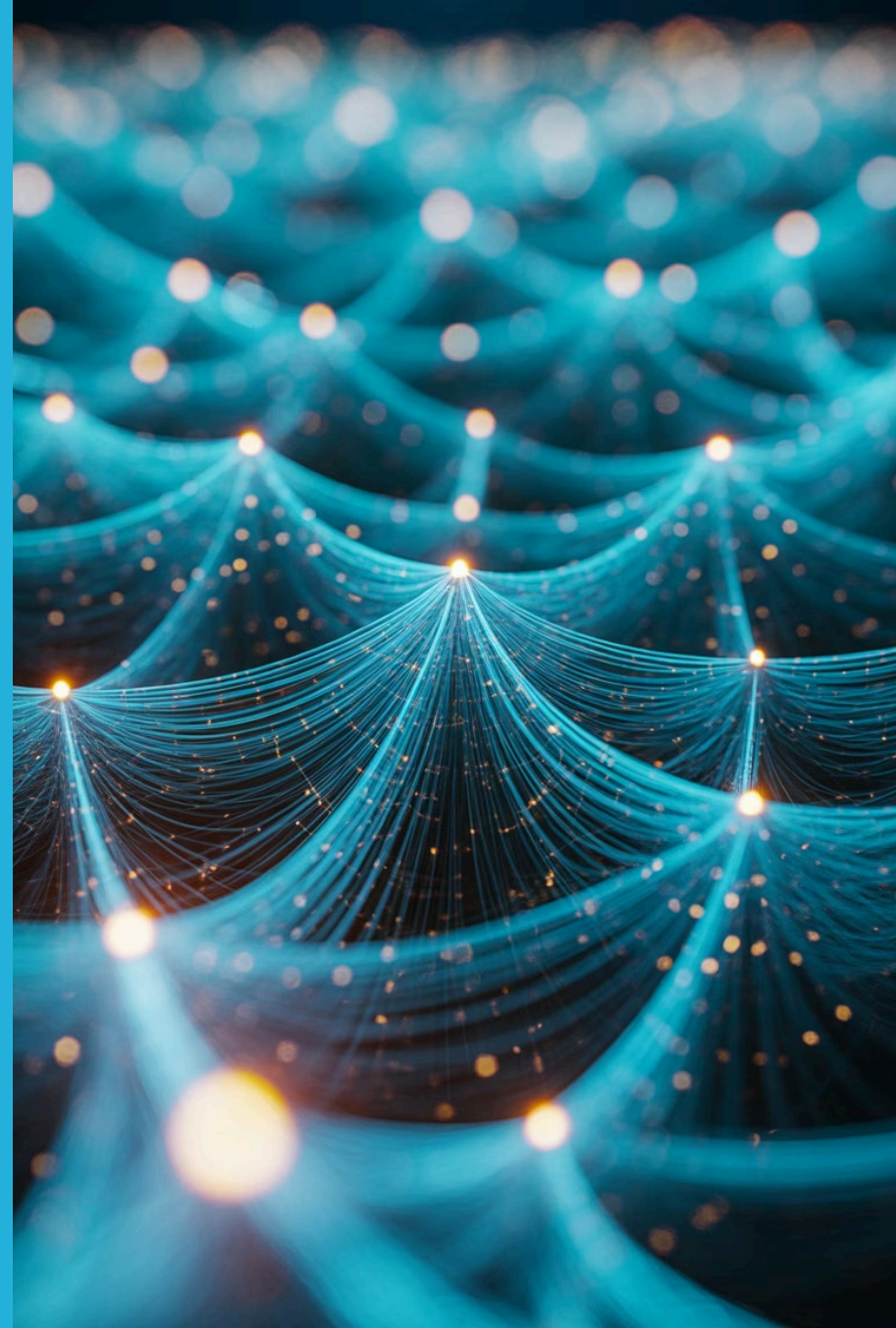


Relaciones Muchos a Muchos en Django

Domina las relaciones ManyToMany: desde la teoría hasta la implementación práctica con modelos intermedios en el proyecto Bikeshop



¿Qué es una relación Muchos a Muchos?

Una relación Muchos a Muchos (M:N) es un tipo de conexión bidireccional donde un registro de la tabla A puede relacionarse con múltiples registros de la tabla B, y viceversa. Es el patrón más flexible de las relaciones en bases de datos.

El ejemplo clásico es la relación entre Estudiantes y Cursos: un estudiante puede inscribirse en múltiples cursos, y cada curso puede tener muchos estudiantes inscritos simultáneamente.

En Django, esta relación se implementa mediante el campo `ManyToManyField`, que automáticamente gestiona la complejidad de las tablas intermedias necesarias para establecer estas conexiones.



Contexto del proyecto

Bikeshop

En nuestro proyecto Bikeshop, necesitamos modelar una situación del mundo real: las órdenes de compra y las bicicletas tienen una relación compleja que no puede resolverse con relaciones simples.

Escenario Real

Cada Orden puede incluir varias bicicletas diferentes (un cliente compra una Mountain Bike y una de Ruta)

Complejidad Adicional

Una Bicicleta del mismo modelo puede aparecer en múltiples órdenes de diferentes clientes

💡 **Pregunta para reflexionar:** ¿Qué sucedería si un cliente compra dos bicicletas del mismo modelo en órdenes diferentes? Esta situación nos lleva directamente a la necesidad de usar una tabla intermedia con información adicional.



