Modelos y migraciones

Antonio Espín Herranz

Introducción

- Esta capa del **MVT** se encarga de interactuar con la BD de una forma homogénea.

- Django coloca una capa de abstracción sobre la BD, de tal forma que podamos trabajar con un modelo de objetos ocultándonos el código SQL y las conexiones con la BD → **ORM** (Object Relational Mapping), mapea el modelo relacional en un modelo de objetos

- Internamente se encarga de generar el código necesario para interactuar con una BD u otra:
  - Las más habituales: SQLite3, MySQL, Oracle, PostGreSQL

Introducción

- Los parámetros de la conexión se indican en el fichero settings.py (del proyecto).
- Por defecto, al generar el proyecto, añade la configuración para SQLite3.
- Incluso crea la BD, en el caso de otros tipos de BD tenemos que crear nosotros la BD.
- Pero a partir de este punto se trabaja de una forma homogénea.
- Y cambiar de BD sería tan sencillo como modificar los datos de la conexión en el fichero de settings.

Configuración de la BD

- **En settings.py**

```
#Esta configuración es para SQLite3
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR /  'db.sqlite3',
    }
}
```

- El nombre del fichero es: db.sqlite3
- Se creará automáticamente dentro de la carpeta del proyecto.

Ejemplo para MySQL

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': ‘nombre_bd',
        'USER': ‘usuario_bd',
        'PASSWORD': ‘password_bd',
        'HOST': 'localhost',
        'DATABASE_PORT': '3306',
    }
}
```

- En este caso la BD la tenemos que crear.
- Tenemos que tener instalado el servidor de MySQL
- Y además necesitaremos tener instalado el paquete de mysql para python: MySQLDB

Ejemplo para MySQL 8

- **# Para MySQL 8**

```
DATABASES = {
    'default': {
    'ENGINE': 'mysql.connector.django',
    'NAME’: ‘base de datos',
    'USER’: ‘usuario',
    'PASSWORD’: password',
    'HOST': 'localhost',
    'DATABASE_PORT': '3306',
}
```

- Instalar el conector de MySQL 8 para Python!

Configuración de la BD

- Adaptadores:

Configuración	Base de datos	Adaptador requerido
django.db.backends.postgresql_psycopg2	PostgreSQL	Psycopg version 2.x, http://www.djangoproject.com/r/python-pgsql/.
django.db.backends.mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysql/.
django.db.backends.sqlite3	SQLite	No necesita adaptador
django.db.backends.oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/.

Configuración de la BD

- Django trabaja con lo que se llama un **ORM** (Object Relational Mapping), mapea objetos del modelo en tablas de la BD y viceversa.

- Con Django no tenemos que crear las tablas de la BD, definimos nuestro modelo con objetos, donde tenemos que indicar las relaciones entre los datos, el tipo y poco más.
  - Las relaciones serán: ***muchos a muchos y uno a muchos***

- Django nos proporciona la clase **models.Model** de la cual tenemos que heredar.

Probar la configuración

- Una vez se han indicado los parámetros de la BD y la hemos creado (en caso de que no sea SQLite3) teclear en el comando:
  - ***python manage.py shell***
  - Desde la carpeta del proyecto (***que sea visible manage.py***)
  - Abre una consola y probamos a crear un cursor si la BD está bien configurada no dará ningún problema.

Probar la configuración

```
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from django.db import connection

In [2]: cursor = connection.cursor()

