, arriba a la derecha tenemos definido un buscador de Personas por nombre:



Este buscador de personas es lo que comúnmente se llama filtro, y también lleva su implementación en las vistas, y su definición de URL.

En nuestro ejemplo, en `views.py` lo tenemos definido de la siguiente manera:

```
class Buscar(ListView):
    """ Buscar persona por nombre, recibiendo el parámetro desde una caja de texto """
    template_name = 'persona/buscar.html'

    def get_queryset(self):
        palabra = self.request.GET.get("texto")
        lista = Persona.objects.filter(
            nombre__icontains = palabra
        )
        return lista
```

Llamamos a la clase “`Buscar`” y la misma hereda de la clase “`ListView`”, clase proporcionada por Django que esta diseñada para listar objetos de un modelo de la base de datos. En este caso será utilizada para mostrar una lista de Personas que coincidan con un término de búsqueda proporcionado por el usuario.

Le indicamos a la clase la ruta al template que debe renderizar y su nombre mediante la variable “`template_name`”, en este caso la plantilla se llama “`buscar.html`”, será el template que se renderizará con los resultados de la búsqueda realizada.

Tenemos definido el método “`get_queryset`”, el cual se utiliza para obtener el conjunto de objetos (`queryset`) que se mostrarán en la vista.

En este caso se realiza una búsqueda en la base de datos para encontrar todas las personas cuyo nombre contiene la palabra proporcionada por el usuario en el parámetro “`texto`”.

Este parámetro “`texto`” nos llega desde el campo de búsqueda del formulario html definido para tal fin, por lo tanto, debemos obtener el valor de dicho campo, y esto lo hacemos en la línea: `self.request.GET.get("texto")` y su contenido queda guardado en la variable “`palabra`”.

Notar que la solicitud (request) en este caso es una solicitud de tipo GET y no POST, este comportamiento esta definido en el formulario html como veremos más adelante.

Luego de obtenido el contenido del campo de texto del formulario html, definimos una variable llamada “`lista`” y la igualamos a una consulta a la base de datos.

La consulta es:

```
Persona.objects.filter(nombre__icontains = palabra)
```

En dicha línea le estamos indicando primero que el modelo al cual se consulta es el modelo Persona, con la palabra “`objects`” accedemos al modelo de consultas de Django el cual nos proporciona muchas funciones para utilizar, dicho modelo nos permite realizar operaciones comunes de base de datos, como crear, actualizar, recuperar y eliminar objetos del modelo asociado, en este caso, se utiliza el método `filter()` para filtrar los objetos del modelo Persona basados en ciertos criterios de búsqueda.

El criterio de búsqueda utilizado para filtrar los objetos del modelo Persona es un filtro que busca objetos donde el campo “nombre” contenga (`icontains`) el valor de la variable “palabra”, ignorando las mayúsculas y minúsculas, lo cual esta indicado por la letra “i” (insensitive) en la palabra “`icontains`”, en caso de no querer ignorar mayúsculas y minúsculas seria solo “`contains`”.

Por lo tanto para este filtro, “`objects`” es el administrador de consultas por defecto de los modelos en la base de datos, “`filter()`” es un método proporcionado por este administrador que permite filtrar objetos del modelo basado en ciertos criterios.

Luego de realizada la consulta cuyos resultados quedaran guardados en la variable “`lista`”, la misma se retorna utilizando la palabra reservada “`return`”

En resumen, la clase “Buscar” es una vista que hereda de “`ListView`” para mostrar un listado de personas que coinciden con un término de búsqueda proporcionado por el usuario, utiliza el método “`get_queryset`” para filtrar los resultados basados en el parámetro “texto” enviado en la solicitud GET del formulario html, y luego renderiza la plantilla buscar.html para mostrar los resultados de la búsqueda.

