---

attachments: [Clipboard_2023-10-24-13-35-42.png, Clipboard_2023-10-24-13-36-21.png, Clipboard_2023-10-24-13-37-51.png, Clipboard_2023-10-24-14-10-11.png]
favorited: true
title: PLANTILLA
created: '2023-12-10T13:32:01.838Z'
modified: '2023-10-24T12:15:27.497Z'

Django

Tabla de contenidos

1.0 Introducción

Tabla de contenidos

1.1. ¿Que es **Python**?

- **Python** es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis muy limpia y un código legible.
- **Python** puede ser una buena alternativa para empezar a programar puesto que es un lenguaje muy sencillo, fácil y con una curva de aprendizaje buena.
- La sintaxis es fácil de entender puesto que es cercana al lenguaje natural.

1.2. ¿Que es **Django**?

- **Django** es un framework web de python gratuito y de codigo abierto que fomenta un desarrollo rapido y un diseño limpio y pragmatico.
- **Django** fue diseñado para ayudar a los desarrolladores a llevar las aplicaciones desde el concepto hasta su finalización lo más rápido posible.
- Ayuda a los desarrolladores a evitar muchos errores de seguridad comunes.
- Es muy escalable, algunos de los sitios más concurridos de la web aprovechan la capacidad de Django para escalar de forma rápida y flexible.
- Su version actual es **Django 5.0** aunque su version mas estable y con errores menores solucionados es **Django 4.2.8**.
- **Django** es un framework que respeta y utiliza el modelo MVC (Modelo Vista Controlador), que basicamente es un patron de diseño arquitectònico que nos permite separar la logica de negocio de la interfaz de usuario.

2.0. Instalación

Tabla de contenidos

2.1. Instalacion de Python en linux

> En este apartado veremos como instalar Python por terminal en linux.

- Para ver si tenemos instalado Python y en el caso de que este instalado ver que version tenemos:

```
python --version
```

> [!NOTE]
> En algunos Sistemas tenemos que utilizar el comando **python3** que es la que yo utilizare.

```
python3 --version
```

> [!IMPORTANT]
> Si tenemos instalada a partir de la version 3 no es necesario seguir los siguientes pasos a no ser que deseemos instalar la ultima version de Python.

- Lo siguiente que tendemos que hacer es actualizar la lista de paquetes con:

```
sudo apt-get update
```

- para descargar la ultima versiòn de python:
    - Primero debemos descargar el PPA(Personal Package Archive) llamado "deadsnake", que proporciona versiones actualizadas de python para tu sistema y volveremos a actualizar la lista de paquetes:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt-get update
```

- Descargamos la version que deseemos con wget y la descomprimimos:

```
wget https://www.python.org/ftp/python/3.x.y/Python-3.x.y.tgz
tar -xvf Python-3.x.y.tgz
```

- Ahora configuramos el entorno de compilacion. Esto analiza la congiguracion del sistema yestablece opciones especificas para compilar Python de acuerdo con las capacidades y configuraciones de tu maquina.
    - --enable-optimizations: Esta opción indica que se deben incluir optimizaciones durante el proceso de compilación. Esto puede aumentar el rendimiento de la ejecución de Python, ya que el código se compilará con optimizaciones específicas para tu arquitectura.

```
cd Python-3.x.y
./configure --enable-optimizations
```

- Ahora compilaremos los archivos necesarios con el comando:(Esto llevara unos minutos)

```
sudo make install
```

Ahora ejecute el comando:

```
python3
```

y podrá usar Python en la terminal.

2.2. Instalación de *Django* en linux

Tabla de contenidos

Lo primero que debemos hacer es saber si tenemos instalado el gestor de paquetes de Python.

```
pip3 --version
```

> En caso de que este instalado nos dira la version del gestor de paquetes.

2.3. Instalar Visual Studio Code

Tabla de contenidos

Empezamos Actualizando paquetes:

```
sudo apt-get update
```

Lo siguiente sera descargar el repositorio de Visual Studio Code:

```
sudo add-apt-repository "deb [arch=amd64] https://packages.microsoft.com/repos/vscode stable main"
```

Ahora actualizamos el sistema de nuevo y e instalamos VSCode.

```
sudo apt update
sudo apt install code
```

Veremos la version instalada con:

```
code -v
```

Ahora podemos abrir VSCode desde la terminal.

```
code
```

3.0. Programación orientada a objetos en PYTHON

Tabla de contenidos

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos". Un objeto es una instancia de una clase, y una clase es un conjunto de atributos y métodos que definen el comportamiento de los objetos.

En Python, todo es un objeto. Las variables, funciones y hasta los tipos de datos son objetos.

3.1. Clases y Objetos:

Tabla de contenidos

- En POO, una clase es una plantilla para crear objetos. Los objetos son instancias de una clase. Vamos a crear una clase simple llamada Persona:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")
```

> [!NOTE]
> En este ejemplo, Persona es una clase que tiene un constructor **init** y un método saludar. El método **init** se llama automáticamente cuando se crea un objeto de la clase, y se utiliza para inicializar los atributos del objeto, "self" hace referencia al objeto actual.

- Crear Objetos:
    - Ahora, podemos crear objetos de la clase Persona:

```
# Crear objetos
persona1 = Persona("Juan", 25)
persona2 = Persona("María", 30)

# Llamar al método saludar
persona1.saludar()
persona2.saludar()
```

Aquí, persona1 y persona2 son instancias de la clase Persona. Al llamar al método saludar, cada objeto imprime un mensaje personalizado.

- Atributos y Métodos:
    - Los atributos son variables asociadas a un objeto, mientras que los métodos son funciones asociadas a un objeto. En el ejemplo anterior, nombre y edad son atributos, y saludar es un método.

```
# Acceder a atributos
print(persona1.nombre)  # Imprime "Juan"
print(persona2.edad)    # Imprime 30
```

- Herencia:
    - La herencia permite que una clase herede atributos y métodos de otra. Vamos a crear una clase llamada Estudiante que hereda de Persona:

```
class Estudiante(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso

    def estudiar(self):
        print(f"{self.nombre} está estudiando en el curso {self.curso}.")
```

En este ejemplo, Estudiante hereda de Persona. El método **init** de Estudiante utiliza super() para llamar al constructor de la clase padre.

- Polimorfismo:
    - El polimorfismo permite que objetos de diferentes clases respondan al mismo método. Por ejemplo:

```
def presentar(objeto):
    objeto.saludar()

# Crear objetos
persona = Persona("Jose Carlos", 38)
estudiante = Estudiante("Ana", 19, "Matemáticas")

# Llamar al método usando polimorfismo
presentar(persona)      # Imprime "Hola, soy Jose Carlos y tengo 38 años."
presentar(estudiante)   # Imprime "Hola, soy Ana y tengo 19 años."
```

> [!NOTE]
> En este caso, el método presentar puede recibir tanto objetos de la clase Persona como de la clase Estudiante gracias al polimorfismo.

3.2. Encapsulamiento

Tabla de contenidos

El encapsulamiento es uno de los conceptos fundamentales de la Programación Orientada a Objetos (POO) que se refiere a la ocultación de los detalles internos de una clase y la restricción del acceso directo a ciertos componentes de esa clase. En otras palabras, encapsular una clase significa que los detalles internos de la implementación de esa clase están ocultos y solo se pueden acceder y manipular a través de una interfaz pública.

En Python, la encapsulación se logra mediante convenciones y prácticas, ya que el lenguaje no impone restricciones estrictas en el acceso a los miembros de una clase. Sin embargo, hay algunas convenciones comunes que se siguen para lograr encapsulación:

- Nombres con un guion bajo (Convención de "Nombre Débil"):
    - Los miembros de una clase que se consideran privados o internos a menudo se nombran con un guion bajo al principio del nombre. Por ejemplo: _variable_privada, _metodo_privado.
- Nombres con dos guiones bajos (Mangling):
    - Los miembros que se deben hacer aún más privados pueden utilizar la técnica de "mangling" agregando dos guiones bajos al principio del nombre. Por ejemplo: __variable_mangle, __metodo_mangle.
- Métodos Getter y Setter:
    - Se pueden proporcionar métodos públicos para obtener (get) y establecer (set) valores de variables privadas. Esto permite un control más preciso sobre la manipulación de los datos internos de la clase.
      A continuación, un ejemplo simple de encapsulamiento en Python:

```
class MiClase:
    def __init__(self):
        self.__variable_privada = 10  # Variable privada

    def obtener_variable_privada(self):
        return self.__variable_privada

    def establecer_variable_privada(self, nuevo_valor):
        if nuevo_valor > 0:
            self.__variable_privada = nuevo_valor
        else:
            print("El valor debe ser mayor que 0.")
```

En este ejemplo, __variable_privada es una variable privada que solo puede ser accedida dentro de la propia clase MiClase. Los métodos obtener_variable_privada y establecer_variable_privada proporcionan una interfaz pública para acceder y modificar el valor de la variable privada. Esto permite que la clase tenga un control más preciso sobre cómo se manipulan sus datos internos.

> [!IMPORTANT]
> Es importante destacar que estas convenciones son simplemente acuerdos entre los programadores y no imponen restricciones reales en el acceso a los miembros de la clase. En Python, la filosofía es "somos todos adultos aquí", confiando en que los programadores seguirán las convenciones para mantener la integridad y la seguridad de la implementación de la clase.

3.3. Abstracción

Tabla de contenidos

- La abstracción es un concepto clave en la Programación Orientada a Objetos (POO) que se refiere a la simplificación de la realidad al aislar y enfocarse en los aspectos esenciales de un objeto o sistema, mientras se ignoran los detalles menos relevantes.

En términos simples, la abstracción permite modelar objetos del mundo real en un programa de manera más simple y manejable.

- En POO, la abstracción se logra a través de la creación de clases y la definición de interfaces. Aquí hay algunos puntos clave relacionados con la abstracción:
- La abstracción de datos implica la representación de datos complejos mediante estructuras de datos más simples y la ocultación de detalles innecesarios.
  Por ejemplo, un objeto que representa una cuenta bancaria puede abstractamente tener métodos como depositar y retirar, sin necesidad de conocer los detalles de cómo se almacenan los datos internamente.
- Abstracción de Procesos:
    - La abstracción de procesos implica representar operaciones complejas como métodos más simples y manejables.

Por ejemplo, un objeto que representa un vehículo puede tener un método arrancar que abstractamente inicia el motor, sin entrar en los detalles específicos de cómo se realiza esa operación.

Aquí hay un ejemplo simple de abstracción en Python:

```
class FiguraGeometrica:
    def calcular_area(self):
        pass

    def calcular_perimetro(self):
        pass

class Circulo(FiguraGeometrica):
    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        return 3.14 * self.radio**2

    def calcular_perimetro(self):
        return 2 * 3.14 * self.radio
```

En este ejemplo, FiguraGeometrica es una clase abstracta que define métodos (calcular_area y calcular_perimetro) sin proporcionar una implementación. La clase Circulo hereda de FiguraGeometrica y proporciona una implementación específica para esos métodos. Los detalles internos de cómo se calcula el área y el perímetro de un círculo están ocultos para el usuario, quien simplemente utiliza la interfaz proporcionada por la clase abstracta. Esto es un ejemplo de cómo la abstracción permite representar de manera simple y manejable conceptos más complejos.