In [3]: _
```

- No tiene que salir ningún tipo de mensaje, con esto confirmamos que NO hay error!

- Si nos permite crear un cursor es que conecta correctamente con la base de datos.

Posibles errores

Mensaje de error	Solución
You haven't set the ENGINE setting yet.	Configura la variable ENGINE con otra cosa que no sea un string vacío. Observa los valores de la tabla 5-1.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Ejecuta el comando python manage.py shell en vez de python.
Error loading _____ module: No module named _____.	No tienes instalado el módulo apropiado para la base de datos especificada (por ej. psycopg o MySQLdb).(Es tu responsabilidad instalarlos)

Posibles errores II

Mensaje de error	Solución
_____ isn't an available database backend.	Configura la variable ENGINE con un motor válido descrito previamente. ¿Habrás cometido un error de tipeo?
database _____ does not exist	Cambia la variable NAME para que *apunte* a una base de datos existente, o ejecuta la sentencia CREATE DATABASE apropiada para crearla.
role _____ does not exist	Cambia la variable USER para que *apunte* a un usuario que exista, o crea el usuario en tu base de datos.
could not connect to server	Asegúrate de que HOST y PORT esta configurados correctamente y que el servidor esté corriendo.

Definir modelos

- Una vez se encuentra creada la BD y la conexión funciona.
- Partimos de un proyecto y una aplicación ya creada.
- Los modelos se ubican en el fichero: **models.py**
  - **Este fichero se ubica dentro de la aplicación**
- Se definen con clases que heredan de: **models.Model**
- Dentro de cada clase se van definiendo los campos y el tipo de estos. Además, se pueden indicar una serie de parámetros, por ejemplo:
  - la longitud del campo,
  - si se permite campos vacíos.

Ejemplo

- class Editor(models.Model):
  - nombre = models.CharField(max_length=30)
  - domicilio = models.CharField(max_length=50)

  - Si en los campos, queremos indicar que se pueden dejar en blanco, utilizar (blank=True),
  - En caso de que el campo sea numérico o fecha (blank=True, null=True)

Crear Tablas

- Si hemos definido un modelo en el fichero: models.py
- Ejecutaremos el comando:
  - **python managed.py makemigrations**
    - Para crear las migraciones
  - Después para confirmarlas:
    - **python managed.py migrate**
  - Comprobar en la base de datos las tablas creadas

Tipos de campos

- models.URLField():
  - una dirección web
- models.EmailField()

A partir de estas definiciones, Django
Es capaz de generar la tabla de la BD.
Al modelo NO es necesario añadir una
Clave primaria, Django lo hace de forma
Automática y la llama: id

- **Texto:**
  - models.CharField(): para campos de texto.
  - models.TextField() : para campos de tipo comentario.
- **Numéricos**:
  - models.**AutoField**(): se utilizan para indicar claves que django incrementa de forma secuencial.
    - ***OJO Django siempre agrega este campo por defecto, si lo añadimos nos dará un error al hacer la migración. Cada tabla solo puede tener un campo de estos.***
  - models.**FloatField**()
  - models.**IntegerField**()
- models.ImageField(upload_to="nombre_carpeta")

Tipos de campos

- **Tiempo**:
  - models.DateField()
  - models.TimeField()
  - models.DateTimeField()

- models.BooleanField()

- Para **subir ficheros**:
  - models.FileField(upload_to='uploads')
  - La ruta será: MEDIA_ROOT/uploads

- Mas información: https://docs.djangoproject.com/es/4.2/ref/models/fields/

Definir modelos

- El modelo **en cualquier momento se puede modificar**, y luego habrá que utilizar **manage.py** para actualizarlo.

- Tener en cuenta que con poco código Django nos genera una **consola de administración** para poder realizar tareas de mantenimiento **sobre estos datos**, como son altas, bajas y modificaciones, buscar, etc.

Comandos relacionados

- ***Antes de lanzar ningún comando*** para la comprobación o generación del modelo ***hay que instalar la aplicación***:

- En el fichero **settings.py** del proyecto, localizar la variable:

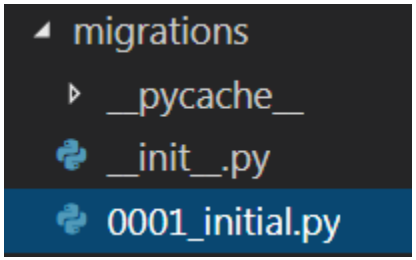
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'mi_aplicacion',
]
```

Comandos relacionados II

- Desde la carpeta del proyecto:
- ***python manage.py check biblioteca***
  - (***el nombre de la aplicación***)
  - Chequea posibles errores en el modelo. **No altera la BD.**
    - **Si no encuentra errores lanza el mensaje:**
    - *System check identified no issues (0 silenced)*

- ***python manage.py makemigrations***
  - ***Antes de lanzar este comando se habrá definido algún modelo en models.py***
  - Guarda las migraciones en la carpeta migraciones.
  - Son archivos de control para que django pueda sincronizar la estructura de la BD. **NO modifica la BD.**

```
Migrations for 'biblioteca':
  biblioteca\migrations\0001_initial.py
    - Create model Autor
    - Create model Editor
    - Create model Libro
```



Las migraciones

- **Las migraciones** son la forma en que Django se encarga de **guardar los cambios que realizamos a los modelos** (Agregando un campo, una tabla o borrando un modelo... etc.) en el esquema de la base de datos.

