



Relaciones Uno a Muchos en Django

¡Bienvenidos al fascinante mundo de las relaciones entre modelos en Django! Hoy exploraremos cómo conectar datos del mundo real de manera profesional y eficiente.

Aprenderemos cómo un cliente puede tener muchas órdenes de compra en nuestra tienda de bicicletas. Descubriremos no solo cómo funcionan estas relaciones, sino también cuándo utilizarlas y cómo implementarlas paso a paso en un proyecto real.

🎯 **Objetivo de hoy:** Comprender, crear e implementar relaciones Uno a Muchos en Django usando el proyecto *bikeshop* con base de datos MySQL, aplicando las mejores prácticas de desarrollo.

¿Qué aprendimos antes?



En nuestra clase anterior exploramos las relaciones Uno a Uno, donde cada cliente tenía exactamente un perfil asociado. Hoy damos un paso adelante y subimos un nivel de complejidad.

Veremos cómo un mismo cliente puede tener múltiples órdenes de compra, reflejando situaciones del mundo real donde las relaciones no son siempre 1:1.



Autor → Libros

Un autor puede escribir múltiples libros a lo largo de su carrera



Profesor → Estudiantes

Un profesor puede enseñar a muchos estudiantes en diferentes cursos



País → Ciudades

Un país contiene múltiples ciudades dentro de su territorio

 **Actividad rápida (2 minutos):** Piensa y comparte con un compañero tres ejemplos cotidianos donde observes relaciones uno a muchos en tu vida diaria.

¿Qué es una relación Uno a Muchos?

Una relación Uno a Muchos (1:N) es un patrón fundamental en bases de datos relacionales que representa cómo un registro "padre" puede estar asociado con varios registros "hijos", pero cada hijo pertenece a un único padre.



Un Cliente

Entidad principal que actúa como registro "padre"



Múltiples Órdenes

Entidades dependientes que actúan como registros "hijos"

En bases de datos relacionales, esta relación se implementa mediante una clave foránea (ForeignKey) en la tabla hija que apunta al identificador único del registro padre. Cada orden almacena la referencia al cliente que la realizó.

- ❏ **Concepto clave:** La tabla "hija" (Orden) contiene una columna especial que almacena el ID del registro "padre" (Cliente), creando así el vínculo entre ambas entidades.

Cuándo y por qué usar una relación Uno a Muchos



Utiliza una relación 1:N cuando necesites modelar situaciones donde una entidad principal puede tener múltiples dependientes, pero cada dependiente pertenece exclusivamente a un único principal.

1

Cliente → Órdenes

Un cliente realiza múltiples compras a lo largo del tiempo, cada una registrada como una orden independiente



Evita duplicación

No necesitas repetir los datos del cliente en cada orden

2

Autor → Libros

Un autor escribe varios libros durante su carrera, pero cada libro tiene un autor principal



Mejora organización

Mantiene la información estructurada y fácil de consultar

3

Categoría → Productos

Una categoría agrupa múltiples productos relacionados, organizando el catálogo eficientemente



Optimiza espacio

Reduce el tamaño de la base de datos significativamente

Caso práctico: Proyecto Bikeshop

En nuestro proyecto bikeshop, necesitamos implementar un sistema que permita a los clientes realizar múltiples compras de bicicletas y accesorios. Cada compra se registra como una orden independiente con sus propios detalles.

El desafío

Queremos poder registrar y gestionar todas las órdenes de compra de cada cliente de manera organizada, manteniendo un historial completo de transacciones.

Cada cliente puede tener muchas órdenes a lo largo del tiempo, pero cada orden pertenece únicamente a un cliente específico.

La solución

Esto se traduce directamente a una relación Uno a Muchos en Django, donde utilizaremos ForeignKey para vincular las órdenes con sus respectivos clientes.

De esta forma, podemos consultar fácilmente todas las compras de un cliente o identificar qué cliente realizó una orden específica.

Paso 1: Crear la app de órdenes

El primer paso en nuestra implementación es crear una nueva aplicación Django dedicada exclusivamente a gestionar las órdenes de compra. Esto nos permite mantener nuestro código organizado y modular.