Representación en la base de datos

Las relaciones Muchos a Muchos requieren una estructura especial en la base de datos que difiere de las relaciones Uno a Uno o Uno a Muchos. No es posible usar una simple clave foránea porque necesitamos almacenar múltiples conexiones en ambas direcciones.

1

Tabla Orden

Contiene información del cliente, fecha, total y estado de la compra

2

Tabla Intermedia (DetalleOrden)

Une Orden y Bicicleta mediante claves foráneas. Aquí guardamos cantidad y precio

3

Tabla Bicicleta

Almacena el catálogo de productos con nombre, precio y características



Estructura de la tabla intermedia: ordenes_bicicletas con campos orden_id y bicicleta_id como claves foráneas

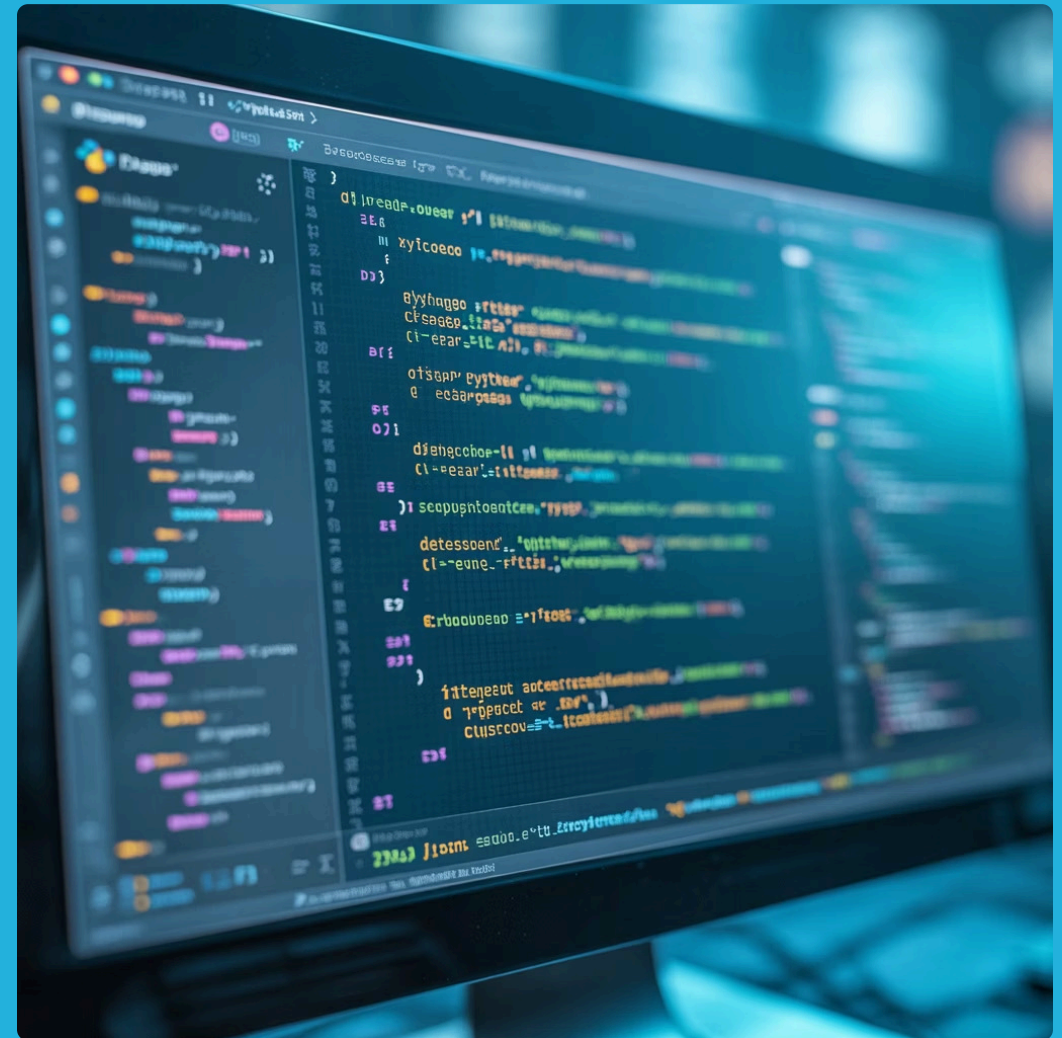


Sintaxis básica en Django

Django proporciona una forma elegante de definir relaciones M:N mediante `ManyToManyField`. La sintaxis más simple crea automáticamente la tabla intermedia sin campos adicionales.

```
class Orden(models.Model):
    bicicletas = models.ManyToManyField(
        'Bicicleta'
    )
```

Limitación importante: Este enfoque básico **no** permite almacenar información adicional sobre la relación, como cantidades o precios específicos de cada bicicleta en la orden.