- Están diseñadas para funcionar en su mayor parte de forma automática, utilizan una versión de control para almacenar los cambios realizados a los modelos y son guardadas en un archivo.

- **makemigrations: es responsable de crear nuevas migraciones basadas en los** cambios aplicados a nuestros modelos.

- **migrate: responsable de aplicar las migraciones y los cambios al esquema de** la base de datos.

Comandos relacionados III

- ***python manage.py sqlmigrate biblioteca 0001***
  - NO altera la BD, genera el SQL necesario para generar las tablas de la BD.
  - Convenciones:
    - Añade una columna id a cada tabla como clave primaria.
    - El nombre de cada tabla se genera con el nombre de la aplicación + el nombre de la clase del modelo.
    - Este código generado es exclusivo de la BD elegida.
    - A las claves foráneas se le añade "_id" al nombre del campo.

- ***python manage.py migrate***
  - Este comando es el que guarda los cambios en la BD.

Comandos relacionados IV

- Al lanzar el comando **migrate**, se produce la siguiente salida:

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

En resumen

- De los 4 comandos indicados sólo 2 son necesarios para modificar la estructura de la BD.

- **Para trabajar**:
  - Se **escriben** en el fichero **models.py** las clases del modelo, o se modifican
  - Se crean las migraciones:
    - **python manage.py makemigrations**
  - Se ejecutan los cambios en la BD.
    - **python manage.py migrate**
    - **Y los cambios se hacen permanentes.**
  - **Así sucesivamente, realizamos cambios en el modelo y después lanzamos los dos comandos.**

Especificar relaciones

- Relación de **1 a Muchos**, clave foránea:

```
class Editor(models.Model):
    pass

class Libro(…):
    editor = models.ForeignKey(Editor, on_delete=models.PROTECT)
```

- Al indicar la clave foránea tenemos que **indicar el nombre de la clase con la que se relaciona**. Lo que se plasma en SQL con la clave foránea aquí se indica con la **entidad / clase**.

- También se indica que ocurre cuando se produce un borrado en este caso del editor con los libros relacionados.

Especificar relaciones

- Al borrar se puede indicar:
  - **PROTECT**: No permite el borrado si hay registros relacionados. Es una buena opción. Se lanza un error si se intenta borrar.
  - **CASCADE**: Borrado en cascada, es peligroso. Borra registros relacionados.
  - **SET_NULL**: Los registros relacionados se ponen a NULL.
  - **SET_DEFAULT**: Se establecen a un valor por defecto.
  - **SET(valor)**: Al valor indicado en la función set().
  - **DO_NOTHING**: No hace nada.

- Esta opción a elegir dependerá del contexto de la BD.

Especificar relaciones

- Relación de Muchos a Muchos:
- autores = models.**ManyToManyField**(Autor)
- Se especifica indicando la entidad con la que se relaciona.
- En este caso se crea una nueva tabla a nivel de BD y arrastra las dos claves primarias:

Relaciones muchos a muchos con atributos

- En algunas situaciones en la tabla que se obtiene de la relación de muchos a muchos, es necesario especificar atributos.

- Por ejemplo, la fecha de venta de un libro. En una relación entre los clientes y los libros.
  - 1 Libro tiene N ventas
  - 1 Cliente tiene N ventas
  - Pero la fecha de la venta o la cantidad de libros será un atributo de la relación.