El formulario de búsqueda html está definido de la siguiente forma:

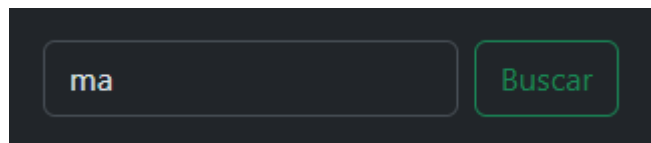
```
<form class="d-flex" role="search" method="GET" id="form" action="{% url 'persona_app:buscar' %}">
  {% csrf_token %}
  <input class="form-control me-2" id="texto" name="texto" type="text" placeholder="Buscar por nombre" aria-label="Search">
  <input class="btn btn-outline-success" type="submit" value="Buscar">
</form>
```

Notar que a la etiqueta `<form>` html se le indica el atributo `method="GET"` y se le define la acción mediante el atributo `action="{% url 'persona__app:buscar' %}"`

Le estamos indicando que los datos sean enviados mediante el método http GET, y que serán procesados por una URL cuyo nombre es “buscar”, dicha URL esta definida en el archivo `urls.py` y es quien matchea con la clase `Buscar` del archivo `views.py`

El campo de texto `<input>` tiene definido un `id="texto"` y un `name="texto"`, a través de estos atributos desde la vista accedemos al contenido ingresado por el usuario en dicho campo html.

Si realizamos una búsqueda, por ejemplo:



ma Buscar

Los resultados obtenidos que se muestran en la plantilla `buscar.html` serán:

Resultado de su búsqueda:

NOMBRE Y APELLIDO	DETALLES
Manuela Santos	VER
Marcos Pereira	VER
Maria Gonzalez	VER
Matias Santos	VER

En caso de no obtener resultados,

Resultado de su búsqueda:

No hay resultados para su búsqueda

La plantilla `buscar.html` está definida de la siguiente manera:

```
buscar.html X
env > personas > templates > persona > buscar.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block content %}
6
7      <h4 class="display-4 mt-3 text-center text-success mb-3">Resultado de su búsqueda:</h4>
8
9      <!-- if object_list|length > 0 -->
10     <section class="col-6 mx-auto mb-3">
11         {% if object_list %}
12             <table class="table">
13                 <thead class="text-center fw-bold text-uppercase">
14                     <tr>
15                         <th>Nombre y apellido</th>
16                         <th>Detalles</th>
17                     </tr>
18                 </thead>
19                 <tbody class="text-center">
20                     {% for p in object_list %}
21                         <tr>
22                             <td>{{ p }}</td>
23                             <td>
24                                 <a href="{% url 'persona_app:detalle' p.id %}" style="text-decoration: none;" class="badge bg-primary p-2">
25                                     VER
26                                 </a>
27                             </td>
28                         </tr>
29                     {% endfor %}
30                 </tbody>
31             </table>
32         {% else %}
33             <p class="alert alert-primary text-center w-75 fw-bold mx-auto mt-5 mb-5">No hay resultados para su búsqueda</p>
34         {% endif %}
35     </section>
36
37     {% include 'persona/paginator.html' %}
38
39 {% endblock %}
```


ORM – Object Relational Mapping

Object – hace referencia a la programación orientada a objetos

Relational – hace referencia a las bases de datos relacionales

Mapping – es un proceso que consiste en emparejar valores

ORM significa mapeo relacional de objetos.

Es una pieza de software que se encarga de hacer consultas e interactuar con la base de datos, usando lenguaje de backend, sin necesidad de escribir consultas SQL.

Hace el mapeo de las estructuras de las bases de datos a objetos.

Un ORM convierte cada tabla de la base de datos en una clase, a cada fila o registro, en un objeto y a cada campo de esos registros, en una propiedad del objeto, y todo en el lenguaje de backend.

Django crea el mapeo de las tablas de la base de datos a estructuras y tipos de datos de Python usando el modelo de datos. Un modelo, como vimos es una clase que corresponde al nombre de una tabla en la base de datos, y cada variable de clase es un campo de dicha tabla, en esta clase también se especifican índices de la tabla, llaves foráneas, ordenamiento de los datos, reglas de validación, entre otras cosas.

Cada aplicación en nuestro sistema Django tiene un modelo asociado, y las tablas creadas son del tipo `app_nombre_del_modelo`, por ejemplo, si tengo una aplicación “contabilidad” y mi modelo tiene la clase “Cliente”, la tabla se llamará “`contabilidad_cliente`”.

Cuando el modelo está definido Django usa las migraciones para crear las tablas y campos necesarios en la base de datos, y si posteriormente hacemos modificaciones al modelo, las migraciones se encargan de replicar esos cambios.

El comando “`python manage.py makemigrations`” busca cambios en todos los modelos de nuestro proyecto y crea las migraciones correspondientes.

Para ejecutar las migraciones se utiliza el comando “`python manage.py migrate`”.

