Proyecto CRUD Admin Simplificado 1 model

0) Requisitos locales

- Python 3.12+
- Git

---

1) Clonar el repositorio y crear entorno virtual

```
# 1.1 Clonar
git clone https://github.com/tu-usuario/tu-repo.git
cd tu-repo

# 1.2 Crear y activar venv
python3 -m venv .venv
# macOS / Linux
source .venv/bin/activate
# Windows PowerShell
# .venv\Scripts\Activate.ps1

#Para ver mas opciones puede revisar los apuntes anteriores, también está la opción de control+shift+p y seguir los pasos
```

---

2) requirements.txt

Crea `requirements.txt` en la raíz con:

```
asgiref==3.9.1
cffi==2.0.0
cryptography==45.0.7
Django==5.2.6
pycparser==2.23
PyMySQL==1.1.2
python-dotenv==1.1.1
sqlparse==0.5.3
```

Instala:

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

Descripción librerías

asgiref==3.9.1

- **Qué es**: “ASGI Reference Implementation”.
- **Para qué sirve**:
  - Django (desde la versión 3) soporta **ASGI** (Asynchronous Server Gateway Interface), que es la evolución de WSGI y permite manejar peticiones **asíncronas** (por ejemplo, WebSockets, long polling, streaming).
  - `asgiref` provee utilidades internas para que Django pueda ejecutar código asíncrono y mantener compatibilidad entre ASGI y WSGI.

cffi==2.0.0

- **Qué es**: C Foreign Function Interface.
- **Para qué sirve**:
  - Permite que Python interactúe con código escrito en **C**.
  - Muchas librerías de seguridad, encriptación o bajo nivel lo usan como dependencia.
  - En tu caso, es requerido indirectamente por `cryptography`.

cryptography==45.0.7

- **Qué es**: Librería de criptografía moderna en Python.
- **Para qué sirve**:
  - Se usa para manejar **encriptación, certificados, claves, firmas digitales**.
  - Django y otras librerías la usan internamente para cosas como:
    - Hash de contraseñas.
    - Manejo de cookies seguras (`SignedCookies`).
    - Tokens de autenticación (ej: CSRF, JWT si usas).
  - En este caso se utiliza para lograr conectarse a la base de datos en Railway

Django==5.2.6

- **Qué es**: El **framework web** principal.
- **Para qué sirve**:
  - El corazón de tu proyecto.
  - Incluye ORM, sistema de templates, seguridad, autenticación, panel admin, migraciones, etc.

pycparser==2.23

- **Qué es**: Un parser escrito en Python para interpretar código en C.
- **Para qué sirve**:
  - Es una dependencia de `cffi`.
  - Permite traducir definiciones de C para que `cffi` pueda exponerlas a Python.

PyMySQL==1.1.2

- **Qué es**: Un **driver MySQL** escrito completamente en Python.
- **Para qué sirve**:

- Django lo usa como reemplazo de `mysqlclient` para conectarse a bases de datos MySQL/MariaDB.
- Al usarlo junto a `pymysql.install_as_MySQLdb()`, Django cree que está usando `MySQLdb` pero realmente es `PyMySQL`.

python-dotenv==1.1.1

- **Qué es**: Una librería para cargar variables desde archivos `.env`.
- **Para qué sirve**:
  - Facilita la gestión de **variables de entorno** como credenciales de la base de datos, claves secretas, configuración de debug.
  - En tu caso, permite que Django tome los valores de `.env` y los use en `settings.py`.

sqlparse==0.5.3

- **Qué es**: Un analizador y formateador de **sentencias SQL**.
- **Para qué sirve**:
  - Django lo usa para formatear y mostrar de forma legible las consultas SQL en consola (cuando haces `python manage.py sqlmigrate`, debug de consultas, etc.).
  - No es un ORM ni ejecuta queries, solo mejora su **lectura y parseo**.

3) Crear el proyecto y la app

```
django-admin startproject modelDemo .
python manage.py startapp modelApp
```

Estructura (esperada):

```
tu-repo/
├─ templates/
│    ├─ base.html          # plantilla para todas las apps
```

```
├─ .env                    # (lo crearás en el paso 6)
├─ requirements.txt
├─ manage.py
├─ modelDemo/
│   ├─ __init__.py
│   ├─ settings.py          # editarás aquí
│   ├─ urls.py              # editarás aquí
│   └─ asgi.py / wsgi.py
└─ modelApp/
   ├─ __init__.py
   ├─ admin.py             # registrarás el modelo aquí
   ├─ apps.py
   ├─ forms.py             # crearás ModelForm aquí
   ├─ models.py            # crearás el modelo aquí
   ├─ urls.py              # rutas de la app
   ├─ views.py             # CRUD
   └─ templates/           # plantilla general
      └─ modelApp/
         ├─ empleado_form.html
         └─ empleado_list.html
```

Crea las carpetas `templates/` y `templates/modelApp/` como se ve arriba.

4) Modelo: Empleado

Django funciona con un **ORM (Object Relational Mapper)**, lo que significa que describes tu modelo en Python y automáticamente Django genera las tablas y columnas en la base de datos (en este caso MySQL en Railway).

`modelApp/models.py`

```
from django.db import models

class Empleado(models.Model):
    nombre = models.CharField(max_length=100)
```

```
    email = models.EmailField(unique=True)
    telefono = models.CharField(max_length=20, blank=True)

    def __str__(self):
        return f"{self.nombre} ({self.email})"
```

Explicación línea por línea

1. `from django.db import models`
   - Importa las clases base de Django para definir modelos.
   - `models.Model` es la clase que todo modelo debe heredar.
2. `class Empleado(models.Model):`
   - Se crea la clase `Empleado`.
   - Cada clase en `models.py` se convierte en una **tabla** en la base de datos.
   - La tabla se llamará automáticamente `modelApp_empleado` (nombre de la app + nombre del modelo, en minúsculas).
3. `nombre = models.CharField(max_length=100)`
   - Crea una **columna tipo texto corto** (VARCHAR(100)).
   - `max_length=100` define la longitud máxima.
   - Sirve para guardar nombres de empleados.
4. `email = models.EmailField(unique=True)`
   - Es un campo especializado para correos electrónicos.
   - Django validará automáticamente que el valor tenga un formato válido (`usuario@dominio.com`).
   - `unique=True` asegura que **no haya correos repetidos** en la tabla (se aplica una restricción única en la BD).
5. `telefono = models.CharField(max_length=20, blank=True)`
   - Otro campo de texto corto para números de teléfono.