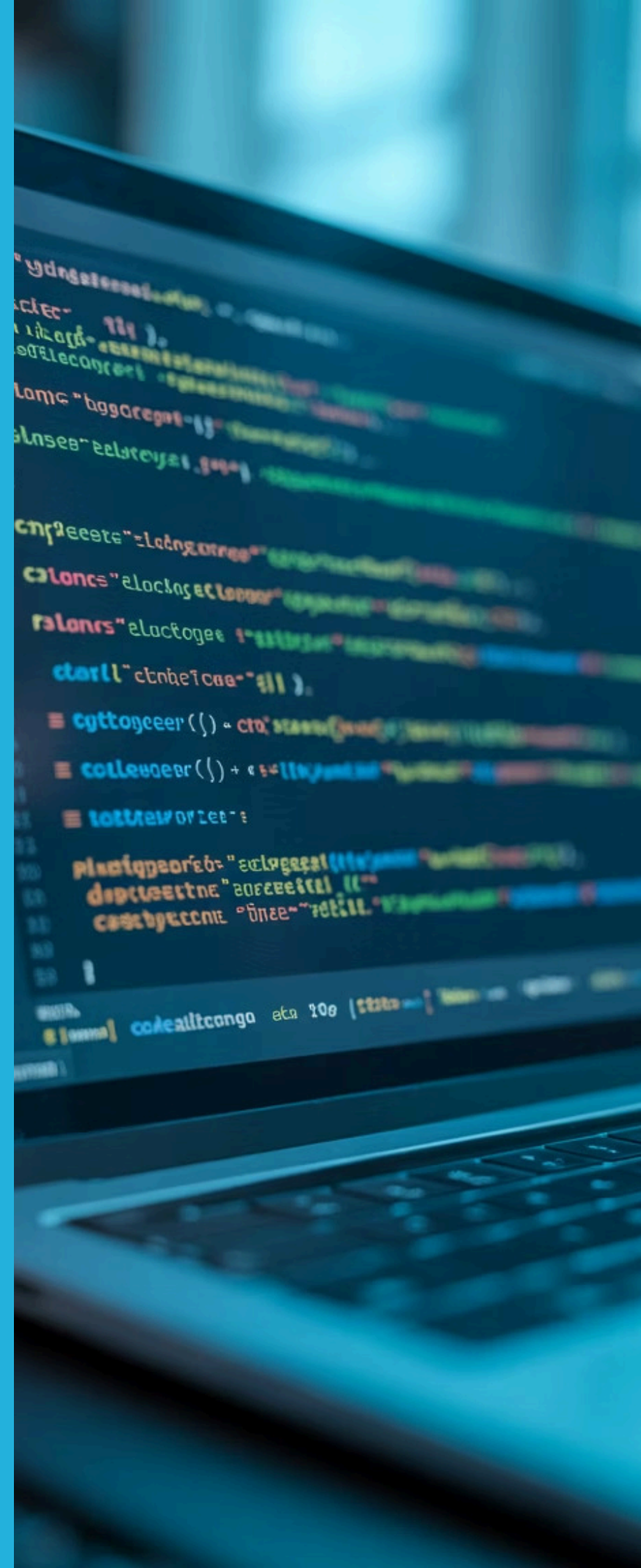
Por eso necesitamos usar `through='DetalleOrden'` para especificar un modelo intermedio personalizado

Implementación con tabla intermedia

La implementación completa requiere tres modelos trabajando en conjunto. Esta es la estructura recomendada para el proyecto Bikeshop:

```
class Orden(models.Model):
    cliente = models.ForeignKey(
        'clientes.Cliente',
        on_delete=models.CASCADE
    )
    bicicletas = models.ManyToManyField(
        'bicicletas.Bicicleta',
        through='DetalleOrden'
    )
    fecha = models.DateTimeField(auto_now_add=True)
    total = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        default=0
    )
    estado = models.CharField(max_length=20)
```

```
class DetalleOrden(models.Model):
    orden = models.ForeignKey(
        Orden,
        on_delete=models.CASCADE
    )
    bicicleta = models.ForeignKey(
        'bicicletas.Bicicleta',
        on_delete=models.CASCADE
    )
    cantidad = models.PositiveIntegerField(default=1)
    precio_unitario = models.DecimalField(
        max_digits=10,
        decimal_places=2
    )
```



Explicación detallada del modelo

Parámetro through

El argumento

`through='DetalleOrden'` establece una relación explícita que pasa por el modelo intermedio, permitiéndonos controlar completamente qué información adicional guardamos sobre cada conexión.

Campos personalizados

En `DetalleOrden`

agregamos `cantidad` para saber cuántas unidades de cada bicicleta se ordenaron, y `precio_unitario` para registrar el precio al momento de la compra (que puede variar del precio actual del catálogo).

Integridad referencial

El parámetro

`on_delete=models.CASCADE`

E mantiene la integridad de los datos: si se elimina una orden o bicicleta, automáticamente se borran los registros relacionados en `DetalleOrden`.



Creando y aplicando migraciones

Después de definir los modelos, Django necesita crear las tablas correspondientes en la base de datos. Este proceso se realiza mediante el sistema de migraciones.