Hay que recordar que en los modelos habitualmente no definimos un campo ID, Django lo crea automáticamente, y es de tipo entero auto incremental, este campo es llave primaria de la tabla.

Luego de creado los modelos, para crear instancias de estos modelos y almacenarlos en la base de datos, tenemos dos opciones, crear el registro y guardarlo automáticamente usando el método `create()`, o crear el registro (una instancia) y guardarlo a posteriori usando el método `save()`.

Depende de nuestro caso de uso, se podría usar cada método en las siguientes situaciones:

Caso de uso	<code>save()</code>	<code>create()</code>
Crear un registro (INSERT)	X	X
Actualizar un registro (UPDATE)	X	
Requiere una instancia del modelo	X	
Modificar un campo antes de grabarlo	X	
No necesita una instancia creada		X

Por lo general `create()` solo se usa cuando vamos a crear un registro nuevo, para todo lo demás se suele utilizar `save()`

Para recuperar los datos del modelo, u obtener algunos registros del modelo, cuando deseamos consultar algo en particular, o ver todos los campos de un modelo, todos los métodos para ello regresan un `querySet` con los registros consultados y podemos iterar sobre los `querySet` para procesar cada registro, y si no encontró registros tendremos un `querySet` vacío.

Un `querySet` es en esencia una lista de objetos de un modelo determinado, nos permite leer los datos de la base de datos, filtrarlos y ordenarlos.

Por ejemplo para ver todos los objetos de un modelo:

`Modelo.objects.all()`

Para obtener un solo registro (en este caso no obtenemos un `querySet`, sino un objeto):

`Modelo.objects.get(id=3)`

A diferencia del método `all()` si `get()` no encuentra ningún registro lanzará una excepción `Modelo.DoesNotExist`, por lo que hay que tenerla en cuenta siempre que usemos `get()` para poder manejarla.

Un ejemplo de cómo manejarla podría ser:

```
try:
    pedro = Cliente.objects.get(id=3)

except Cliente.DoesNotExist:

    pedro = Cliente.objects.create(
        nombre="Pedro",
        apellidos="Aguilar",
        fecha_nacimiento=fecha_nacimiento
    )
```

Esta forma de manejar el comportamiento rompe con el principio DRY (Don't Repeat Yourself), Django incorpora un método para estos casos:

`Modelo.objects.get_or_create(...)`

Por ejemplo:

```
pedro = Cliente.objects.get_or_create(
    nombre="Pedro",
    apellidos="Aguilar Ramírez",
    fecha_nacimiento=fecha_nacimiento
)
```

Solo tenemos que hacer la búsqueda usando todos los campos que no tengan valores default, para que el registro pueda ser creado si no existe.

`filter()`

El método `filter()` se utiliza para recuperar objetos de la base de datos que cumplen con ciertos criterios de búsqueda. La sintaxis general para este método sería:

`querySet = Modelo.objects.filter(campo1=valor1, campo2=valor2, ...)`

Django también incorpora el método `order_by()`, para usarlo solo hay que especificar los nombres de los campos por los que se desea ordenar, y si queremos un ordenamiento descendente antepone un guion al nombre del campo.

```
clientes = Cliente.objects.filter(apellidos__startswith="Garcia").order_by(
    "apellidos", "-fecha_nacimiento")
```

Funciones `Aggregate` y `Annotate`

Estas funciones nos posibilitan usar métodos como `Sum`, `Avg`, `Count`, `Max`, `Min`, etc.

`Aggregate` genera un resultado de valores sobre todo un `querySet`.

`Annotate` generará un resultado de valores por cada elemento de nuestro `querySet`.

La función `Annotate` se utiliza para agregar campos calculados a cada objeto en el `querySet` resultante. Realiza operaciones como las mencionadas, sumar, promedio, contar, etc., y agrega el resultado como un nuevo campo a cada objeto en el `querySet`.

`Aggregate` se utiliza para realizar operaciones de agregación en un `querySet` y retornar un único valor resultante. Puede realizar operaciones como contar, sumar, promediar, etc., sobre un conjunto de objetos en lugar de agregar campos a cada objeto individual.

`Q` objects

Los “objetos Q” son una forma de construir consultas complejas y dinámicas en el ORM, permiten combinar múltiples condiciones de búsqueda utilizando operadores lógicos como `AND`, `OR` y `NOT`.

Necesita ser importado el módulo “`from django.db.models import Q`”

Son útiles cuando necesitamos construir consultas que involucren a múltiples condiciones y queremos mantener nuestro código limpio y legible.