4.0. Generar nuestro primer proyecto con *Django*

4.1. Instalando el modulo de Django

Tabla de contenidos

Para instalar este paquete debemos utilizar

```
sudo apt install python3-django
```

Ahora con el siguiente comando comprobamos que Django esta instalado y su version:

```
pip list | grep Django
```

> [!NOTE]
> En algunos Sistemas tenemos que utilizar el comando **pip3**.

```
pip3 list | grep Django
```

Crearemos un directorio donde trabajaremos con Django.

```
mkdir AprendiendoDjango
```

> [!IMPORTANT]
> Deberas usar camelCase, UpperCamelCase o snake_case.

4.2. ¿Que es un entorno virtual?

Tabla de contenidos

Un entorno virtual (también conocido como virtualenv) es una herramienta que ayuda a gestionar las dependencias de un proyecto de software de manera aislada del sistema operativo y de otros proyectos. En el caso de Django (un marco de desarrollo web para Python), el uso de un entorno virtual es común y recomendado por varias razones:

- Aislamiento de Dependencias:
    - Un entorno virtual permite tener una copia independiente de Python y las bibliotecas específicas que tu proyecto necesita. Esto asegura que las dependencias de un proyecto no interfieran con las de otro.
- Versiones de Python:
    - Permite utilizar diferentes versiones de Python para diferentes proyectos. Esto es útil cuando tienes proyectos que requieren versiones específicas de Python.
- Facilita la Reproducibilidad:
    - Al especificar las dependencias de tu proyecto en un archivo requirements.txt, puedes fácilmente recrear el entorno virtual en otro lugar o compartirlo con otros desarrolladores, asegurando que todos estén utilizando las mismas versiones de las bibliotecas.
- Evita Conflictos con el Sistema Operativo:
    - Evita conflictos entre las bibliotecas del sistema operativo y las bibliotecas del proyecto. Un entorno virtual proporciona un espacio aislado donde las bibliotecas pueden instalarse sin afectar al sistema global.
- Facilita la Mantenibilidad:
    - Hace que la gestión de dependencias sea más fácil y mantenible. Puedes tener diferentes versiones de bibliotecas para diferentes proyectos sin afectar el sistema principal.

- Mejora la Seguridad:
    - Al tener un entorno virtual específico para cada proyecto, limitas el acceso a las bibliotecas y dependencias específicas de ese proyecto, reduciendo el riesgo de conflictos y mejorando la seguridad.

4.3. Cómo usar un entorno virtual en Django

Tabla de contenidos

Crear un Entorno Virtual:

Ejecuta el siguiente comando para crear un entorno virtual en la carpeta de tu proyecto:

```
python3 -m venv venv
```

Activar el Entorno Virtual:

```
source venv/bin/activate
```

Desactivar el Entorno Virtual:

```
deactivate
```

Instalar Dependencias:

Una vez que el entorno virtual está activado, puedes instalar las dependencias del proyecto utilizando:

```
pip3 install -r requirements.txt.
```

Usar un entorno virtual es una buena práctica en el desarrollo de software en Python y es especialmente útil en proyectos web como Django, donde la gestión de dependencias es crucial.

> [!NOTE]
> Es buena práctica crear un archivo requeriments.txt para indicar la librerias y versiones utilizadas en el proyecto.

```
pip3 freeze > requeriments.txt
```

De este modo podemos instalar todos las dependencias necesarias para nuestro proyecto.

Para iniciar un proyecto con django nos vamos a la carpeta creada y utilizamos.

4.4. Crea tu proyecto

Tabla de contenidos

```
django-admin startproject aprendiendoDjango
```

Esto nos creara un directorio con el nombre del proyecto con varios ficheros que veremos mas adelante entre ellos esta manage.py.

4.5. manage.py

Tabla de contenidos

El fichero creado manage.py es un script de line de comandos que se utiliza para realizar tareas administrativas realacionadas con un proyecto Django y se encuentra en el directorio raiz del proyecto.

Con el siguiente comando veremos todos los comandos y que poodemos hacer con ellos:

```
python3 manage.py help
```

4.5.1 Lista de comandos de manage.py

- [auth]
    - **changepassword**: permite cambiar la contraseña de un usuario.
    - **createsuperuser**: Interactivamente crea un nuevo superusuario para el panel de administracion.
- [contenttypes]
    - **remove_stale_contenttypes**: Elimina los tipos de contenido obsoletos de la base de datos.
- [django]
    - **check**: verifica la configuracion de Django en busca de problemas.
    - **compilemessages**: compila archivos de mensajes(.po) a archivos binarios(.mo).
    - **createcachetable**: Crea la tabla de cache en la base de datos.
    - **dbshell**: Accede a la shell de la base de datos.
    - **diffsettinng**: Muestra las diferencias entre la configuración actual y la configuración por defecto.
    - dumpdata: Exporta datos de la base de datos en formato JSON.
    - **flush**: Borra todos los datos de lña base de datos sin afectar a la estructura.
    - **inspectdb**: Genera modelos de Django basados en una base de datos existente.
    - **loaddata**: Carga datos desde archivos fixtures.
    - **makemessages**: Crea archivos de mensajes(.po) a partir del codigo fuente.
    - **makemigrations**: crea nuevas migraciones.
    - **migrate**: Aplica migraciones a la base de datos.
    - **optimizemigration**: Optimiza las migraciones existentes.
    - **sendtestemail**: Envía un correo electrónico de prueba.
    - **shell**: Inicia la consola interactiva de Django.
    - **showmigrations**: Muestra todas las migraciones y su estado.
    - **sqlflush**: Muestra las declaraciones SQL para realizar un flush de la base de datos.

    - **sqlmigrate**: Muestra las declaraciones SQL para una migración específica.
    - **sqlsequencereset**: Muestra las declaraciones SQL para reiniciar secuencias de la base de datos.
    - **squashmigrations**: Combina migraciones antiguas en una sola.
    - **startapp**: Crea una nueva aplicación Django.
    - **startproject**: Crea un nuevo proyecto Django.
    - **test**: Ejecuta las pruebas definidas en la aplicación.
    - **testserver**: Inicia un servidor para pruebas.
- [sessions]
    - **clearsessions**: Elimina las sesiones de usuarios expiradas.
- [staticfiles]
    - **collectstatic**: Recopila archivos estáticos en una ubicación central.
    - **findstatic**: Muestra la ubicación de un archivo estático.
    - **runserver**: Inicia el servidor de desarrollo de Django.

5.0. Principales componentes de Django

Tabla de contenidos

1) **ORM(Object-Relational-Mapping)**

> Django incluye un ORM que facilita la interacción con bases de datos relacionales. Los modelos de Django definen en Python y se mapean a tablas de base de datos,permitiendo a los desarrolladores realizar operaciones de base de datos de manera más intuitiva.

2) **MVC(Model-View-Controller)**

> Django sigue el patrón de diseño Modelo-Vista-Controlador, aunque a menudo se le llama Modelo-Vista-Plantilla (MVT) en el contexto de Django. Los modelos representan la estructura de datos, las vistas gestionan la lógica de presentación y las plantillas definen la presentación de los datos.

3) **Sistema de plantillas**

> Django proporciona un sistema de plantillas que permite separar la logica de presentacion del codigo Python, facilitando la ceracion de páginas web dinámicas.

4) **Enrutamiento y Vistas**

> El enrutador de Django dirige las solicitudes HTTP a las vistas correspondientes. Las vistas correspondientes son funciones de Python que toman solicitudes y devuelven respuestas, y son responsables de procesar la lógica de la aplicación.

5) **Admin Site**

> Django incluye un panel de administración automático que facilita la gestión de modelos y datos de la base de datos sin tener que crear una interfaz de administración personalizada.

6) **Sistema de formularios**

> Django proporciona herramientas para crear y procesar formularios de manera sencilla, facilitando la interaccion con los usuarios.

7) **Manejo de URL**

> Django utiliza un sistema de enrutamiento basados en patrones de URL para dirigir lsa solicitudes a las vistas correspondientes.

6.0. Cómo funciona Django

Tabla de contenidos

8) **Solicitud y enrutamiento**

> Cuando se recibe una solicitud HTTP, el sistema de enrutamiento de Django(definido en el archivo "urls.py") dirige la solicitud a una vista especifica.

9) **Vistas y lógica de Aplicación**

> La vista es una funcion de python que maneja la logica de la aplicación. Puede acceder a la base de datos procesar datos y devolver una respuesta.

10) **Modelo y Base de Datos**

> Los modelos de Django definen la estructura de la base de datos. El ORM se encarga de traducir las operaciones de la base de datos a operaciones en objetos Python.

11) **Platillas y Presentación**

> Las plantillas de Django se utilizan para definir la presentación de los datos generados por las vistas. Estas plantillas son procesadas y enviadas al cliente.

12) **Respuesta HTTP**

> Django genera una respuesta HTTP que se envia de vuelta al navegador del usuario.

- Django facilita la creación de aplicaciones web robustas al proporcionar una estructura organizada y herramientas poderosas, permitiendo a los desarrolladores construir aplicaciones de manera eficiente y mantenible. Además, su énfasis en la seguridad y las mejores prácticas lo convierten en una opción popular para el desarrollo web en Python.

7.0. Primer contacto con Django

Tabla de contenidos

7.1. Creando una app

7.1.1. ¿Que es una *app*?

- Si miramos el directorio creado en el capitulo anterior dentro veremos una serie de archivos y una carpeta con el nombre de nuestro proyecto, esto es un paquete donde tenemos el archivo ***settings.py***.
- Empezaremos a configurarlo por ***INSTALLED_APPS***.

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

> Esto es una lista con con varias aplicaciones que generan distintas funcionalidades.

- **admin**: Proporciona la interfaz de administracion de Django que te permite gestinar modelos y datos directamente desde el navegador.
- **auth**: Gestiona la autentificación y autorización de usuarios.
- **contenttypes**: Permite la asociación de modelos con tipos de contenido.
- **sessions**: Proporciona soporte para sesiones de usuario.
- **messages**: Maneja mensajes de un lado a otro entre vistas.
- **staticfiles**: Gestiona archivos estaticos como CSS, JavaScript e imágenes para tu aplicación.

> Puedes agregar tanatas aplicaciones como desees.

- En resumen, una aplicación es un paquete dentro de Django y todas estas aplicaciones forman nuestro proyecto.

7.1.2. Como crear una app

1) abre la terminal en el directorio creado para el proyecto(AprendiendoDjango).
2) Para crear una app:

```
python3 manage.py startapp
```

> Esto generara un directorio con varios archivos que iremos viendo.

7.2 Vistas

Tabla de contenidos

- Las **vistas** se refieren a las funciones o clases que manejan las solicitudes HTTP y devuelven respuestas. Las vistas son responsables de procesar la entrada del usuario (la solicitud) y producir la salida (la respuesta).