- `blank=True` permite que este campo quede vacío en los formularios (opcional).
- Se almacena como texto porque un teléfono no es un número con el que vayas a hacer cálculos.

6. `def __str__(self): return f"{self.nombre} ({self.email})"`
   - Define cómo se mostrará el objeto cuando aparezca en el **admin de Django** o en la consola interactiva.
   - En vez de ver `Empleado object (1)`, verás algo como:

     ```
     Juan Pérez (juan@correo.com)
     ```

Para ir a mas detalles de los tipos de Field que se pueden utilizar visita la documentación oficial: Ver

5) Admin habilitado

El **Django Admin** es un panel web que se genera automáticamente para gestionar los modelos de tu aplicación sin que tengas que programar todo el CRUD desde cero.

Cuando defines un modelo (ej. `Empleado`), debes **registrarlo en** `admin.py` para que aparezca en el panel de administración.

`modelApp/admin.py`

```
from django.contrib import admin
from .models import Empleado

@admin.register(Empleado)
class EmpleadoAdmin(admin.ModelAdmin):
    list_display = ("id", "nombre", "email", "telefono")
    search_fields = ("nombre", "email")
```

Explicación línea por línea

1. `from django.contrib import admin`
   `from .models import Empleado`
   - Importamos el **sistema de administración de Django** y el modelo `Empleado`.
2. `@admin.register(Empleado)`
   `class EmpleadoAdmin(admin.ModelAdmin):`
   - Usamos un **decorador** para registrar el modelo `Empleado` en el admin.
   - Definimos una clase `EmpleadoAdmin` que hereda de `admin.ModelAdmin` para personalizar cómo se muestra.
3. `list_display = ("id", "nombre", "email", "telefono")`
   - Define las **columnas que se verán en la lista del admin** para este modelo.
   - En vez de mostrar solo `Empleado object`, verás una tabla con ID, Nombre, Email y Teléfono.
4. `search_fields = ("nombre", "email")`
   - Permite **buscar registros** desde la barra de búsqueda del admin usando nombre o email.

6) Variables de entorno (.env)

Un archivo `.env` es un **archivo de texto plano** que contiene **variables de entorno** (par clave=valor).

En Django (y en muchos proyectos modernos) se usa para **separar la configuración sensible del código**, por ejemplo:

- Contraseñas
- Usuarios de la base de datos
- Claves secretas (`SECRET_KEY`)
- Hosts permitidos

Esto evita **hardcodear (escribir valores fijos) credenciales** en `settings.py` o subirlas a GitHub.

Crea un archivo `.env` en la raíz del proyecto (nunca lo subas al repo si es público). Ejemplo (rellena con credenciales de Railway o donde levantes tu base de datos mysql):

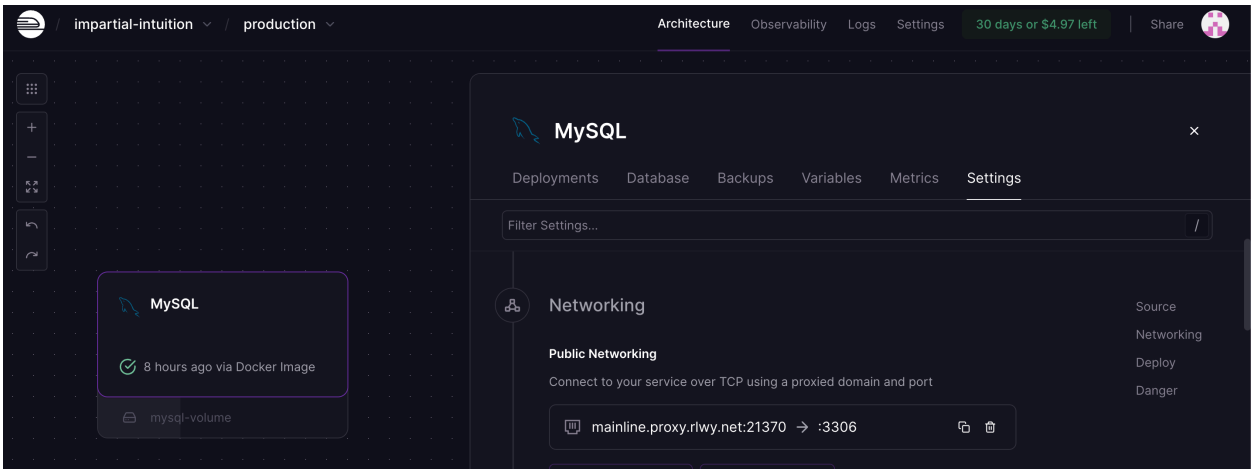
```
# DB (Railway)
MYSQL_HOST=mainline.proxy.rlwy.net
MYSQL_PORT=21370
MYSQL_DATABASE=railway
MYSQL_USER=root
MYSQL_PASSWORD=VOISSbeWvXIsfasdkNkMMVUjCvqCpyDHPdXPUiz

# Django
DEBUG=True
SECRET_KEY=django-insecure-4d&6n=0#f49kup^z609m5=8c5@$-cg8m&=n5$b2ae**=j-zpi0
USE_PYMYSQL=1
```