Por ejemplo, construir un objeto Q para buscar productos con precio menor a \$50 o con existencia menor que 10.

```
query = Q(precio__lt=50) | Q(existencia__lt=10)
```

Aplicamos la consulta usando el método `filter`:

```
Productos = Productos.objects.filter( query )
```

Expresiones F

Las expresiones F nos permiten realizar operaciones de la base de datos a nivel de campo utilizando valores de otros campos en la misma fila. Son útiles en situaciones donde necesitamos actualizar o filtrar objetos basándonos en valores de campos relacionados.

También debe ser importado el módulo “`from django.db.models import F`”

Las funciones “`selected_related()`” y “`prefetch_related()`” son métodos de optimización que nos permiten reducir la cantidad de consultas a la base de datos y mejorar el rendimiento al recuperar objetos relacionados.

La función “`selected_related()`” se utiliza para realizar un unión anticipada de modelos relacionados. Cuando la utilizamos Django realizará una sola consulta SQL que recuperará tanto los objetos del modelo principal como los objetos de los modelos relacionados mediante una JOIN en la base de datos.

Esto es útil cuando necesitamos acceder a los campos de los modelos relacionados y queremos evitar realizar múltiples consultas SQL para recuperar los datos.

`selected_related` solo funciona para relaciones de clave externa (foreign key) y relaciones uno a uno (OneToOneField)

Por ejemplo:

```
autores = Autor.objects.selected_related('editorial')
```

Esta consulta asegura que cuando accedas a los objetos de Autor, también se recuperarán los objetos relacionados de Editorial en una sola consulta.

`prefetch_related` se utiliza para recuperar objetos relacionados de una manera diferida o retrasada, a diferencia de `selected_related` que realiza una unión anticipada, en este caso se realiza una consulta adicional a la base de datos después de recuperar los objetos del modelo principal.

Esto es útil cuando necesitamos acceder a múltiples relaciones inversas o relaciones de muchos a muchos, ya que `prefetch_related()` permite reducir la cantidad de consultas al recuperar todos los objetos relacionados en una sola consulta adicional

URL de la documentación oficial:

<https://docs.djangoproject.com/en/5.0/topics/db/>

Archivo managers.py

El archivo `managers.py` en Django se utiliza para definir managers personalizados para nuestros modelos de datos. Un manager es una clase que provee métodos para realizar consultas en la base de datos.

Por defecto Django provee un manager llamado “`objects`” para cada modelo que definimos, sin embargo, a veces necesitamos métodos de consulta personalizados o queremos modificar el comportamiento predeterminado del manager, en estos casos podemos definir nuestro propio manager en el archivo “`managers.py`”

Un ejemplo simple, tenemos un modelo llamado `Productos` y queremos crear un manager personalizado para recuperar solo los productos activos, aquellos que no están descontinuados.

```
#managers.py
from django.db import models

class ProductoManager( models.Manager ):
    def activos(self):
        return self.filter( activo = True )
```

Aquí construimos un manager personalizado llamado `ProductoManager`, este contiene un método llamado “`activos()`” que retorna solo los productos activos.

Para utilizar este manager personalizado en nuestro modelo `Producto`, debemos asignarlo al atributo “`objects`” dentro de la definición del modelo.

```
# models.py

from django.db import models
from .managers import ProductoManager

class Producto(models.Model):
    nombre = models.CharField(...)
    precio = ...
    activo = models.BooleanField(default=True)

    objects = ProductoManager()
```

Ahora cuando queramos recuperar solo los productos activos, podemos hacerlo llamando al método “`activos()`” en el manager, por ejemplo:

```
productos_activos = Producto.objects.activos()
```

