

Crear Un Nuevo Proyecto Django

Para crear un nuevo proyecto se debe abrir una terminal y posicionarse en la carpeta que va a contener el proyecto de Django. Luego se usa el comando `django-admin startproject NombreDeMiProyecto` (reemplazar `NombreDeMiProyecto` por el nombre adecuado para tu proyecto).

Por ejemplo, para crear un proyecto en la ubicación `Documentos/ProyectosDjango`, suponiendo que previamente se ha creado la carpeta `ProyectosDjango` dentro de `Documentos`, debería hacerlo del siguiente modo:

```
~$ cd Documentos/ProyectosDjango/  
~/Documentos/ProyectosDjango$ django-admin startproject Catalogo  
~/Documentos/ProyectosDjango$
```

En este ejemplo se crea el proyecto "Catalogo" dentro de la carpeta `ProyectosDjango` que se encuentra dentro de la carpeta `Documentos`. Esto crea una carpeta con el nombre del proyecto, en este caso "Catalogo" que contiene toda la estructura del proyecto.

Al abrir la carpeta del proyecto, continuando con el ejemplo de `Catalogo`, se encuentra una carpeta con el mismo nombre del proyecto. Esta carpeta es la **aplicación** principal del proyecto. También se encuentra el archivo **`manage.py`**.

Documentos/

└─ ProyectosDjango/

└─ Catalogo/ <- Carpeta del proyecto Django

 └─ Catalogo/ <- Carpeta de la aplicación principal

 └─ manage.py

- **manage.py**: Es un script de utilidad proporcionado por Django para realizar diversas tareas de administración y gestión del proyecto. Permite ejecutar comandos desde la línea de comandos para realizar acciones como crear aplicaciones, ejecutar el servidor de desarrollo, realizar migraciones de base de datos, crear un superusuario para el panel de administración, entre otras tareas comunes en un proyecto Django.
- **Aplicaciones**: Las "aplicaciones" son componentes reutilizables y modulares que ayudan a organizar y dividir la funcionalidad del proyecto en partes más pequeñas y manejables. Cada aplicación tiene su propio conjunto de *modelos*, *vistas*, *templates* y *rutas*, lo que permite una estructura organizada y facilita la reutilización en otros proyectos. Realizan una tarea específica o manejan una característica particular de

la aplicación en general, como gestionar usuarios, mostrar contenido, manejar comentarios, etc.

Por ejemplo en el proyecto se podrían crear las aplicaciones *Productos*, *Categorías*, *CarritoDeCompras*, *Usuarios*, *Comentarios* y cada una de estas aplicaciones tendría su propia estructura.

Cuando abra el proyecto desde un IDE, voy a abrir la carpeta del proyecto.

Estructura De La Aplicación Principal Del Proyecto

En la carpeta de la aplicación principal del proyecto se van a encontrar los siguientes archivos:

```
└─ Catalogo/          <- Carpeta del proyecto Django
    └─ Catalogo/      <- Carpeta de la aplicación principal
        │   └─ __init__.py
        │   └─ asgi.py
        │   └─ settings.py
        │   └─ urls.py
        │   └─ wsgi.py
    └─ manage.py
```

- `__init__.py`: Marca el directorio como un paquete de Python, utilizado para organizar el código y permitir importaciones.
- `asgi.py`: (Asynchronous Server Gateway Interface). Punto de entrada para la comunicación asincrónica (eventos en tiempo real) con servidores compatibles con ASGI.
- `settings.py`: Es uno de los archivos más importantes de un proyecto Django, ya que define cómo la aplicación se comportará y se conectará con otros componentes. Este archivo es fundamental para la configuración de la aplicación Django. Contiene diversas variables y configuraciones que determinan el comportamiento de la aplicación, como la base de datos que se utilizará, el middleware, la configuración de aplicaciones instaladas, la configuración de la interfaz de administración, configuraciones de caché, internacionalización, etc.
- `urls.py`: En este archivo se define el enrutamiento de URLs para la aplicación. Aquí se mapean las URL a las vistas correspondientes, lo que

significa que cuando un usuario visita cierta URL, Django sabe qué vista debe llamar para procesar la solicitud y devolver una respuesta adecuada.

- `wsgi.py`: (Web Server Gateway Interface) Punto de entrada para servidores web compatibles con WSGI, permite la comunicación con el servidor para servir la aplicación.

El servidor web utiliza la especificación WSGI para comunicarse con la aplicación Django, lo que permite que la aplicación reciba solicitudes HTTP, las procese y devuelva respuestas al servidor web.

Iniciar El Servidor

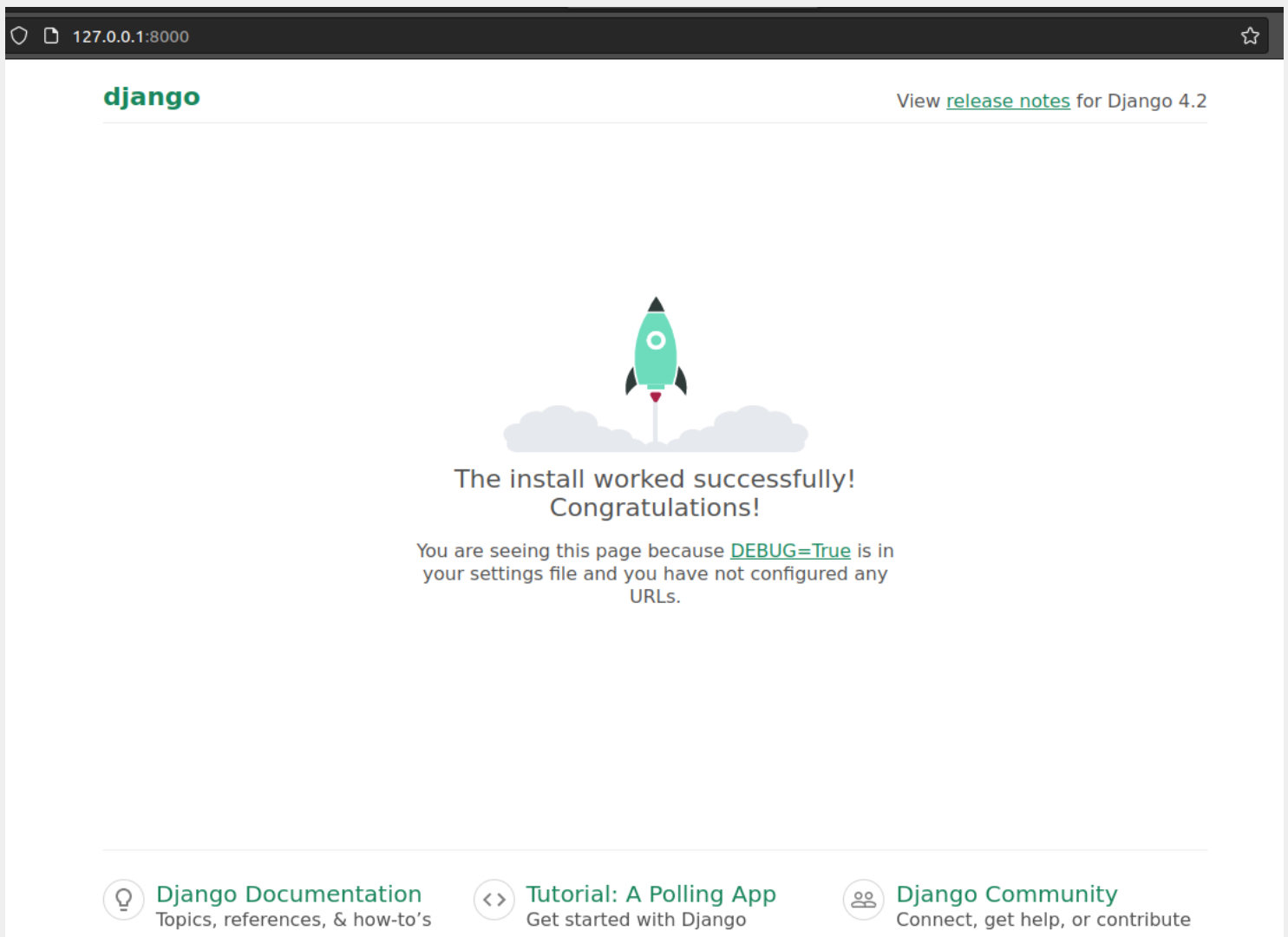
Para ejecutar la aplicación en el entorno de desarrollo, se debe usar la terminal, puede ser la del sistema operativo o una integrada que tenga el IDE que se esté usando.