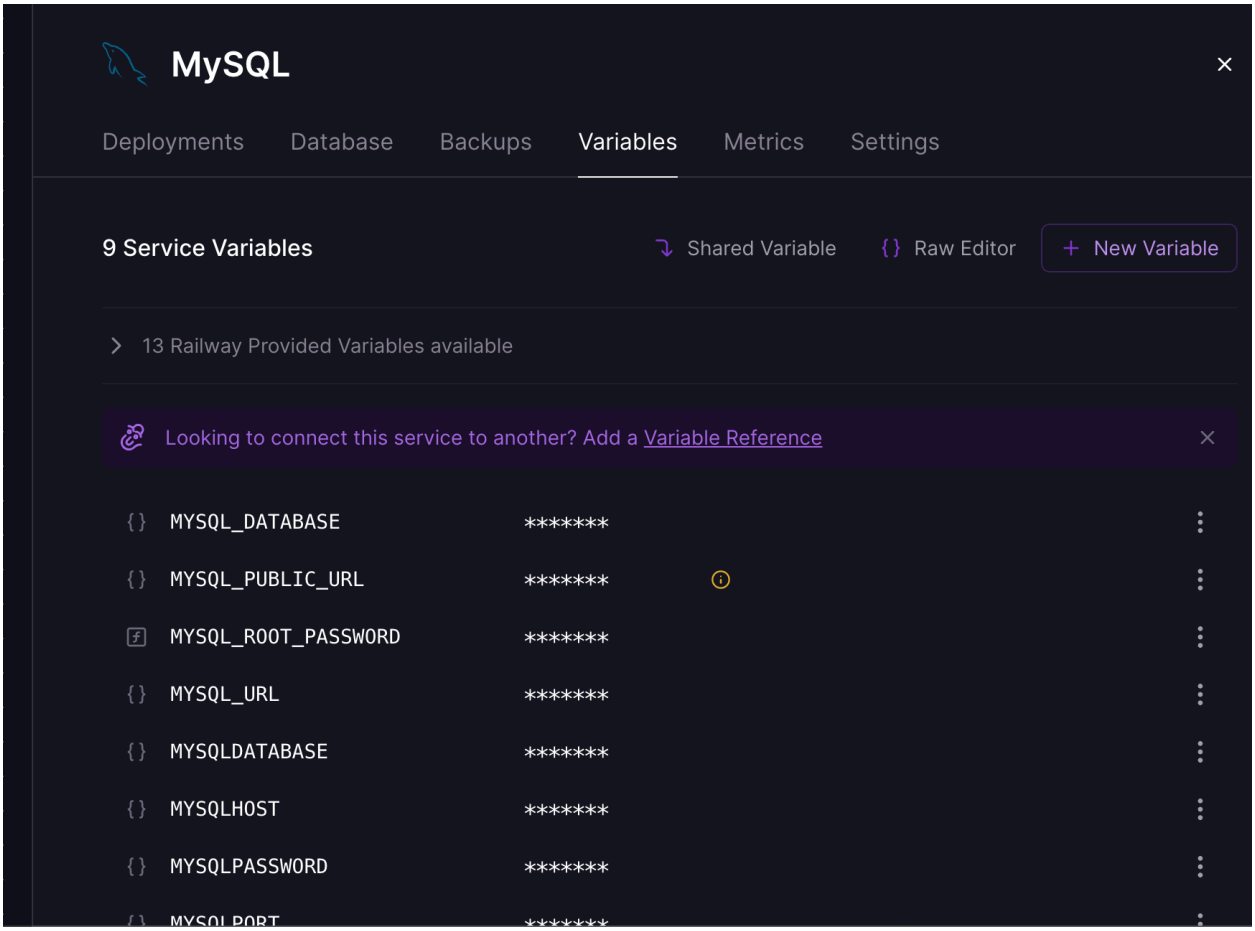
> Si Railway te entrega una URL tipo mysql://user:pass@host:port/db, separa los componentes y colócalos como arriba.

En Railway deberá crear un proyecto que levante mysql.

El host se obtiene de la siguiente menu:



Luego vamos a la pestaña Variables y copiamos los demás datos en el archivo .env



7) Configurar settings.py (Django + PyMySQL + .env)

modelDemo/settings.py

```
import os
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATES_DIR=os.path.join(BASE_DIR,'templates')
# Cargar .env local
try:
    from dotenv import load_dotenv
    load_dotenv(os.path.join(BASE_DIR,'.env'))
```

```
except Exception:
    pass
# Quick-start development settings - unsuitable for production

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.getenv("SECRET_KEY")

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = os.getenv("DEBUG", "True") == "True"

ALLOWED_HOSTS = []



# Application definition

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

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'modelDemo.urls'
```

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATES_DIR],
        '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',
            ],
        },
    },
]

WSGI_APPLICATION = 'modelDemo.wsgi.application'

# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

# Usar PyMySQL
if os.getenv("USE_PYMYSQL", "0") == "1":
    import pymysql
    pymysql.install_as_MySQLdb()

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": os.getenv("MYSQL_DATABASE"),
        "USER": os.getenv("MYSQL_USER"),
        "PASSWORD": os.getenv("MYSQL_PASSWORD"),
        "HOST": os.getenv("MYSQL_HOST"),
        "PORT": os.getenv("MYSQL_PORT", "3306"),
        "OPTIONS": {
```

```
            "charset": "utf8mb4",
            "init_command": "SET sql_mode='STRICT_TRANS_TABLES'",
            **(
                {"ssl": {"ca": os.getenv("MYSQL_SSL_CA")}}
                if os.getenv("MYSQL_SSL_CA")
                else {}
            ),
        },
    }
}
# Password validation
# https://docs.djangoproject.com/en/4.1/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]

# Internationalization

LANGUAGE_CODE = 'es-es'
```

```
TIME_ZONE = 'UTC'

USE_I18N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)

STATIC_URL = 'static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR,'static')]

# Default primary key field type
# https://docs.djangoproject.com/en/4.1/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

Explicación sección por sección

1. Importaciones y Paths

```
import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATES_DIR=os.path.join(BASE_DIR,'templates')
```

- `BASE_DIR` → ruta base del proyecto (donde está `manage.py`).
- `TEMPLATES_DIR` → apunta a la carpeta global `templates/` (donde guardas `base.html`).

---

2. Carga del archivo `.env`

```
try:
    from dotenv import load_dotenv
    load_dotenv(BASE_DIR / ".env")
except Exception:
    pass
```

- Usa **python-dotenv** para cargar las variables de entorno desde un archivo `.env`.
- Si el `.env` no existe, lo ignora (para no romper el proyecto).

---

3. Seguridad básica

```
SECRET_KEY = os.getenv("SECRET_KEY")
DEBUG = os.getenv("DEBUG", "True") == "True"
ALLOWED_HOSTS = []
```

- `SECRET_KEY` → clave secreta de Django, usada para firmar cookies, sesiones y seguridad.
- `DEBUG` → si está en `True` muestra errores detallados (solo en desarrollo).
- `ALLOWED_HOSTS` → lista de dominios/IPs permitidos para acceder a la app (en producción debes configurarlo).