Comandos esenciales:

```
python manage.py makemigrations ordenes
```

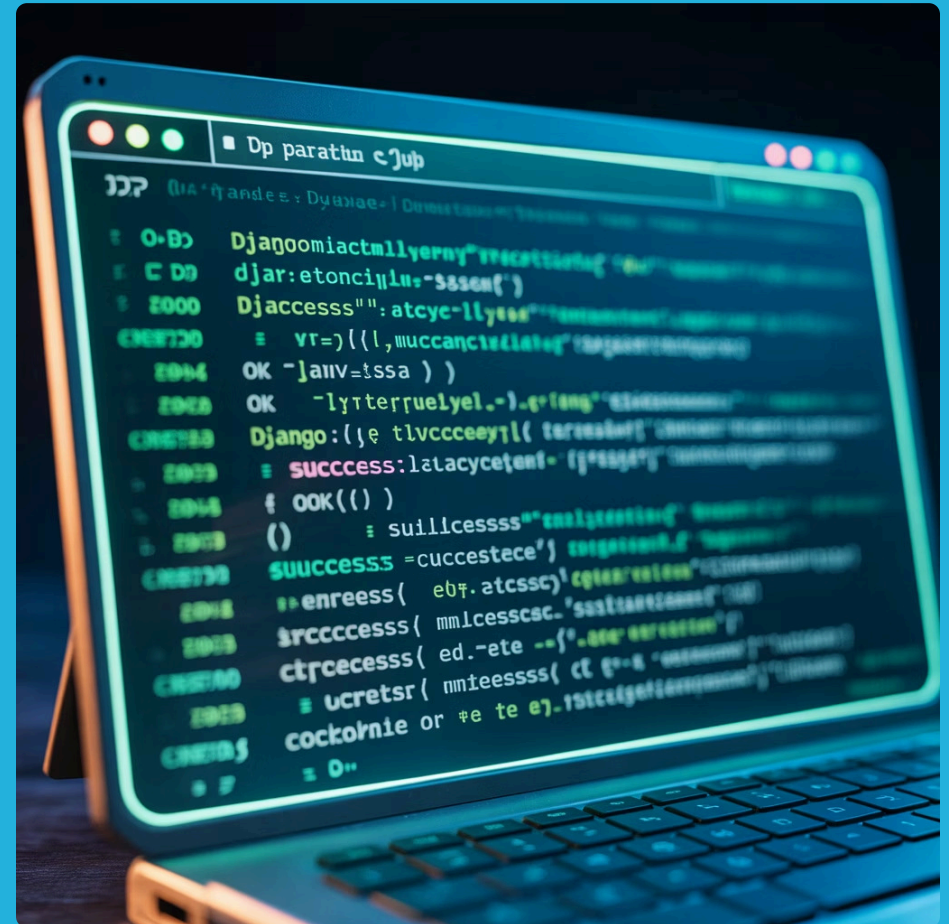
Este comando analiza los cambios en tus modelos y genera archivos de migración con las instrucciones SQL necesarias.

```
python manage.py migrate
```

Ejecuta las migraciones pendientes y modifica la estructura de la base de datos.

Verificación:

- Abre phpMyAdmin o DBeaver
- Localiza la tabla `ordenes_detalleorden`
- Confirma las claves foráneas hacia `orden_id` y `bicicleta_id`



Registro en el panel de administración

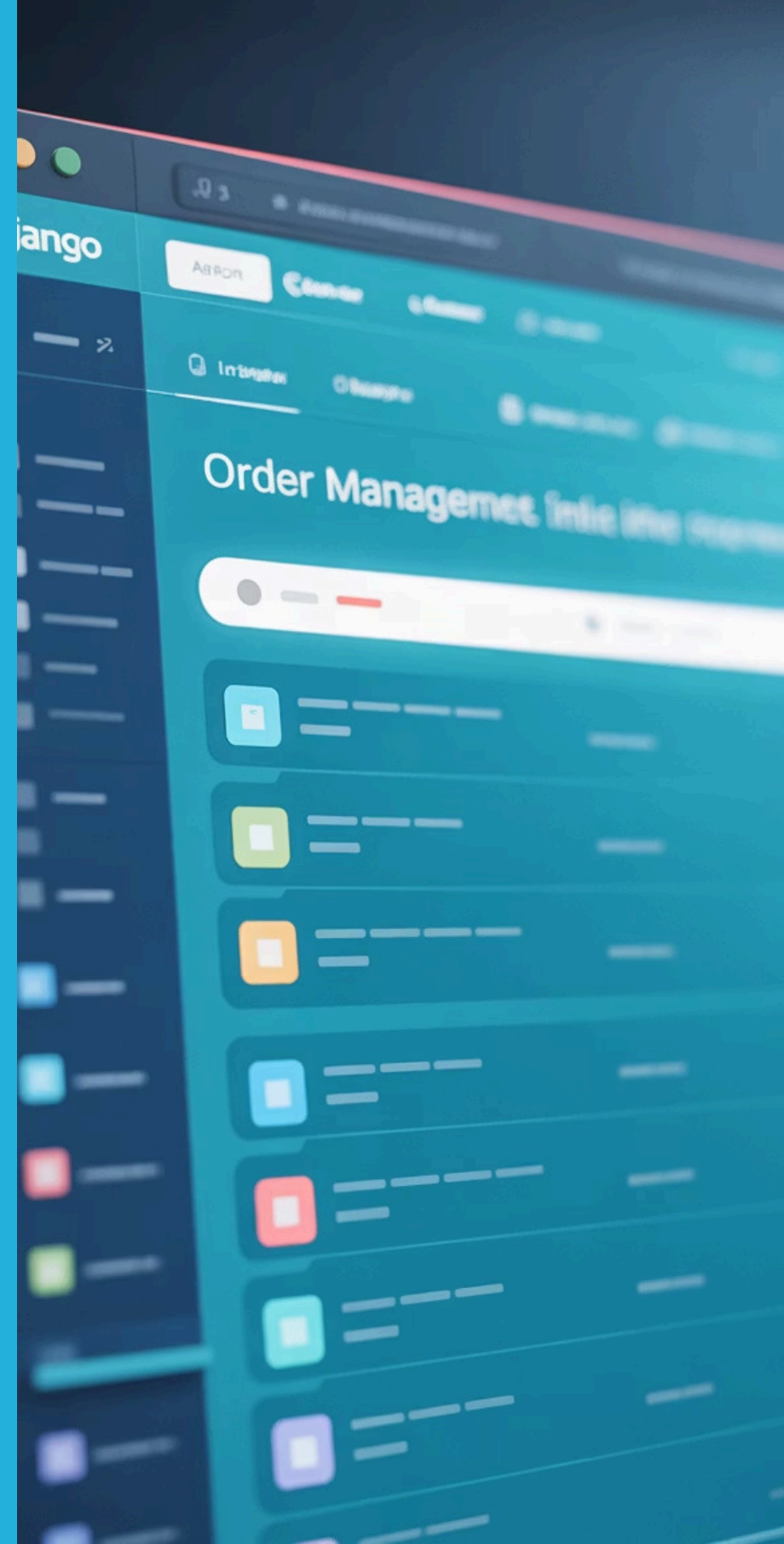
Django proporciona un poderoso panel de administración que podemos personalizar para gestionar nuestras órdenes y sus detalles de forma intuitiva. La configuración con `InlineModelAdmin` permite editar modelos relacionados en la misma pantalla.