01

Ejecutar comando de creación

En la terminal, dentro del directorio de tu proyecto, ejecuta el comando para crear la nueva app

02

Registrar en settings.py

Agrega la app 'ordenes' a la lista `INSTALLED_APPS` para que Django la reconozca

03



Verificar estructura

Confirma que se crearon correctamente los archivos `models.py`, `views.py` y otros archivos base

```
python manage.py startapp ordenes
```

Ahora agrega la app a tu configuración en `settings.py`:

```
INSTALLED_APPS = [  
    ...,  
    'clientes',  
    'ordenes',  
]
```

  **Actividad práctica:** Crea la app en tu proyecto siguiendo estos pasos y verifica que aparezca correctamente en la estructura de carpetas.

Paso 2: Crear el modelo Orden

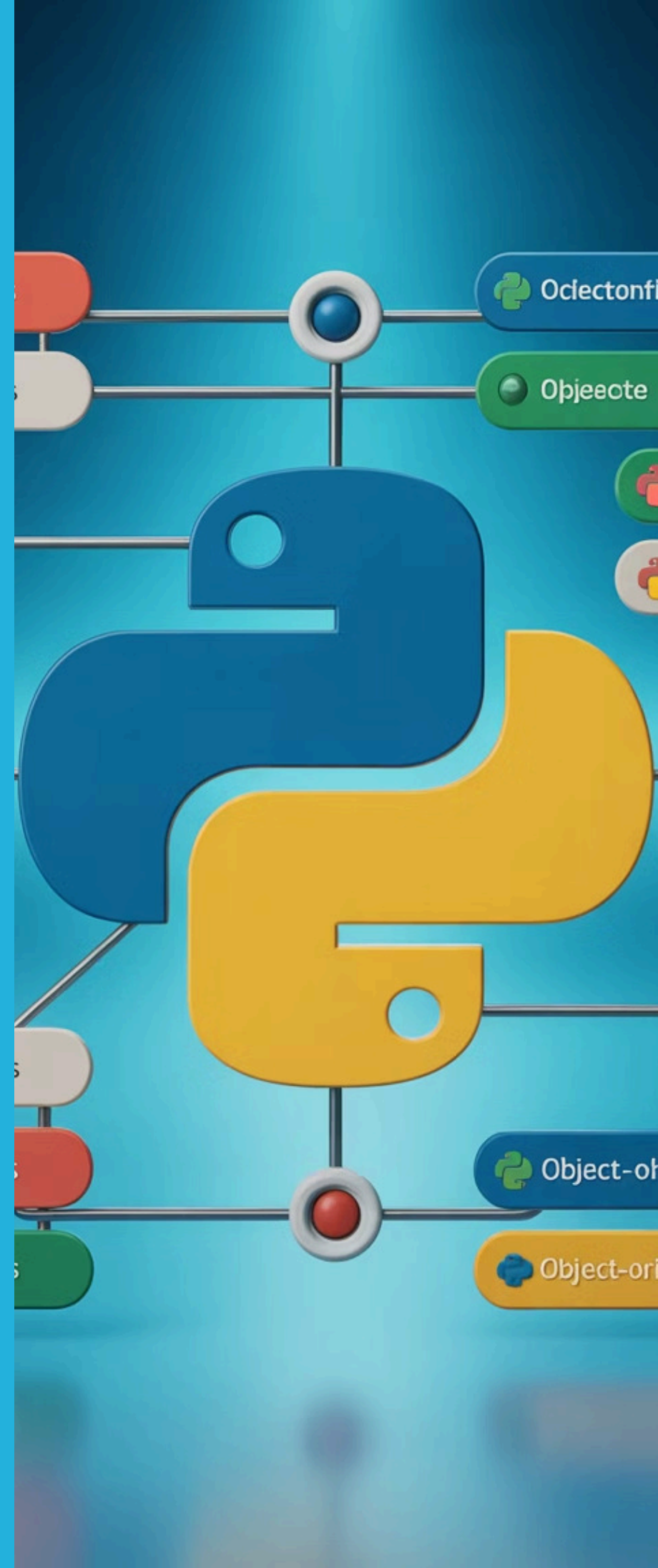
Ahora definiremos el modelo Orden en el archivo `ordenes/models.py`. Este modelo representa cada compra realizada por un cliente y contiene toda la información relevante de la transacción.

```
from django.db import models
from clientes.models import Cliente

class Orden(models.Model):
    cliente = models.ForeignKey(
        Cliente,
        on_delete=models.CASCADE,
        related_name='ordenes'
    )
    fecha = models.DateTimeField(auto_now_add=True)
    total = models.DecimalField(
        max_digits=10,
        decimal_places=2
    )
    estado = models.CharField(
        max_length=20,
        choices=[
            ('pendiente', 'Pendiente'),
            ('pagada', 'Pagada'),
            ('cancelada', 'Cancelada')
        ],
        default='pendiente'
    )

    def __str__(self):
        return f"Orden {self.id} - Cliente: {self.cliente.nombre}"
```

