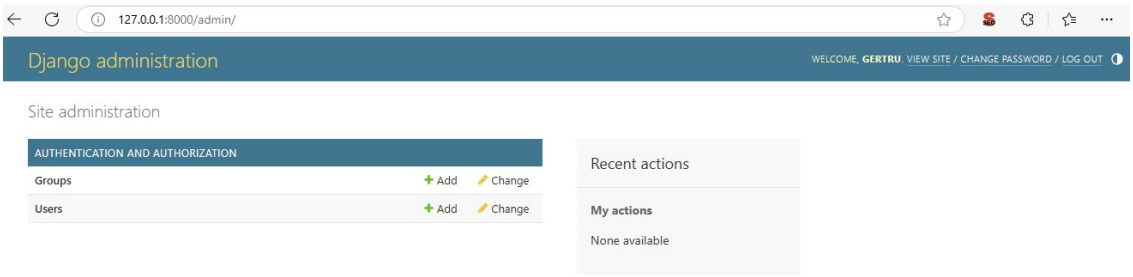
1.Registro de modelos

El punto de partida sería tener nuestra tabla Author en nuestra BD, creada en el archivo models.py. Tras ejecutar el comando `python manage.py makemigrations` se nos creó el fichero 0001Initial.py, pero aún no teníamos la tabla en la BD. Había que ejecutar el scripts y para ello utilizamos la instrucción `Python manage.py mígrate.`
Una vez realiza la secuencia de instrucciones la tabla ya estaba creada en le BD.

En admin.py vamos a registrar los modelos para que sea visible y utilizado dentro de la administración de Django. Aquí registraremos todos los modelos que queremos que sean recocidos por el sitio de administración de Django y nos permitan realizar un CRUD de forma automática.

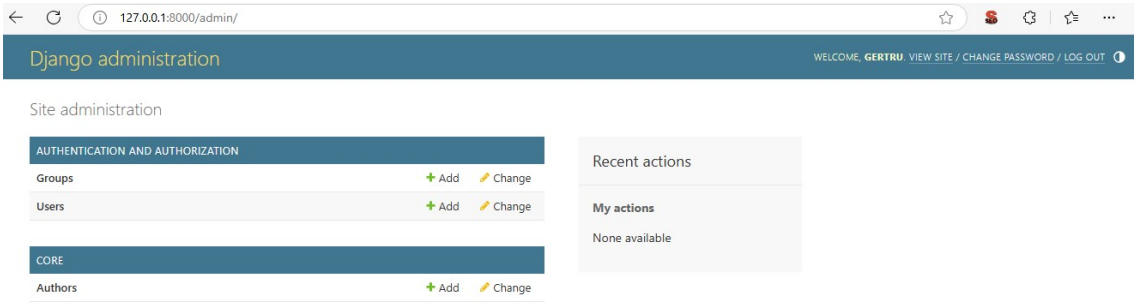
Si accedemos al sitio de administración actualmente tenemos lo siguiente:



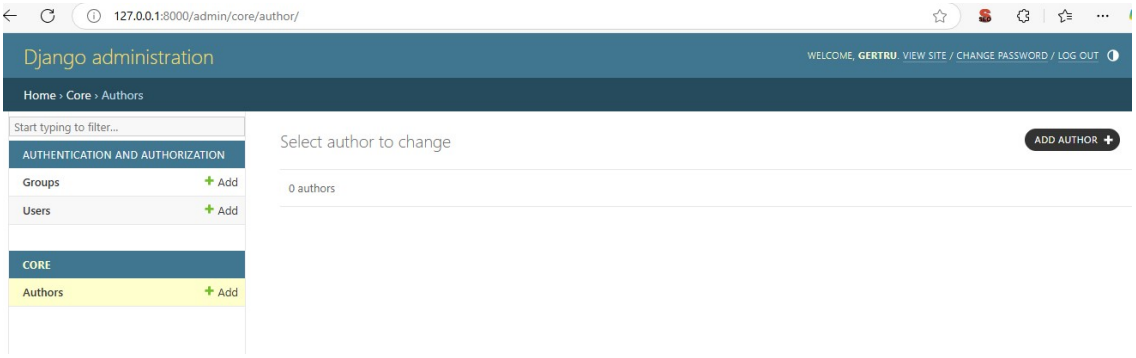
Para que podamos realizar los CRUDs en nuestra tabla/modelo, hay que registrarla en el fichero admin.py de nuestra aplicación, incluyendo el siguiente código.

```
from django.contrib import admin

#Añadimos esta linea para comenzar con el registro.
from core.models import Author

#Forma de registrar la tabla Author
admin.site.register(Author)
```