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

class DetalleOrdenInline(admin.TabularInline):
    model = DetalleOrden
    extra = 1
    fields = ['bicicleta', 'cantidad', 'precio_unitario']

@admin.register(Orden)
class OrdenAdmin(admin.ModelAdmin):
    inlines = [DetalleOrdenInline]
    list_display = ('id', 'cliente', 'fecha', 'total', 'estado')
    list_filter = ('estado', 'fecha')
    search_fields = ('cliente__nombre',)
```

Beneficio clave: Desde el panel de administración puedes crear una orden completa, seleccionar el cliente, agregar múltiples bicicletas con sus cantidades y precios, todo en una sola pantalla integrada.



Probando en Django Shell – Parte 1

El Django Shell es una herramienta interactiva fundamental para probar y experimentar con nuestros modelos. Vamos a crear datos de ejemplo paso a paso para entender cómo funcionan las relaciones.

01

Iniciar el shell

Ejecuta `python manage.py shell`

02

Importar modelos

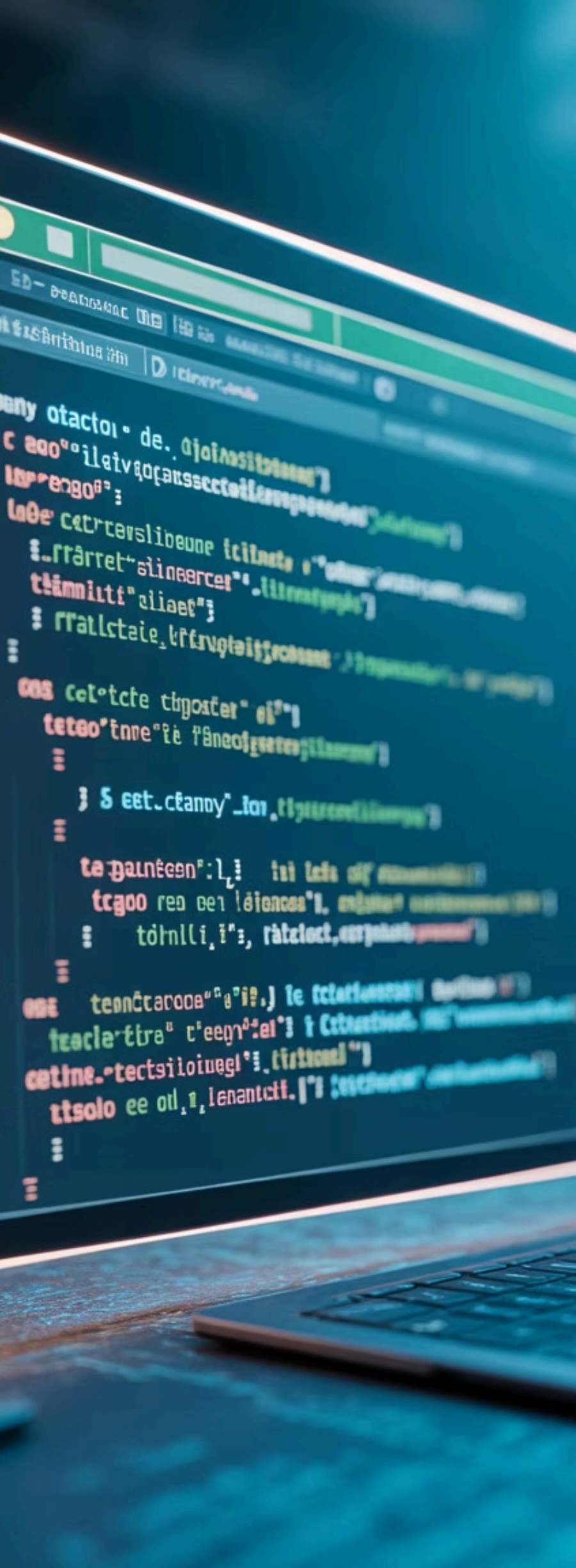
```
from clientes.models import
Cliente
from bicicletas.models import
Bicicleta
from ordenes.models import
Orden, DetalleOrden
```

03

Crear objetos base

```
c = Cliente.objects.create(
    nombre="Laura Gómez",
    email="laura@email.com"
)
b1 = Bicicleta.objects.create(
    nombre="Mountain Bike",
    precio=500
)
b2 = Bicicleta.objects.create(
    nombre="Ruta Pro",
    precio=800
)
o = Orden.objects.create(cliente=c)
```

💡 **Actividad:** Escribe estos comandos en tu Django Shell y observa cómo se crean los registros en las tablas correspondientes.



Probando en Django Shell – Parte 2

Crear los detalles de la orden