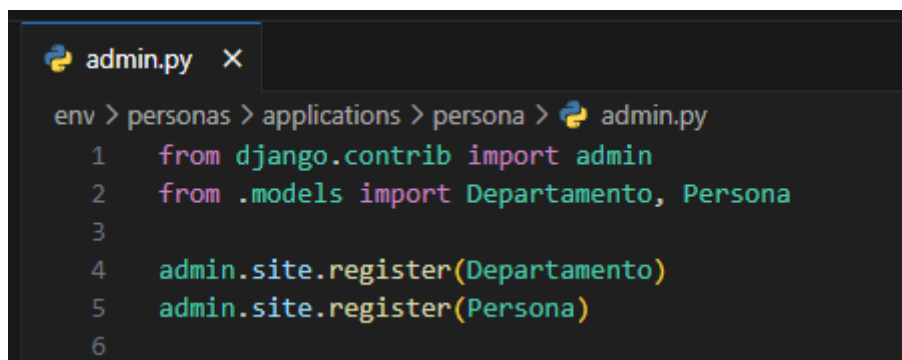
admin.py

El archivo `admin.py` de Django sirve para el registro de los modelos para la interfaz de administración de Django. En este archivo podemos registrar los modelos de las aplicaciones para que sean gestionables a través de la interfaz de administración de Django. Esto nos permite realizar operaciones CRUD en los objetos de esos modelos desde el panel de administración.

También nos sirve para personalizar la interfaz de administración, personalizar la forma en que se muestran y se editan los objetos de nuestros modelos en la interfaz de administración, utilizando clases `ModelAdmin` personalizadas, esto nos permite controlar que campos se muestran, que acciones están disponibles, etc.

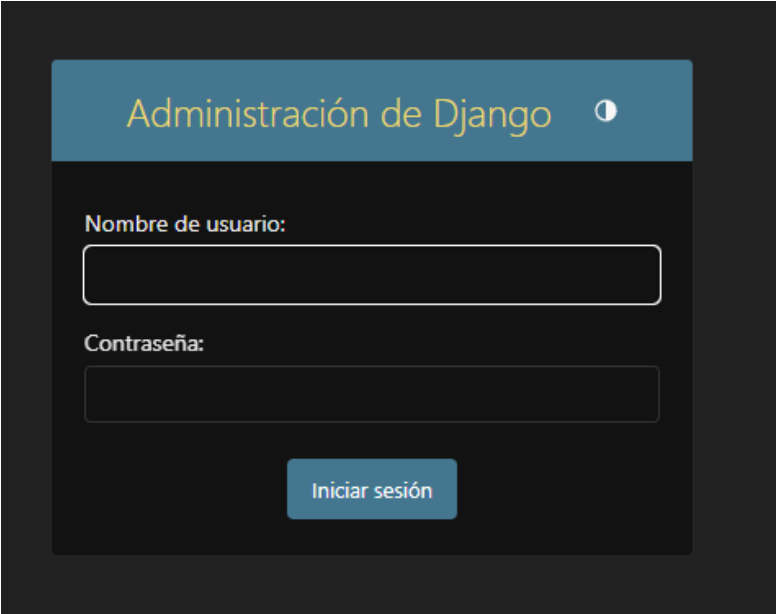
También nos permite configurar filtros de búsqueda para facilitar la navegación y administración de grandes conjuntos de datos en la interfaz de administración.

Por ejemplo, el `admin.py` de nuestra aplicación `Personas`, registra en el sitio de administración de Django los dos modelos definidos en el proyecto: `Personas` y `Departamento`

A screenshot of a code editor window titled 'admin.py'. The editor shows a Django project path 'env > personas > applications > persona' followed by the filename 'admin.py'. The code contains six lines: line 1 imports 'admin' from 'django.contrib'; line 2 imports 'Departamento' and 'Persona' from '.models'; line 3 is a blank line; line 4 calls 'admin.site.register(Departamento)'; line 5 calls 'admin.site.register(Persona)'; and line 6 is a blank line.

```
admin.py X
env > personas > applications > persona > admin.py
1  from django.contrib import admin
2  from .models import Departamento, Persona
3
4  admin.site.register(Departamento)
5  admin.site.register(Persona)
6
```

Para acceder al sitio de administración de Django debemos acceder a la URL `http://localhost:8000/admin`

The image shows the Django Admin login interface. It has a dark blue header with the text "Administración de Django" and a small circular icon. Below the header, there are two input fields: "Nombre de usuario:" and "Contraseña:". At the bottom, there is a blue button labeled "Iniciar sesión".