- En Django las vistas se crean o modifican desde el fichero *views.py* que esta en el directorio de nuestra app.

> En este fichero vemos como se importan los ***shortcuts***,

- los "shortcuts" (atajos) son funciones y clases proporcionadas por el módulo django.shortcuts que facilitan y simplifican tareas comunes en el desarrollo de aplicaciones web. Estos atajos son una manera conveniente de realizar operaciones que se realizan frecuentemente sin tener que escribir una cantidad extensa de código.

> Para mas informacion sobre -> SHORTCUTS.
>
> Para continuar a partir de este punto deberia tener unas nociones basicas de fundamentos de la programacion y una base de programacón y sintaxis de Python.

7.2.1. Creando una vista(Ejemplo)

Tabla de contenidos

- En el fichero views.py, debemos importar HttpResponse, que es un objeto que representa la respuesta HTTP que sera enviada al navegador de lusuario cuando se realiza una solicitud a la vista.

```
from django.shortcuts import render, HttpResponse

def hola_mundo(request):
    return HttpResponse("hola mundo con Django!!")
```

> Por convención los métodos y funciones en python seran con camelCase.
> ***Contructor de HttpResponse:***

```
HttpResponse(content=b'', status=200, content_type='text/html', charset=None)
```

- Si nos fijamos esta vista es una simple función de Python que recibe un parametro llamado *request*, que es un parametro que me permite recibir datos de peticiones que haga a esta URL y devuelve una cadena de texto.

Para ver el funcionamiento de nuestra vista necesitamos cargar el metodo o utilizarlo en una URL.

7.2.2. Utilizar el metodo en una URL

Tabla de contenidos

Para utilizar el metodo en una URL debemos abrir el fichero llamado *urls.py* del directorio con el mismo nombre creado dentro de nuestro proyecto.

- Primero importaremos la funcion:

```
from myapp import views
```

- Despues modificaremos la lista urlpatterns:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hola-mundo/', views.hola_mundo, name="hola_mundo")
]
```

- Desglosamos el objeto *path*:

- path('hola-mundo/', ...) -> define la URL que activara la vista.
- views.hola_mundo -> especifica la funcion que se ejecutara cuando la URL coincida.
- name="hola_mundo" -> Asigna un nombre a la URL. Este nombre puede ser utilizado en otras partes del código para referenciar esta URL de manera más fácil. Por ejemplo, en las plantillas o en la generación de URLs dentro de las vistas.

7.2.3. Hello World

Tabla de contenidos

Ya tenemos nuestra primera vista preparada para verla en el navegador.

¿Como lo hacemos?

- Debemos arrancar el servidor de Django:

```
python3 manage.py runserver
```

- Introducimos la URL que nos da el servidor en el navegador o pulsamos ctrl y hacemos click en ella.
- En esta ocasión en vez de mostrar la pagina por defecto de Django nos devuelve un error 404. Esto sucede porque la primera ruta(path) que tenemos en *urls.py* es "hola-mundo/", asi que debemos añadirle esto a la URL.

Ya tenemos El hola mundo!

7.2.4. Multiples vistas y URLs

Tabla de contenidos

Ahora crearemos otro método en *views.py*, por ejemplo una pagina de inicio:

```
def index(request):
    return HttpResponse("""
        <h1>Inicio</h1>
    """)
```

Ya tenemos nuestro segundo método, solo nos queda asignarle una ruta, para ello volvemos al fichero *urls.py*:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name="index"),
    path('index', views.index, name="inicio"),
    path('hola-mundo/', views.hola_mundo, name="hola_mundo")
]
```

En este ejemplo dejamos vacia la cadena que recoge como primer argumento para que al abrir la raiz de la URL nos muestre nuestra pagina de inicio.

7.3. Navegación entre rutas

Tabla de contenidos

- Para entender como funciona lo haremos con un ejemplo:
  En el fichero views.py declararemos la siguiente variable:

```
layout = """
<h1>Sitio web con Django | Jose Carlos Limones</h1>
<hr/>
<ul>
   <li>
   <a href="/">Inicio</a>
   </li>
   <li>
   <a href="/hola-mundo">Hola mundo</a>
   </li>
   <li>
   <a href="/pagina">Pagina 1</a>
   </li>
</ul>
<hr/>
"""
```

- Y modificamnos las funciones ya creadas:

```
def index(request):
   return HttpResponse(layout+"""
      <h1>Inicio</h1>
   """)

def hola_mundo(request):
   return HttpResponse(layout+"""
      <h1">hola mundo con Django!!</h1>
      <h3>Soy Jose Carlos Limones</h3>
      """)

def pagina(request):
   return HttpResponse(layout+"""
      <h1>Esta es la Pagina 1!!</h1>
      """)
```

> *Recuerda modificar el fichero urls.py*.

7.4. parámetros en ruta

Tabla de contenidos

Como sabemos en la URL podemos pasar parametros, en este apartado veremos como se tratan.

Creamos una supuesta pagina llamada "contacto":

```
def contacto(request, nombre):
 return HttpResponse(layout+f"<h1>Contacto {nombre} </h1>")
```

- Para poder tratar estos parametros debemos indicarle al path que tipo de parametro es y su nombre de esta forma:

```
 path('contacto/<str:nombre>', views.contacto, name="contacto")
```

Ahora probamos en nuestro navegador:

```
http://127.0.0.1:8000/contacto/jose%20Carlos
```

> el "%20" el navegador lo tratara como un espacio.

Si queremos añadir mas parametros simplemente lo haremos asi:

- En views.py:

```
def contacto(request, nombre, apellido):
  return HttpResponse(layout + f"<h1>Contacto {nombre} </h1>")
```

- En urls.py:

```
    path('contacto/<str:nombre>/<str:apellido>', views.contacto, name="contacto")
```

- En el navegador:

```
http://127.0.0.1:8000/contacto/Jose%20Carlos/Limones
```

7.5. Parámetros opcionales

Tabla de contenidos

- Podemos hacer que los parámetros sean opcionales inicializandolos a null o vacío de esta forma:

```
def contacto(request, nombre="", apellido=""):
    html = ""

    if nombre and apellido:
        html = f"<h3>{nombre}  {apellido}"
    elif nombre:
        html = f"<h3>{nombre}"

    return HttpResponse(layout + f"<h2>Contacto </h2>" + html)
```

- Despues en urls.py debemos darle las opciones de cada path:

```
path('contacto/', views.contacto, name="contacto"),
path('contacto/<str:nombre>', views.contacto, name="contacto"),
path('contacto/<str:nombre>/<str:apellido>', views.contacto, name="contacto")
```

7.6. Redirecciones

Tabla de contenidos

Django viene con una aplicación de redirecciones opcional. Te permite almacenar una redireccion en una base de datos y maneja la redirección por ti.

- En este ejemplo vamos a dirigir la pagina de contacto con los parametros necesarios:
    - importar el *shortcut* ***redirect***.

```
from django.shortcuts import redirect
```

    - Metemos una redireccion en nuestro metodo o función.

```
def pagina(request, redirigir = 0):
    if redirigir == 1:
        return redirect('contacto', nombre="Jose Carlos", apellido="Limones")
```

```
    return HttpResponse(layout+"""
       <h1>Esta es la Pagina 1!!</h1>
       """)
```

- Escribimos el nuevo path:

```
    path('pagina/<int:redirigir>', views.pagina, name="pagina"),
```

- Ahora nos redirigira a contacto con los paramtros pasados escribiendo en el navegador:

```
http://127.0.0.1:8000/pagina/1
```

> Recuerda que debemos mantener el path anterior.

7.7. Vistas Base

Tabla de contenidos

- Django proporciona una serie de vista(objetos) ya creadas. Estas vistas han sido creadas para usarlas como plantilla, es decir crear clases que hereden de de esta base.
- Todas las demás vistas basadas en clases heredan de esta base. clase. No es estrictamente una vista genérica y, por lo tanto, también se puede importar desde *django.views*.
- El diagrama de flujo del metodo es:
  - setup() -> Se llama al inicio de cada solcitud par realizar cualquier configuracion necesaria.
  - dispatch() -> Se llama para maneja la solicitud HTTP. Determina qué método HTTP se está utilizando y llama al método correspondiente (por ejemplo, get(), post(), etc.).
  - http_method_not_allowed() -> Este metodo se invoca si el metodo de la solicitud HTTP no esta permitido.
  - options() -> Este método se invoca cuando se recibe una solicitud OPTIONS en la vista. La implementación predeterminada devuelve una respuesta HTTP con los métodos permitidos en el encabezado "Allow".

Ejemplo views.py:

```
from django.http import HttpResponse
from django.views import View


class MyView(View):
    def get(self, request, *args, **kwargs):
        return HttpResponse("Hello, World!")
```

Ejemplo urls.py

```
from django.urls import path

from myapp.views import MyView

urlpatterns = [
```

```
    path("mine/", MyView.as_view(), name="my-view"),
]
```

- La lista de métodos permitidos:

```
["get", "post", "put", "patch", "delete", "head", "options", "trace"]
```

7.8. Templates en Django

Tabla de contenidos

Hasta ahora hemos trabajado con el codigo html directamente en las vista, pero esa no es la forma correcta de utilizar Django.

En Django, los templates son archivos de texto que contienen marcadores o placeholders que luego son reemplazados por valores específicos cuando la página se renderiza dinámicamente. Estos marcadores están rodeados por doble llave y siguen la sintaxis del lenguaje de plantillas de Django

7.8.1 ¿Como se utlizan?

Si estas utilizando Visual Studio Code puedes descargarte su extension, pichando en la pestaña de la barra de navegacion izquierda y poniendo en el buscador *Django*. Te aparecera la primera.

- Los templates hacen que nustras app esten mejor estructuradas y organizadas para ello hay que crear nuestra carpeta de templates dentro de el directorio de la app.

```
cd myapp
mkdir templates
```

- En esta carpeta crearemos todas nuestras plantillas siempre organizandola de la mejor manera y la mas intuitiva para cuando llegue el mantenimiento de nuestro proyecto. En este caso practico las crearemos directamente en el directorio.

- Creamos un fichero por template.
  Ejemplo myapp/templates/pagina.html:

```
<h1>inicio</h1>
<p>Esta es la pagina de inicio</p>
```

- Ahora tenemosque modificar views.py:

```
def pagina(request, redirigir = 0):
    if redirigir == 1:
        return redirect('contacto', nombre="Jose Carlos", apellido="Limones")

    return render(request, 'pagina.html')
```

Usamos la funcion **render()** cuando queremos que nos devuelva un template, pasandole como argumento la avariable **request** y el **nombre del template** como minimo.

De esta forma nos mostrara nuestro primer template.

7.8.2. Layout, bloques y herencia de plantillas

- Como vemos el codigo no es limpio ¿como soluciona esto Django?

- Django utiliza un sistema de bloques y herencia que nos deja un proyecto estructurado y codigo limpio.

Ejemplo:

1) Creamos *layout.html* en nuestro directorio de templates y le añadimos una estructura basica de html.
2) En la etiqueta title creamos un bloque con el nobre title
3) Dentro del body crearemos un *div* que nos servira de ontenedor con el atributo *id* y el valor de *content*.
4) Dentro del div crearemos nuestro bloque.