```
DetalleOrden.objects.create(  
    orden=o,  
    bicicleta=b1,  
    cantidad=2,  
    precio_unitario=500  
)
```

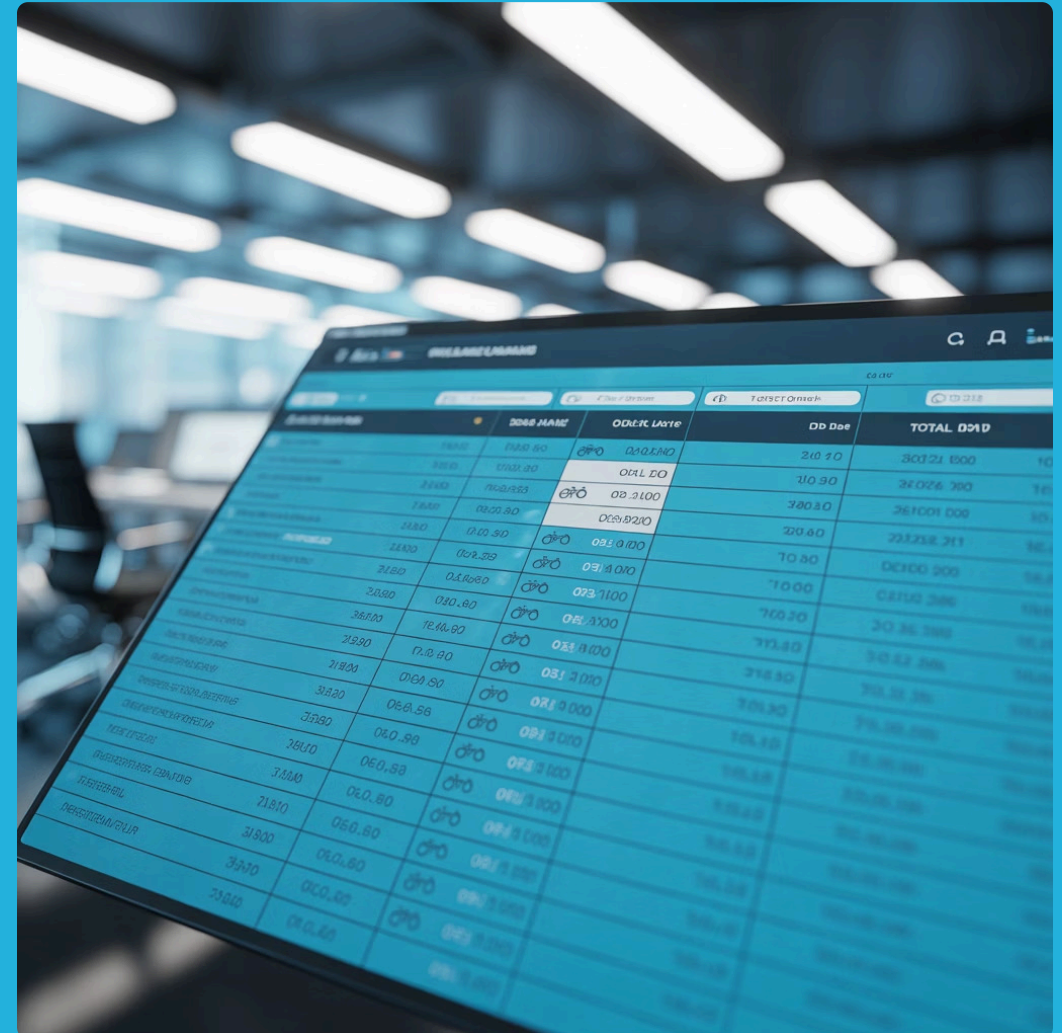
```
DetalleOrden.objects.create(  
    orden=o,  
    bicicleta=b2,  
    cantidad=1,  
    precio_unitario=800  
)
```

Consultas bidireccionales

```
# Desde la orden a las bicicletas  
o.bicicletas.all()
```

```
# Desde la bicicleta a las órdenes  
b1.orden_set.all()
```

Django crea automáticamente relaciones inversas usando `_set` cuando no especificamos `related_name`.



ORDEN	BIKES	CANTIDAD	PRECIO UNITARIO	TOTAL
1	1	2	500	1000
1	2	1	800	800
2	1	1	500	500
2	2	1	800	800
3	1	1	500	500
3	2	1	800	800
4	1	1	500	500
4	2	1	800	800
5	1	1	500	500
5	2	1	800	800
6	1	1	500	500
6	2	1	800	800
7	1	1	500	500
7	2	1	800	800
8	1	1	500	500
8	2	1	800	800
9	1	1	500	500
9	2	1	800	800
10	1	1	500	500
10	2	1	800	800

📄 Verificación en la base de datos: Abre phpMyAdmin y revisa la tabla `ordenes_detalleorden`. Deberías ver dos filas con los IDs correspondientes y los valores de cantidad y precio que ingresaste.

Cálculo automático del total

Un requisito común en sistemas de órdenes es calcular automáticamente el total basándose en los items incluidos. Django nos permite implementar esta lógica de múltiples formas.

Método básico con loop

```
total = sum(
    d.cantidad * d.precio_unitario
    for d in DetalleOrden.objects.filter(orden=o)
)
o.total = total
o.save()
```

Este enfoque calcula la suma iterando sobre todos los detalles y multiplicando cantidad por precio unitario.

Método optimizado con agregación

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

total = DetalleOrden.objects.filter(
    orden=o
).aggregate(
    total=Sum(F('cantidad') * F('precio_unitario'))
)['total']
o.total = total or 0
o.save()
```

Usando agregación de Django, el cálculo se realiza en la base de datos, lo cual es mucho más eficiente para órdenes grandes.