---

4. Aplicaciones instaladas

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

```
    'modelApp',
]
```

- Aquí Django sabe **qué apps están activas** en el proyecto.
- Incluye las apps “core” de Django (admin, auth, sesiones, etc.) + tu app `modelApp`.

---

5. Middleware

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

- El **middleware** es como una cadena de filtros que procesan cada petición/respuesta.
- Ej: CSRF para seguridad en formularios, sesiones, mensajes, protección contra clickjacking.

---

6. URLs y plantillas

```
ROOT_URLCONF = 'modelDemo.urls'
```

- Indica que el archivo principal de URLs está en `modelDemo/urls.py`.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATES_DIR],
```

```
        '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',
            ],
        },
    },
]
```

- Configuración del sistema de **templates**:
  - `DIRS`: busca primero en la carpeta global `templates/`.
  - `APP_DIRS=True`: también busca dentro de cada app (ej. `modelApp/templates/`).
  - `context_processors`: añade variables útiles a todos los templates (ej. `request`, `user`, `messages`).

---

7. WSGI

```
WSGI_APPLICATION = 'modelDemo.wsgi.application'
```

- Define el punto de entrada **WSGI** (para servidores como Gunicorn/Apache).
- Para **ASGI** (channels, websockets) usarías `asgi.py`.

---

8. Base de datos

```
if os.getenv("USE_PYMYSQL", "0") == "1":
    import pymysql
    pymysql.install_as_MySQLdb()
```

- Permite usar **PyMySQL** como driver si se define la variable `USE_PYMYSQL`.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": os.getenv("MYSQL_DATABASE"),
        "USER": os.getenv("MYSQL_USER"),
        "PASSWORD": os.getenv("MYSQL_PASSWORD"),
        "HOST": os.getenv("MYSQL_HOST"),
        "PORT": os.getenv("MYSQL_PORT", "3306"),
        "OPTIONS": {
            "charset": "utf8mb4",
            "init_command": "SET sql_mode='STRICT_TRANS_TABLES'",
            **(
                {"ssl": {"ca": os.getenv("MYSQL_SSL_CA")}}
                if os.getenv("MYSQL_SSL_CA")
                else {}
            ),
        },
    }
}
```

- Configura la conexión a **MySQL** usando las variables de entorno (`.env`).
- Incluye soporte opcional de **SSL** si Railway u otro proveedor lo requiere.

---

9. Validación de contraseñas

```
AUTH_PASSWORD_VALIDATORS = [
    {'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'},
    {'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator'},
    {'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator'},
    {'NAME': 'django.contrib.auth.password_validation.NumericPasswordVal
```

```
idator'},
]
```

- Reglas para que las contraseñas sean seguras:
  - No parecidas al usuario.
  - Longitud mínima.
  - No comunes (como "123456").
  - No solo números.

10. Internacionalización

```
LANGUAGE_CODE = 'es-es'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_TZ = True
```

- `LANGUAGE_CODE`: idioma por defecto → español (España).
- `TIME_ZONE`: zona horaria del servidor (UTC por defecto).
- `USE_I18N`: habilita traducciones.
- `USE_TZ`: usa **timezone-aware datetimes** (más seguro con fechas).

11. Archivos estáticos

```
STATIC_URL = 'static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR,'static')] if (os.path.join(BASE
_DIR,'static')).exists() else []
```

- Define cómo servir CSS, JS e imágenes.
- `STATIC_URL`: la URL base (`/static/`).

- `STATICFILES_DIRS`: carpeta local `static/` (si existe).

---

12. Clave primaria por defecto

```
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

- Django usará `BigAutoField` (entero grande) para IDs automáticos en los modelos.

8) Crear la BD en Railway

1. Entra a Railway → **New Project** → **Provision MySQL**.
2. Copia las credenciales (host, port, user, password, database).
3. Pégalas en tu `.env` (paso 6).
4. Asegúrate que la instancia permita conexión externa (Railway lo permite por defecto) y usa el puerto indicado.

> No necesitas crear tablas manualmente; Django lo hará con migrate.

---

9) Migraciones e inicio de admin

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

python manage.py createsuperuser
# ingresa usuario/clave para entrar al /admin
```

¿Qué hace cada comando?

1. **`python manage.py makemigrations`**
   - Escanea tus modelos (p. ej., `Empleado`) y **genera archivos de migración** en `modelApp/migrations/` (p. ej., `0001_initial.py`).
   - **No** toca la base de datos aún; solo crea el “plan” de cambios.

2. `python manage.py migrate`
    - Ejecuta el “plan” anterior **contra tu MySQL de Railway** y crea/modifica tablas.
    - Aplica migraciones **de Django** (auth, sessions, admin, contenttypes) y **de tu app** (`modelApp_empleado`).
    - Crea también la tabla `django_migrations` para llevar control de lo aplicado.
3. `python manage.py createsuperuser`
    - Crea un usuario con permisos de **administrador** (`is_staff=True`, `is_superuser=True`).
    - Lo usas para entrar a **/admin** y gestionar registros sin programar formularios.

¿Qué se crea en la BD?

- Tablas de Django: `auth_user`, `auth_group`, `django_admin_log`, `django_session`, `django_content_type`, etc.
- Tu tabla: `modelApp_empleado` (con `id`, `nombre`, `email`, `telefono`).
- Registro de migraciones: `django_migrations`.

10) ModelForm con validaciones y Bootstrap

En Django, un `ModelForm` es una clase que genera automáticamente un formulario HTML **basado en un modelo** de tu aplicación.

Tenemos un modelo `Empleado`, el `ModelForm` crea los campos `nombre`, `email`, `telefono` con sus validaciones básicas sin que tengas que escribir todo a mano.

Puedes añadir **validaciones extra** y personalizar los **widgets** (inputs, selects, etc.) para aplicar estilos como **Bootstrap**.