En la terminal hay que posicionarse en la carpeta principal del proyecto, la que contiene el archivo **`manage.py`**. Luego se debe usar el comando **`python manage.py runserver`**.

La terminal mostrará varios mensajes, pero por ahora solo nos concentraremos en el siguiente:

Starting development server at <http://127.0.0.1:8000/>

Que indica que el servidor está corriendo en *localhost* en el puerto 8000. Para acceder a la aplicación se debe abrir el navegador y navegar a la ruta <http://127.0.0.1:8000/>. El navegador debe mostrar la página de bienvenida del framework, similar a la siguiente imagen.



URLs, Views y Templates

Vamos a crear una ruta que nos devuelva un template simple, pero que va a servir para ver el concepto de cómo funcionan las rutas, vistas y templates dentro del framework.

Es muy poco frecuente incluir templates y vistas dentro de la aplicación principal, pero para este ejemplo que es simple resulta una buena opción.

En la aplicación principal, vamos a crear un template que muestre la página de *Home* del proyecto. Para esto hay que agregar en la aplicación principal una carpeta llamada **templates** y dentro de esta, otra carpeta que debe tener el mismo nombre de la aplicación, en este caso, **Catalogo**. Esta última es la que va a contener el template HTML que, para este ejemplo, se va a llamar *home*.

También debemos agregar un nuevo archivo llamado **views.py**. El archivo views, es el que va a contener el código que va a manejar la lógica de la aplicación, es decir son las vistas. Hay que recordar que Django usa el patrón de diseño MTV (Model-View-Template).

Con estas modificaciones la estructura del proyecto se ve del siguiente modo:

```
└─ Catalogo/          <- Carpeta del proyecto Django
   └─ Catalogo/       <- Carpeta de la aplicación principal
      └─ templates
         └─ Catalogo
            └─ home.html
      └─ __init__.py
      └─ asgi.py
      └─ settings.py
      └─ urls.py
      └─ views.py
      └─ wsgi.py
└─ manage.py
```

También vamos a editar el archivo **settings.py** para agregar la aplicación a la lista de aplicaciones disponibles del proyecto. El archivo contiene una lista de python con el nombre **INSTALLED_APPS** y simplemente hay que agregar un elemento de tipo string a la lista con el nombre de la aplicación.

Ya está lista la estructura de archivos y carpetas necesarios, ahora hay que trabajar en el template, la vista y la ruta.

Template

En el template **home.html** (Catalogo/Catalogo/templates/Catalogo) basta con incluir algún texto que indique que se está viendo la página principal. En este caso va a incluir el texto "**Home**" dentro de una etiqueta **h1**.

Vista

En django se pueden usar dos tipos de vistas, las basadas en clases y las que son funciones. Vamos a ver vistas basadas en funciones.

En el archivo **views.py** vamos a importar la función **render**:

```
from django.shortcuts import render
```

Luego definimos una función que simplemente va a tener como tarea renderizar el template **home**. El nombre de la función también va a ser **home** como el template, pero puede ser cualquier otro.

```
def home(request):  
    return render(request, 'Catalogo/home.html')
```

Las vistas siempre deben tener como argumento **request**. En Django, el elemento request es un objeto que contiene toda la información sobre la solicitud (request) que el cliente (generalmente un navegador web) hace al servidor.

Request es un objeto de la clase HttpRequest, proporcionado por Django, y contiene una variedad de información útil, como los parámetros enviados en la URL, los datos del formulario enviado por el cliente, la dirección IP del cliente, la sesión del usuario, las cookies, entre otros.

El nombre request es una convención comúnmente utilizada en Django para representar el objeto que contiene la solicitud (request) del cliente.

Las vistas, además de request, pueden tomar más argumentos si son necesarios.

Por último está la función render, que devuelve un objeto **HttpResponse**, a la que en este caso le pasamos request y un string con la ruta al template.

HttpResponse es la respuesta HTTP con el contenido de la plantilla HTML renderizada, listo para ser mostrado en el navegador.

Ruta

Por último, en el archivo **urls.py** definimos la ruta para acceder al *home*. En este caso la ruta va a ser un tanto particular, ya que va a apuntar a la ruta raíz o principal del proyecto.

Al abrir el archivo vamos a encontrar el siguiente contenido:

```
from django.contrib import admin  
  
from django.urls import path,  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
]
```

Hay que importar la vista creada anteriormente.

```
from . import views (importamos todo el módulo)
```

urlpatterns es una lista que contiene las urls que van a estar disponibles en la aplicación. Para agregar una nueva ruta hay que agregar un nuevo elemento a

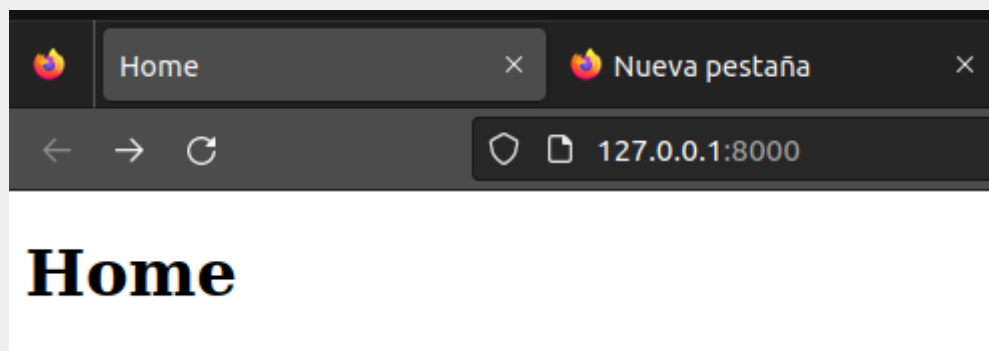
la lista, usando la función **path** como en la ruta *admin* que está definida en el archivo. Quedaría del siguiente modo:

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', views.home, name='home')  
]
```

El primer argumento de **path** es un string que representa la ruta que el cliente va a usar para hacer la petición al servidor. El segundo es la vista que resuelve la petición al servidor, en este caso la vista **home** dentro del módulo **views**. El tercero es un nombre que luego puede ser utilizado dentro de la aplicación para hacer referencia a la ruta, es opcional pero muy útil.

Una vez terminado, se resuelve el objetivo que era renderizar un template que muestre la página *Home* del proyecto.

Ahora, si ejecutamos nuevamente el servidor de desarrollo desde la terminal y en el navegador nos dirigimos a <http://127.0.0.1:8000> se ve la página que definimos como home del proyecto.



Al definir una ruta que resuelva una petición en la ruta raíz del proyecto, ya no se muestra la página de bienvenida del framework.

El funcionamiento es bastante simple. El cliente hace una petición al servidor a través de la **ruta**. En la configuración la ruta tiene definida una **vista** que va a resolver la petición, en este caso, la vista devuelve una respuesta con el renderizado el **template** al cliente.

cliente -> ruta -> vista -> template -> cliente

Como se señaló anteriormente, es muy poco frecuente incluir templates y vistas dentro de la aplicación principal. Este fue un simple ejemplo que sirvió para mostrar el funcionamiento de las rutas, vistas y templates. Para el resto del proyecto se crean aplicaciones con sus propias rutas, vistas y templates.

Crear Aplicaciones Dentro Del Proyecto

Siguiendo con el ejemplo del Catálogo, vamos a crear dentro del proyecto, la aplicación *Productos*. Para crear aplicaciones abrimos la terminal, vamos a estar posicionados en la carpeta principal del proyecto, la que contiene el archivo `manage.py`, y ejecutamos el siguiente comando:

python manage.py startapp NombreDeLaAplicacion.

En nuestro caso es: *python manage.py startapp Productos.*

Ahora la estructura del proyecto se ve del siguiente modo:

Documentos/

└─ ProyectosDjango/

└─ Catalogo/ ← Carpeta del proyecto

| └─ Catalogo/ ← Carpeta de la aplicación principal

| └─ templates

| | └─ Catalogo

| | | home.html

| └─ __init__.py

| └─ asgi.py

| └─ settings.py

| └─ urls.py

| └─ views.py

| └─ wsgi.py

└─ manage.py

└─ Productos/ ← Nueva aplicación

| └─ __init__.py

| └─ admin.py

| └─ apps.py

| └─ models.py

| └─ tests.py

| └─ views.py

Para que la aplicación tenga su estructura completa, falta crear **dentro de la carpeta de la aplicación**, que en este caso es “Productos”, un archivo para registrar las rutas, ***urls.py*** y una carpeta para los templates. La carpeta de templates debe tener como nombre “***templates***” y debe incluir una carpeta con el mismo nombre de la aplicación, en este caso “***Productos***”.

Ahora la estructura de la aplicación se ve del siguiente modo:

```
Productos/  
├─ migrations/  
|   └─ ...  
├─ templates/  
|   └─ Productos/  
├─ __init__.py  
├─ admin.py  
├─ apps.py  
├─ models.py  
├─ tests.py  
├─ urls.py  
└─ views.py
```

Registrar La Aplicación

En la aplicación principal, en el archivo settings.py se agrega la aplicación a la lista ***INSTALLED_APPS***.

Con esta estructura de proyecto vamos a desarrollar dos tareas de ejemplos.

- La primera tarea es mostrar un template para Productos, muy parecido al home.
- La segunda tarea es crear un CRUD de Productos.

Desarrollando estos ejemplos, vamos a ver varios conceptos:

urls, views, templates, modelos, conexión a una base de datos, ORM de Django, Formularios basados en modelos, [Lenguaje del motor de template de Django](#), peticiones por get y post, request, response.

Template Productos

Dentro de la carpeta de templates de la aplicación *Productos* (Productos/templates/Productos) creamos un nuevo template, que para este ejemplo lleva como nombre "*productos.html*". Por ahora solo va a contener el texto "Productos" dentro de una etiqueta h1.

View Productos

Por ahora este view va a ser igual al de la aplicación principal. Es decir que simplemente va a renderizar el template. El view se ve del siguiente modo:

```
from django.shortcuts import render  
  
def productos(request):  
    return render(request, 'Productos/productos.html')
```

URLs Productos

En este punto vamos a hacer dos cosas:

- Definir la ruta de la aplicación
- Incluir las rutas de *Productos* en la aplicación principal

Las rutas de las aplicaciones se incluyen en el archivo `urls.py` de la aplicación principal para centralizar y organizar la configuración de las URL del proyecto. Esto permite mantener una estructura clara y modular, separando las funcionalidades de las diferentes aplicaciones y facilitando su mantenimiento y escalabilidad. Además, al ubicar las rutas en la aplicación principal, se pueden agregar o quitar aplicaciones sin afectar la configuración global del proyecto.

En `urls.py` de la aplicación *Productos*, definimos la ruta que va a apuntar al view. El contenido del archivo se va a ver del siguiente modo:

```
from django.urls import path  
from . import views      <!-- importo el módulo views de la aplicación  
  
urlpatterns = [  
    path('productos', views.productos, name='productos')  
]
```

Ahora incluimos las urls de la aplicación *Productos* en las urls de la aplicación principal. Para esto editamos `urls.py` de la aplicación principal (Catalogo/Catalogo/urls.py). Se importa la función *include* y en `urlpatterns` se agrega un path que apunta al archivo de rutas de *Productos*. El archivo editado queda del siguiente modo:

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

from . import views

urlpatterns = [

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

    path('', views.home, name='home'),

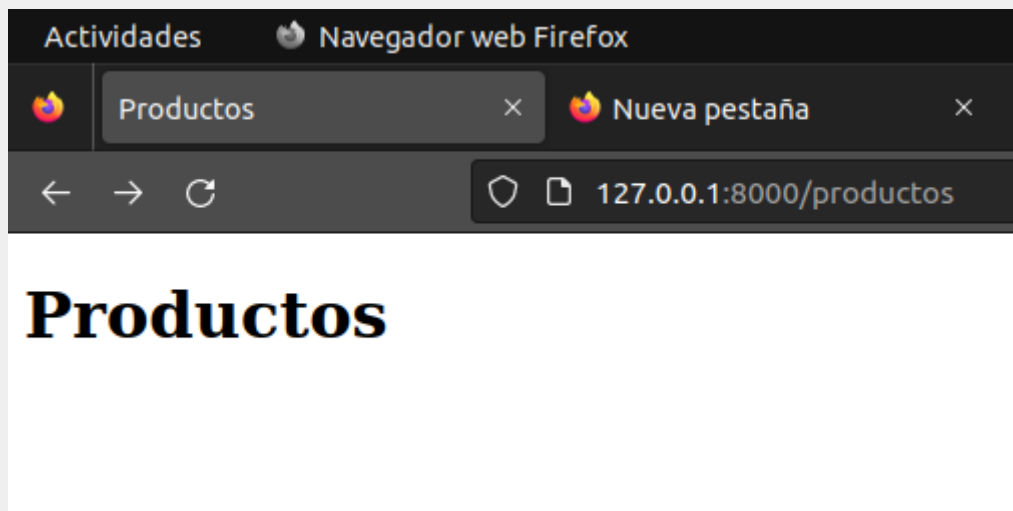
    path('', include('Productos.urls'))

]
```

Probar La Ruta

Ya se creó el template de *Productos*, la vista que renderiza el template, la ruta que apunta la vista y el archivo de urls de *Productos* se incluyó en las urls de la aplicación principal.

Para probar la nueva ruta hay que ejecutar el servidor local (*python manage.py runserver*) y en el navegador ir a la ruta 127.0.0.1:8000/productos



CRUD

Para crear un crud primero vamos a definir un modelo, luego vamos a migrar a la base de datos (sqlite) para crear la tabla. También vamos a crear rutas, vistas y templates para cada una de las funcionalidades.

Modelos

En Django, un modelo es una clase de Python que representa una entidad del proyecto (por ejemplo, usuario, producto, factura, remito, inscripción, etc). Cada modelo va a representar una tabla en la base de datos, cada atributo de la clase representa un campo de la tabla, y los métodos pueden definir funcionalidades asociadas a los datos.

Django utiliza estos modelos para crear y manipular tablas en la base de datos, permitiendo interactuar con los registros utilizando objetos Python, facilitando la creación, consulta, actualización y eliminación de registros.

Los modelos se registran en el archivo ***models.py*** de la aplicación. Siguiendo con el ejemplo del Catálogo, vamos a crear el modelo que representa a los productos dentro de la aplicación 'Productos'.

El modelo tendrá los campos: nombre, descripción, precio, stock, fecha de creación, fecha de actualización y disponible.

Los modelos siempre van a heredar de la clase ***Model***. Los modelos en Django heredan de `models.Model` para obtener funcionalidades y propiedades que facilitan la interacción con la base de datos. Esta clase base proporciona métodos y atributos necesarios para definir la estructura de la tabla en la base de datos, gestionar campos y relaciones, y realizar operaciones CRUD de manera eficiente y orientada a objetos. Al heredar de `models.Model`, los modelos adquieren la capacidad de interactuar con la base de datos mediante el sistema ORM (Mapeo Objeto-Relacional) de Django, simplificando la gestión de datos y la sincronización entre el código Python y la estructura de la base de datos.