🤔 **Concepto activo:** Piensa cómo podrías automatizar este cálculo usando señales de Django (signals) para que el total se actualice automáticamente cada vez que se crea o modifica un DetalleOrden.

Consultas avanzadas y optimización

A medida que tu aplicación crece, la eficiencia de las consultas se vuelve crítica. Django ofrece herramientas poderosas para optimizar el acceso a datos relacionados y evitar el problema de consultas N+1.

Problema: Consultas N+1

```
ordenes = Orden.objects.prefetch_related('bicicletas').all()
for o in ordenes:
    print(o.cliente.nombre, [b.nombre for b in o.bicicletas.all()])
```

Buenas prácticas en relaciones M:N



Nombres descriptivos

Usa `related_name='ordenes'` en lugar del automático `_set` para hacer el código más legible y explícito.



Transacciones atómicas

Envuelve la creación de orden + detalles con `transaction.atomic()` para garantizar consistencia: o se guardan todos los datos o ninguno.



Validación de datos

Implementa validaciones en el modelo: cantidad debe ser mayor que 0, precio no puede ser negativo, etc.



Documentación clara

Documenta tus relaciones con docstrings explicando qué representa cada modelo y cómo se relacionan entre sí.



Ejemplo de transacción: `with transaction.atomic():` seguido del código de creación de objetos relacionados garantiza que todos los cambios se guarden juntos o se revierten en caso de error.

Errores comunes y soluciones



IntegrityError

Error: IntegrityError (clave foránea inválida)

Causa: Intentas crear un DetalleOrden referenciando una Orden o Bicicleta que no existe en la base de datos.

Solución: Verifica que los objetos existan antes de crear la relación usando `Orden.objects.filter(id=x).exists()`



Through Model Error

Error: "You must specify through model"

Causa: Intentas usar `.add()` o `.create()` en una relación M:N con modelo intermedio personalizado.

Solución: Crea instancias de DetalleOrden directamente en lugar de usar métodos del ManyToManyField.



Duplicate Key

Error: duplicate key value violates unique constraint

Causa: Intentas crear el mismo detalle de orden dos veces (misma orden + misma bicicleta).

Solución: Usa `get_or_create()` o verifica existencia antes de crear: `if not DetalleOrden.objects.filter(...).exists()`

Actividad práctica en equipo

Desafío de implementación completa

Es momento de aplicar todo lo aprendido en un ejercicio práctico colaborativo. Trabaja con tu equipo para completar los siguientes pasos:



Crear bicicletas

Define tres nuevas bicicletas en el catálogo con nombres creativos y precios realistas.



Generar orden

Crea una orden para un cliente existente o nuevo, y asocia dos de las tres bicicletas creadas.



Calcular total


Implementa el cálculo automático del total usando agregación de Django.



Consulta inversa

Muestra en consola todas las órdenes que incluyan una bicicleta específica.

 Tiempo asignado: 20 minutos

 Propósito: Consolidar el conocimiento teórico mediante práctica colaborativa y resolución de problemas reales.



Entidades intermedias enriquecidas

Las entidades intermedias no se limitan a conectar dos tablas. Pueden convertirse en modelos ricos en información que describen completamente la naturaleza de la relación.

Campos adicionales útiles:

- **Timestamp:** fecha_creacion para auditoría
- **Usuario:** quién agregó el item a la orden
- **Descuento:** porcentaje o monto de descuento aplicado
- **Observaciones:** notas especiales sobre ese item
- **Estado:** pendiente, confirmado, enviado

```
class DetalleOrden(models.Model):
    # Relaciones básicas
    orden = models.ForeignKey(...)
    bicicleta = models.ForeignKey(...)

    # Información comercial
    cantidad = models.PositiveIntegerField()
    precio_unitario = models.DecimalField(...)
    descuento = models.DecimalField(
        max_digits=5,
        decimal_places=2,
        default=0
    )

    # Metadatos
    fecha_agregado = models.DateTimeField(
        auto_now_add=True
    )
    observaciones = models.TextField(
        blank=True
    )
```



Este enriquecimiento convierte la tabla intermedia en una entidad de negocio completa, permitiendo trazabilidad, análisis y auditoría detallada de cada transacción.

Ejercicio individual: Autores y Libros

Ahora es tu turno de aplicar estos conceptos en un contexto diferente. Este ejercicio individual te ayudará a reforzar tu comprensión de las relaciones M:N con modelos intermedios.

1

Diseñar los modelos

Crea dos modelos: `Autor` (con nombre, biografia) y `Libro` (con titulo, año_publicacion, ISBN).

2

Relación M:N con información adicional

Implementa un modelo intermedio `Colaboracion` que incluya el campo `rol` con opciones: "principal", "coautor", "editor".

3

Poblar con datos

Crea al menos 3 autores y 2 libros, donde un libro tenga múltiples autores con diferentes roles.

