

AE1 – Integración del Framework Django con Bases de Datos

Módulo: Desarrollo de Aplicaciones Web con Python y Django

Profesor: Cristian Iglesias

Objetivo general: Comprender cómo Django se integra con bases de datos relacionales utilizando su ORM (Object-Relational Mapping), permitiendo manipular datos de forma elegante y eficiente sin necesidad de escribir SQL manualmente.



Objetivos del Aprendizaje

Al finalizar esta clase, desarrollarás competencias fundamentales para trabajar con Django y bases de datos de manera profesional:



Integración con BD

Comprender cómo Django se comunica con diferentes motores de bases de datos



Motores Compatibles

Identificar SQLite, MySQL, PostgreSQL y Oracle como opciones disponibles



Configuración

Configurar correctamente la conexión en settings.py según el motor elegido



Operaciones CRUD

Usar el ORM de Django para crear, leer, actualizar y eliminar registros



Migraciones

Ejecutar migraciones y consultas personalizadas de forma eficiente

🧠 ¿Por qué Django usa un ORM?

Concepto Fundamental

ORM (Object-Relational Mapping) significa "mapeo objeto-relacional". Es una técnica de programación que permite manipular datos de una base de datos utilizando clases y objetos en Python, sin necesidad de escribir SQL directamente.

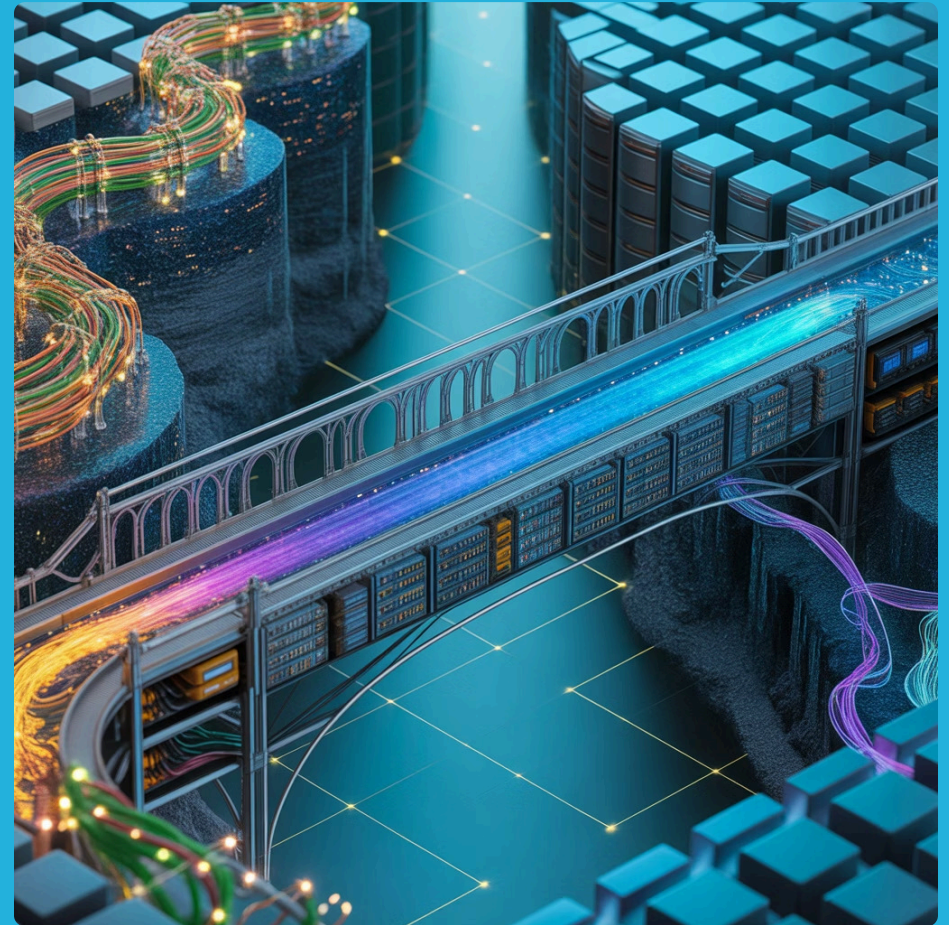
Esta abstracción hace que el código sea más legible, mantenible y portable entre diferentes motores de bases de datos.