Este modelo incluye todos los campos necesarios para gestionar una orden: la referencia al cliente, la fecha de creación automática, el monto total y el estado actual del pedido.



Entendiendo cada parte del modelo

Analicemos en detalle cada componente del modelo Orden para comprender completamente su funcionamiento y las decisiones de diseño que tomamos.

1

ForeignKey
Establece la relación Uno a Muchos con el modelo Cliente. Es el campo que conecta ambas tablas en la base de datos mediante una clave foránea.

2

on_delete=CASCADE
Define el comportamiento cuando se elimina un cliente: todas sus órdenes también se eliminarán automáticamente, manteniendo la integridad referencial.


3

related_name='ordenes'
Permite acceder a todas las órdenes de un cliente desde el objeto cliente usando la sintaxis intuitiva: `cliente.ordenes.all()`

4

auto_now_add=True
Registra automáticamente la fecha y hora exacta cuando se crea una nueva orden, sin necesidad de especificarla manualmente.

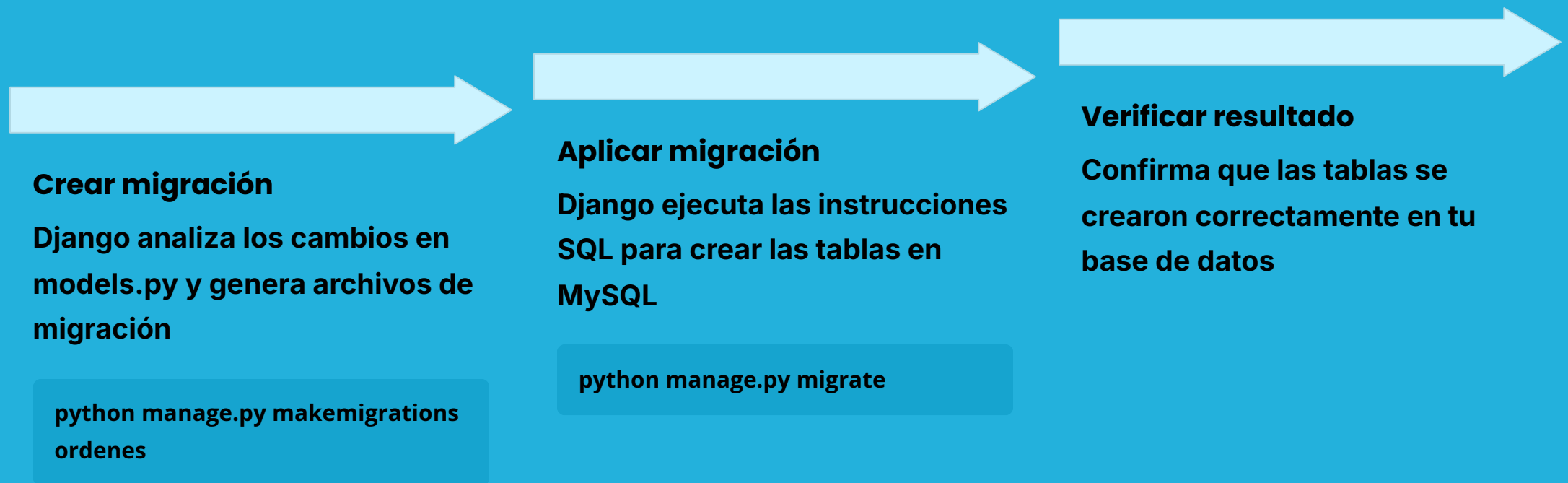


 **Pregunta para reflexionar:** ¿Qué pasaría si no definimos el parámetro `related_name` en nuestro `ForeignKey`?

Respuesta: Django crearía un nombre automático usando el patrón `orden_set`, que es menos intuitivo y descriptivo.