`modelApp/forms.py`

```
from django import forms
from django.core import validators
```

```
from modelApp.models import Empleado  # importa tu modelo

class EmpleadoForm(forms.ModelForm):
    class Meta:
        model = Empleado
        fields = ['nombre', 'telefono', 'email']
        widgets = {
            'nombre': forms.TextInput(attrs={'class': 'form-control'}),
            'telefono': forms.TextInput(attrs={'class': 'form-control'}),
            'email': forms.EmailInput(attrs={'class': 'form-control'}),

        }

    # Validadores extra que no estén definidos en el modelo
    nombre = forms.CharField(
        min_length=5,
        max_length=20,
        validators=[
            validators.MinLengthValidator(5),
            validators.MaxLengthValidator(20),
        ],
        widget=forms.TextInput(attrs={'class': 'form-control'})
    )

    def clean_nombre(self):
        inputNombre = self.cleaned_data['nombre']
        if inputNombre == "Luis":
            raise forms.ValidationError("No se aceptan más Luis")
        return inputNombre
```

Explicación sección por sección

1. Importaciones

```
from django import forms
from django.core import validators
from modelApp.models import Empleado
```

- `forms`: módulo para manejar formularios en Django.
- `validators`: trae validadores básicos (mínimo/máximo de caracteres, etc.).
- `Empleado`: el modelo que conecta el formulario con la base de datos.

2. Definición del `ModelForm`

```
class EmpleadoForm(forms.ModelForm):
```

- El formulario está ligado al modelo `Empleado`.
- Esto significa que cuando llames `form.save()`, se guardará directamente en la tabla `modelApp_empleado`.

3. Clase interna `Meta`

```
class Meta:
    model = Empleado
    fields = ['nombre', 'telefono', 'email']
    widgets = {
        'nombre': forms.TextInput(attrs={'class': 'form-control'}),
        'telefono': forms.TextInput(attrs={'class': 'form-control'}),
        'email': forms.EmailInput(attrs={'class': 'form-control'}),
    }
```

- `model`: dice qué modelo usa este formulario.
- `fields`: lista los campos del modelo que quieres mostrar.
- `widgets`: personaliza cómo se verán los campos en HTML (aplicando **Bootstrap** con `class="form-control"`).

Ejemplo renderizado en HTML:

```
<input type="text" name="nombre" class="form-control">
<input type="text" name="telefono" class="form-control">
<input type="email" name="email" class="form-control">
```

4. Validaciones adicionales en `nombre`

```
nombre = forms.CharField(
    min_length=5,
    max_length=20,
    validators=[
        validators.MinLengthValidator(5),
        validators.MaxLengthValidator(20),
    ],
    widget=forms.TextInput(attrs={'class': 'form-control'})
)
```

- Aquí se **sobrescribe el campo** `nombre` definido en `Meta` para agregar validaciones personalizadas:
  - Debe tener entre 5 y 20 caracteres.
  - Sigue usando el mismo widget de Bootstrap.

Nota: si solo definieras los validadores en el modelo, se aplicarían también, pero aquí lo haces directamente en el formulario.

5. Validación personalizada `clean_nombre`

```
def clean_nombre(self):
    inputNombre = self.cleaned_data['nombre']
    if inputNombre == "Luis":
        raise forms.ValidationError("No se aceptan más Luis")
    return inputNombre
```

- `clean_<campo>` es un método especial de Django para validar un campo.
- En este caso:

    - Si el usuario escribe **“Luis”**, el formulario será inválido.
    - Mostrará el mensaje: **“No se aceptan más Luis”**.
- Si no hay problema, devuelve el valor limpio para guardarlo.

11) Vistas

Encontramos **funciones de vista (FBV = Function-Based Views)** que implementan el **CRUD** (Crear, Leer, Actualizar y Eliminar) para el modelo `Empleado` usando el `ModelForm`.

`modelApp/views.py`

```
from modelApp.models import Empleado
from django.shortcuts import render,redirect
from . import forms

def index(request):
    form =forms.EmpleadoForm()
    if request.method=='POST':
        form=forms.EmpleadoForm(request.POST)
        if form.is_valid():
            print("Formulario OK")
            print("Nombre: ",form.cleaned_data['nombre'])
            #ver información procesada desde el form
            form.save()
            return listar_empleados(request)
    data={'form':form}
    return render(request,'modelApp/index.html',data)

def listar_empleados(request):
    empleados = Empleado.objects.all()
    data={'empleados':empleados}
    return render(request,'modelApp/empleados.html',data)

def editar_empleado(request, id):
    empleado = Empleado.objects.get(id=id)  # Buscar el registro a editar
```

```
    form = forms.EmpleadoForm(instance=empleado)
    if request.method == 'POST':
        form = forms.EmpleadoForm(request.POST, instance=empleado)
        if form.is_valid():
            form.save()  # Aquí actualiza el registro en lugar de crear uno nuevo
        return listar_empleados(request)#redirect('employee_list')
    else:
        data = {'form':form}
        return render(request, 'modelApp/index.html', data)

def eliminar_empleado(request,id):
    empleado=Empleado.objects.get(id=id)
    empleado.delete()
    return redirect('trabajadores:empleados')
```

Explicación del código

```
from modelApp.models import Empleado
from django.shortcuts import render,redirect
from . import forms
```

- Importa el modelo **Empleado**.
- Importa **render/redirect** para devolver templates o redirigir.
- Importa el archivo `forms.py` que contiene el `EmpleadoForm`.

1.- Crear empleado – `index(request)`

```
def index(request):
    form =forms.EmpleadoForm()
    #cuando envía el formulario el boton guardar
    if request.method=='POST':
        form=forms.EmpleadoForm(request.POST)
        if form.is_valid():
            print("Formulario OK")
```

```
            print("Nombre: ",form.cleaned_data['nombre'])
            print("Email: ",form.cleaned_data['email'])
            print("Telefono: ",form.cleaned_data['Telefono'])
            form.save()
            return listar_empleados(request)
    data={'form':form}
    return render(request,'modelApp/index.html',data)
```

Explicación:

- Si la petición es `GET`: se muestra un formulario vacío (`EmpleadoForm()`).
- Si es `POST` (cuando se envía el formulario):
  - Django llena el formulario con los datos (`request.POST`).
  - `is_valid()` valida las reglas (longitud, email válido, etc.).
  - Si es válido:
    - Se imprimen los valores en consola (`cleaned_data`).
    - Se guarda un nuevo registro en la BD (`form.save()`).
    - Se redirige a la lista de empleados (`listar_empleados`).