Ejemplo layout.html

```
<!DOCTYPE html>
<html lang="es">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>
      {% block title %}

      {% endblock title %}
      | Jose Carlos Limones
   </title>
</head>
<body>
   <h1>Sitio web con Django | Jose Carlos Limones</h1>
   <hr/>
   <ul>
      <li>
         <a href="/">Inicio</a>
      </li>
      <li>
         <a href="/hola-mundo">Hola mundo</a>
      </li>
      <li>
         <a href="/pagina">Pagina 1</a>
      </li>
      <li>
         <a href="/contacto">contacto</a>
      </li>
   </ul>
   <hr/>
   <div id ="content">
      {% block content %}

      {% endblock content %}
   </div>
   <footer>
      Master en Python &copy Jose Carlos Limones
   </footer>
</body>
</html>
```

Ejemplo index.html

```
{% extends "layout.html" %}

{% block title %}
index
{% endblock title %}

{% block content %}
<h1>inicio</h1>
<p>Esta es la pagina de inicio</p>
{% endblock content %}
```

- En la primera linea le indicamos a Django el template de donde tiene que heredar, esto herda la estructura html.
- Como vemos los bloques son como etiquetas deben abrirse y cerrarse.
- A las etiquetas de bloques se le añadira el nombre del bloque donde debe ir.

> Si en nuestro contenedor tenemos algun contenido este se machacara al cargar el bloque.

Para solucinar esto debemos indicar a nuestro template que debe heredar el contenido insertando en el bloque lo siguiente:

```
{{ block.super }}
```

7.8.3. Vistas, Templates y Variables

Django nos da la posibilidad de enviar variables de nuestras vistas a los templates.

La forma de hacerlo es añador un tercer argumento al método **render()**:

Ejemplo views.py:

```
def index(request):
  return render(request, 'index.html', {
    'title': 'Inicio',
    'nombre': 'JC Limones',
    'mi_variable': 'Soy un dato en la vista'
    })
```

Ahora nos vamos a nuestro template y con la doble llave podemos insertar nuestra variable en el codigo.

Ejemplo index.html:

```
{% extends "layout.html" %}

{% block title %}

{{title}}

{% endblock title %}

{% block content %}

  <h1 class="title" >{{title}}</h1>
  <p>mi variable: {{mi_variable}}</p>
```

```
 <p>nombre: {{nombre}}</p>
 <p>Esta es la pagina de inicio</p>

{% endblock content %}
```

7.8.4. URLs en las templates

Tabla de contenidos

Si nos observamos el fichero *layout.html*, en los atributos href de los enlaces tenemos las direcciones *hardcodeadas*, es decir si tenemos que cambiar alguna url tambien la tendremos que cambiar en este fichero.

Podemos utilizar una instruccion de una template pasandole el nombre(name) de nuestra path en urls.py.

```
<li>
   <a href="{% url 'index' %}">Inicio</a>
</li>
<li>
   <a href="{% url  'hola_mundo'%}">Hola mundo</a>
</li>
<li>
   <a href="{% url 'pagina' %}">Pagina 1</a>
</li>
<li>
   <a href="{% url 'contacto' %}">contacto</a>
</li>
```

Desde esta instrucción podemos enviar los valores de las variables que necesitamos en orden de declaración.

```
<a href="{% url 'contacto' 'Jose Carlos' 'Limones'%}">contacto</a>
```

o asi:

```
<a href="{% url 'contacto' nombre='Jose Carlos' apellido='Limones'%}">contacto</a>
```

> Ya podemos cambiar la url del path en el fichero urls.py sin tener que cambiar los enlaces a esa pagina.

7.8.5.Fechas

Tabla de contenidos

Podemos mostrar la fecha de mnuestro servidor de la siguiente forma:

```
{% now "Y:M:D h:m:s" %}
```

7.8.6. Condicionales -if templates Django

Tabla de contenidos

Los condicionales if funcionan exactamente como en python, solo cambia su sintaxis. Si borramos la variable "nombre" pasada en el diccionario de la funcion render() veremos como funciona.

Ejemplo index.html:

```
{% if nombre %}
   <p>nombre: {{nombre}}</p>
{% else %}
      <strong>El nombre no existe</strong>
{% endif %}
```

7.8.7. Bucle -for template Django

Tabla de contenidos

Los bucles for al igual que los if solo cambia su sintaxis, veamos un ejemplo.

Ejemplo views.py:

```
def index(request):
   lenguajes = ['C', 'C++', 'Python', 'PHP', 'JavaScript']
   return render(request, 'index.html', {
      'title': 'Inicio',
      'nombre': 'JC Limones',
      'mi_variable': 'Soy un dato en la vista',
      'lenguajes': lenguajes
   })
```

Ejemplo index.html:

```
<h3>Listado de lenguajes</h3>
<ul>
   {% for lenguaje in lenguajes %}
   <li>{{lenguaje}}</li>
   {% endfor %}
</ul>
```

7.8.8. Funcionalidades extras de bucle -for

Tabla de contenidos

- for...empty:
    - Puedes utilizar **empty** para manejar el caso de que la secuencia este vacia

```
{% for item in my_list %}
   {{ item }}
{% empty %}
   La lista está vacía.
{% endfor %}
```

- for...if:
    - puedes usar la etiqueta if para filtrar los elementos que cumplen ciertas condiciones

```
{% for item in my_list %}
   {{ item }}
{% empty %}
   La lista está vacía.
{% endfor %}
```

- forloop:
    - La variable forloop proporciona informacionadicional sobre el bucle actual.

```
{% for item in my_list %}
   {{ forloop.counter }}: {{ item }}
   {% if forloop.last %}(Último elemento){% endif %}
{% endfor %}
```

- forloop.counter: Número de la iteración actual (comienza en 1).
- forloop.counter0: Número de la iteración actual, indexado desde 0.
- forloop.revcounter: Número de iteraciones desde el final (comienza en 1).
- forloop.revcounter0: Número de iteraciones desde el final, indexado desde 0.
- forloop.first: True si es la primera iteración.
- forloop.last: True si es la última iteración.

7.8.9. Filtros

Tabla de contenidos

En Django, los filtros son funciones que se aplican a las variables en las plantillas para modificar su contenido o formato. Los filtros permiten realizar diversas operaciones, desde el formateo de cadenas hasta la manipulación de datos en las plantillas. Los filtros se aplican a las variables utilizando el siguiente formato:

```
{{ variable|filtro }}
```

> Estos son unos ejemplos de filtros:

1) **date:**
    - **Descripción:** Formatea una fecha según el formato especificado.
    - **Ejemplo:** `{{ my_date|date:"F j, Y" }}`
2) **default:**
    - **Descripción:** Establece un valor predeterminado si la variable es nula.
    - **Ejemplo:** `{{ my_variable|default:"No disponible" }}`
3) **length:**
    - **Descripción:** Obtiene la longitud de una lista o cadena.
    - **Ejemplo:** `{{ my_list|length }}`
4) **lower y upper:**
    - **Descripción:** Convierte una cadena a minúsculas o mayúsculas.
    - **Ejemplo:** `{{ my_string|lower }}` / `{{ my_string|upper }}`
5) **default_if_none:**
    - **Descripción:** Establece un valor predeterminado solo si la variable es `None`.

- **Ejemplo:** `{{ my_variable|default_if_none:"No disponible" }}`

6) **floatformat:**
   - **Descripción:** Formatea un número de punto flotante según la cantidad de decimales especificada.
   - **Ejemplo:** `{{ my_float|floatformat:2 }}`
7) **slice:**
   - **Descripción:** Obtiene una porción de una lista o cadena.
   - **Ejemplo:** `{{ my_list|slice:":2" }}`
8) **yesno:**
   - **Descripción:** Retorna un valor específico según si la variable es True, False o None.
   - **Ejemplo:** `{{ my_bool|yesno:"Sí,No" }}`
9) **linebreaks:**
   - **Descripción:** Convierte saltos de línea en etiquetas de párrafo HTML.
   - **Ejemplo:** `{{ my_text|linebreaks }}`
10) **pluralize:**
   - **Descripción:** Añade una "s" al final de una palabra según el valor numérico proporcionado.
   - **Ejemplo:** `Hay {{ count }} mensaje{{ count|pluralize }}.`

Encuentra una lista completa de filtros en la documentación oficial de Django.

7.8.10. Crear filtros personalizados

Tabla de contenidos

Vamos a ver un ejemplo de como crear un filtro personalizado.

11) En el directorio de nuestra app creamos un subdirectorio llamado templatetags.
12) Dantro de este directorio creamos un fichero llamado **init**.py para que se convierta en un paquete.
13) Creamos el fichero donde iran nuestros filtros, ***filters.py***.
14) En el fichero lo primero que tenemos que hacer es importar las templates.
15) El siguiente paso e crear la función que ejecutara el filtro.
16) Ahora debemos cargar los filtros en la template para poder usarlos.
    Ejemplo filters.py:

```
from django import template

register = template.Library()
```

```
@register.filter(name='saludo')

def saludo(value):
    return f"<h1 style='background:green;'>Bienvenido, {value}.</h1>"
```

Ejemplo pagina.html (template):

```
{% load filters %}
{{ "jclimones" | saludo | safe }}
```

> ***safe*** convierte el html. Si no lo ejecutamos nos mjuestra el codigo como tal.

7.9. Includes en Django

7.9.1. {% include %}

Tabla de contenidos

En Django, el tag {% include %} se utiliza para incluir el contenido de otro archivo de plantilla dentro de la plantilla actual. Esto es útil para dividir grandes plantillas en partes más pequeñas y manejables, o para reutilizar bloques de contenido en varias plantillas.

La sintaxis basica es:

```
{% include '/ruta/nombre_de_archivo.html' %}
```

Ejemplo:

- Creamos una nueva template, por ejemplo **contenido_template.html**.

```
<p>Este es el contenido de la nueva template</p>
```

- Despues nos vamos a nuestro **index.html** y colocamos en la posicion que deseemos los siguiente:

```
{% include "contenido_template.html" %}
```

> Esto nos mostrara el contenido de la nueva template en el index.

Veamos que pasaria si le quisiera pasar en la nueva template una de las variables existente en la vista.

Ejemplo contenido_template.html:

```
<p>Este es el contenido de la nueva template y mi nombre es {{nombre}}</p>
```

Y si, respeta el valor de las variables de la vista.

7.9.2. {% include with %}

Tabla de contenidos

En Django, el {% include %} con la opción with permite pasar variables adicionales al archivo de plantilla incluido. Esto es útil cuando necesitas proporcionar datos específicos para el contenido que estás incluyendo.

La sintaxis básica es la siguiente:

```
{% include 'nombre_de_archivo.html' with variable1=valor1 variable2=valor2 %}
```

Ejemplo:

- En el archivo index.html modificamos el include de esta forma:

```
{% include 'contenido_template.html' with edad=", mi edad es de 38 años" %}
```

- El archivo contenido_template.html lo modificamos asi:

```
<p>Este es el contenido de la nueva template y mi nombre es {{nombre}} {{edad}}</p>
```

Esta tipo de includes te permite varias opciones como **only** que indica que solo debe utilizar las variables pasadas en el include.