Ejemplo sencillo de traducción:

```
bicis = Bicicleta.objects.filter(tipo="mtb")
```

equivale a

```
SELECT * FROM bicicletas WHERE tipo = 'mtb';
```



💡 **Ventaja clave:** Escribes código Python orientado a objetos y Django se encarga de traducirlo al lenguaje SQL específico de tu base de datos.



⚙️ Bases de Datos Soportadas por Django

Django ofrece compatibilidad nativa con múltiples motores de bases de datos, permitiendo elegir el más adecuado según las necesidades del proyecto:

1

SQLite

Motor por defecto en Django. Ideal para desarrollo y prototipado rápido. No requiere servidor separado, almacena todo en un archivo local.

Instalación: Incluido en Python

2

PostgreSQL

Base de datos robusta y avanzada, ideal para producción. Ofrece características empresariales como transacciones ACID completas y soporte para JSON.

Instalación: `pip install psycopg2`

3

MySQL

Ampliamente usada por su velocidad, confiabilidad y compatibilidad. Excelente para aplicaciones web de alto tráfico.

Instalación: `pip install mysqlclient`

4

Oracle

Común en entornos empresariales grandes. Proporciona escalabilidad y características avanzadas para sistemas críticos.


Instalación: `pip install cx_Oracle`

¿Y las bases de datos NoSQL?

Django fue diseñado originalmente para trabajar con bases de datos relacionales (SQL), siguiendo el modelo tradicional de tablas con relaciones definidas.

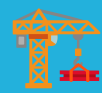
Sin embargo, es posible conectar Django a bases de datos NoSQL como MongoDB utilizando paquetes externos desarrollados por la comunidad:

```
pip install djongo
```

 **Nota importante:** El soporte a NoSQL es limitado y experimental. Se recomienda principalmente para proyectos académicos o de investigación, no para producción empresarial.



Recomendación: Para proyectos profesionales, mantén el uso de bases de datos SQL donde Django ofrece soporte completo y estable.



Configuración de la Base de Datos (SQLite por defecto)

Todo proyecto Django nuevo viene preconfigurado con SQLite. La configuración se encuentra en el archivo `settings.py` dentro del directorio principal del proyecto:

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

ENGINE

Especifica el motor de base de datos que Django utilizará. En este caso, `django.db.backends.sqlite3` indica SQLite.

NAME

Define el nombre o la ruta completa del archivo de base de datos. Para SQLite, es simplemente un archivo en el sistema.



Actividad guiada: Abre tu proyecto en VS Code, localiza el archivo `settings.py` y examina la sección `DATABASES` para familiarizarte con su estructura.



Configurar MySQL (conceptual)

Para conectar Django con MySQL, necesitas modificar la configuración en `settings.py` proporcionando los datos de conexión al servidor:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'mi_base',  
        'USER': 'root',  
        'PASSWORD': '1234',  
        'HOST': 'localhost',  
        'PORT': '3306',  
    }  
}
```

Paso previo obligatorio: Crear la base de datos en el servidor MySQL antes de ejecutar las migraciones de Django:

```
CREATE DATABASE mi_base;
```



Importante: A diferencia de SQLite, MySQL requiere un servidor de base de datos ejecutándose y credenciales de acceso configuradas.



Creando un Proyecto Django

Vamos a crear un proyecto completo desde cero para una tienda de bicicletas. Sigue estos pasos en tu terminal:



📁 Estructura generada:

```
bikeshop/
├── settings.py ← configuración DB
├── bicicletas/
│   └── models.py ← modelos ORM
```

```
veesshot llochostr >>,
jango Django developmneñst sereeri
thuttjovetruuietiagootnetttkhostcting
aanttuvlig rue t at-ut
teesstit raunvetestgfortiilat >
vessthully, siathesting,
tccthnclo >
tc.ilott t >
localhosts .a -wosellssst}>>10st>
a
```




Django como ORM: Definiendo Modelos

Los modelos en Django son clases Python que representan tablas en la base de datos. Cada atributo de la clase se convierte en una columna de la tabla:

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

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

```
    marca = models.CharField(max_length=50)
```

```
    modelo = models.CharField(max_length=50)
```

```
    tipo = models.CharField(max_length=20)
```

```
    precio = models.DecimalField(max_digits=10, decimal_places=2)
```

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