Aquí se implementa el **Create** del CRUD.

2.- Listar empleados – `listar_empleados(request)`

```
def listar_empleados(request):
    empleados = Empleado.objects.all()
    data={'empleados':empleados}
    return render(request,'modelApp/empleados.html',data)
```

Explicación:

- Recupera todos los empleados con `.objects.all()`.
- Envía la lista a la plantilla `empleados.html`.
- En el template puedes mostrar los registros en una tabla.

Aquí se implementa el **Read** del CRUD.

3.- Editar empleado – `editar_empleado(request, id)`

```
def editar_empleado(request, id):
    empleado = Empleado.objects.get(id=id)  # Buscar el registro a editar
    form = forms.EmpleadoForm(instance=empleado)
    if request.method == 'POST':
        form = forms.EmpleadoForm(request.POST, instance=empleado)
        if form.is_valid():
            form.save()  # Aquí actualiza el registro en lugar de crear uno nuevo
        return listar_empleados(request)
    else:
        data = {'form':form}
        return render(request, 'modelApp/index.html', data)
```

Explicación:

- Busca al empleado por su `id`.
- Si la petición es `GET`:
  - Muestra el formulario con los datos actuales (`instance=empleado`).
- Si la petición es `POST`:
  - Carga el formulario con los nuevos datos (`request.POST`).
  - Si es válido, guarda los cambios con `form.save()`.
  - Vuelve a mostrar la lista de empleados.

Aquí se implementa el **Update** del CRUD.

4.- Eliminar empleado – `eliminar_empleado(request, id)`

```
def eliminar_empleado(request,id):
    empleado=Empleado.objects.get(id=id)
    empleado.delete()
    return redirect('trabajadores:empleados')
```

Explicación:

- Busca al empleado por su `id`.
- Lo elimina con `.delete()`.
- Redirige a la lista de empleados (`redirect` a la URL llamada `trabajadores:empleados`).

Aquí se implementa el **Delete** del CRUD.

12) URLs del proyecto y de la app

En Django, el **sistema de enrutamiento (URL dispatcher)** conecta cada URL que el usuario visita con una **vista (view)**. Aquí definimos las rutas para que los controladores de tu CRUD de `Empleado` funcionen.

`modelDemo/urls.py`

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

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

Explicación:

- `path('admin/', admin.site.urls)`

  → Activa la interfaz de **admin de Django** en `/admin`.

- `path('', include("modelApp.urls"))`

  → Indica que todas las rutas principales del sitio se cargarán desde `modelApp/urls.py`.

  → Ejemplo: si visitas `/empleados/`, Django buscará esa ruta dentro del archivo `modelApp/urls.py`.

`modelApp/urls.py`

```
from django.urls import path
from . import views

app_name="trabajadores"

urlpatterns=[
    path('',views.index,name="agregar"),
    path('empleados/',views.listar_empleados,name='empleados'),
    path('actualizarEmpleado/<int:id>',views.editar_empleado,name="editar"),
    path('eliminarEmpleado/<int:id>',views.eliminar_empleado,name='elimina
r'),
]
```

- `app_name = "trabajadores"` → Define un namespace para identificar las rutas de esta app.
- `path()` → Cada ruta conecta una **URL → vista** y recibe un `name` para llamarla desde templates o `redirect()`.
- **Rutas del CRUD:**
  - `/` → `index` → Crear empleado (**Create**).
  - `/empleados/` → `listar_empleados` → Ver lista (**Read**).
  - `/actualizarEmpleado/<id>` → `editar_empleado` → Editar (**Update**).
  - `/eliminarEmpleado/<id>` → `eliminar_empleado` → Borrar (**Delete**).

13) Templates (con herencia desde base.html)

13.1 templates/base.html

Plantilla general con Bootstrap + menú:

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    {% load static %}
    <title>{% block title %}Página base{% endblock %}</title>

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-sRIl4kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLzeN8tE5YBujZqJLB" crossorigin="anonymous">

    <!-- Bootstrap Icons -->
    <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.13.1/font/bootstrap-icons.min.css">

    {% block extra_css %}{% endblock %}
</head>

<body>
    <!-- Header con navbar -->
    <header>
        <nav class="navbar navbar-expand-lg bg-dark" data-bs-theme="dark">
            <div class="container-fluid">
                <a class="navbar-brand" href="{% url 'trabajadores:empleados' %}">Django 05</a>

                <!-- Botón hamburguesa (modo móvil) -->
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
                    data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent"
                    aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
```

```
            </button>

            <!-- Contenido colapsable -->
            <div class="collapse navbar-collapse" id="navbarSupportedConten
t">
                <!-- Menú de navegación -->
                <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                    <li class="nav-item">
                        <a class="nav-link active" aria-current="page"
                            href="{% url 'trabajadores:empleados' %}">Trabajadores
</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Libre</a>
                    </li>
                    <li class="nav-item dropdown">
                        <a class="nav-link dropdown-toggle" href="#" role="button"
data-bs-toggle="dropdown"
                            aria-expanded="false">
                            Ejemplo
                        </a>
                        <ul class="dropdown-menu">
                            <li><a class="dropdown-item" href="#">Libre</a></li>
                            <li><a class="dropdown-item" href="#">Libre</a></li>
                            <li>
                                <hr class="dropdown-divider">
                            </li>
                            <li><a class="dropdown-item" href="#">Libre</a></li>
                        </ul>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link disabled" aria-disabled="true">Libre</a>
                    </li>
                </ul>

                <!-- Formulario de búsqueda -->
```

```
            <form class="d-flex" role="search" method="get" action="#">
              <input class="form-control me-2" type="search" name="q" pla
ceholder="Buscar"
                aria-label="Buscar">
              <button class="btn btn-outline-success" type="submit">Busca
r</button>
            </form>
          </div>
        </div>
      </nav>
    </header>

    <!-- Contenido principal -->
    <main class="container py-4">
      {% if messages %}
        {% for message in messages %}
          <div class="alert alert-{{ message.tags }} alert-dismissible fade sho
w" role="alert">
            {{ message }}
            <button type="button" class="btn-close" data-bs-dismiss="alert"
aria-label="Cerrar"></button>
          </div>
        {% endfor %}
      {% endif %}