Ejemplo de **only** en index.html:

```
{% include 'contenido_template.html' with edad=", mi edad es de 38 años" only %}
```

> Vemos como la variable nombre ya no aparece.

Las opciones son:

- **parsed**: La opción parsed se utiliza para indicar que el contenido de la plantilla incluida ya ha sido analizado. Esto es útil cuando se incluyen fragmentos de HTML preprocesados.

```
{% include 'nombre_de_archivo.html' with variable_html=mi_contenido_html parsed %}
```

- **language**: La opción language se utiliza para establecer el idioma en el que se debe procesar la plantilla incluida. Por ejemplo:

```
{% include 'nombre_de_archivo.html' with variable1=valor1 variable2=valor2 language='es' %}
```

> Estas opciones proporcionan flexibilidad al utilizar el tag include y permiten adaptar su comportamiento según las necesidades específicas de tu aplicación.

7.9.3. Comentarios

Tabla de contenidos

- Los comentarios en nuestras templates se pueden hacer como en html. El problema de esto es que si le damos al inspector nos aparecera entre el codigo html.
- Pero si lo hacemos con Django sera un comentario que queda para el codigo fuente.
- Se pueden hacer de dos formas:

```
1)   {% comment "" %}
     Esto es un comentario.
     {% endcomment %}

2)   {% comment "Esto es un comentario." %}

     {% endcomment %}
```

7.10. Archivos estáticos

Tabla de contenidos

Los archivos estáticos en el contexto del desarrollo web se refieren a recursos como archivos CSS, JavaScript e imagenes. Son aquellos que no cambian dinámicamente según la solicitud

del usuario. A diferencia de los archivos dinámicos que son generados y servidos por el servidor en tiempo real.

7.10.1. Estilos y apariencia visual con Django

Tabla de contenidos

Para incluir nuestro css en el proyecto debemos seguir los siguientes pasos:

- Creamos en nuestra app un directorio llamado static y dentro de esta otro llamado css.

```
mkdir static
cd static
mkdir css
```

- Ahora crearemos nuestro archivo **styles.css**.
- En ete Archivo daremos los estilos.

Ejemplo styles.css

```
body {
   font-family: Verdana, Geneva, Tahoma, sans-serif;
   background-color: aquamarine;
}
```

- El siguiente paso es cargar y linkear nuestro archivo.css. En la primera linea, antes de la linea de ***< !DOCTYPE html >*** cargamos los archivo estáticos.

Ejemplo layout.html:

```
{% load static %}
```

- En el ***< head >***

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

8.0. Modelos

Tabla de contenidos

En Django, un "modelo" es una representación de una tabla en una base de datos relacional. Los modelos son utilizados para definir la estructura de la base de datos y para interactuar con ella mediante consultas. Los modelos en Django son parte del sistema de Object-Relational Mapping (ORM), que permite interactuar con la base de datos utilizando objetos de Python en lugar de consultas SQL directas.

Aquí hay algunas características clave de los modelos en Django y para qué sirven.

8.1. Migraciones en Django

Tabla de contenidos

Las migraciones son una forma de realizar cambios en la estructura de la base de datos de manera controlada y evolutiva. Django utiliza un sistema de migraciones para llevar a cabo estas modificaciones.

Pasos Comunes:

- **Crear una Migración:**
    - Cuando realizas cambios en tus modelos, como agregar un nuevo campo o modificar una relación, necesitas crear una nueva migración para reflejar esos cambios.

```
python3 manage.py makemigrations
```

- **Aplicar Migraciones:**
    - Luego de crear una migración, debes aplicarla para actualizar la base de datos.

```
python3 manage.py migrate
```

- **Desplegar Migraciones Específicas:**
    - Puedes especificar el nombre de una migración para aplicar solo hasta cierto punto.

```
python3 manage.py migrate myapp 0003_migration_name
```

- **Desplegar Todas las Migraciones:**
    - Para aplicar todas las migraciones pendientes.

```
python3 manage.py migrate
```

- **Estado de Migraciones:**
    - Puedes verificar el estado actual de las migraciones.

```
python3 manage.py showmigrations
```

Rollback y Deshacer Migraciones:
- Para revertir la última migración:

```
python3 manage.py migrate myapp zero
```

- Para deshacer todas las migraciones y volver a un estado vacío:

```
python3 manage.py migrate myapp zero
```

8.2. Ejemplos para entender los modelos y migraciones

Tabla de contenidos

Recomendacion

> Es aconsejable utilizar el paquete de python llamado pylint.

- Pylint es una herramienta de análisis estático para código Python. Su objetivo principal es detectar errores y problemas en el código fuente, así como aplicar convenciones de estilo definidas en las PEP (Python Enhancement Proposals), como PEP 8.

Para más información -> pylint.

```
pip3 install pylint-django
```

Para usarlo solo debemos hacer esto:

```
pylint mi_archivo.py
```

> Para los ejemplos vamos a utilizar la base de datos que viene por defecto SQLite que esta almacenada en el fichero ***db.sqlite3***.

Si observamos el archivo de configuración de Django(*settings.py*) vemos que hay declarado un diccionario llamado ***DATABASES***. A continuacion te doy una explicacion con detalle:

1) **default**:
    - Esto indica que estamos configurando la base de datos predeterminada para nuestra aplicación. Puedes tener múltiples bases de datos en una aplicación Django y asignar nombres distintos a cada una.
2) **ENGINE**:
    - Aquí especificamos el motor de la base de datos que Django utilizará. En este caso, 'django.db.backends.sqlite3' indica que estamos usando SQLite como motor de base de datos. SQLite es una base de datos ligera que se almacena en un archivo local y es una opción común para el desarrollo.
3) **NAME**:
    - Esto especifica la ruta al archivo de la base de datos.
        - *BASE_DIR* es una variable que apunta al directorio base de tu proyecto Django. Con la expresión BASE_DIR / 'db.sqlite3', estamos construyendo la ruta completa al archivo db.sqlite3 dentro del directorio base (Que en el caso de nuestra aplicación esta en la raiz).

8.3. Crear y aplicar el primer modelo

Tabla de contenidos

- Localizamos y abrimnos el archivo **models.py**
- Creamos 2 clases, **Empleado** y **Puesto** con los siguientes atributos:

```
from django.db import models

class Puesto(models.Model):
    nombre = models.CharField(max_length=100)
    nivel = models.CharField(max_length=150)
    antiguedad = models.DateField()

class Empleado(models.Model):
    nombre = models.CharField(max_length=100)
    apellidos = models.CharField(max_length=100)
    edad = models.models.IntegerField()
    autorizado = models.BooleanField()
    carta_presentacion = models.TextField()
    fecha_alta = models.DateTimeField(auto_now_add=True)
```

> Para informacion sobre los tipos de campos -> models

- Tenemos que crear las migraciones, vamos al directorio donde se encuentra el archivo manage.py y ejecutamos:

```
python3 manage.py makemigrations
```

Esto generara una salida parecida a esta:

```
Migrations for 'myapp':
  myapp/migrations/0001_initial.py
    - Create model Puesto
    - Create model Empleado
```

- Aplicamos las migraciones:

```
python3 manage.py migrate
```

Salida:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, myapp, sessions
Running migrations:
  Applying myapp.0001_initial... OK
```

Para poder ver si todo se ha creado correctamente necesitamos una aplicación como ***DB Browser for SQLite***.

- Instalacion *SQLite*
  Debian/Ubuntu:

```
sudo apt-get update
sudo apt install sqlite3
```

- Instalacion *DB Browser for SQLite*

```
sudo apt-get update
```

- Con el siguiente comando se abrira la interfaz grafica de esta aplicacion.

```
sqlitebrowser
```

- En la pestaña file, pinchamos en abrir base de datos y buscamos en el directorio de nuestra aplicación, ahora en la barra de navegación izquierda deberia aparecernos nuestros modelos en forma de tabla SQL.

8.4. Relaciones entre Modelos

Tabla de contenidos

- Django facilita la definición de relaciones entre modelos. Por ejemplo, una relación de clave foránea se puede agregar para representar una relación de muchos a uno.

```
class Author(models.Model):
    name = models.CharField(max_length=50)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    publication_date = models.DateField()
```

- Crear un objeto relacionado:

```
author = Author.objects.create(name='F. Scott Fitzgerald')
book = Book.objects.create(title='The Great Gatsby', author=author, publication_date='1925-04-10')
```

- Consultar objetos relacionados:

```
books_by_fitzgerald = Book.objects.filter(author__name='F. Scott Fitzgerald')
```

> Estos son solo ejemplos básicos de cómo trabajar con bases de datos en Django. Django proporciona una API rica y potente para realizar consultas, gestionar relaciones y realizar operaciones en la base de datos de manera eficiente.

9.0. Bases de datos y Consiltas en Django

9.1. Consultas básicas (CRUD)

Tabla de contenidos

Django proporciona una API para realizar consultas a la base de datos de manera sencilla.

- **Obtener todos los objetos de un modelo**

```
all_books = Book.objects.all()
```

- **Filtrar resultados:**

```
recent_books = Book.objects.filter(publication_date__gte='2022-01-01')
```

- **Ordenar resultados:**

```
ordered_books = Book.objects.order_by('-publication_date')
```

- **Limitar resultados:**

```
first_two_books = Book.objects.all()[:2]
```

- **Obtener un solo objeto:**

```
specific_book = Book.objects.get(title='The Great Gatsby')
```

9.2. Ejemplo para CREAR(Create) un objeto en la base de datos

Tabla de contenidos

Como hemos aprendido no necesitamos utilizar(aunque se puede) consultas SQL para comunicarnos con la base de datos, si no que vamos a utilizar las capas de abstracción que nos ofrece este framework.

Ahora vemaos como se hace:

- Ejemplo views.py:

```
from myapp.models import Empleado, Puesto
def crear_empleado(request):
    empleado = Empleado(
        nombre = "Jose Carlos",
        apellidos = "Limones Hdez" ,
        edad = 38,
        autorizado = True,
        carta_presentacion = "Hola, esta es mi primera consulta a la base de datos desde Django",
    )

    empleado.save()
```

```
    return HttpResponse(f"Empleado creado: {empleado.nombre}")
```

- Ejemplo urls.py

```
path("crear_empleado", views.crear_empleado, name="crear_empleado")
```

Ahora si abrimos en el navegador la ***< url servido Django >/crear_empleado*** nos creara nuestro primer empleado en la tabla myapp_empleado.

- Si lo que queremos es pasarle los atributos de nuestro objeto por la url, nos vamos a urls.py

```
    path("crear_empleado/<str:nombre>/<str:apellidos>/<int:edad>/<str:autorizado>/<str:carta_presentacion>",
views.crear_empleado, name="crear empleado")
```

- Ejemplo views.py