```
    anio = models.IntegerField()
```

```
    def __str__(self):
```

```
        return f"{self.marca} {self.modelo}"
```

Tipos de campos más comunes:

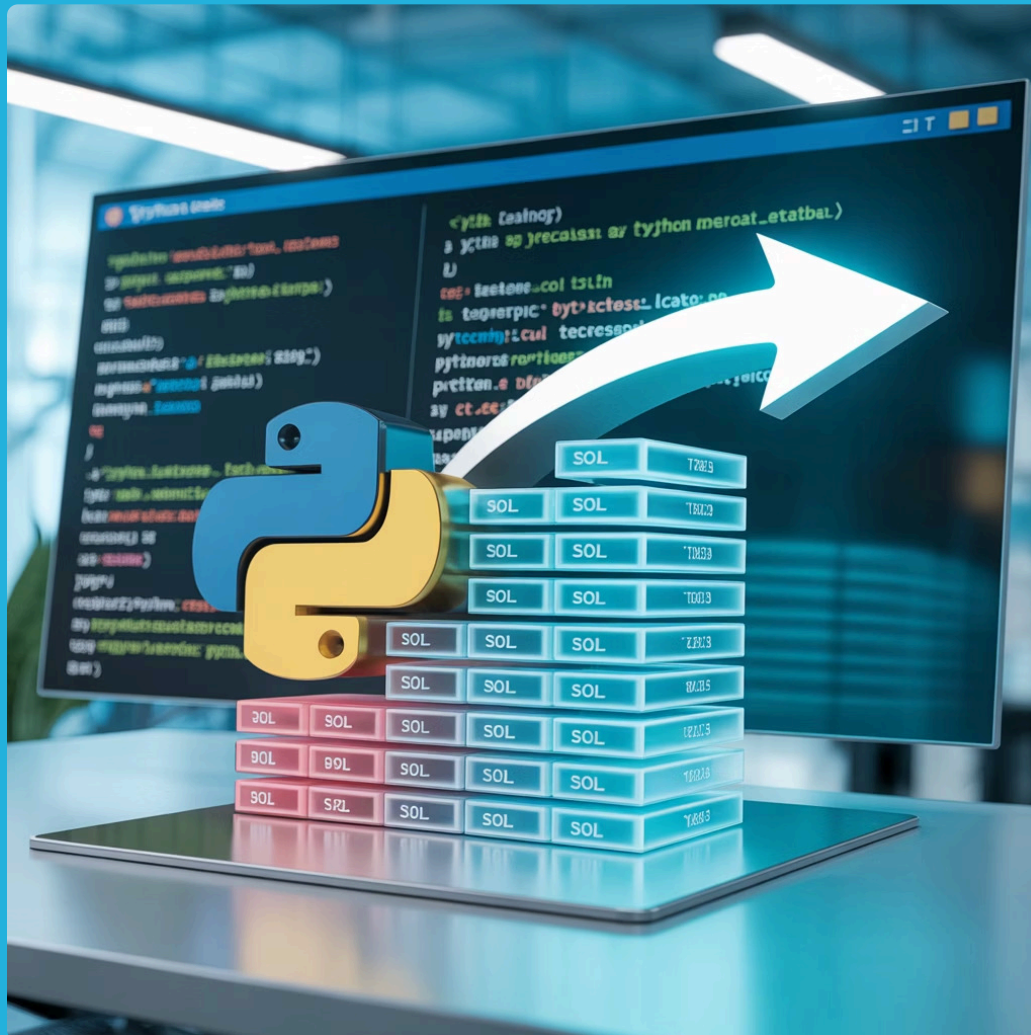
- **CharField**: Texto corto con longitud máxima definida
- **DecimalField**: Números decimales precisos (ideal para precios)
- **BooleanField**: Valores verdadero/falso
- **IntegerField**: Números enteros
- **DateTimeField**, **DateField**: Fechas y horas

¿Qué hace el ORM?

El ORM de Django actúa como un traductor inteligente que convierte automáticamente tus clases Python en estructuras de base de datos SQL.

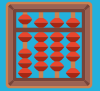
Ejemplo de traducción:

La clase `Bicicleta` se convierte en la tabla `bicicletas_bicicleta` en la base de datos.



Ventajas del ORM

- ☐ **Sin SQL manual**
No necesitas escribir consultas SQL directamente, Django lo hace por ti
- ☐ **Portabilidad**
El mismo código funciona con SQLite, MySQL, PostgreSQL u Oracle sin cambios
- ☐ **Validación automática**
Django valida los datos automáticamente según los tipos de campo definidos
- ☐ **Seguridad**
Protección automática contra inyección SQL y otros ataques comunes



Migraciones: Sincronizando Código y Base de Datos

Las migraciones son el mecanismo de Django para crear y actualizar la estructura de la base de datos basándose en los cambios en tus modelos. Es un sistema de control de versiones para tu esquema de base de datos.

1

makemigrations