      {% block content %}
      {% endblock %}
    </main>

    <!-- Bootstrap JS -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.
bundle.min.js"
      integrity="sha384-FKyoEForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVc
XK/BmnVDxM+D2scQbITxI"
      crossorigin="anonymous"></script>
```

```
    {% block extra_js %}{% endblock %}
</body>

</html>
```

Explicación paso a paso

13.1.1 Estructura general del documento

```
<!DOCTYPE html>
<html lang="es">
```

- `<!DOCTYPE html>`: activa el modo estándar del navegador.
- `lang="en"`: idioma del documento. Si todo está en español, te conviene `lang="es"`.

13.1.2 `<head>`: metadatos, Bootstrap y estáticos

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  {% load static %}
  <title>{% block title %}Página base{% endblock %}</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css" rel="stylesheet" ...>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.13.1/font/bootstrap-icons.min.css">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js" ...></script>
</head>
```

- **`{% load static %}`**: habilita la etiqueta `{% static %}` para enlazar tus propios CSS/JS (por ejemplo: `<link rel="stylesheet" href="{% static 'css/app.css' %}">`).

- `<title>` **con** `{% block title %}`: cada template hijo puede cambiar el título sin duplicar todo el `<head>`.
- **Bootstrap 5.3.8 + Icons**: traes estilos y componentes listos. El *bundle* incluye Popper (necesario para dropdowns y tooltips).

> Sugerencia: si agregas CSS/JS propios, crea bloques opcionales:
>
> ```
> {% block extra_css %}{% endblock %}
> {% block extra_js_head %}{% endblock %}
> ```

13.1.3 `<header>`: Navbar responsiva

```
<nav class="navbar bg-dark navbar-expand-lg bg-body-tertiary" data-bs-the
me="dark">
```

- `navbar`: barra de navegación Bootstrap.
- `navbar-expand-lg`: se expande a partir de *large*; antes de eso se colapsa (modo móvil).
- `data-bs-theme="dark"`: aplica tema oscuro a componentes hijos (contraste correcto).

Contenido interno

- **Brand**:

  ```
  <a class="navbar-brand" href="{% url 'trabajadores:agregar' %}">Django
  05</a>
  ```

  Enlaza a la vista agregar de empleados. Usa el **namespace** `trabajadores` (definido en `app_name`) + nombre de URL `agregar`.

- **Toggler (móvil)**:

Botón hamburguesa que muestra/oculta el menú en pantallas pequeñas.

- **Menú principal**:

```
<ul class="navbar-nav me-auto mb-2 mb-lg-0">
  <li class="nav-item">
    <a class="nav-link active" aria-current="page" href="{% url 'trabajador
es:empleados' %}">Trabajadores</a>
  </li>
  ...
  <li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" ...>Ejemplo</a>
    <ul class="dropdown-menu">...</ul>
  </li>
</ul>
```

  - `active` + `aria-current="page"`: indica la opción actual (accesibilidad).
  - `dropdown`: ejemplo de submenú.

- **Formulario de búsqueda (decorativo por ahora)**:

```
<form class="d-flex" role="search">
  <input class="form-control me-2" type="search" placeholder="Buscar"
aria-label="Search" />
  <button class="btn btn-outline-success" type="submit">Buscar</button
>
</form>
```

  - Sin `action`, enviará a la misma URL.
  - Si planeas implementarlo, usa `method="get"` y un `name` en el input (ej. `name="q"`), para leerlo en la vista.

13.1.4 `<main>`: zona de contenido variable

```
<main>
  {% block content %}
  {% endblock %}
</main>
```

- `{% block content %}`: es el *placeholder* principal que cada template hijo rellenará con su contenido (formularios, tablas, etc.).
- Así evitas repetir el header, CSS y scripts en cada pantalla.

13.2 templates/modelApp/empleados.html

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

{% block title %}Empleados{% endblock %}

{% block content %}
<div class="container mt-5">
    <!-- Encabezado con logo -->
    <div class="alert alert-dark display-5 text-center">
        <img src="{% static 'imagenes/logo.png' %}" width="100px" alt="">
        Listado Trabajadores
    </div>

    <!-- Si hay empleados -->
    {% if empleados %}
    <table class="table">
        <thead>
            <tr>
                <th>Id</th>
                <th>Nombre</th>
                <th>Email</th>
```

```
            <th>Fono</th>
            <th>Opciones</th>
          </tr>
        </thead>
        <tbody>
          {% for emp in empleados %}
          <tr>
            <td scope="row">{{ emp.id }}</td>
            <td>{{ emp.nombre }}</td>
            <td>{{ emp.email }}</td>
            <td>{{ emp.fono }}</td>
            <td>
              <!-- Botón editar -->
              <a href="{% url 'trabajadores:editar' emp.id %}" class="btn btn-s
uccess btn-sm">
                <i class="bi bi-pencil"></i>
              </a>
              <!-- Botón eliminar -->
              <a href="/eliminarEmpleado/{{ emp.id }}" class="btn btn-danger
btn-sm">
                <i class="bi bi-trash"></i>
              </a>
            </td>
          </tr>
          {% endfor %}
        </tbody>
    </table>
    {% else %}
    <!-- Si no hay empleados -->
    <div class="alert alert-info">No hay Empleados</div>
    {% endif %}
</div>
{% endblock %}
```

Explicación paso por paso

3.2.1. Herencia y carga de estáticos

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

- `extends` → hereda la estructura del `base.html`.
- `load static` → permite usar `{% static %}` para rutas a imágenes, CSS o JS dentro de `static/`.

3.2.2. Bloque de título

```
{% block title %}Empleados{% endblock %}
```

- Reemplaza el bloque `title` del `base.html`.
- El `<title>` del navegador mostrará **"Empleados"**.

3.2.3. Contenido principal (`block content`)

```
{% block content %}
<div class="container mt-5">
```

- Todo lo que está dentro de este bloque reemplazará el `block content` del `base.html`.
- `container mt-5` → clase Bootstrap que centra y agrega margen superior.

3.2.4. Encabezado con logo