```
def crear_empleado(request, nombre, apellidos, edad, autorizado, carta_presentacion):
    empleado = Empleado(
        nombre = nombre,
        apellidos = apellidos ,
        edad = edad,
        autorizado = autorizado,
        carta_presentacion = carta_presentacion,
    )

    empleado.save()

    return HttpResponse(f"Empleado creado: {empleado.nombre}")
```

- En el navegador:

```
< url servidor django >/Jose Carlos/Limones Hdez/38/True/Hola, esta es mi primera consulta a la base de datos
desde Django
```

- Ahora utilizaremos ***create*** en vez de ***save***:

- Ejemplo views.py

```
def crear_empleado(request, nombre, apellidos, edad, autorizado, carta_presentacion):
    empleado = Empleado.objects.create(
        nombre = nombre,
        apellidos = apellidos ,
        edad = edad,
        autorizado = autorizado,
        carta_presentacion = carta_presentacion,
    )
    return HttpResponse(f"Empleado creado: {empleado.nombre}")
```

9.3. Ejemplo LEER(Read) datos de la base de datos

Tabla de contenidos

- Ejemplo 1 views.py

```
def mostrar_empleado(request):
  empleado = Empleado.objects.get(pk=6)
  return HttpResponse(f"El empleado a mostrar es {empleado.nombre}")
```

- Ejemplo 2 views.py

```
def mostrar_empleado(request):
  empleado = Empleado.objects.get(nombre="Helena")
  return HttpResponse(f"El empleado a mostrar es {empleado.nombre}")
```

- Ejemplo urls.py

```
path('empleado', views.empleado, name="mostrar_empleado")
```

Ahora si escribimos la url correcta en el navegador nos mostrara la consulta.

9.3.1. Listar Elementos de la base de datos

- Creamos una template nueva empleados.html:

```
{% extends "layout.html" %}

{% block title %}
Listados de Empleados
{% endblock title %}

{% block content %}
<h1>Lista de empleados por nombre</h1>
<ul>
    {% for empleado in  empleados_por_nombre %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
    {% endfor %}
<ul>
<h1>Lista de empleados por id reverso</h1>
<ul>
    {% for empleado in  empleados_id_reverso %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
    {% endfor %}
<ul>
<h1>Lista de empleados por defecto</h1>
<ul>
    {% for empleado in  empleados_por_defecto %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
    {% endfor %}
<ul>
<h1>Lista de empleados por defecto(4 elementos)</h1>
<ul>
    {% for empleado in  empleados_hasta_id4 %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
    {% endfor %}
<ul>
<h1>Lista de empleados por defecto(del 5 al 7)</h1>
<ul>
    {% for empleado in  empleados_hasta_id4 %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
    {% endfor %}
<ul>
{% endblock content %}
```

- Ejemplo views.py

```
def empleados(request):
    empleados_por_nombre = Empleado.objects.order_by('nombre')
    empleados_id_reverso = Empleado.objects.order_by('-id')
    empleados_por_defecto = Empleado.objects.all()
    empleados_hasta_id4 = Empleado.objects.all()[:4]
    empleados_del_5_al_7 = Empleado.objects.all()[5:7]

    return render(request, 'empleados.html', {
        'empleados_por_nombre': empleados_por_nombre,
        'empleados_id_reverso': empleados_id_reverso,
        'empleados_por_defecto': empleados_por_defecto,
        'empleados_hasta_id4': empleados_hasta_id4,
        'empleados_del_5_al_7': empleados_hasta_id4,
    })
```

- Ejemplo urls.py

```
path('empleados/', views.empleados, name="empleados")
```

9.4. Ejemplo ACTUALIZAR(Update) datos de la base de datos

Tabla de contenidos

- Ejemplo 1 views.py

```
def editar_empleado(request, id):
    empleado = Empleado.objects.get(pk=id)
    empleado.autorizado = False
    empleado.save()
    return HttpResponse(f"La autorizacion de {empleado.nombre} a cambiado a {empleado.autorizado}")
```

- Ejemplo urls.py

```
path('editar_empleado/<int:id>', views.editar_empleado, name="editar_empleado")
```

9.5. Ejemplo ELIMINAR(Delete) datos y otros ejemplos de consultas a la base de datos.

Tabla de contenidos

- Creamos una template nueva empleados.html:

```
<h1>Lista de empleados por con opcion de borrado</h1>
<ul>
    {% for empleado in  empleados_por_defecto %}
        <li>{{empleado.nombre}} {{empleado.apellidos}}</li>
        <a href="{% url 'borrar' empleado.id %}">Borrar empleado</a>
    {% endfor %}
<ul>
{% endblock content %}
```

- Ejemplo views.py

```
def borrar_empleado(request, id):
    empleado = Empleado.objects.get(pk=id)
    empleado.delete()

    return redirect('empleados')
```

> Practica con otras opciones para apredender mas sobre consultas.

9.6. Lookups

Tabla de contenidos

Supongamos que tienes algunos registros de empleados en tu base de datos.

- Obtener todos los empleados:

```
empleados = Empleado.objects.all()
```

- Obtener empleados mayores de 30 años:

```
empleados_mayores_30 = Empleado.objects.filter(edad__gt=30)
```

- Obtener empleados autorizados:

```
empleados_autorizados = Empleado.objects.filter(autorizado=True)
```

- Obtener empleados cuya carta de presentación contiene la palabra 'Django':

```
empleados_con_django = Empleado.objects.filter(carta_presentacion__icontains='Django')
```

- Obtener empleados que fueron dados de alta después de una fecha específica:

```
from datetime import datetime
fecha_limite = datetime(2023, 1, 1)
empleados_despues_de_fecha = Empleado.objects.filter(fecha_alta__gt=fecha_limite)
```

- Obtener un empleado por nombre y apellidos exactos:

```
empleado_exacto = Empleado.objects.get(nombre='NombreExacto', apellidos='ApellidosExactos')
```

- Obtener empleados con imágenes nulas:

```
empleados_sin_imagen = Empleado.objects.filter(imagen='null')
```

- Obtener empleados ordenados por edad de forma descendente:

```
empleados_ordenados_por_edad = Empleado.objects.order_by('-edad')
```

- Estos son solo algunos ejemplos. Puedes combinar múltiples condiciones y utilizar diferentes operadores para crear consultas más complejas según tus necesidades

9.7. Consultas Avanzadas y Agregaciones en Django.

Tabla de contenidos

- Vamos a explorar algunas características más avanzadas relacionadas con consultas y agregaciones en Django.

9.7.1. Consultas Avanzadas:

Tabla de contenidos

- Consultas con múltiples condiciones:

> Puedes encadenar múltiples condiciones utilizando el método filter().

```
recent_fiction_books = Book.objects.filter(
    genre='Fiction',
```

```
    publication_date__gte='2022-01-01'
)
```

- Consultas con operadores lógicos:

> Django permite utilizar operadores lógicos como Q para realizar consultas más complejas.

```
from django.db.models import Q

books_query = Book.objects.filter(
    Q(genre='Fiction') | Q(genre='Mystery'),
    publication_date__gte='2022-01-01'
)
```

- Agregaciones en Django:

> Django proporciona funciones de agregación para realizar cálculos en conjuntos de datos.

- Conteo de objetos:

```
total_books = Book.objects.count()
```

- Agregación de valores:

```
from django.db.models import Sum

total_pages = Book.objects.aggregate(total_pages=Sum('num_pages'))
```

- Agrupación y Anotación:

> Puedes agrupar objetos y realizar anotaciones para agregar información adicional.

```
from django.db.models import Count

authors_with_book_count = Author.objects.annotate(book_count=Count('book'))
```

- Transacciones:

> Django maneja las transacciones de manera automática, pero también puedes utilizarlas explícitamente.

```
from django.db import transaction

try:
    with transaction.atomic():
        # Realizar operaciones en la base de datos
        book = Book.objects.create(title='New Book', author=author, publication_date='2023-01-01')
except IntegrityError:
    # Manejar errores de integridad
    pass
```

Estos son solo algunos ejemplos de las capacidades avanzadas de consultas y agregaciones en Django. La documentación oficial de Django es una excelente referencia para explorar más opciones y casos de uso específicos: Django QuerySet API.

- En nuestro ejemplo:
    - Ejemplo views.py

```
def empleados(request, empleado_id):
empleado = Empleado.objects.get(id=empleado_id)
puesto = empleado.puesto.name
nombre_empleado = empleado.nombre

context = {
   'nombre_empleado': nombre_empleado,
   'puesto': puesto,
}

return render(request, 'empleados.html', context)
```

- Ejemplo urls.py

```
path('empleados/<int:empleado_id>', views.empleados, name="empleados_id"),
```

- Ejemplo empleados. html

```
<h1>Empleado y cargo</h1>
<p>Nombre: {{ nombre_empleado }}</p>
<p>Puesto: {{ puesto }}</p>
```

10.0. Consultas(CRUD) utilizando SQL

Tabla de contenidos

Puedes realizar consultas directamente en SQL utilizando el método raw de Django. Aquí tienes algunos ejemplos de cómo podrías hacerlo:

- Crear datos:
  - Ejemplo views.py

```
from django.db import connection

def crear_empleado_sql(request):
   with connection.cursor() as cursor:
      nombre = "Ivan"
      apellidos = "Rodriguez"
      edad = 45
      autorizado = True
      carta_presentacion = "No uses div!"
      fecha_alta = "2023-12-22 12:00:00"

      # Consulta SQL para la inserción
      query = """
         INSERT INTO myapp_empleado (nombre, apellidos, edad, autorizado, carta_presentacion,
fecha_alta)
         VALUES (%s, %s, %s, %s, %s, %s);
      """

      # Ejecutar la consulta SQL
      cursor.execute(query, [nombre, apellidos, edad, autorizado, carta_presentacion, fecha_alta])

      # Importante: Hacer commit para aplicar los cambios
```

```
        connection.commit()
        return redirect('empleados')
```

> En el caso del ejemplo que hemos estado suiguiendo deberia borrar el campo de imagen de la base de datos para hacer la llamada mas facil.

- Leer datos:

> Ahora veremos un método mucho mas facil pero solo se puede usar en el caso e leer datos, el método ***raw()***

- Ejemplo views.py

```
def empleados(request):
    empleados_mayores_30 = Empleado.objects.raw("SELECT * FROM myapp_empleado WHERE edad > %s", [30])

    return render(request, 'empleados.html', {
        'empleados_mayores_30': empleados_mayores_30,
    })
```

  - Ejemplo empleados.html

```
<h1>Empleados Mayores de 30</h1>
<ul>
    {% for empleado in empleados_mayores_30 %}
        <li>{{ empleado.nombre }} {{ empleado.apellidos }} - Edad: {{ empleado.edad }}</li>
    {% endfor %}
</ul>
```

- Actualizar datos:

  - Ejemplo views.py