```
python manage.py makemigrations
```

Detecta cambios en tus modelos y genera archivos de migración que describen esos cambios

2

migrate

```
python manage.py migrate
```

Aplica las migraciones pendientes a la base de datos, creando o modificando tablas según corresponda



Actividad práctica: Después de definir el modelo Bicicleta, ejecuta ambos comandos y observa el archivo de migración generado en `bicicletas/migrations/0001_initial.py`. Verás código Python que Django traduce a SQL.

Flujo completo: Modelo → makemigrations → archivo .py → migrate → tabla en BD



Consultas con el ORM

El ORM de Django proporciona una API elegante y poderosa para realizar consultas a la base de datos sin escribir SQL. Aquí los métodos más utilizados:

Consultas básicas

Obtener todas las bicicletas

`Bicicleta.objects.all()`

Filtrar por tipo específico

`Bicicleta.objects.filter(tipo="mtb")`

Bicicletas disponibles

`Bicicleta.objects.filter(disponible=True)`

Ordenar por precio descendente

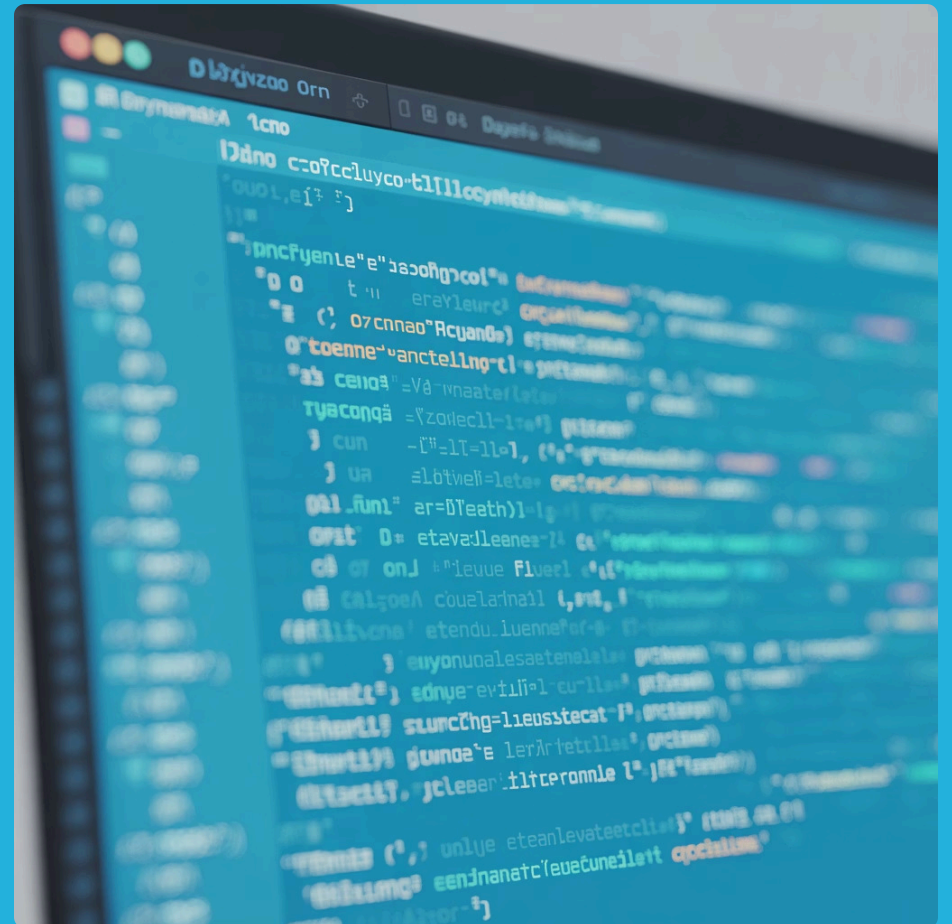
`Bicicleta.objects.order_by('-precio')`


Obtener un único objeto

`Bicicleta.objects.get(id=1)`

Contar registros

`Bicicleta.objects.count()`



 **Actividad práctica:** Abre la consola interactiva de Django con `python manage.py shell` y prueba estas consultas en tiempo real.

Tip profesional: Usa el signo `-` delante del nombre del campo para ordenar de forma descendente.

Insertar, Actualizar y Eliminar Datos

El ORM hace que las operaciones CRUD (Create, Read, Update, Delete) sean extremadamente simples y elegantes en Python:

Crear