Paso 3: Crear y aplicar migraciones

Las migraciones son la manera en que Django traduce nuestros modelos Python a estructuras de base de datos reales. Este proceso es fundamental para mantener sincronizada nuestra aplicación con la base de datos.



📌 ⚠ Solución de problemas comunes:

- Asegúrate de usar el motor de almacenamiento InnoDB en MySQL, ya que MyISAM no soporta claves foráneas
- Verifica que el usuario de MySQL tenga los permisos necesarios para crear tablas y relaciones
- Confirma que la app 'clientes' esté migrada antes de migrar 'ordenes'

👤 **Actividad:** Ejecuta las migraciones en tu proyecto y verifica en MySQL Workbench o phpMyAdmin que las tablas se crearon con las claves foráneas correctas.

Paso 4: Registrar en el admin de Django

El panel de administración de Django nos proporciona una interfaz gráfica poderosa para gestionar nuestros datos sin necesidad de escribir SQL. Configurémoslo para nuestro modelo Orden.

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

```
@admin.register(Orden)
class
OrdenAdmin(admin.ModelAdmin):
    list_display = (
        'id',
        'cliente',
        'fecha',
        'total',
        'estado'
    )
    list_filter = ('estado', 'fecha')
    search_fields = (
        'cliente_nombre',
        'cliente_email'
    )
    date_hierarchy = 'fecha'
```

Características añadidas:

- **list_display:** columnas visibles en la lista
- **list_filter:** filtros laterales
- **search_fields:** búsqueda por cliente
- **date_hierarchy:** navegación por fechas

👤 Configuración avanzada: Puedes usar `InlineModelAdmin` para mostrar las órdenes directamente dentro de la página de detalle del cliente, permitiendo gestionar todo desde un solo lugar.



Probando en Django Shell

El shell interactivo de Django es una herramienta invaluable para probar consultas y experimentar con nuestros modelos en tiempo real. Veamos cómo crear y consultar órdenes desde aquí.

Inicia el shell con el comando `python manage.py shell` y ejecuta las siguientes consultas:

```
from clientes.models import Cliente
from ordenes.models import Orden

# Crear un nuevo cliente
c = Cliente.objects.create(
    nombre="Juan Pérez",
    email="juan@example.com"
)

# Crear órdenes para ese cliente
o1 = Orden.objects.create(
    cliente=c,
    total=1200.50
)

o2 = Orden.objects.create(
    cliente=c,
    total=850.00,
    estado='pagada'
)

# Consultar todas las órdenes del cliente
ordenes_juan = c.ordenes.all()
print(f"Juan tiene {ordenes_juan.count()} órdenes")

# Iterar sobre las órdenes
for orden in ordenes_juan:
    print(f"Orden #{orden.id}: ${orden.total} - {orden.estado}")
```

📋 🧠 **Desafío práctico:** Crea otro cliente con al menos tres órdenes de diferentes estados y montos. Luego practica las consultas inversas para obtener el cliente desde una orden usando `orden.cliente`.

Consultas frecuentes en relaciones 1:N

Dominar las consultas en relaciones Uno a Muchos es esencial para trabajar eficientemente con Django. Estas son las operaciones más comunes que realizarás en tu día a día como desarrollador.

Todas las órdenes

```
Orden.objects.all()
```

Obtiene la lista completa de órdenes en el sistema

Órdenes por cliente

```
Orden.objects.filter(cliente=c)
```

Filtra todas las órdenes de un cliente específico

Órdenes por estado

```
Orden.objects.filter(estado='pagada')
```

Encuentra todas las órdenes con un estado determinado

→ Consultas combinadas

Puedes encadenar filtros para consultas más específicas:


```
Orden.objects.filter(
    cliente=c,
    estado='pendiente'
).order_by('-fecha')
```

→ Agregaciones útiles

Calcula totales y estadísticas directamente en la base de datos:

```
from django.db.models import Sum, Count

total_ventas = Orden.objects.aggregate(
    Sum('total')
)
```

 **Pregunta de reflexión:** ¿Qué método usarías para obtener únicamente las órdenes pendientes de un cliente específico, ordenadas de la más reciente a la más antigua?

select_related: consultas más rápidas

Una de las optimizaciones más importantes en Django es el uso de `select_related()`. Este método previene el problema de "consultas N+1" que puede ralentizar significativamente tu aplicación.