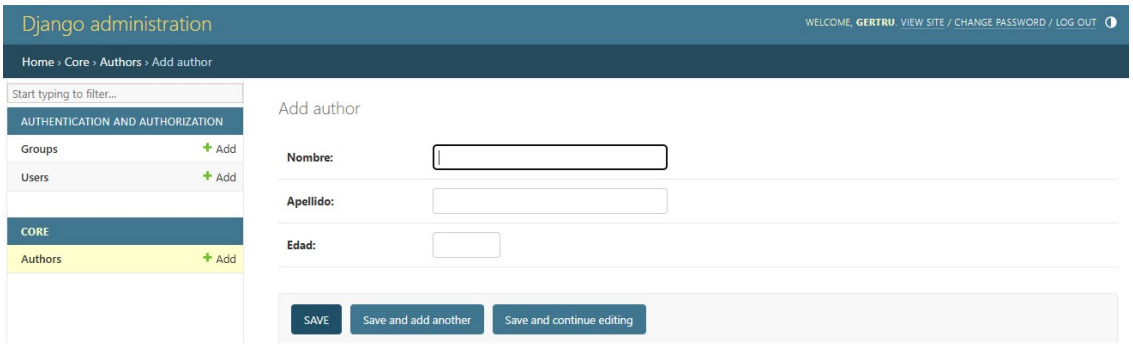
Tras incluir el código, hay que reiniciar el servidor y recargar la página de admin. Podemos observar que ya se ve la tabla bajo la aplicación CORE.



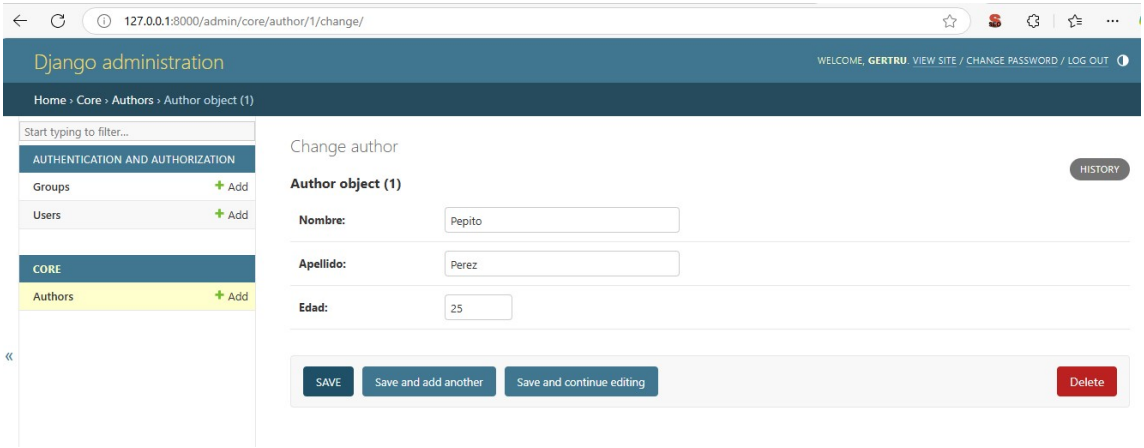
Pinchando en Authors, veo que no hay nada:

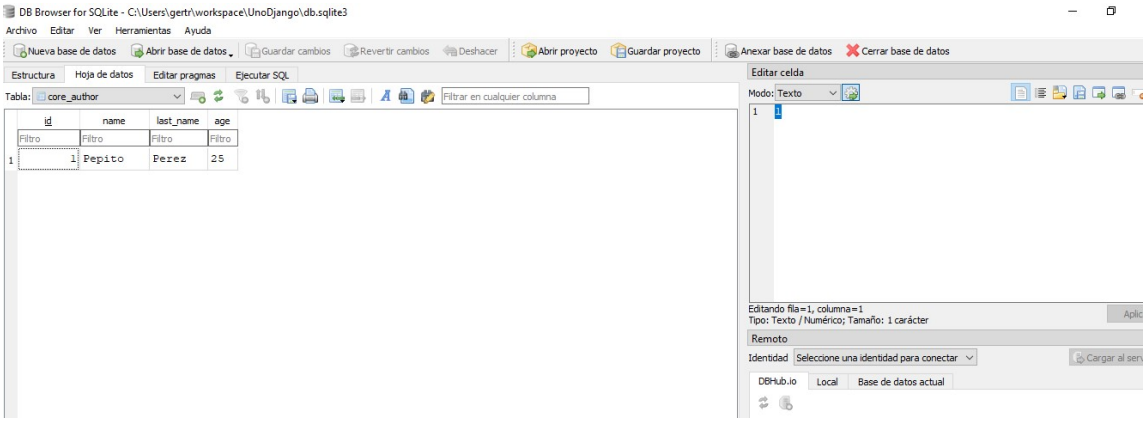


Pero tengo la opción de pinchar en ADD AUTHORS, permitiéndonos crear una autor introduciendo los campos que ya habíamos creado el fichero models.py de la aplicación.



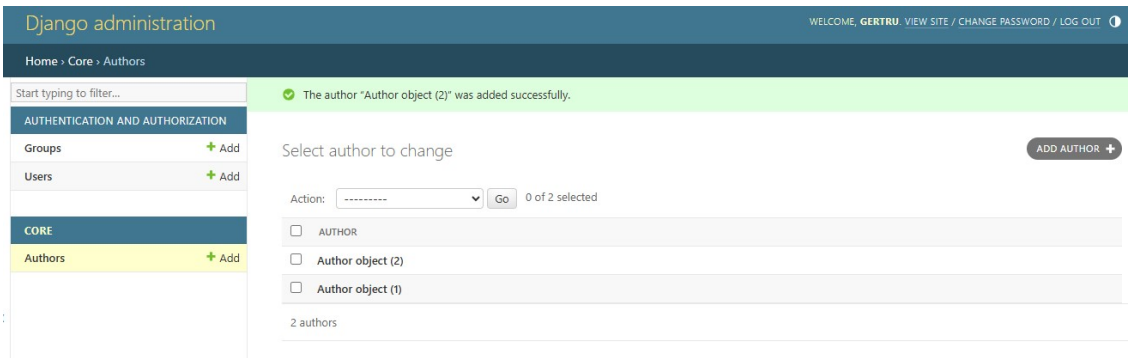
Si añadimos un autor y luego nos vamos a la BD abierta desde el BD browser SQLlite, pues nos aparecerá el nuevo autor.





Como podemos observar el interfaz utilizado para crear el nuevo autor, viene automáticamente con Django.

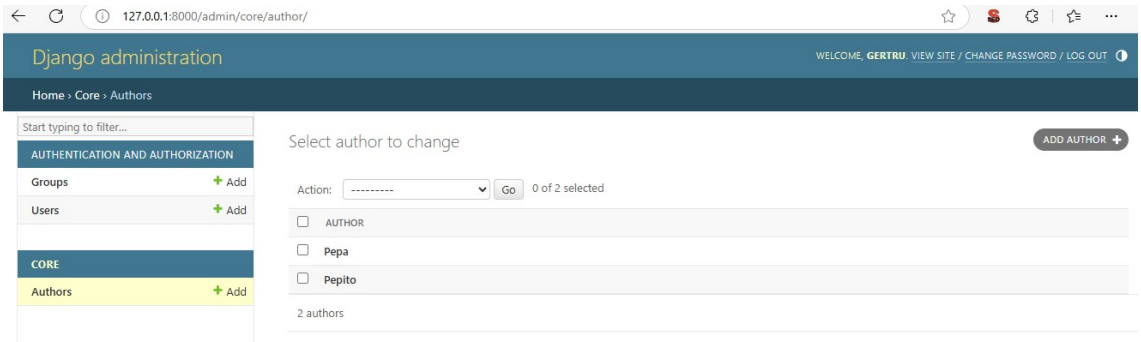
Aquí más abajo, vemos que por cada autor que introduzcamos nos crea un Author object con el ID de ese registro, pues al fin y al cabo es un objeto de la clase Author definida en models.py.