```
Bicicleta.objects.create(  
    marca="Trek",  
    modelo="Marlin 7",  
    tipo="mtb",  
    precio=799,  
    anio=2023  
)
```

Crea un nuevo registro en la base de datos instantáneamente

Actualizar

```
bici =  
Bicicleta.objects.get(id=1)  
bici.precio = 900  
bici.save()
```

Modifica un registro existente y guarda los cambios

Eliminar

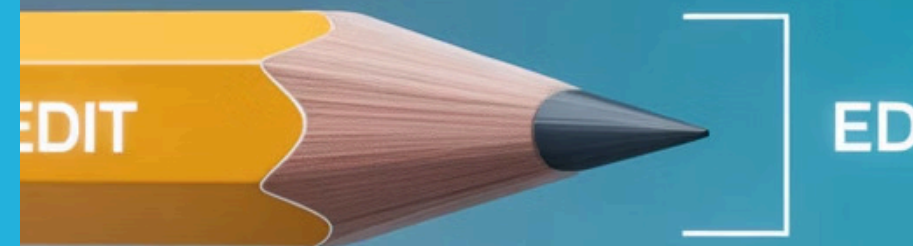
```
bici = Bicicleta.objects.get(id=1)  
bici.delete()
```

Elimina un registro de forma permanente

✨ **Ventaja principal:** Todo esto sin escribir una sola línea de SQL. Django traduce automáticamente estas operaciones a INSERT, UPDATE y DELETE según corresponda.



CREATE



DELETE



Consultas SQL Manuales

Aunque el ORM cubre la mayoría de necesidades, hay situaciones donde necesitas ejecutar SQL directamente para consultas complejas o para optimización de rendimiento:


Método 1: Usando `connection.cursor()`

```
from django.db import connection

with connection.cursor() as cursor:
    cursor.execute("SELECT * FROM bicicletas_bicicleta WHERE disponible = %s", [True])
    resultados = cursor.fetchall()
    for fila in resultados:
        print(fila)
```

Método 2: Usando `.raw()`

```
bicis = Bicicleta.objects.raw("SELECT * FROM bicicletas_bicicleta WHERE precio > 500")
for bici in bicis:
    print(bici.marca, bici.modelo)
```

 **Precaución:** Al usar SQL manual, pierdes algunas ventajas del ORM como la portabilidad entre motores y la protección automática. Úsalo solo cuando sea realmente necesario.

Consola Interactiva (Shell)

Django incluye una consola interactiva Python que es extremadamente útil para probar código, experimentar con el ORM y depurar sin necesidad de levantar el servidor web completo.

Iniciar la consola

```
python manage.py shell
```

Una vez dentro, puedes importar y usar tus modelos:

```
from bicicletas.models import Bicicleta
```

```
# Crear una bicicleta
```

```
Bicicleta.objects.create(  
    marca="Scott",  
    modelo="Scale 940",  
    tipo="mtb",  
    precio=1100,  
    anio=2024  
)
```

```
# Listar todas
```

```
Bicicleta.objects.all()
```

```
# Filtrar
```

```
Bicicleta.objects.filter(precio__gte=1000)
```



Casos de uso:

- Probar consultas ORM complejas
- Insertar datos de prueba rápidamente
- Verificar relaciones entre modelos
- Depurar problemas con queries

 **Actividad práctica:** Abre el shell y crea 3 bicicletas diferentes, luego lista todas con `.all()` y filtra por tipo con `.filter(tipo="mtb")`.

⚙️ Verificación en la Base de Datos

Es importante verificar que Django realmente está creando y manipulando los datos en la base de datos. Puedes acceder directamente al shell de la base de datos:

```
python manage.py dbshell
```

Una vez dentro, puedes ejecutar comandos SQL nativos para inspeccionar las tablas:

Comandos útiles en SQLite

Listar todas las tablas

```
.tables
```

Ver estructura de una tabla

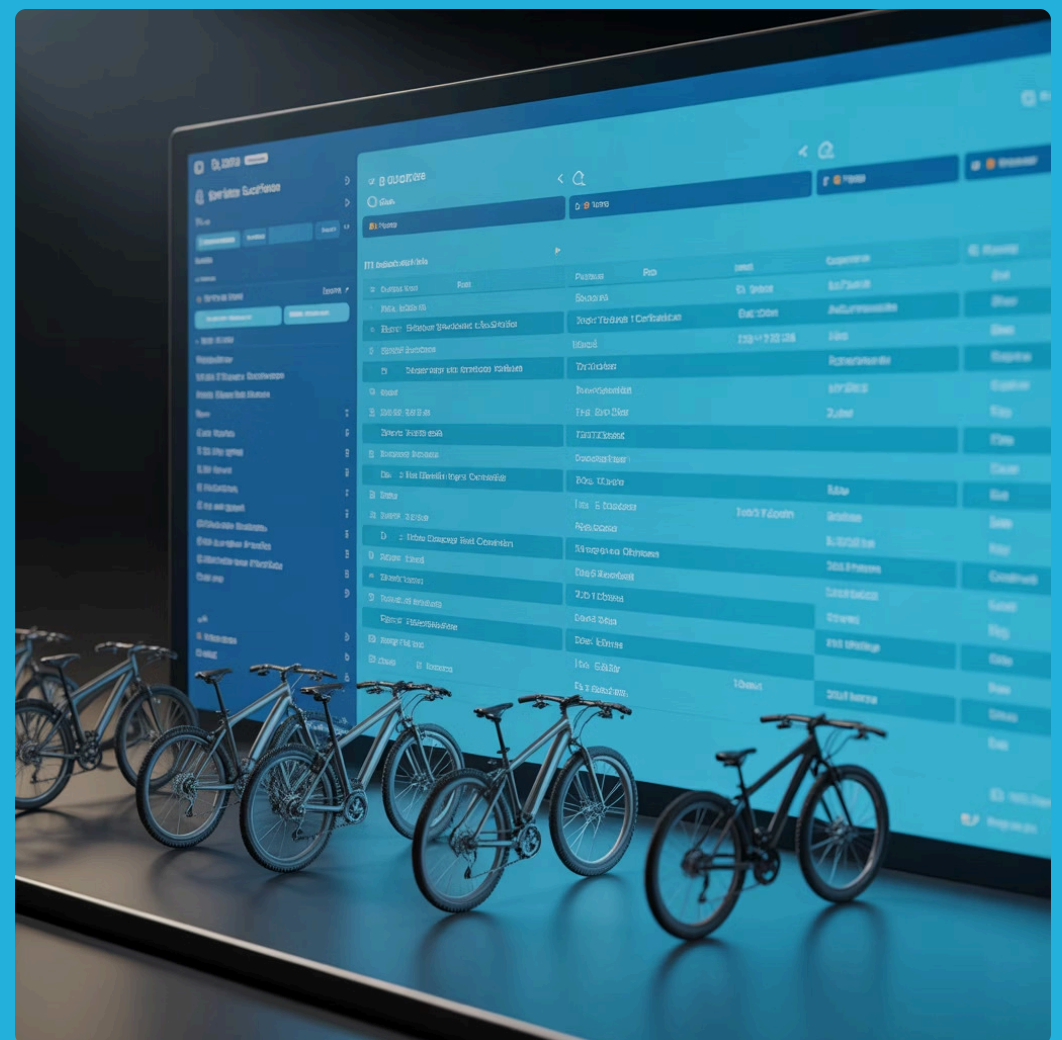
```
.schema bicicletas_bicicleta
```

Consultar datos