✗ Sin optimización

```
ordenes = Orden.objects.all()
for o in ordenes:
    # Genera 1 consulta por orden
    print(o.cliente.nombre)
```

Si tienes 100 órdenes, esto genera 101 consultas a la base de datos (1 inicial + 100 para cada cliente).

1

Consulta total

Una sola consulta con JOIN recupera toda la información necesaria

✓ Con optimización

```
ordenes = Orden.objects.select_related('cliente').all()
for o in ordenes:
    # Sin consultas adicionales
    print(o.cliente.nombre)
```

Con 100 órdenes, solo genera 1 consulta usando un JOIN en SQL. ¡100 veces más rápido!

99%

Reducción tiempo

Puede reducir el tiempo de respuesta hasta en un 99% en consultas grandes

0

Consultas extra

Elimina completamente las consultas adicionales al acceder a objetos relacionados

🚀 **Beneficio clave:** `select_related()` realiza un JOIN automático en SQL, cargando los objetos relacionados en una sola operación. Es fundamental para optimizar el rendimiento en aplicaciones con muchos datos.

¿Qué hace on_delete?

El parámetro `on_delete` en un ForeignKey define qué sucede con los registros "hijos" cuando se elimina el registro "padre". Esta decisión tiene implicaciones importantes para la integridad de tus datos.

CASCADE

Comportamiento: Elimina automáticamente todas las órdenes cuando se borra el cliente.

Cuándo usar: Cuando los registros hijos no tienen sentido sin el padre (ej: órdenes sin cliente).

SET_NULL

Comportamiento: Establece el campo cliente en NULL, conservando las órdenes.

Cuándo usar: Para mantener historial de órdenes aunque el cliente ya no exista (requiere `null=True`).

PROTECT

Comportamiento: Impide eliminar el cliente si tiene órdenes asociadas.

Cuándo usar: Para prevenir eliminaciones accidentales de datos importantes.

DO_NOTHING

Comportamiento: No realiza ninguna acción automática.

Cuándo usar: Raramente recomendado, puede causar errores de integridad referencial.

🌟 **Mini-desafío:** ¿Qué opción de `on_delete` utilizarías si quieres conservar el historial completo de todas las órdenes para análisis financiero, incluso después de que un cliente cancele su cuenta?

Respuesta: `SET_NULL` con `null=True`, permitiendo mantener las órdenes para auditoría y análisis histórico.

¡Manos al código!

Es momento de poner en práctica todo lo aprendido. Este ejercicio guiado te ayudará a consolidar los conceptos de relaciones Uno a Muchos mediante la implementación práctica.

01

Crear clientes

Crea al menos 3 clientes diferentes con nombres y correos únicos usando `Cliente.objects.create()`

02

Generar órdenes

Para cada cliente, crea 2 órdenes con diferentes montos y estados (pendiente, pagada, cancelada)

03

Consultar relaciones

Usa `cliente.ordenes.all()` para mostrar todas las órdenes de cada cliente y verifica los resultados

04

Probar CASCADE

Elimina uno de los clientes y observa qué sucede con sus órdenes asociadas en la base de datos



Tiempo asignado: 10 minutos para completar todos los pasos



Tip profesional: Usa el método `related_name` para acceder rápidamente a las órdenes: `cliente.ordenes.all()` es más intuitivo que `cliente.orden_set.all()`

Al finalizar, compara tus resultados con un compañero y discutan cualquier diferencia o duda que surja durante el ejercicio.

Errores comunes y cómo solucionarlos

Aprender de los errores más frecuentes te ahorrará horas de depuración. Aquí están los problemas que encuentran la mayoría de desarrolladores al trabajar con relaciones Uno a Muchos.