Administración de Django

Nombre de usuario:

Contraseña:

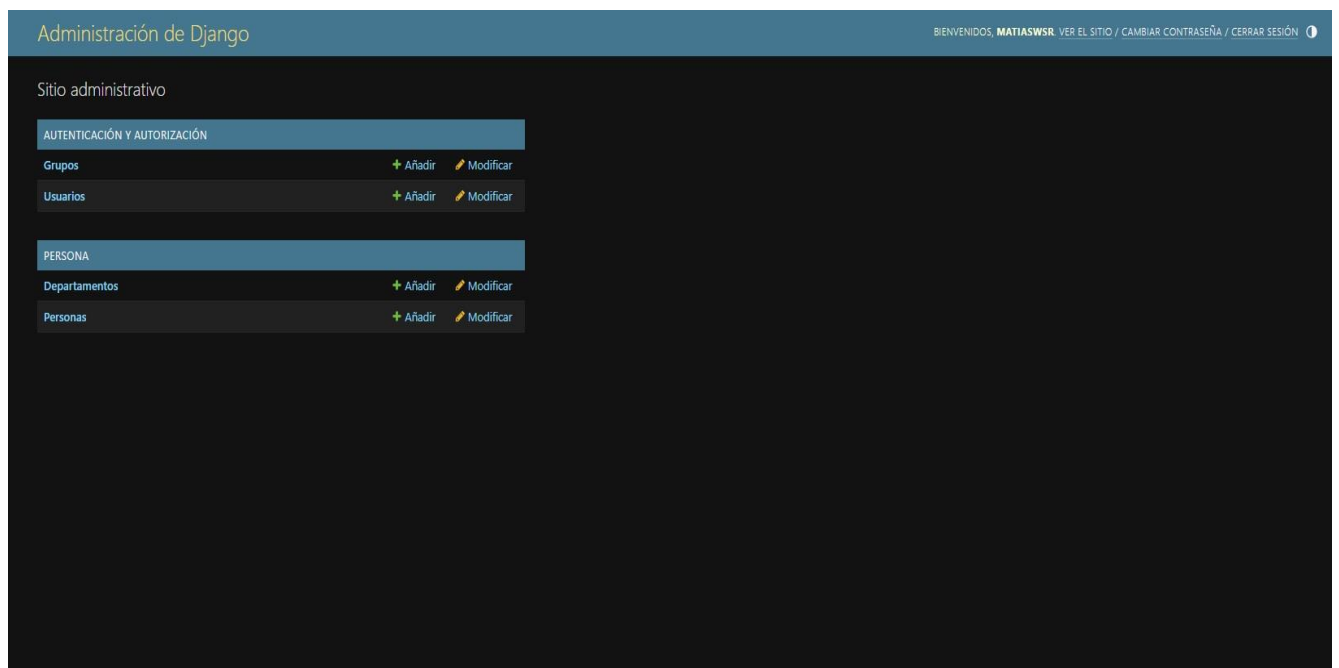
Iniciar sesión

Para acceder al sitio de administración de Django, previamente debemos ejecutar el comando para crear un super usuario, que será el usuario administrador del sitio.

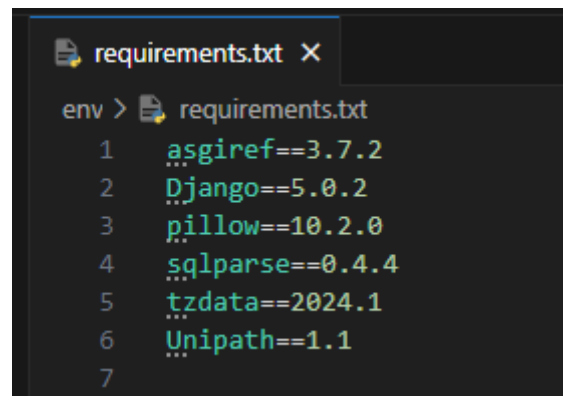
Para ello ejecutamos el comando:

```
python manage.py createsuperuser
```

Luego de creado el usuario y acceder podremos ver una interfaz similar a la siguiente:

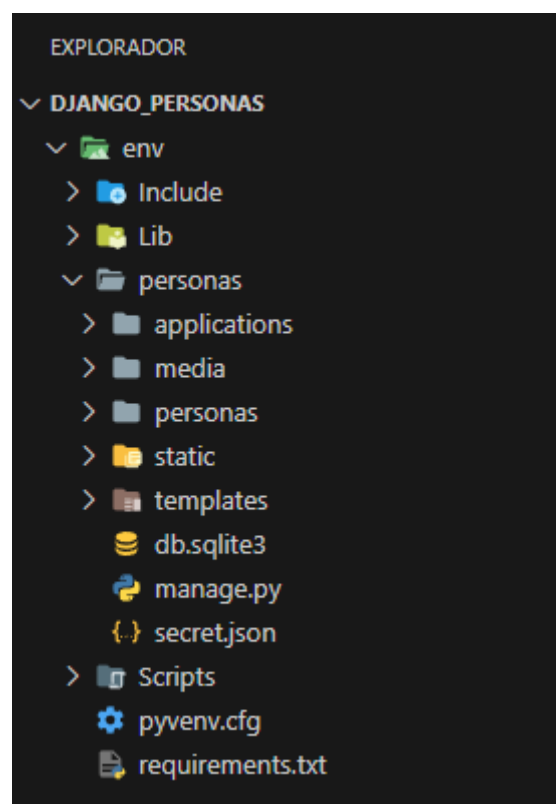


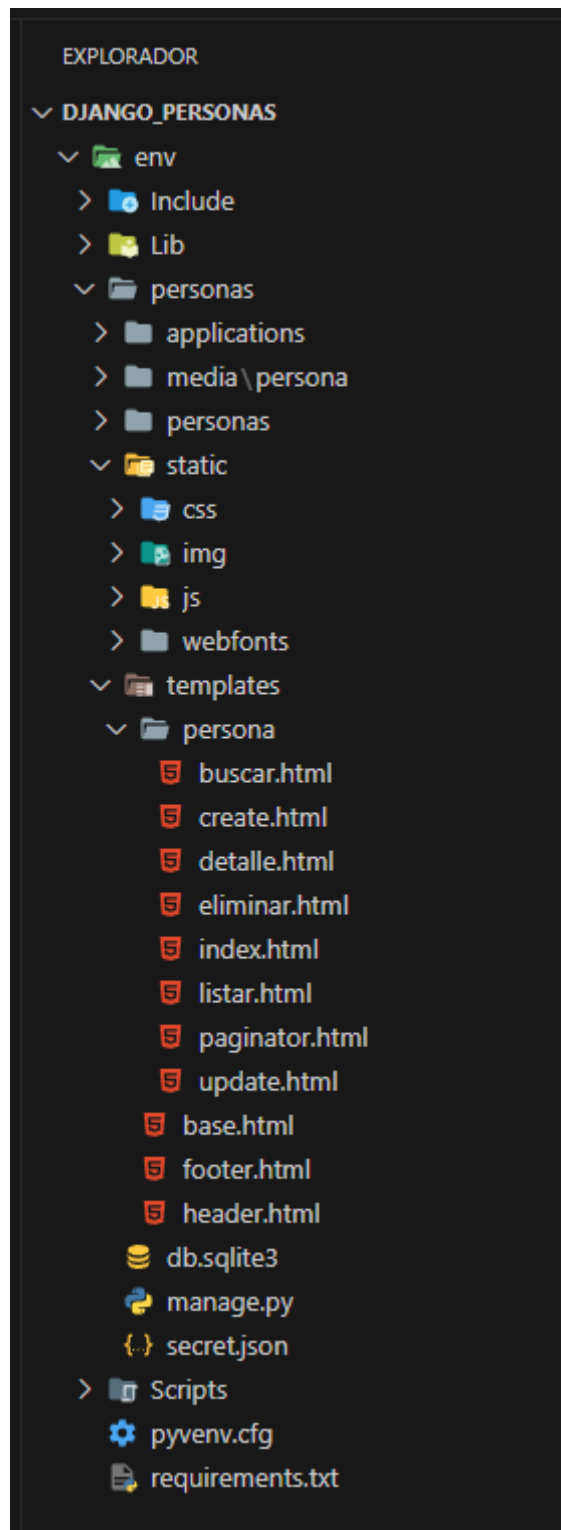
Para el desarrollo de este proyecto el archivo requirements.txt contiene los siguientes módulos a instalar:

A screenshot of a code editor window titled 'requirements.txt'. The editor shows a list of Python dependencies with their version constraints. The text is as follows:

```
env > requirements.txt
1  asgiref==3.7.2
2  Django==5.0.2
3  pillow==10.2.0
4  sqlparse==0.4.4
5  tzdata==2024.1
6  Unipath==1.1
7
```

Estructura usada en el proyecto:





La elaboración de este manual es de mi autoría con el único fin de afianzar y compartir conocimientos, utilizando un proyecto base para practicar las operaciones básicas CRUD y un ejemplo de filtro.

Para las explicaciones teóricas la fuente principal es la documentación oficial de Django, así como también algunos sitios de internet.