4

Consulta filtrada

Escribe una consulta que muestre todos los autores principales:

```
Colaboracion.objects.filter(rol='principal').select_related('autor', 'libro')
```



Tip: Usa `choices` en el campo `rol` para asegurar que solo se ingresen valores válidos: `rol = models.CharField(max_length=20, choices=ROL_CHOICES)`

Conexión con la próxima clase

Has dominado las relaciones Muchos a Muchos, pero el viaje apenas comienza. En la siguiente lección integraremos todo lo aprendido hasta ahora.

- 1** — **Uno a Uno**
Usuario ↔ Perfil
- 2** — **Uno a Muchos**
Cliente ↔ Órdenes
- 3** — **Muchos a Muchos**
Orden ↔ Bicicletas
- 4** — **Integración completa**
Aplicación real

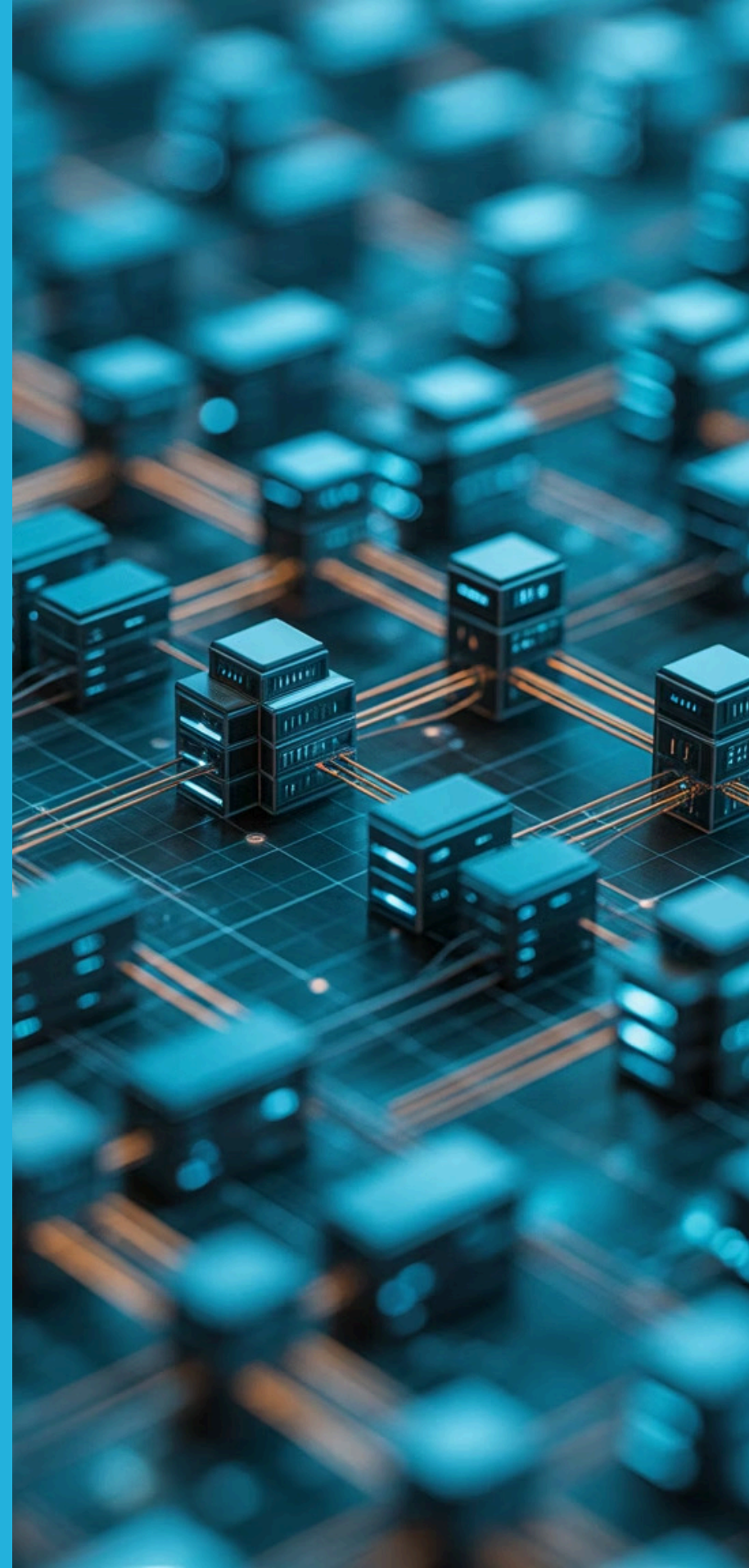
Lo que aprenderemos:

Consultas cruzadas

- Filtros a través de múltiples relaciones
- Agregaciones complejas
- Anotaciones con datos relacionados

Optimización de rendimiento

- Estrategias de caching
- Índices de base de datos
- Perfilado de consultas



Resumen y conceptos clave

ManyToManyField

Une múltiples registros
bidireccionales entre dos
modelos de forma elegante

Herramientas esenciales


`related_name`, `prefetch_related`,
`transaction.atomic()`



Modelo through

Permite agregar campos
personalizados a la relación
usando tabla intermedia

Relaciones bidireccionales

Orden  Bicicleta: navega en
ambas direcciones con facilidad

Las relaciones Muchos a Muchos son fundamentales para modelar la complejidad del mundo real en aplicaciones Django. Dominar este concepto te permite crear sistemas robustos y escalables que reflejan fielmente las relaciones complejas de tu dominio de negocio.

🎯 Recuerda: La clave está en elegir la relación correcta para cada caso de uso. M:N es poderoso, pero solo cuando realmente necesitas esa flexibilidad bidireccional.