❌ Olvidar `related_name`

Problema: Django genera automáticamente `orden_set` que es menos intuitivo

Solución: Siempre define `related_name='ordenes'` para mayor claridad

❌ No especificar `on_delete`

Problema: Las migraciones fallan con error de sintaxis

Solución: Siempre incluye `on_delete=models.CASCADE` o la opción apropiada

❌ Tipos de campo incompatibles

Problema: Usar `IntegerField` para montos en lugar de `DecimalField`

Solución: Usa `DecimalField(max_digits=10, decimal_places=2)` para dinero

- **Problema de migraciones circulares**

Si dos apps se referencian mutuamente, puede causar errores. Solución: usa cadenas de texto para referencias:
`ForeignKey('otra_app.Modelo')`

- **Consultas N+1 no detectadas**

El código funciona pero es lento. Solución: usa Django Debug Toolbar para identificar consultas innecesarias y aplica `select_related()`

- **Integridad referencial violada**

Intentar crear órdenes con clientes inexistentes. Solución: valida siempre la existencia del objeto relacionado antes de crear la relación

✅ Consejo profesional: Ejecuta regularmente `python manage.py check` para detectar problemas potenciales antes de que causen errores en producción.

Buenas prácticas en relaciones Uno a Muchos

Seguir las mejores prácticas desde el principio te ayudará a construir aplicaciones Django robustas, mantenibles y escalables. Estos son los principios que debes tener siempre presentes.



related_name descriptivos

Usa nombres que describan claramente la relación:

`related_name='ordenes'` en lugar de nombres genéricos como `'items'`



Usa `transaction.atomic()`

Al crear múltiples registros relacionados, envuélvelos en una transacción para garantizar consistencia: todo se guarda o nada



Modelos simples y claros

Mantén cada modelo enfocado en una sola responsabilidad. Si un modelo crece demasiado, considera dividirlo en múltiples modelos relacionados



Optimiza con `select_related`

Siempre usa `select_related()` cuando sepas que vas a acceder a objetos relacionados para evitar consultas N+1



Considera índices

Para campos que consultas frecuentemente, agrega `db_index=True` para mejorar el rendimiento de búsquedas



Documenta tus decisiones

Añade docstrings explicando por qué elegiste `CASCADE`, `PROTECT` u otras opciones en `on_delete`



Patrón recomendado para transacciones:

```
from django.db import transaction

with transaction.atomic():
    cliente = Cliente.objects.create(nombre="Ana")
    Orden.objects.create(cliente=cliente, total=100)
    Orden.objects.create(cliente=cliente, total=200)
```

Desafío grupal: Mini sistema de órdenes

Ha llegado el momento de aplicar todo lo aprendido en un proyecto colaborativo. Trabajarán en equipos para construir una extensión más completa del sistema de órdenes.

Objetivo del desafío

Crear un modelo adicional llamado Producto y vincularlo a Orden. Esto será un adelanto de las relaciones Muchos a Muchos que veremos en la próxima clase.



Organización

Formen equipos de 3 personas. Cada equipo trabajará en su propio branch de Git para mantener el código organizado.



Tiempo disponible


20 minutos para completar la implementación, pruebas y preparar una breve presentación de su solución.

Tareas a completar

1. Crear el modelo Producto con campos: nombre, precio, descripción
2. Agregar al menos 5 productos diferentes a la base de datos
3. Asociar 2-3 productos a cada orden creada
4. Mostrar los resultados en el shell de Django
5. Registrar Producto en el admin para visualización