```
def actualizar_empleado_sql(request, id):
# Obtener los nuevos valores que deseas asignar
nuevo_nombre = "Nuevo nombre"
nuevos_apellidos = "Nuevos apellidos"
nueva_edad = 30
nuevo_autorizado = False
nueva_carta_presentacion = "Nueva carta de presentación"
nueva_fecha_alta = "2023-12-22 12:00:00"

with connection.cursor() as cursor:
    # Construir y ejecutar la consulta SQL de actualización
    query = """
        UPDATE myapp_empleado SET nombre=%s, apellidos=%s, edad=%s, autorizado=%s, carta_presentacion=%s, fecha_alta=%s WHERE id=%s;
        """
    cursor.execute(query, [nuevo_nombre, nuevos_apellidos, nueva_edad, nuevo_autorizado, nueva_carta_presentacion, nueva_fecha_alta, id])

return redirect('empleados')
```

- Eliminar datos:

  - Ejemplo views.py

```
    def eliminar_empleado_sql(request, nombre):
    with connection.cursor() as cursor:
        # Construir y ejecutar la consulta SQL de eliminación
        query = """
            DELETE FROM myapp_empleado
            WHERE nombre = %s;
        """
        cursor.execute(query, [nombre])

    return redirect('empleados')
```

10.1. Consultas con JOIN

Tabla de contenidos

En SQL, la cláusula JOIN se utiliza para combinar registros de dos o más tablas en base a una condición relacionada entre ellas. El propósito de un JOIN es combinar información de diferentes tablas que comparten una relación a través de claves primarias y foráneas.

- Ejemplo de JOIN:

```
SELECT *
FROM tabla1
LEFT JOIN tabla2 ON tabla1.columna = tabla2.columna;
```

- Ahora empezaremos a utilizar la relación entre tablas declarando un nuevo atributo a nuestra clase **Empleado**:

```
puesto = models.ForeignKey(Puesto, on_delete=models.CASCADE)
```

- Si deseas realizar una consulta que involucre ambas tablas (Empleado y Puesto), puedes hacerlo de la siguiente manera:

```
from django.db import connection

with connection.cursor() as cursor:
   cursor.execute("SELECT * FROM tuapp_empleado WHERE edad > %s", [30])
   empleados_mayores_30 = cursor.fetchall()
```

- Si deseas agregar filtros y ordenar los resultados:

```
with connection.cursor() as cursor:
   cursor.execute("""
      SELECT nombre, apellidos, edad
      FROM myapp_empleado
      WHERE edad > %s AND autorizado = %s
      ORDER BY edad DESC
   """, [30, True])
   resultados_filtrados = cursor.fetchall()
```

> Para aprender mas sobre SQL te recomiendo ir a SQL

11.0. Django admin Y operaciones CRUD

Tabla de contenidos

El panel de administración de Django es una herramienta poderosa que facilita la administración de modelos y la realización de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sin necesidad de crear interfaces de administración personalizadas.

- Activar el Panel de Administración:
    - Ejemplo myapp/admin.py

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

admin.site.register(Empleado)
admin.site.register(Puesto)
```

- Configura el superusuario usando:

```
python3 manage.py createsuperuser
```

- En el navegador añadiremos **/admin** a la direccion de nustro servidor local.
- Una vez dentro veremos los modelos que se crean por defecto y despues las de nuestra aplicación.

11.1. Operaciones CRUD básicas con admin:

Tabla de contenidos

- Crear un Nuevo Objeto:
    - En el panel de administración, selecciona tu modelo y haz clic en "Add".
- Leer (Ver) Objetos:
    - Visualiza todos los objetos y realiza búsquedas y filtros en el panel.
- Actualizar un Objeto:
  -Edita un objeto existente en el panel de administración.
- Eliminar un Objeto:
    - Selecciona un objeto y haz clic en "Delete".
- Personalización del Panel de Administración:
    - Puedes personalizar la apariencia y el comportamiento del panel de administración.
- Personalizar Listado de Objetos:
    - En el archivo **admin.py**, puedes definir la clase **ModelAdmin** y personalizar el listado de objetos.
        - Ejemplo admin.py

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

admin.site.register(Empleado)
admin.site.register(Puesto)
```

- Personalizar Listado de objetos:

```
class EmpleadoAdmin(admin.ModelAdmin):
    list_display = (
    'nombre',
    'apellidos',
    'edad',
    'autorizado',
    'carta_presentacion',
    'fecha_alta',
    )

admin.site.register(Empleado, EmpleadoAdmin)
```

- Filtros y busquedas por agrupaciones:

```
class EmpleadoAdmin(admin.ModelAdmin):
list_filter = ('nombre', 'fecha_alta')
search_fields = ('nombre', 'puesto__name')
date_hierarchy = 'fecha_alta'

admin.site.register(Empleado, EmpleadoAdmin)
```

- Personalización del Formulario de Edición:

> En este caso tenemos que cambiar nuestro modelo de empleado por < puestos = models.ManyToManyField(Puesto) >

```
fields = ('nombre', 'apellidos', 'edad', 'autorizado', 'carta_presentacion')
filter_horizontal = ('puestos',)
```

- Acceso a Modelos Relacionados:
  Si tienes relaciones entre modelos, el panel de administración facilita el acceso a los objetos relacionados.
    - Inlines:
        - Puedes usar clases inline para mostrar y editar objetos relacionados dentro del formulario principal.

```
class PuestoInline(admin.TabularInline):
model = Puesto
extra = 1


class EmpleadoAdmin(admin.ModelAdmin):
inlines = [PuestoInline]
```

> Estos son solo algunos ejemplos de cómo persolnalizar el panel de administración, para mas informacion visite Django admin.

11.2.0. Formularios en Django

Tabla de contenidos

A continuación vamos a aprender como utilizar Django con formularios utilizando varios métodos.

11.2.1. Método GET

Con este método obtenemos información de la base de datos mediante un formulario. Enviamos los datos de la consulta mediante la url y recibimos la respuesta del servidor.

- Ejemplo views.py:

```
from django.shortcuts import render, HttpResponse, redirect, get_object_or_404

def mostrar_el_empleado(request):
    return render(request, 'mostrar_1empleado.html')

def get_empleado(request):
    if request.method == 'GET':
        id_empleado = request.GET.get('id_empleado')
        print(f"El id enviado es: {id_empleado}")

        try:
            # Intenta obtener el objeto Empleado con el ID proporcionado
            empleado = get_object_or_404(Empleado, id=id_empleado)
            return render(request, 'mostrar_el_empleado.html', {'empleado': empleado})
        except:
            # Si no se encuentra el empleado, muestra un mensaje
            return render(request, 'mostrar_el_empleado.html', {'empleado': None})
    else:
        return HttpResponse("<h2>No se puede mostrar el empleado</h2>")
```

- Ejemplo urls.py:

```
path('mostrar_el_empleado/', views.mostrar_el_empleado, name="mostrar_el_empleado"),
path('get_empleado/', views.get_empleado, name="get"),
```

- Ejemplo template, mostrar_el_empleado.html:

```
{% extends "layout.html" %}

{% block title %}
El emlpeado es:
{% endblock title %}

{% block content %}
<h1 class="title">Busca el empleado</h1>
<form action="{% url 'get' %}" method="GET">
    <label for="id_empleado">id_empleado</label>
    <input type="number" name="id_empleado" placeholder="Pon el id del empleado">
    <input type="submit" value="Mostrar">
</form>
{% if empleado %}
    <ul>
        <li>El empleado es {{ empleado.nombre }} y su edad es de {{empleado.edad}} años</li>
    </ul>
{% else %}
    <p>Empleado no encontrado</p>
{% endif %}
{% endblock content %}
```

> En el capitulo excepciones estan las explicaiones sobre el manejo de excepciones.

11.2.2. Método POST

Tabla de contenidos

El método POST es uno de los métodos HTTP utilizados para enviar datos encriptados al servidor desde el cliente. Este método se utiliza comúnmente para enviar información sensible, como datos de formularios.

- Ejemplo views.py

```
def save_empleado(request):
if request.method == 'POST':
    # Obtener los datos del request
    nombre = request.POST['nombre']
    apellidos = request.POST['apellidos']
    edad = request.POST['edad']
    autorizado = request.POST['autorizado']
    carta_presentacion = request.POST['carta']

    empleado = Empleado.objects.create(
        nombre=nombre,
        apellidos=apellidos,
        edad=edad,
        autorizado=autorizado,
        carta_presentacion=carta_presentacion,
    )

    return redirect('empleados')
else:
    return HttpResponse("<h2>No se ha podido crear el nuevo empleado</h2>")

def create_empleado(request):
return render(request, 'create_empleado.html')
```

- Ejemplo urls.py

```
path('save_empleado/', views.save_empleado, name="save"),
path('create_empleado/', views.create_empleado, name="create_empleado"),
```

- Ejemplo template, create_empleado.html

```
{% extends "layout.html" %}

{% block title %}
Formularios en Django
{% endblock title %}


{% block content %}
<h1 class="title">Formularios en Django</h1>
<form action="{% url 'save' %}" method="POST">
   {% csrf_token %}
   <label for="nombre">Nombre</label>
   <input type="text" name="nombre" placeholder="Pon tu nombre">
   <label for="apellidos">Apellidos</label>
   <input type="text" name="apellidos" placeholder="Pon tus apellidos">
   <label for="edad">Edad</label>
   <input type="number" name="edad" id="edad" placeholder="Pon tu edad">
```

```
    <label for="check">Lo autorizas?</label>
    <select name="autorizado" id="autorizado">
      <option value="1">Si</option>
      <option value="0">No</option>
    </select>
    <label for="carta">Carta</label>
    <textarea name="carta" id="carta"></textarea>
    <input type="submit" value="Guardar">
</form>

{% endblock content %}
```

11.3. csrf_token

Tabla de contenidos

El token CSRF (Cross-Site Request Forgery) es una medida de seguridad que ayuda a proteger las aplicaciones web contra ataques CSRF. Un ataque CSRF ocurre cuando un atacante engaña a un usuario para que realice una acción no deseada en una aplicación web en la que el usuario está autenticado. El token CSRF en Django es una característica que ayuda a prevenir este tipo de ataques.

Detalles sobre {% csrf_token %} en Django:

- Generación del Token:
  - En un formulario HTML en Django, debes incluir el token CSRF para proteger contra ataques CSRF. Puedes agregar el token utilizando el tag {% csrf_token %}.

```
<form method="post" action="/mi_ruta/">
    {% csrf_token %}
    <!-- Campos del formulario -->
    <input type="text" name="nombre" />
    <input type="submit" value="Enviar" />
</form>
```

- Ubicación en el Formulario:
  - El tag {% csrf_token %} generalmente se coloca dentro del formulario, preferiblemente justo después de la apertura de la etiqueta `<form>` Django genera y maneja automáticamente el token.
- Necesidad del Token:
  - Django requiere que el token CSRF se incluya en todas las solicitudes POST para proteger contra ataques CSRF. Si intentas enviar un formulario POST sin incluir el token, Django generará un error Forbidden (403).
- Protección de Formularios Ajax:
  - Si estás enviando formularios a través de peticiones Ajax, también debes incluir el token CSRF en la solicitud. Puedes obtener el valor del token desde el atributo csrfmiddlewaretoken del formulario.