El modelo Producto se ve del siguiente modo:

```
from django.db import models
```

```
class Producto(models.Model):
```

```
    # Campos para el modelo Producto
```

```
    nombre = models.CharField(max_length=100)
```

```
    descripcion = models.TextField()
```

```
    precio = models.DecimalField(max_digits=8, decimal_places=2)
```

```
    stock = models.PositiveIntegerField()
```

```
    fecha_creacion = models.DateTimeField(auto_now_add=True)
```

```
    fecha_actualizacion = models.DateTimeField(auto_now=True)
```

```
disponible = models.BooleanField(default=True)
```

```
def __str__(self):  
    return self.nombre
```

En este modelo los campos se definen del siguiente modo:

nombre:

Campo de texto corto con longitud máxima de 100 caracteres.

Tipo en Python: str (cadena de texto)

descripcion:

Campo de texto largo para almacenar descripciones extensas.

Tipo en Python: str (cadena de texto)

precio:

Campo numérico decimal para almacenar valores monetarios con 8 dígitos en total y 2 decimales.

Tipo en Python: Decimal (decimal de Python)

stock:

Campo numérico entero positivo para almacenar la cantidad disponible en stock.

Tipo en Python: int (entero)

fecha_creacion:

Campo de fecha y hora que se actualiza automáticamente cuando se crea un nuevo producto.

Tipo en Python: datetime.datetime (objeto de fecha y hora de Python)

fecha_actualizacion:

Campo de fecha y hora que se actualiza automáticamente cada vez que se modifica un producto existente.

Tipo en Python: `datetime.datetime` (objeto de fecha y hora de Python)

disponible:

Campo booleano que indica si el producto está disponible (True) o no (False).

Tipo en Python: `bool` (booleano)

Base De Datos Sqlite

Por defecto Django trabaja con una base de datos Sqlite, que para estos ejemplos, es más que suficiente . Sqlite está configurado en todos los proyectos de django en el archivo ***settings.py*** de la aplicación principal, en el diccionario ***DATABASES*** que de manera predeterminada guarda la siguiente configuración:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    }  
}
```

Con esta configuración se crea una base de datos sqlite en la raíz del proyecto.

Django da soporte para trabajar de forma nativa con bases de datos PostgreSQL, MariaDB, MySQL, Oracle y SQLite.

También se puede trabajar con otras bases de datos instalando librerías de terceros.

Migraciones

Las migraciones son scripts generados automáticamente que reflejan los cambios en la estructura de la base de datos a medida que evoluciona tu modelo de datos. Estos scripts permiten crear, modificar o eliminar tablas y campos en la base de datos de manera coherente y controlada, lo que simplifica la gestión de cambios en la estructura de datos a lo largo del desarrollo de tu aplicación. Las migraciones aseguran que la base de datos esté sincronizada con la definición de modelos en tu código, lo que facilita la colaboración y el mantenimiento del proyecto.

Para crear la migración para el modelo Producto, lo hacemos desde la terminal, ubicado en la raíz del proyecto con el siguiente comando:

```
python manage.py makemigrations
```

Este comando va a generar la siguiente salida:

```
Migrations for 'Productos':
```

```
Productos/migrations/0001_initial.py
```

```
- Create model Producto
```

Para que esta migración impacte en la base de datos usamos el siguiente comando:

```
python manage.py migrate
```

Este comando va a crear las tablas en la base de datos, no solo las tablas de los modelos que definamos, sino que también va a crear las tablas necesarias para el funcionamiento del framework.

La salida de la consola luego de ejecutar el comando se ve del siguiente modo:

```
Operations to perform:
```

```
Apply all migrations: Productos, admin, auth, contenttypes, sessions
```

```
Running migrations:
```

```
Applying Productos.0001_initial... OK
```

```
Applying contenttypes.0001_initial... OK
```

```
Applying auth.0001_initial... OK
```

```
Applying admin.0001_initial... OK
```

```
Applying admin.0002_logentry_remove_auto_add... OK
```

```
Applying admin.0003_logentry_add_action_flag_choices... OK
```

```
Applying contenttypes.0002_remove_content_type_name... OK
```

```
Applying auth.0002_alter_permission_name_max_length... OK
```

```
Applying auth.0003_alter_user_email_max_length... OK
```

```
Applying auth.0004_alter_user_username_opts... OK
```

```
Applying auth.0005_alter_user_last_login_null... OK
```

```
Applying auth.0006_require_contenttypes_0002... OK
```

```
Applying auth.0007_alter_validators_add_error_messages... OK
```

```
Applying auth.0008_alter_user_username_max_length... OK
```

```
Applying auth.0009_alter_user_last_name_max_length... OK
```

```
Applying auth.0010_alter_group_name_max_length... OK
```

```
Applying auth.0011_update_proxy_permissions... OK
```

```
Applying auth.0012_alter_user_first_name_max_length... OK
```

```
Applying sessions.0001_initial... OK
```

En este caso la primera migración que aplica es la del modelo Producto. El resto son las que el framework crea automáticamente.

Una vez realizadas las migraciones, se crean las tablas en la base de datos. En el caso de las tablas de las aplicaciones que creamos Django les pone como nombre **NombreDeLaAplicacion_NombreDeTabla**

Entonces para el modelo Producto, va a crear la tabla **Productos_producto**.

Crear Registros En Base De Datos

Para crear registros del modelo producto, vamos a trabajar en la aplicación Productos.

En nuestra aplicación vamos a tener un formulario que va a tener los campos necesarios para que el usuario complete los datos de un producto y un botón para guardar los registros.

Vamos a crear un **formulario**, un **template** que muestre el formulario, una **vista** que renderice el template y una **ruta** que apunte a la vista.

Formulario (ModelForm)

Para trabajar con formularios hay diferentes opciones. En este caso vamos a usar Formularios basados en Modelos. La clase ModelForm permite crear un formulario partiendo de un modelo. Esta es una clase que luego permite renderizar un formulario en un template.

Para empezar a trabajar con ModelForm vamos a crear en la aplicación Productos un nuevo archivo llamado **forms.py**. Este archivo va a contener los diferentes formularios de la aplicación.

En el archivo importamos ModelForm y los modelos con los que se va a trabajar. En nuestro caso el archivo con el formulario para productos queda del siguiente modo:

```
from django.forms import ModelForm
from .models import Producto

class productoForm(ModelForm):

    class Meta:

        model = Producto

        fields = '__all__'
```

Este es un formulario muy simple, que está asociado al modelo Producto y se indica que muestre campos en el template para todas las propiedades del modelo Producto.

Vista Para Crear Nuevo Registro

En la vista se va a crear una instancia del formulario visto anteriormente y luego se va a enviar al template. Lo que tiene de diferente este template respecto de los anteriores es que se le va a pasar valores desde el backend. Para enviar valores al template lo vamos a hacer con la función **render** que usamos anteriormente, pero pasando un tercer argumento llamado **context** (contexto). El contexto es un diccionario que va a contener los valores que se

quieren enviar al frontend. Dado que el contexto se utiliza para pasar datos desde la vista al template, se puede incluir prácticamente cualquier tipo de datos que se quiera mostrar o manipular en el template.

En la lógica de la vista se va a contemplar la petición por Get y Post. Si el usuario hace la petición por get, se muestra el formulario vacío. Cuando la petición sea por post se envían los valores del formulario al backend para crear el registro en la base de datos.

La vista queda del siguiente modo:

```
#importa el modelo Producto
from .models import Producto

#importa el formulario
from . forms import productoForm

def crearProducto(request):

    #se crea la instancia del formulario
    formulario = productoForm()

    #Se crea el contexto para enviar valores al template
    contexto ={

        'form': formulario,

        'mensaje': 'Crear Producto'

    }

    #Si la peticion es por POST se procesan los valores del formulario
    if request.method == 'POST':

        formularioPOST = productoForm(request.POST)

        #Si los valores del formulario son validos se crea un nuevo registro
        #en la tabla Productos_producto
        if formularioPOST.is_valid():

            formularioPOST.save()

            #luego de guardar se redirecciona a la ruta 'productos'
            return redirect('productos')

    #Si los valores no son validos
```

else:

```
#Se actualizan los valores del contexto
contexto['mensaje']+=' - Error En El Formulario'

contexto['form']= formularioPOST
```

```
#Se vuelve a mostrar el template con los valores
#del formulario para que el usuario pueda ver
#en donde esta el error
```