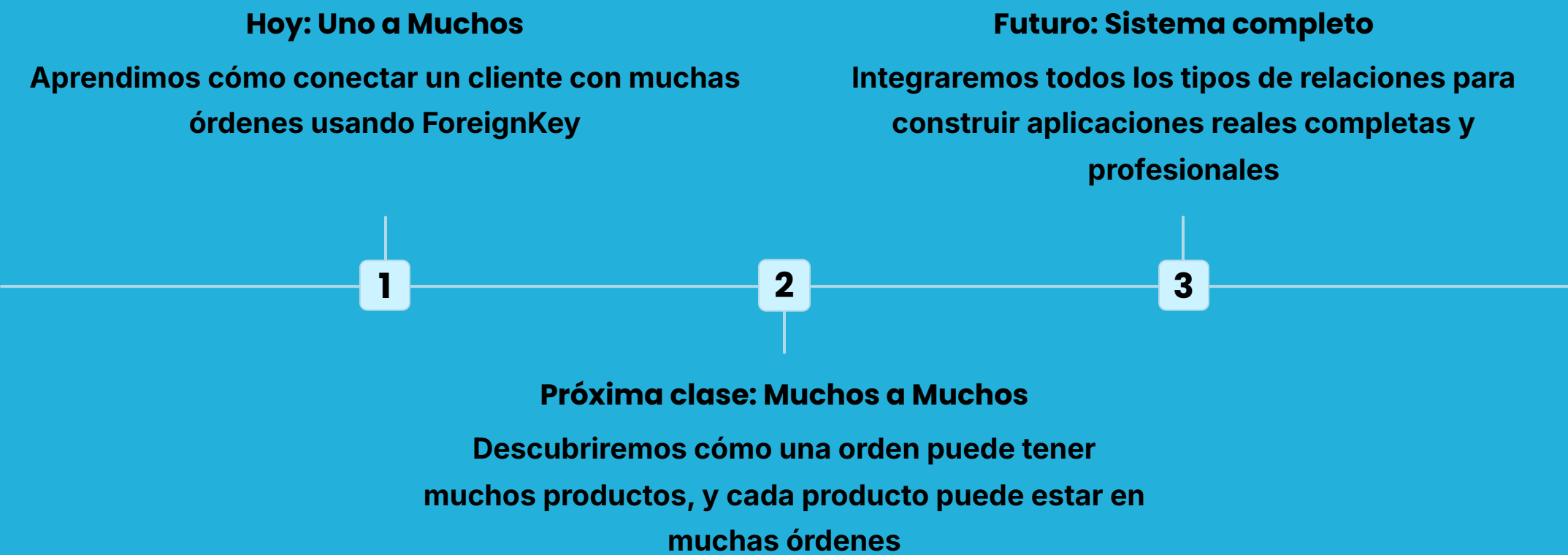
Criterios de evaluación

- Funcionalidad: El código funciona correctamente
- Buenas prácticas: Uso de related_name, on_delete apropiado
- Claridad: Código limpio y bien documentado
- Presentación: Explicación clara de las decisiones tomadas

 Al finalizar: Cada grupo presentará su código durante 3 minutos, explicando las decisiones de diseño y mostrando una demostración funcionando en el shell.



Lo que viene...

Hoy hemos dado un paso fundamental en el dominio de las relaciones en Django. Ahora que comprendes las relaciones Uno a Muchos, estás preparado para el siguiente nivel de complejidad.



Cliente	↓ Orden ↓	Producto
Una persona que realiza compras en nuestra tienda	Una transacción específica realizada por el cliente	Los artículos incluidos en cada orden




🔄 La arquitectura completa: Cliente → Orden → Producto representa un flujo de comercio electrónico real. Cada capa construye sobre la anterior, y pronto tendrás el conocimiento para implementar sistemas completos.



Adelanto emocionante: Las relaciones Muchos a Muchos usarán `ManyToManyField`, permitiendo que un producto aparezca en múltiples órdenes y cada orden contenga múltiples productos. ¡Es más simple de lo que imaginas!

Resumen y reflexión final

¡Felicidades! Has completado exitosamente esta clase sobre relaciones Uno a Muchos en Django. Repasemos los conceptos clave que ahora dominas y reflexionemos sobre su aplicación práctica.



		
Concepto dominado	Opciones on_delete	Proyecto práctico
Relaciones Uno a Muchos con ForeignKey	CASCADE, SET_NULL, PROTECT, DO_NOTHING	Sistema completo Cliente-Órdenes funcionando
<div><div></div><div>ForeignKey es la clave</div><div>Una relación Uno a Muchos se implementa con ForeignKey en el modelo "hijo", estableciendo la conexión con el modelo "padre"</div></div>	<div><div></div><div>Flexibilidad y escalabilidad</div><div>Cada registro padre puede tener ilimitados registros hijos, mientras cada hijo pertenece a un único padre</div></div>	<div><div></div><div>Aplicación real exitosa</div><div>Implementamos un caso práctico completo: Cliente → Órdenes en el proyecto bikeshop con MySQL</div></div>



Reflexión final

Pregunta para llevar a casa: ¿Dónde aplicarías relaciones Uno a Muchos en tu próximo proyecto Django?

Piensa en sistemas reales: redes sociales (usuario-publicaciones), educación (curso-estudiantes), logística (almacén-productos). Las posibilidades son infinitas.



Próximos pasos recomendados: Practica creando tus propias relaciones en un proyecto personal, experimenta con diferentes opciones de on_delete, y prepárate para la próxima clase donde exploraremos las poderosas relaciones Muchos a Muchos.