Otro ejemplo

```python
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

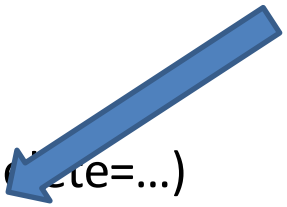
Notas

- Hay veces que necesitamos referirnos a una clase que no está previamente declarada:

```
class Libro(models.Model):
    #datos del libro

class LibrosCliente(models.Model):
    libro = models.ForeighKey(Libro, on_delete=…)
    cliente = models.ForeighKey("Cliente", on_delete=…)
    fecha_venta = models.DateField()

class Cliente(models.Model):
    # datos del cliente
    # Si tenemos un campo many to many pero hay
    # que asignar el tipo LibrosCliente.
    libros = models.ManyToManyField(Libro, thought=LibrosCliente)
```



Acceso básico a los datos

- Django proporciona un API de alto nivel para trabajar con el modelo.
- Se puede lanzar el comando:

  python manage.py shell

- Desde la consola importar el modelo:
  - ***from biblioteca.models import Editor***
  - Se importan las entidades del modelo de la app.
  - A partir de aquí se **instancia un objeto** y se llama al método **save**()
  - ***OJO, cuando instanciamos, dar nombre a los parámetros, no utilizar parámetros posicionales.***

Acceso básico a los datos

```
In [7]: p1=Editor(nombre='Anaya SL',apellidos='',email='webmaster@anaya.es')

In [8]: p1
Out[8]: <Editor: Editor object (None)>

In [9]: p1.save()

In [10]: _
```

- Utilizar parámetros con nombre y no es necesario enviar el id, **Django** lo genera automáticamente.
- Una vez grabado, **p1.id** ya está disponible.

Operaciones disponibles

- Dentro de cada clase del modelo: por ejemplo, Editor, Autor, Libro, etc. Se dispone de una propiedad **objects** que le podemos aplicar métodos para filtrar, contar, sumar columnas, calcular la media.

- Mediante **métodos del ORM** se pueden realizar las **consultas similares** a las de **SQL**.

Propiedad objects

- La propiedad objects nos permite realizar operaciones de este estilo:
  - numEmpresas = Empresa.**objects**.count()
  - Siendo **Empresa** un **objeto del modelo**.

- Si estamos **utilizando un IDE y** la propiedad **objects** no está disponible porque Django la agrega automáticamente a la clase.
  - Si no queremos tener el error en la vista, agregar a cada clase del modelo la siguiente declaración:
    - **objects = models.Manager()**

Propiedad objects

- Otra posible solución recopilada de stackOverflow:
  - Instalar: ***NO ESTA PROBADA***
    - **pip install pylint-django**
  - Después en Visual Studio Code:
    - File > Preferences > Settings

```
{"python.linting.pylintArgs": [
        "--load-plugins=pylint_django"
],}
```

  - https://stackoverflow.com/questions/45135263/class-has-no-objects-member

Operaciones disponibles

- **objeto.save()**: grabar el objeto, → insert into
- Se el objeto ya existe, se lanza → update
  - Se pueden modificar propiedades del objeto:
    - objeto.propiedad = valor

- Recuperar una **lista de objetos**:
  - L = **Editor.objects.all()**
    - Añadir a las clases del modelo, el método

    def __str__(self):
    return "…"
    - Para que se vean correctamente los datos de los objetos
    - Como lo que devuelve all() es una lista podemos utilizar **slicing**.

Operaciones disponibles

- **Filtrado de datos**:
  - Editor.objects.**filter**(nombre='xxx')
  - Se pueden establecer condiciones.
  - Es como si colocáramos la cláusula **where** en la consulta.
  - Si se indican dos filtros es equivalente a un **AND**.
    - Editor.objects.filter(nombre='xxx',apellidos='yyy')
  - Para la **OR**: (utilizar **|**)
    - Book.objects.filter(price__gt=100) **|** Book.objects.filter(price__lt=24)
- Se puede forzar a que utilice **Like** en la consulta:
  - Select * from editores where nombre **like** 'A%'
  - L = Editor.objects.filter(nombre**__contains**='An')

Operaciones disponibles

- **Recuperar un objeto individual**:
  - obj = Editor.objects.**get**(id=1)
  - Si ponemos una condición que devuelva más de 1 objeto, se producirá una excepción.
  - De la misma forma si no se recupera un objeto, también lanzará una excepción: **DoesNotExist**

- **Ordenar datos:**
  - L = Editor.objects.**order_by**('nombre')
    - Produce ordenación ASC.
    - Para **DESC**, signo menos delante del nombre: **-nombre**
    - Se pueden utilizar varias columnas, separadas por comas.

Operaciones disponibles

- También es posible establecer un criterio de orden en la declaración del modelo:
- Tenemos que añadir una clase Meta anidada.