```
return render(request, 'Productos/formProducto.html', contexto)
```

```
# Si la peticion no es por POST entonces es por GET
# En la peticion por GET simplemente se muestra el
# formulario para que el usuario cargue los valores
# en los campos.
```

else:

```
return render(request, 'Productos/formProducto.html', contexto)
```

Cuando se cree el template debe tener el mismo nombre que se especificó en los argumentos del método **render**, en este caso *"formProducto.html"*

Template Para El Formulario

Para mostrar el formulario, lo vamos a incluir en un template. Este va a ser un template muy simple que solo va a incluir el formulario y un botón para enviar los valores del formulario al backend.

Para hacer referencia en el template a los valores que se reciben por contexto, se utiliza la clave del diccionario, encerrada en llaves dobles.

Por ejemplo:

```
{{claveDiccionario}}
```

El diccionario de la vista **'crearProducto'** tiene dos elementos, entonces para mostrarlos en el template se debe usar la notación `{{form}}` y `{{mensaje}}` como se muestra en el siguiente ejemplo:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>{{mensaje}}</title>

</head>

<body>

    <h2>{{mensaje}}</h2>


    <form method="post">

        {%csrf_token%}

        {{form.as_p}}


        <input type="submit" value="Guardar">

    </form>

</body>

</html>
```

Lo importante en este punto es notar cómo se hace referencia a los valores enviados desde la vista por medio del **contexto** utilizando la sintaxis de dobles llaves `{{key}}` para hacer referencia a las claves del diccionario.

En nuestro ejemplo desde la vista son enviados dos valores, un string para usar a modo de título en el template y un objeto que es una instancia del formulario ***productoForm***.

En el *body* del *html* simplemente usamos un formulario de *html* y le especificamos el envío con el método *post*. Dentro del formulario simplemente incluimos `{%csrf_token%}`, `{{form}}` y un botón de tipo *submit*.

La sintaxis `{% csrf_token %}` se utiliza para agregar un token de seguridad a los formularios en el *template* para prevenir [ataques CSRF](#) y garantizar la autenticidad de las solicitudes *POST*.

Cuando se hace referencia al formulario que se envía desde la vista, en este caso con la variable `{{form}}`, lo que se obtiene como resultado es un formulario que muestra todos los campos necesarios para que el usuario complete los valores correspondientes al modelo con el que el formulario trabaja, en este caso al modelo *Producto*.

En el ejemplo en vez de usar `{{form}}` se usa `{{form.as_p}}` para que los campos del formulario se muestren uno debajo del otro.

Ruta Para La Función De Crear Producto

Por último se crea una ruta que apunte a la vista ***crearProducto***. Para esto se agrega una nueva ruta al archivo *Productos/urls.py*

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

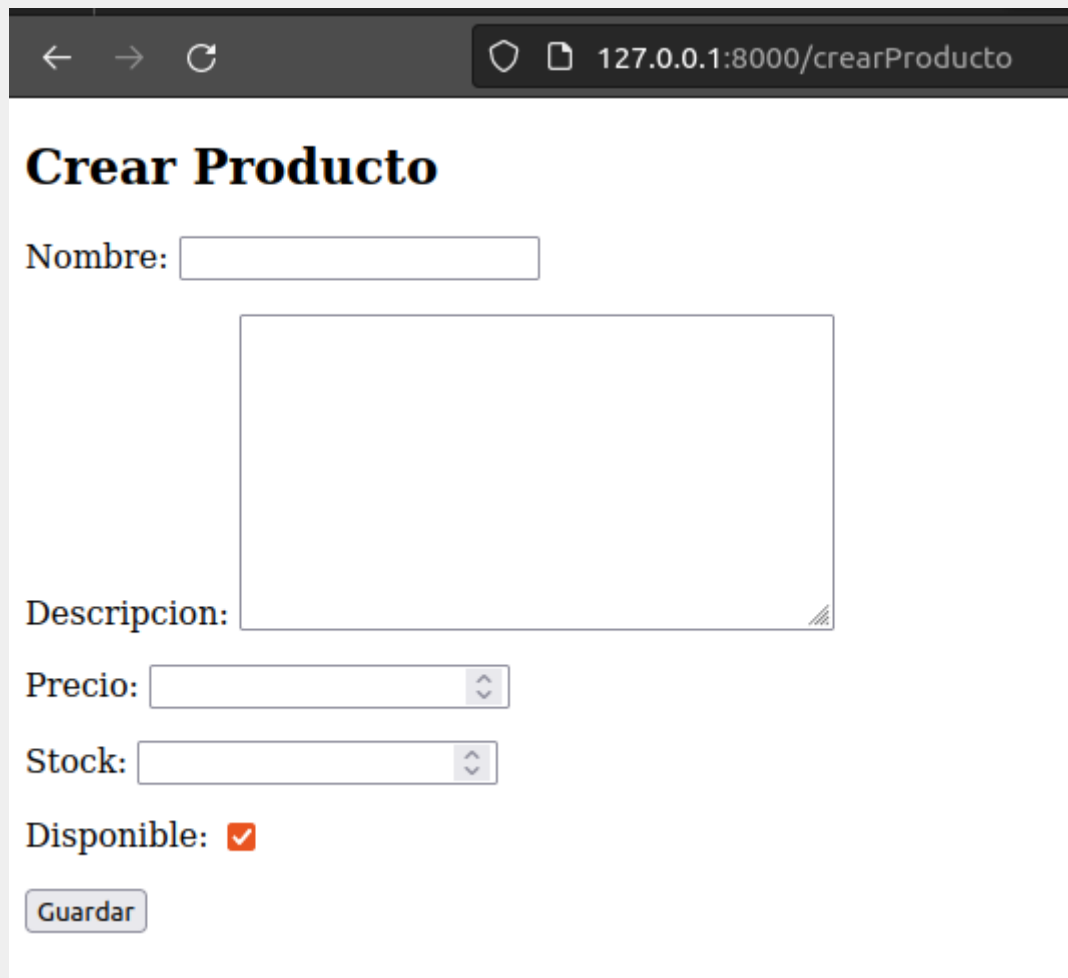
urlpatterns = [

    path('productos', views.productos, name='productos'),
    path('crearProducto', views.crearProducto, name='crearProducto')
]
```

En este punto la aplicación ya tiene lo necesario para poder crear registros del modelo *Producto*.

Para poder probarlo hay que ejecutar el servidor de Django y en el navegador ir a la ruta `http://127.0.0.1:8000/crearProducto`

El resultado se ve como en la siguiente imagen



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/crearProducto'. The page title is 'Crear Producto'. The form contains the following elements:

- Nombre:** A text input field.
- Descripcion:** A large text area.
- Precio:** A numeric input field with a spinner.
- Stock:** A numeric input field with a spinner.
- Disponible:** A checkbox that is currently checked.
- Guardar:** A button to submit the form.

En este punto la aplicación tiene la capacidad de guardar registros del modelo producto en la base de datos.

Listar Registros

Este es un ejemplo simple que muestra como obtener registros de un modelo y enviarlos a un template para poder mostrarlos, en este caso, en forma de lista.

Al igual que en los ejemplos anteriores, para cumplir con el objetivo, se va a crear una **ruta**, una **vista** y un **template**.

Vista Para Listar Productos

Dentro del archivo `Productos/views.py` creamos una nueva vista. Esta es una vista simple, que obtiene los registros desde la base de datos que corresponden al modelo `Producto`, y los envía al template por medio del contexto. Para poder trabajar con el modelo, primero hay que importarlo.