```
// Ejemplo de enviar un formulario mediante Ajax con jQuery
$.ajax({
    type: 'POST',
```

```
        url: '/mi_ruta/',
        data: {
            csrfmiddlewaretoken: $('input[name=csrfmiddlewaretoken]').val(),
            nombre: 'valor_del_nombre',
        },
        success: function(response) {
            console.log(response);
        },
    });
```

- Manejo en Vistas:
    - En el lado del servidor, Django maneja automáticamente la validación del token CSRF. No es necesario realizar ninguna acción especial en tus vistas para verificar el token, siempre y cuando el formulario se envíe correctamente.
- Doble Verificación en Formularios Ajax (Opcional):
    - Si estás construyendo una aplicación con formularios Ajax, puedes duplicar la verificación del token CSRF en el servidor accediendo a request.POST['csrfmiddlewaretoken'] en tus vistas para garantizar la seguridad adicional.
      El uso de {% csrf_token %} es una buena práctica en el desarrollo de aplicaciones Django para garantizar la seguridad de las solicitudes POST. Este enfoque ayuda a proteger tu aplicación contra ataques CSRF y mantiene un nivel adicional de seguridad en la interacción entre el cliente y el servidor.

11.4. Formularios Basados en Clases

Tabla de contenidos

En Django, los formularios basados en clases son una forma de definir y trabajar con formularios utilizando clases de Python en lugar de funciones. Estos formularios están diseñados para aprovechar la orientación a objetos y proporcionan una estructura más organizada y reutilizable para definir la lógica del formulario.

- Ejemplo views.py

```
from myapp.forms import FormEmpleado # Importamos el forms.py que crearemos a continuación

def create_full_empleado(request):
if request.method == 'POST':
    formulario = FormEmpleado(request.POST)


    if formulario.is_valid():
        data_form = formulario.cleaned_data

        nombre = data_form['nombre']
        apellidos = data_form['apellidos']
        edad = data_form['edad']
        autorizado = data_form['autorizado']
        carta = data_form['carta']
        print('Esta entrando')
        empleado = Empleado.objects.create(
            nombre = nombre,
            apellidos = apellidos ,
```

```
            edad = edad,
            autorizado = autorizado,
            carta_presentacion = carta,
        )
        return redirect('empleados')
else:
    formulario = FormEmpleado()
    print('No esta entrando')

return render(request, 'create_full_empleado.html', {
    'form': formulario
})
```

- Método HTTP:
  - Verifica si la solicitud es una solicitud POST.
  - Si es POST, significa que el formulario ha sido enviado.
- Creación del formulario:
  - Crea una instancia del formulario FormEmpleado utilizando los datos de la solicitud POST (si está presente). Si no hay datos POST, se crea un formulario vacío.
  - FormEmpleado es un formulario basado en una clase, presumiblemente definido en algún lugar de tu aplicación, y contiene campos para los atributos de un Empleado, como nombre, apellidos, edad, etc.
- Validación del formulario:
  - Verifica si el formulario es válido llamando al método is_valid(). Este método realiza validaciones en los datos ingresados y devuelve True si el formulario es válido, o False en caso contrario.
- Acceso a los datos del formulario:
  - Si el formulario es válido, se accede a los datos limpios (cleaned_data) del formulario. Estos son los datos que han pasado las validaciones del formulario.
  - Se extraen los valores específicos del formulario, como nombre, apellidos, edad, etc.
- Creación del objeto Empleado:
  - Se utiliza la función Empleado.objects.create() para crear un nuevo objeto Empleado en la base de datos utilizando los valores extraídos del formulario.
- Redirección:
  - Después de crear el objeto Empleado, redirige al usuario a la vista 'empleados'. Puedes ajustar esto según tus rutas y nombres de vistas específicos.
- Formulario no válido o solicitud GET:

- Si la solicitud no es POST (es decir, una solicitud GET) o si el formulario no es válido, se renderiza la plantilla 'create_full_empleado.html' con el formulario para que el usuario pueda completar los datos.

- Ejemplo urls.py

```
path('create_full_empleado/', views.create_full_empleado, name="create_full_empleado")
```

- A continuacion crearemos un archivo en el directorio de nuestra aplicacion ***myapp*** llamado forms.py
    - Ejemplo forms.py

```
from django import forms

class FormEmpleado(forms.Form):
    nombre = forms.CharField(
        label = 'Nombre'
    )
    apellidos = forms.CharField(
        label = 'Apellidos'
    )
    edad = forms.IntegerField(
        label = 'Edad'
    )
    opciones = [
        (1, 'Si'),
        (0, 'No'),
    ]
    autorizado = forms.TypedChoiceField(
        label = '¿Autorizado?',
        choices=opciones,
        widget=forms.Select
    )
    carta = forms.CharField(
        label = 'Carta de presentacion',
        widget=forms.Textarea
    )
```

- Ahora cargaremos nuestro formulario en la template:
    - Ejemplo template, create_full_empleado.html

```
{% extends "layout.html" %}

{% block title %}
Formularios en Django
{% endblock title %}


{% block content %}
<h1 class="title">Formularios en Django</h1>
<form action="" method="POST">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Guardar">
</form>
```

```
{% endblock content %}
```

11.5. Validacion de formularios "Validators"

Tabla de contenidos

- Los "validators" son funciones o clases que se utilizan para validar los datos ingresados en los campos de un modelo. Estos validators permiten establecer reglas específicas sobre qué datos son aceptables y cuáles no en un campo determinado.
- En la template debemos insertar una condicion para mostrar el error:

```
{% if form.errors %}
<strong>Hay Errores en el formulario</strong>
{% endif %}
```

- Ejemplo forms.py

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

class FormEmpleado(forms.Form):
    nombre = forms.CharField(
        label = 'Nombre',
        required=True,
        validators=[
            validators.MaxLengthValidator(20, 'El nombre es demasiado largo'), # El número máximo de carácteres es de 20.
            validators.RegexValidator('^[A-Za-z0-9ñ ]*$', 'Caracteres no validos', 'Invalid_name') # Solo se puede utilizar los carácteres especificados.
        ]
    )
```

> Esto es solo un ejemplo, para mas detalles visite Django validators .

11.6. Mensajes/Sesiones Flash

Tabla de contenidos

Las sesiones flash son un mecanismo utilizado en muchos frameworks web, incluyendo Django, para enviar mensajes temporales entre las solicitudes del usuario. Estos mensajes generalmente se utilizan para mostrar notificaciones o mensajes informativos después de realizar una acción, como enviar un formulario.

- Ejemplo viwes.py

```
from django.contrib import messages

def create_full_empleado(request):
    if request.method == 'POST':
        formulario = FormEmpleado(request.POST)

        if formulario.is_valid():
            data_form = formulario.cleaned_data

            nombre = data_form['nombre']
            apellidos = data_form['apellidos']
```

```
            edad = data_form['edad']
            autorizado = data_form['autorizado']
            carta = data_form['carta']
            empleado = Empleado.objects.create(
               nombre = nombre,
               apellidos = apellidos ,
               edad = edad,
               autorizado = autorizado,
               carta_presentacion = carta,
            )

            messages.success(request, f'El empleado: {empleado.id} llamado {empleado.nombre} se ha creado correctamente')

            return redirect('empleados')
      else:
         formulario = FormEmpleado()

      return render(request, 'create_full_empleado.html', {
         'form': formulario
      })
```

- Ejemplo template:

```
{% if messages %}
{% for message in messages %}
    <div class="message">
        {{ message }}
    </div>
{% endfor %}
{% endif %}
```

12.0. Excepciones

Tabla de contenidos

En Django, el manejo de excepciones se utiliza para controlar situaciones inesperadas o errores durante el procesamiento de una solicitud. A continuación, te proporcionaré una visión general del manejo de excepciones en Django:

1. Http404:
   En el contexto de Django, Http404 es una excepción específica que se utiliza para indicar que la página solicitada no se ha encontrado. Puedes levantar esta excepción manualmente en tu vista si deseas mostrar una página de error 404 personalizada.

```
from django.http import Http404

def mi_vista(request):
   # Algo salió mal y queremos mostrar una página de error 404
   raise Http404("La página que buscas no se encuentra.")
```

- Manejo de Excepciones en Vistas:
    - En las vistas de Django, puedes utilizar bloques try y except para manejar excepciones y tomar acciones específicas cuando se producen.

```
from django.http import HttpResponse

def mi_vista(request):
    try:
        # Código que puede generar una excepción
        resultado = 1 / 0
    except ZeroDivisionError:
        # Manejo específico para la excepción ZeroDivisionError
        return HttpResponse("No se puede dividir por cero.")
```

- Manejo de Excepciones en Plantillas:
    - Puedes manejar excepciones directamente en plantillas de Django utilizando el bloque {% try %} ... {% except %} ... {% endtry %}.

```
{% try %}
    {{ variable|default:"No disponible" }}
{% except %}
    Ocurrió un error al obtener la variable.
{% endtry %}
```

- Middleware de Manejo de Excepciones:
    - Django proporciona un middleware llamado django.middleware.common.CommonMiddleware que maneja automáticamente las excepciones Http404 y redirige a la página de error 404 definida en tu configuración.

```
MIDDLEWARE = [
    # ...
    'django.middleware.common.CommonMiddleware',
    # ...
]
```

- Personalización de Páginas de Error:
    - Puedes personalizar las páginas de error 404 y 500 en tu aplicación. Django busca plantillas específicas, como 404.html y 500.html, en tu directorio de plantillas para mostrar páginas de error personalizadas.
- Manejo de Excepciones Genéricas:
    - Django también proporciona clases de vistas genéricas, como django.views.generic.base.View, que manejan automáticamente ciertas excepciones, como Http404.

> [!NOTE]
> En resumen, el manejo de excepciones en Django es una parte fundamental para mejorar la robustez y la experiencia del usuario al enfrentar situaciones imprevistas durante la ejecución de tu aplicación web.

12.1. get_object_or_404

Tabla de contenidos

get_object_or_404 es una función útil en Django que se utiliza para recuperar un objeto de la base de datos según ciertos criterios de búsqueda. Su propósito principal es simplificar el manejo de errores cuando se trabaja con bases de datos.

- Función:
    - get_object_or_404 es una función proporcionada por Django que reside en django.shortcuts.
    - Se utiliza para intentar obtener un objeto de la base de datos. Si el objeto no existe, en lugar de devolver None, devuelve un error Http404 que se puede manejar para mostrar una página de error personalizada.
- Parámetros:
    - El primer argumento es el modelo desde el cual deseas obtener el objeto.
    - Los argumentos siguientes son utilizados para filtrar el objeto. En el ejemplo, id=id_objeto significa que se está filtrando por el campo id del modelo, y se compara con el valor proporcionado en id_objeto.
- Manejo de Errores:
    - Si el objeto se encuentra, get_object_or_404 lo devuelve.
    - Si el objeto no se encuentra, lanza una excepción Http404.
      Esta excepción se captura comúnmente en una vista de Django, y luego puedes personalizar cómo manejarla, como redirigir a una página de error 404 personalizada o mostrar un mensaje específico.

13.0. Fuentes

Tabla de contenidos

- Django projects
- Tutorial Python
- Instalacion de Python
- Instalacion VSCode