```
<div class="alert alert-dark display-5 text-center">
  <img src="{% static 'imagenes/logo.png' %}" width="100px" alt="">
  Listado Trabajadores
</div>
```

- `alert alert-dark` → usa un estilo de alerta gris oscuro de Bootstrap como encabezado.

- `display-5` → tipografía grande.
- `text-center` → centra el contenido.
- Imagen cargada desde `/static/imagenes/logo.png`.

3.2.5. Condicional: mostrar empleados o mensaje

```
{% if empleados %}
  ... tabla ...
{% else %}
  <div class="alert alert-info">No hay Empleados</div>
{% endif %}
```

- Si la vista pasó una lista `empleados` con registros → muestra la tabla.
- Si no hay datos → muestra mensaje “No hay Empleados”.

3.2.6. Tabla de empleados

```
<table class="table">
  <thead>...</thead>
  <tbody>
    {% for emp in empleados %}
    <tr>
      <td scope="row">{{ emp.id }}</td>
      <td>{{ emp.nombre }}</td>
      <td>{{ emp.email }}</td>
      <td>{{ emp.fono }}</td>
```

- Recorre con `{% for emp in empleados %}` la lista de objetos enviada desde la vista `listar_empleados`.
- Muestra los atributos del modelo `Empleado` (`id`, `nombre`, `email`, `fono`).

3.2.7. Botones de acciones

```
<td>
  <a href="{% url 'trabajadores:editar' emp.id %}" class="btn btn-success bt
n-sm">
    <i class="bi bi-pencil"></i>
  </a>
  <a href="/eliminarEmpleado/{{ emp.id }}" class="btn btn-danger btn-sm">
    <i class="bi bi-trash"></i>
  </a>
</td>
```

- Botón **editar**:
  - Usa la URL nombrada: `{% url 'trabajadores:editar' emp.id %}`.
  - Aplica estilos Bootstrap (`btn btn-success btn-sm`).
  - Ícono de lápiz (`bi bi-pencil`).
- Botón **eliminar**:
  - Redirige a `/eliminarEmpleado/<id>`.
  - Estilo rojo (`btn btn-danger`).
  - Ícono de basurero (`bi bi-trash`).
  - En el boton eliminar la url se creó distinto a la de editar, la buena práctica es como el editar, utilizando url usando nombres de las rutas.

13.3 templates/modelApp/index.html

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

{% block title %}Agregar Trabajador{% endblock %}

{% block content %}
<div class="container mt-5">
  <!-- Encabezado con logo —>
```

```
    <div class="alert alert-dark display-5 text-center">
      <img src="{% static 'imagenes/logo.png' %}" width="100px" alt="">
      Agregar Trabajador
    </div>

    <!-- Formulario -->
    <form action="" method="post">
      <table class="table">
        {{ form.as_table }}
        {% csrf_token %}
        <tr>
          <td colspan="2">
            <input type="submit" value="Guardar" class="btn btn-dark">
          </td>
        </tr>
      </table>
    </form>
  </div>
{% endblock %}
```

Explicación paso a paso

13.3.1. Herencia y carga de estáticos

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

- Hereda la estructura del `base.html` (navbar, header, bloques, etc.).
- `load static`: permite insertar imágenes, CSS o JS almacenados en la carpeta `static/`.

13.3.2. Bloque del título

```
{% block title %}Agregar Trabajador{% endblock %}
```

- Reemplaza el `<title>` definido en `base.html`.
- En la pestaña del navegador se mostrará **“Agregar Trabajador”**.

---

3. Contenedor principal

```
<div class="container mt-5">
```

- Centra el contenido con **Bootstrap** y le da margen superior (`mt-5`).

13.3.4. Encabezado con logo

```
<div class="alert alert-dark display-5 text-center">
  <img src="{% static 'imagenes/logo.png' %}" width="100px" alt="">
  Agregar Trabajador
</div>
```

- Estilo **alert-dark** para el título (banda gris oscuro).
- `display-5`: tipografía grande de Bootstrap.
- `text-center`: centra el texto.
- Inserta un **logo** desde la carpeta `static/imagenes/logo.png`.

13.3.5. Formulario

```
<form action="" method="post">
  <table class="table">
    {{ form.as_table }}
    {% csrf_token %}
    <tr>
      <td colspan="2">
        <input type="submit" value="Guardar" class="btn btn-dark">
      </td>
    </tr>
```

```
    </table>
</form>
```

- `action=""`: envía el formulario a la misma URL.
- `method="post"`: usa POST para enviar los datos.
- `{{ form.as_table }}`:
  - Renderiza automáticamente los campos del `ModelForm` como filas de una tabla HTML.
  - Ejemplo (si el modelo tiene `nombre`, `email`, `fono`):

    ```
    <tr><th><label for="id_nombre">Nombre:</label></th><td><input type="text" name="nombre"></td></tr>
    ```
- `{% csrf_token %}`: token de seguridad obligatorio en formularios POST de Django.
- Fila extra:
  - Botón de **submit** con estilo oscuro de Bootstrap (`btn btn-dark`).

---

14) Levantar el servidor

```
python manage.py runserver
```

- Admin: http://127.0.0.1:8000/admin/
- App (lista): http://127.0.0.1:8000/
- Crear: http://127.0.0.1:8000/

---

15) Flujo mínimo para verificar todo

1. `python manage.py makemigrations && python manage.py migrate`
2. `python manage.py createsuperuser`

3. Revisa `/admin`: debes ver **Empleados**. Crea uno desde admin para probar.
4. En la app: entra al listado y verifica que se muestre.
5. Crea, edita y elimina desde las vistas del CRUD.

---

16) Notas y tips

- Si ves error de conexión a MySQL: revisa host/puerto/usuario/clave en `.env`.
- En Railway, si regeneras credenciales, actualiza tu `.env`.
- `pymysql.install_as_MySQLdb()` permite que Django use PyMySQL como si fuera `mysqlclient`.
- Para producción, pon `DEBUG=False` y ajusta `ALLOWED_HOSTS`.
- Si usarás Bootstrap en más lugares, crea `static/` y organiza CSS/JS propios (no es obligatorio para este ejemplo).

17) Enlaces de interes

Database Functions

Working with forms

Widgets

Form and field validation