```
class Editor(models.Model):
    …
    …
    class Meta:
        ordering = ['nombre']
```

Cuando hacemos estos cambios
En el modelo tenemos que hacer:
python managed.py migrate

- Dentro de la clase Meta también se puede especificar el nombre plural para esa entidad: (se utiliza luego en el panel de Administración).

```
class Meta:
    …
    verbose_name_plural = 'Editores'
    # Se utilizará sobre todo en el panel de administración.
```

Operaciones disponibles

- Se pueden **encadenar búsquedas**.
  - Editor.objects.filter(pais="U.S.A.").order_by("nombre")

- **Actualizar varios campos** a la vez:
  - Carga 1 objeto y lo actualiza:
    - Editor.objects.filter(id=1).**update**(nombre='Apress Publishing')
  - Carga varios objetos y los actualiza:
    - Editor.objects.all().**update**(ciudad='USA')

Operaciones disponibles

- Encadenamientos con operaciones de borrado:
  - Carga un objeto y luego lo borra:
    - p = Editor.objects.get(nombre="AddisonWesley")
    - **p.delete()**
  - Carga varios objetos y luego los borra:
    - Con filtrado previo:
      - Editor.objects.filter(ciudad='USA').**delete()**
    - Borrado de TODOS:
      - Editor.objects.all().**delete()**

Funciones de Agregado

- Para **contar** registros: (contar filas en SQL)
  - Autor.objects.count()
  - Autor.objects.all().count()

- **all**(): Devuelve un **queryset** con todos los registros. Si queremos ver los datos de estos objectos implementar el método **__str__()** en la clase del modelo.

Funciones de Agregado

- **Contar** las filas que **cumplen una condición**:
- Cuantos libros ha publicado un editor concreto:
  - Book.objects.filter(publisher__name='…).count()

  - **Filter** devuelve los objetos que cumplen la condición y después se cuentan.
  - Si los libros tienen una **foreignkey** que hace referencia a los editores, podemos referenciar a la propiedad name dentro de la propiedad: publisher.

Funciones de Agregado

- Las operaciones sobre todas las filas:
  - Media: **Avg**
  - Máximo: **Max**
  - Mínimo: **Min**
  - Suma: **Sum**

- Lo primero importar del paquete **django.db.models**

- Ejemplo:

  from django.db.models import Avg,Min,Max

Funciones de Agregado

- Para calcular el precio medio de los libros:
  Book.objects.all().aggregate(media_id=Avg('price'))

- En este caso está creando un alias llamado media_id para almacenar el valor. Viene dentro de un diccionario: {'media_id':52.7}

- Se puede ejecutar sin el alias:
  Book.objects.all().aggregate(Avg('price'))

- Y lo genera django, en este caso será:
  nombreColumna__avg
  - Con un formato similar para cada una de las operaciones.

Funciones de Agregado

- Podemos calcular más de una operación en la misma consulta, separando por comas (pudiendo poner alias o no).
  - **Book.objects.all().aggregate(Avg('price'), Sum('price'), …)**
- Sin **all**() también funciona.

- **aggregate**: Está operando para TODAS las filas.

Funciones de Agregado

- Cuando la función de agregado se aplica a grupos de filas y contamos, sumamos, etc. (lo que sería el **group by en SQL**), se aplica **annotate**()

- Por ejemplo, número de libros por cada editor.

  from django.db.models import Count

  Publisher.objects.**annotate**(num_books=Count('book'))

  – num_books: es un alias.

  – Devuelve un QuerySet.

Funciones de Agregado

- Los resultados se pueden encadenar con una ordenación y tomar los n primeros.
  - Utilizar **slicing**.

- Ordenando con el alias de forma decreciente:
  - Publisher.objects.annotate(num_books=Count('book')).order_by('**-**num_books')**[:5]**

Funciones de Agregado

- Django proporciona **dos formas de crear agregados**:

- Dependiendo si queremos aplicar la función de agregado **a todo el QuerySet** o **cada valor del QuerySet**.

- Esto se traduce a SQL:
  - Select sum(precio), avg(precio) from libros
    - Esto se aplica a todo el conjunto → **aggregate**()
  - Select categoria, sum(precio), avg(precio) from libros **group by** categoria
    - Por cada categoría se aplica la operación → **annotate**()

Funciones de Agregado

- **aggregate**():
  - Cláusula terminal para un QuerySet que devuelve un diccionario de pares clave / valor.
  - Si no se utiliza un alias, el nombre se genera automáticamente a partir del nombre del campo y la función de agregado → **price__avg**
  - Precio medio de todos los libros:
    - Book.objects.all().aggregate(Avg('price'))
    - Book.objects.aggregate(Avg('price'))

Funciones de Agregado

- **annotate**():
  - **NO** es cláusula terminal, la salida es un QuerySet puede ser modificado con filter, order_by, etc.
  - Se pueden utilizar alias y la sintaxis es la misma que aggregate().