```
from .models import Producto
```

El código se ve del siguiente modo:

```
def listarProductos(request):

    #Consulta a la base de datos por medio del
    #ORM de django que obtiene todos los registros
    #del modelo Producto
    productos = Producto.objects.all()

    context = {
        'titulo': 'Lista De Productos',
        'productos': productos
    }

    return render(request, 'Producto/listaProductos.html', context)
```

En esta vista se puede ver como se usa el [ORM de Django](#) para obtener los registros desde la base de datos en la línea **productos = Producto.objects.all()**. En los próximos ejemplos se van a ver otros tipos de consultas que permite hacer el ORM, aunque es importante leer la [Documentación del Framework](#) ya que es una herramienta muy amplia.

La variable **productos** va a contener el resultado de la consulta a la base de datos. Esta variable va a ser de tipo **queryset**.

El tipo de dato queryset es específico de Django, no es propio de Python en sí. Django ha desarrollado su propio sistema de consultas (ORM) que utiliza los querysets para interactuar con la base de datos de manera orientada a objetos. Los querysets no son listas comunes de Python, sino que están diseñados para optimizar las consultas a la base de datos y proporcionar una interfaz intuitiva para trabajar con los datos del modelo.

El queryset contiene una colección de objetos, en este caso de tipo **Producto**. Es decir que lo que se está enviando al template son objetos de tipo Producto.

Template Para Listar Productos

Vamos a crear en la carpeta de templates (Productos/templates/Productos) un nuevo archivo llamado **listaProductos.html**.

El template para la lista de productos va a contener una lista y dentro de esta se va a usar notación propia del [lenguaje de template de Django](#). En este caso se usa un bucle for, que va a permitir iterar el queryset **productos** y por cada elemento del queryset se va a renderizar un , un elemento de la lista, el cual va a contener datos del objeto correspondiente a cada iteración.

Como resultado se obtienen tantos como objetos contenga el queryset.

Además, teniendo en cuenta que estamos iterando sobre objetos, vamos a mostrar información de cada uno de los atributos del objeto dentro de cada ``.

El código se ve del siguiente modo:

```
<h2>{{titulo}}</h2>

<ul>

    {% for producto in productos %}

    <li>

        <p>

            Producto ID: {{producto.id}}<br>

            Nombre: {{producto.nombre}}<br>

            Descripcion: {{producto.descripcion}}<br>

            Precio: {{producto.precio}}<br>

            Stock: {{producto.stock}}<br>

            Disponible: {{producto.disponible}}<br>

            Fecha De Creacion: {{producto.fecha_creacion}}<br>

            Fecha De Actualizacion: {{producto.fecha_actualizacion}}<br>

        </p>

    </li>

    <hr>

    <br>

    {% endfor %}

</ul>
```




Url Para Listar Productos

Por último queda crear la url que permita acceder a la vista que renderiza el template en el que se listan los productos.

Para esto se debe agregar en `Productos/urls.py` una nueva ruta. En la lista **`urlpatterns`** agregamos un nuevo path con la ruta **`listarProductos`** apuntando a la vista **`listarProductos`** creada anteriormente.

```
path('listarProductos', views.listarProductos, name='listarProductos')
```

Para probar la nueva funcionalidad, se ejecuta el servidor local y en el navegador vamos a la ruta ***http://127.0.0.1:8000/listarProductos***. El resultado se ve como en la siguiente imagen:

127.0.0.1:8000/listarProductos

Lista De Productos

- Producto ID: 2
Nombre: Segundo Producto
Descripcion: Texto De Descripcion
Precio: 5.20
Stock: 185
Disponible: True
Fecha De Creacion: May 1, 2018, 7:25 p.m.
Fecha De Actualizacion: Aug. 23, 2023, 7:25 p.m.

- Producto ID: 3
Nombre: Tercer Producto
Descripcion: Descripcion tercer producto
Precio: 18.30
Stock: 300
Disponible: True
Fecha De Creacion: May 1, 2018, 7:26 p.m.
Fecha De Actualizacion: Aug. 23, 2023, 7:26 p.m.

- Producto ID: 4
Nombre: Cuarto Producto
Descripcion: Descripcion tercer producto
Precio: 15.25
Stock: 400
Disponible: True
Fecha De Creacion: May 1, 2018, 7:27 p.m.
Fecha De Actualizacion: Aug. 23, 2023, 7:27 p.m.

- Producto ID: 5
Nombre: Quinto Producto
Descripcion: Descripcion quinto producto

Editar Registros

En este punto vamos a reutilizar alguno de los elementos que se usaron para crear registros de Productos.

Lo que se va a reutilizar es el formulario (*productoForm*) y el template que renderiza el formulario (*formProducto.html*), ya que son dos elementos necesarios para poder mostrarle al usuario los valores actuales de un determinado objeto que se quiere editar y los controles necesarios para poder hacerlo.

Vista Para Editar Productos

Se va a crear la función **editarProducto** que como parámetro, además de *request* va a recibir el **id** de un producto. El parametro id que recibe la funcion va a estar definido en la ruta. Este id se va a usar para recuperar un registro de la base de datos y cargar sus valores en los campos del formulario. Luego se envía el formulario al template con los valores del objeto para que el usuario lo edite.

Por último se guardan los datos actualizados en la base de datos.

También se van a usar los métodos http get y post en la lógica de la vista para mostrar el formulario y para guardar los valores actualizados.

El código de la vista se ve del siguiente modo:

```
def editarProducto(request, id):
```

```
    #se obtiene el registro en base a su id
```

```
    productoEditar = Producto.objects.get(pk=id)
```

```
    if request.method == 'GET':
```

```
        #Se crea una instancia del formulario pero pasandole como parametro
```

```
        #una instancia del modelo Producto que corresponde al obtenido en
```

```
        #la consulta a la base de datos.
```

```
        formEditar = productoForm(instance=productoEditar)
```

```
        contextoGet = {
```

```
            'form': formEditar,
```

```
            'mensaje': 'Editar Producto'
```

```
        }
```

```
        return render(request, 'Productos/formProducto.html', contextoGet)
```

```
    else:
```

```
        #Si la peticion es por POST
```

```

formGuardar = productoForm(request.POST, instance=productoEditar)

if formGuardar.is_valid():
    formGuardar.save()
    return redirect('listarProductos')

else:
    return render(request, 'Productos/formProducto.html',
        {'form':formGuardar, 'mensaje':'Error - Editar Producto'})

```

Ruta Para Editar Productos

Por último solo queda definir la ruta para acceder a la vista en el archivo ***Productos/urls.py***.

Esta ruta va a tomar como parámetro un valor entero que va a ser el id del registro que se quiera editar.

La definición de la ruta se ve del siguiente modo:

```

from django.urls import path

from . import views

urlpatterns = [

    path('productos', views.productos, name='productos'),

    path('crearProducto', views.crearProducto, name='crearProducto'),

    path('listarProductos', views.listarProductos, name='listarProductos'),

    path('editarProducto/<int:id>', views.editarProducto, name='editarProducto')

]

```

Si bien se podría escribir en el navegador la ruta para editar un producto, por ejemplo, ***localhost:8000/editarProducto/1***, se podría agregar al listado de clientes un enlace para cada producto, de manera dinámica, para que cada producto en el listado tenga su ruta para acceder a la función de editar.

Para esto editamos el template ***Productos/listaProductos.html*** y agregamos al un enlace con notación del lenguaje de template de Django. El template modificado queda del siguiente modo:

```
<h2>{{titulo}}</h2>
```

```
<ul>
```

```
    {% for producto in productos %}
```

```

<li>

    <p>

        Producto ID: {{producto.id}}<br>

        Nombre: {{producto.nombre}}<br>

        Descripcion: {{producto.descripcion}}<br>

        Precio: {{producto.precio}}<br>

        Stock: {{producto.stock}}<br>

        Disponible: {{producto.disponible}}<br>

        Fecha De Creacion: {{producto.fecha_creacion}}<br>

        Fecha De Actualizacion: {{producto.fecha_actualizacion}}<br>

        <a href=" {% url 'editarProducto' producto.id %} ">

            <button>Editar</button>

        </a>

    </p>

</li>

<hr>

<br>

{% endfor %}

</ul>

```

Ahora cada elemento de la lista tiene su propio botón que apunta a la ruta para editar con su respectivo id.

Borrar Registros

Vista Para Borrar Productos

La vista para borrar un registro, en principio es muy simple, solo toma un registro en base a su id y luego lo borra de la base de datos. Al igual que la vista para editar, esta también recibe como parámetro el id del producto desde la url. El código se ve del siguiente modo:

```
def borrarProducto(request, id):

    productoBorrar = Producto.objects.get(pk=id)

    productoBorrar.delete()

    return redirect('listarProductos')
```

Url Para Borrar Productos

Esta url es igual a la usada para editar productos. Toma un parametro de tipo entero para pasarlo a la vista y apunta a la vista ***borrarProducto***.

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

urlpatterns = [

    path('productos', views.productos, name='productos'),

    path('crearProducto', views.crearProducto, name='crearProducto'),

    path('listarProductos', views.listarProductos, name='listarProductos'),

    path('editarProducto/<int:id>', views.editarProducto, name='editarProducto'),

    path('borrarProducto/<int:id>', views.borrarProducto, name='borrarProducto')

]
```

Del mismo modo que incluimos un botón para editar en el listado de productos, podemos incluir uno para borrar un producto.