```
SELECT * FROM bicicletas_bicicleta;
```

Contar registros

```
SELECT COUNT(*) FROM bicicletas_bicicleta;
```



💡 **Observación importante:** Verás que Django creó automáticamente campos adicionales como **id** (clave primaria) que no definiste explícitamente en tu modelo.

🔍 **Ejercicio de verificación:** Crea una bicicleta usando el ORM, luego verifica con dbshell que el registro realmente existe en la tabla de la base de datos.



Integración Django–Base de Datos: Flujo Completo

Ahora que hemos visto todas las piezas, entendamos cómo se conectan en el flujo de trabajo completo de desarrollo con Django:

1. Definir Modelos

Creamos clases Python en `models.py` que representan nuestras entidades de negocio

4. Usar ORM para CRUD

Manipulamos datos usando métodos Python elegantes sin tocar SQL



2. Ejecutar Migraciones

Usamos `makemigrations` y `migrate` para sincronizar con la base de datos

3. Crear Tablas

Django genera automáticamente las tablas SQL con la estructura correcta

🤔 Pregunta reflexiva para debate: ¿Cómo simplifica el ORM la vida del desarrollador web? ¿Qué ventajas tiene sobre escribir SQL manualmente? ¿Hay desventajas?

Ejercicio Guiado en Equipo

Es momento de aplicar todo lo aprendido. Trabajarán en equipos de 2-3 personas para completar el siguiente desafío:

01

Crear el modelo Cliente

En tu aplicación, define un modelo `Cliente` con los campos: `nombre` (`CharField`), `email` (`EmailField`), y `telefono` (`CharField`)

02

Ejecutar las migraciones

Usa `makemigrations` y `migrate` para crear la tabla en la base de datos

03



Insertar tres registros

Usando el shell de Django, crea tres clientes con datos diferentes y realistas

04

Filtrar por nombre

Escribe una consulta ORM que filtre y muestre solo los clientes cuyo nombre comience con la letra "A"

  **Objetivo pedagógico:** Practicar la definición de modelos, migraciones y consultas ORM en un contexto realista. Tiempo estimado: 20-25 minutos.

 **Pista:** Para filtrar nombres que empiecen con "A", usa `Cliente.objects.filter(nombre__startswith="A")`



Buenas Prácticas en Django ORM

Para desarrollar aplicaciones Django profesionales y mantenibles, sigue estas recomendaciones esenciales:

Nombres claros y consistentes

Usa nombres descriptivos en español o inglés de forma consistente. Ejemplo: `FechaCreacion` o `created_at`, pero no mezcles idiomas

Migraciones en control de versiones

Siempre incluye los archivos de migración en Git. Son parte esencial del código y documentan la evolución del esquema de base de datos

No edites la base directamente

Nunca modifiques manualmente la estructura de la base de datos. Siempre crea migraciones para que Django mantenga todo sincronizado

Usa el ORM siempre que sea posible

Recorre a SQL manual solo cuando realmente lo necesites. El ORM ofrece seguridad, portabilidad y mantenibilidad

Prueba en el shell primero

Antes de automatizar consultas complejas, pruébalas en el shell interactivo para verificar que funcionan correctamente

Cierre y Conclusiones

¡Felicidades! Has completado exitosamente el módulo de integración de Django con bases de datos. Hoy adquiriste competencias fundamentales:

☀️ ORM de Django

Comprendes qué es el ORM, cómo funciona y por qué es una herramienta poderosa para el desarrollo web moderno

☀️ Motores Soportados

Conoces las bases de datos compatibles (SQLite, PostgreSQL, MySQL, Oracle) y sus casos de uso

☀️ Configuración y Migraciones

Sabes configurar la conexión en settings.py y manejar el sistema de migraciones de Django

☀️ Operaciones CRUD y SQL

Dominas las operaciones básicas con el ORM y conoces cómo ejecutar SQL manual cuando es necesario

Desafío Final

Para consolidar tu aprendizaje, te propongo el siguiente reto: Conecta tu proyecto Django a MySQL o PostgreSQL (en lugar de SQLite) y migra todos tus modelos para verlos funcionar en un motor de base de datos real de producción.

Este ejercicio te dará experiencia práctica con configuraciones más cercanas a entornos profesionales y te preparará para proyectos de mayor escala.

¡Éxito en tu camino como desarrollador Django! 🎓