```
>>> q = Book.objects.annotate(Count('authors'))
>>> q[0].authors__count

>>> q = Book.objects.annotate(num_authors = Count('authors'))
>>> q[0].num_authors
```

Funciones de Agregado

- **Join y funciones de agregado:**
- Por ejemplo:
  - En el modelo: Store tiene una **relación de muchos a muchos** con los libros.
    - Cada almacén tiene muchos libros y un libro puede estar en varios almacenes.

La definición:

```
class Book(models.Model):
    price =models.FloatField()
    …

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
```

Funciones de Agregado

- **Join y funciones de agregado:**
  - Se pueden aplicar operaciones de agregado sobre cada almacén → annotate
  - Y Python mediante cadenas separadas por _ _ nos permite navegar al campo de otra clase, siempre y cuando esté relacionada.

- Calcular los baremos de precios (min / max) por cada almacén:

```
Store.objects.annotate(min_price=Min('books__price'),
    max_price=Max('books__price'))
```

Funciones de Agregado

- **filter / exclude**
  - Para estos métodos y get se pueden utilizar múltiples condiciones.

    https://docs.djangoproject.com/en/4.2/ref/models/querysets/#range

  - Tener en cuenta que filter se puede añadir después de **annotate** → equivale a **Having** en el SQL (una **condición para el grupo**).
    - Book.objects.annotate(num_authors=Count('authors')).filter(num_authors__gt=1)

  - Los libros que tienen más de 1 autor.
    - having num_authors > 1

Funciones de Agregado

- **Orden de aplicación** cláusulas **filter** y **annotate**

- No son conmutativas.
  - Publisher.objects.annotate(num_books=Count('book')).filter(book__rating__gt=3.0)
  - Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))

  – El filtro delante de la anotación restringe los objetos antes de hacer la anotación.
  – Ambas devolverán los editores que tiene un libro con un ranking > 3.0.

  – La primera consulta proporcionará el número total de los libros publicados por la editorial.
  – Y en la segunda incluirá solamente los buenos libros en el contador de la anotación.

Funciones de Agregado

- **values**()
  - Normalmente cuando se aplica una anotación los resultados se generan para cada objeto.
  - La cláusula values se puede utilizar para restringir las columnas que forman parte del resultado.
    - Author.objects.values('name').annotate(average_rating =Avg('book__rating'))
    - En este caso, solo mostrará el nombre.

PRÁCTICA Nº 4: Tutorial_Biblio

Resumen de comandos

- **python manage.py check**
  - Chequea el modelo. NO actualiza la BD.
- **python manage.py sqlmigrate biblioteca 0001**
  - Genera el SQL asociado a una migración, es de consulta
  - El número irá cambiando, según vayamos generando migraciones.
- **python manage.py shell**
  - Abre una consola de administración para probar los modelos.
- **python manage.py makemigrations**
  - Crea una migración, NO actualiza la BD, pero es necesario lanzarlo cuando se modifica el modelo.
- **python manage.py migrate**
  - Actualiza la BD con la migración previamente generada.
  - También actualiza los objetos que representan el modelo cuando se añaden métodos a una clase pero que no implica una modificación en la Base de datos.

Resumen de comandos

- Se puede realizar la operación al revés para que django explore una base de datos y construya el modelo.

- python managed inspectdb
- python managed inspectdb > models.py

- Documentación:
  - https://docs.djangoproject.com/en/4.2/ref/django-admin/#django-admin-inspectdb