El template editado se ve del siguiente modo:

```
<h2>{{titulo}}</h2>
```

```
<ul>
```

```
    {% for producto in productos %}
```

```
    <li>
```

```
        <p>
```

```
            Producto ID: {{producto.id}}<br>
```

```
            Nombre: {{producto.nombre}}<br>
```

```
            Descripcion: {{producto.descripcion}}<br>
```

```

    Precio: {{producto.precio}}<br>
    Stock: {{producto.stock}}<br>
    Disponible: {{producto.disponible}}<br>
    Fecha De Creacion: {{producto.fecha_creacion}}<br>
    Fecha De Actualizacion: {{producto.fecha_actualizacion}}<br>
    <a href=" {% url 'editarProducto' producto.id %} ">
        <button>Editar</button>
    </a>

    <a href=" {% url 'borrarProducto' producto.id %} ">
        <button>borrar</button>
    </a>
</p>
</li>
<hr>
<br>
{% endfor %}
</ul>

```

Resumen Crud

Hasta este punto se vieron los conceptos mínimos e indispensables para poder llevar a cabo la tarea de crear un CRUD de un modelo simple con Django.

Hay que tener en cuenta que en los ejemplos vistos se omitieron conceptos importantes, como bloques try except para hacer las consultas a la base de datos o controles para confirmar acciones por parte del usuario, entre otros, ya que la idea es que el código sea lo más resumido posible.

Relaciones Entre Modelos - 1 a 1, 1 a N, N a N

Vamos a desarrollar ejemplos simples que sirvan para poder ver los conceptos de relaciones entre modelos. Para esto se agregan los modelos Ubicacion, Almacen y Pedido.

Relación 1 a 1:

Para el ejemplo de relación 1 a 1 vamos a desarrollar el modelo "Ubicacion" que va a representar la ubicación de un producto en un almacén o depósito. Cada almacén está dividido en secciones, cada sección tiene varias estanterías y las estanterías tienen varios niveles.

De este modo es posible asignarle a un producto una ubicación específica en un almacén al especificar su sección, estantería y nivel. Cada producto tendrá una ubicación y cada ubicación le pertenece a un único producto.

Agregando el modelo "Ubicacion" en models.py (Catalogo/Productos/models.py) el módulo se ve del siguiente modo:

```
from django.db import models

class Ubicacion(models.Model):
    seccion = models.CharField(max_length=10)
    estante = models.CharField(max_length=10)
    nivel = models.IntegerField()

    def __str__(self) -> str:
        return f"""
        Seccion: {self.seccion}\n
        Estante: {self.estante}\n
        Nivel: {self.nivel}"""

class Producto(models.Model):
    # Campos para el modelo Producto
    nombre = models.CharField(max_length=100)
```

```

descripcion = models.TextField()

precio = models.DecimalField(max_digits=8, decimal_places=2)

stock = models.PositiveIntegerField()

fecha_creacion = models.DateTimeField(auto_now_add=True)

fecha_actualizacion = models.DateTimeField(auto_now=True)

disponible = models.BooleanField(default=True)

ubicacion = models.OneToOneField(Ubicacion, on_delete=models.SET_NULL,
blank=True, null=True)

def __str__(self):
    return self.nombre

```

Luego de crear las migraciones y de aplicarlas a la base de datos se pueden empezar a hacer pruebas para ver el funcionamiento de la relación entre los modelos.

Para estos ejemplos vamos a usar la herramienta de shell que facilita Django. Los ejemplos los desarrollamos en terminal ya que de este modo nos ahorramos la estructura de rutas, vistas, templates, etc de los nuevos modelos y se enfoca el ejemplo en la creación de las relaciones.

En la terminal ejecutamos ***python manage.py shell*** para ejecutar el shell de Django. Luego procedemos a crear nuestros objetos y relaciones.

Lo primero que debemos tener en cuenta es que solo se van a poder establecer relaciones entre registros existentes. Esto quiere decir que primero se deben crear los objetos, guardar los registros en y luego establecer las relaciones.

Creamos un registro de Ubicación:

```

>>> # Importamos los modelos con los que vamos a trabajar
>>> from Productos.models import Producto, Ubicacion
>>>
>>> #creamos una instancia de Ubicacion
>>> u1 = Ubicacion(seccion="CCH", estante="CJA", nivel=5)
>>>
>>> #guardamos en base de datos
>>> u1.save()
>>>
>>> #vemos el objeto guardado
>>> u1
<Ubicacion: Seccion: CCH

```

Estante: CJA

Nivel: 5>

>>>

Ahora se puede asignar esta ubicación a un producto. Para esto recuperamos un producto de los guardados en los ejemplos anteriores y le asignamos la ubicación creada:

```
>>> #Obtenemos un producto desde la base de datos
```

```
>>> prod = Producto.objects.get(pk=2)
```

```
>>>
```

```
>>> #vemos el valor actual del atributo "ubicacion"
```

```
>>> print(prod.ubicacion)
```

```
None
```

```
>>>
```

```
>>> #establecemos la relacion entre el producto y la ubicacion
```

```
>>> prod.ubicacion = u1
```

```
>>>
```

```
>>>#Volvemos a ver el valor del atributo "ubicacion" del producto
```

```
>>> print(prod.ubicacion)
```

Seccion: CCH

Estante: CJA

Nivel: 5

```
>>>
```

```
>>> #guardamos los cambios en el producto
```

```
>>> prod.save()
```

```
>>>
```

Podemos crear un nuevo producto y relacionarlo con una ubicación existente en un solo paso.

```
>>> #Creamos una nueva ubicacion
```

```
>>> u2 = Ubicacion(seccion="ACH", estante="BJA", nivel=3)
```

```
>>>
```

```
>>> # Guardamos el registro
```

```
>>> u2.save()
```

```
>>>
```

```
>>> #Creamos un nuevo producto
```

```
>>> prodU2 = Producto(nombre="1 a 1", descripcion="Relaciona ubicacion al  
crear el registro", precio=10.99, stock=36, ubicacion=u2)
```

```
>>>
```

```
>>> #guardamos los cambios
```

```
>>> prodU2.save()
```

```
>>>
```


También es posible ver con qué producto se relaciona una determinada ubicación.

```
>>> u1.producto.descripcion
'Texto De Descripcion'
>>>
>>>
>>> u2.producto.descripcion
'Relaciona ubicacion al crear el registro'
>>>
```

Relación 1 a N:

Para el ejemplo vamos a desarrollar la relación entre Ubicación y Almacén. En este ejemplo un Almacén puede tener varias ubicaciones y una ubicación puede pertenecer a un Almacén. Es decir que se va a dar la relación 1 a N.

Primero creamos el modelo Almacen en (Catalogo/Productos/models.py). El modelo es muy simple, se ve del siguiente modo:

```
class Almacen(models.Model):
    nombre = models.CharField(max_length=30)

    def __str__(self) -> str:
        return "Almacen: "+self.nombre
```

Claro que este modelo debería tener parámetros que definan de mejor modo a un almacén, pero la idea es concentrarnos en las relaciones.

Editamos el modelo Ubicación para establecer la relación

```
class Ubicacion(models.Model):
    almacen = models.ForeignKey(Almacen, on_delete=models.CASCADE, null=True)
    seccion = models.CharField(max_length=10)
    estante = models.CharField(max_length=10)
    nivel = models.IntegerField()

    def __str__(self) -> str:
        return f"""Almacen: {self.almacen}\n
\tSeccion: {self.seccion}\n
\tEstante: {self.estante}\n
\tNivel: {self.nivel}\n"""
```

ForeignKey es la clase que se utiliza para establecer la relación 1 a N. En este caso 1 Almacen se puede relacionar con N Ubicaciones.

Hay que tener en cuenta, que en este punto hay registros de Ubicación existentes en la base de datos. Entonces se establece *null=True* en la relación, para evitar errores al crear las migraciones a la base de datos.

El modelo Ubicación no debería permitir valores nulos en el campo *almacen*, así que hay que resolverlo. Lo vamos a resolver al mismo tiempo que veamos los ejemplos en donde se establecen las relaciones.

Luego de crear las migraciones usamos el shell para crear al menos un registro de Almacén y asignamos ese almacén a las ubicaciones existentes.

```
>>> # Importo los modelos
>>> from Productos.models import Almacen, Ubicacion, Producto
>>>
>>> # Obtengo todas las ubicaciones existentes
>>> ubicaciones = Ubicacion.objects.all()
>>>
>>> # Creo un registro de Almacen
>>> a1 = Almacen(nombre="Primer Almacen")
>>>
>>> # Guardo el registro
>>> a1.save()
>>>
>>> # Asigno el almacen creado a las ubicaciones existentes
>>>
>>> for u in ubicaciones:
...     u.almacen=a1 #se establece la relación
...     u.save()
...     print(u)
...     print("\n*****")
...
Almacen: Almacen: Primer Almacen
          Seccion: CCH
          Estante: CJA
          Nivel: 5
*****

Almacen: Almacen: Primer Almacen
          Seccion: ACH
          Estante: BJA
          Nivel: 3
*****

>>>
```

Si bien el modelo Almacen no cuenta con un campo "ubicacion" es posible ver las ubicaciones relacionadas con un determinado Almacen.

```
>>># Obtengo todas las ubicaciones relacionadas con el almacen "a1"
>>> ubicacionesAlmacen1 = a1.ubicacion_set.all()
>>>
>>>
>>> # ubicacionesAlmacen1 es un iterable de tipo QuerySet que contiene
>>> # objetos del tipo Ubicacion
>>> type(ubicacionesAlmacen1)
<class 'django.db.models.query.QuerySet'>
>>>
>>>
>>> for ua1 in ubicacionesAlmacen1:
...     print(f"Ubicación id: {ua1.id}")
...
Ubicación id: 2
Ubicación id: 3
>>>
```

Ahora todas las ubicaciones existentes tiene asignadas un Almacen. Entonces es posible editar el modelo Ubicacion para dejar de permitir valores nulos en el campo "almacen"

```
almacen = models.ForeignKey(Almacen, on_delete=models.CASCADE)
```

Ahora al crear la migración para registrar este cambio en el modelo, con *python manage.py makemigrations*, el framework muestra el siguiente mensaje:

```
It is impossible to change a nullable field 'almacen' on ubicacion to non-nullable without providing a default.
This is because the database needs something to populate existing rows.
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Ignore for now. Existing rows that contain NULL values will have to be handled manually, for example with a
RunPython or RunSQL operation.
 3) Quit and manually define a default value in models.py.
Select an option: 2
Migrations for 'Productos':
  Productos/migrations/0004_alter_ubicacion_almacen.py
    - Alter field almacen on ubicacion
```

Como ya no hay ubicaciones con valor null en el campo "almacen" en la base de datos, indico la opción dos.

Luego se migra con *python manage.py migrate*

```
Operations to perform:
  Apply all migrations: Productos, admin, auth, contenttypes, sessions
Running migrations:
  Applying Productos.0004_alter_ubicacion_almacen... OK
```

Ahora Ubicacion ya no permite valores nulos en el campo "almacen"

Relación N a N:

Para el ejemplo vamos a desarrollar la relación entre Producto y Pedido.

En este ejemplo un Pedido puede *tener* N productos y un Producto puede estar en N pedidos.

Primero modelamos un Pedido.

```
class Pedido(models.Model):  
    fecha = models.DateField(auto_now_add=True)  
    productos = models.ManyToManyField(Producto)
```

Claro que un pedido debería tener muchas mas información, pero la idea es simplemente desarrollar el ejemplo de la relación N a N.

Luego de crear las migraciones y migrar, en la base de datos se van a crear dos tablas. Una corresponde al modelo, en este caso el modelo Pedido, que en base de datos va a tener la tabla "Productos_pedido" (recordar que todas las tablas que creemos van a llevar como nombre "NombreAplicacion_nombreModelo").

La segunda tabla que se crea es la tabla intermedia que registra las relaciones N a N. En este caso lleva el nombre "Productos_pedido_producto".

Pruebas en shell

```
>>> from Productos.models import Producto, Pedido  
>>>  
>>> # Recuperamos algunos registros de Productos  
>>> p2 = Producto.objects.get(pk=2)  
>>> p3 = Producto.objects.get(pk=3)  
>>> p4 = Producto.objects.get(pk=4)  
>>> p5 = Producto.objects.get(pk=5)  
>>> p11 = Producto.objects.get(pk=11)  
  
>>> # Creamos un pedido  
>>> pedido1 = Pedido()  
>>> pedido1.save()  
>>>  
>>> # Ahora se puede establecer relaciones  
>>> # entre el pedido y los Productos  
>>>  
>>> pedido1.productos.add(p2, p3, p4, p5, p11)  
>>> pedido1.save()  
>>>
```

Al igual que vimos con la relación 1 a N, el campo "productos" del modelo Pedido también es una colección y al crear una consulta se obtiene un QuerySet que, en este caso, contiene objetos de tipo Producto.

```
>>>
>>> pedido1.productos.all()

<QuerySet [<Producto: Segundo Producto>, <Producto: Tercer Producto>,
<Producto: Cuarto Producto>, <Producto: Quinto Producto>, <Producto: 1 a 1>]>

>>>
>>>
```

Vamos a crear dos pedidos más para otros ejemplos

```
>>>
>>> pedido2 = Pedido()
>>> pedido2.save()
>>>
>>> pedido2.productos.add(p3, p5)
>>>
>>> pedido2.productos.all()
<QuerySet [<Producto: Tercer Producto>, <Producto: Quinto Producto>]>
>>>
```

También se puede crear un Producto y establecer la relación con un Pedido existente en un solo paso.

```
>>># Primero se crea el Pedido
>>>
>>> pedido3 = Pedido()
>>> pedido3.save()
>>>
>>># Luego se crea el Producto y se relaciona con el pedido
>>> productoNuevo = pedido3.productos.create(nombre="Relacion N a N",
descripcion="Relacion al Crear el Objeto", precio=10.2, stock=36)
>>>
>>>
```

El producto ya está creado, relacionado con el pedido y guardado en la base de datos, no es necesario usar *productoNuevo.save()*.

```
>>>
>>> # Podemos seguir agregando productos al pedido
>>> pedido3.productos.add(p11)
>>>
>>> # Listamos sus productos
>>> pedido3.productos.all()
<QuerySet [<Producto: 1 a 1>, <Producto: Relacion N a N>]>
```

Si bien en el modelo Producto no hay un atributo asociado a Pedido, es posible hacer una consulta a la base de datos para ver con que Pedidos está relacionado un Producto.

```
>>>
>>> # Producto ID 2. Vemos en qué pedidos está presente
>>> p2.pedido_set.all()
<QuerySet [<Pedido: Pedido object (1)>]>
>>>
>>>
>>> # Producto ID 3. Vemos en qué pedidos está presente
>>> p3.pedido_set.all()
<QuerySet [<Pedido: Pedido object (1)>, <Pedido: Pedido object (2)>]>
>>>
>>>
>>> # Producto ID 11. Vemos en qué pedidos está presente
>>> p11.pedido_set.all()
<QuerySet [<Pedido: Pedido object (1)>, <Pedido: Pedido object (3)>]>
>>>
```