Si quisiéramos cambiar ese nombre por defecto que se nos crea, nos iríamos a models.py y añadimos una definición de __str__

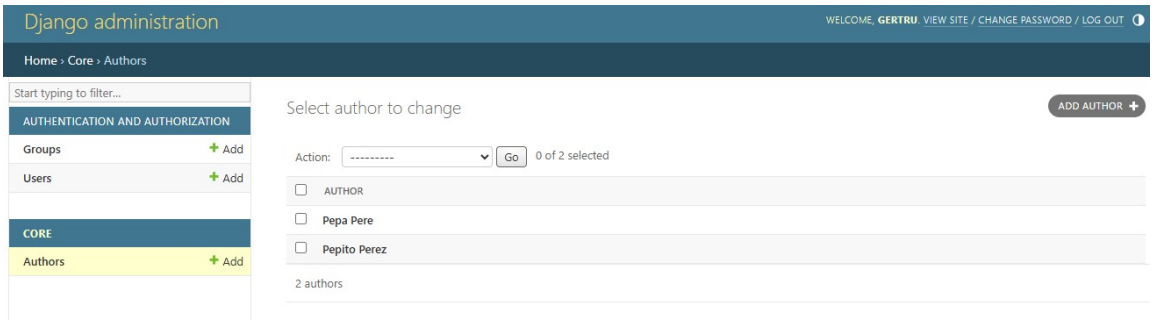
```
from django.db import models

# Create your models here.
# Al aplicar esta herencia, Django va sa saber que Author es una tabla en la BD
class Author (models.Model):
    name=models.CharField (verbose_name='Nombre', # etiqueta dentro de la tabla
    max_length= 100,
    default=''
    )
    last_name=models.CharField(verbose_name='Apellido',
    max_length=150,
    default='')
    age=models.PositiveSmallIntegerField (verbose_name='Edad',
    )

#Añadimos este código para que en el administrador al acceder a los registros que contiene la tabla aparezca no Author object ID, sino que nos de más información del registro que hay dentro.

#Con el método __str__, sobreescribimos la información por defecto.Indicamos ahí lo que queramos que nos retorne
#Ojo, está dentro de la clase
    def __str__(self):
      return self.name
```

El resultado después de recargar la página de admin sería :



También podríamos incluir esta definición, para que nos devuelva el nombre y apellido

```
def __str__(self):
      return f'{self.name} {self.last_name}'
```



Hemos combinado lo que tenemos en nuestro modelo (models.py) con lo que nos ofrece django a través del sitio de administración de Django o lo que es lo mismo hemos creado nuestro modelo y agregado al sitio de administración de Django.

2.Relaciones en modelos

Hasta ahora habíamos creado una tabla sencilla. Vamos a crear un nuevo modelo que se llame Book. Hay varias formas en Djando de indicar que tenemos una relación entre dos tablas.

Si añadimos el nuevo modelo Book dentro de models.py:

--Relación uno a uno:

```
   class Book (models.Model):

   title=models.CharField('Título del libro',max_length=255,unique=True)
#Unique=True , Django no va a permitir introducir dos libros con títulos iguales. Cuando cree este campo en la BD solo permitir valores diferentes.
   cod=models.CharField('Codigo',max_length=15,unique=True)
   author=models.OneToOneField(Author,on_delete=models.CASCADE)
#Cascade ,de esta forma indicamos que cuando se borre un autor en la Bd, también se borrará el libro.
author=models.OneToOneField(Author,on_delete=models.SET_NULL,null=True)
```

```
#De esta forma indicamos que en el campo author de aquellos libros en los
que el author se haya borrado, django pondrá el valor null. Para ello se le
indica que pude aceptar valor null a True.
```

--Relación uno a mucho. Un autor puede tener varios libros, pero libro sólo pertenece a un autor.

```
#Relación uno a muchos, igualmente el borrado puede ser en cascada o bien
set_null. Estas son las dos clásicas situaciones aunque Django contempla más
posibilidades.
   author=models.ForeignKey(Author, on_delete=models.CASCADE)
```

--Relación mucho a mucho, se crea una tercera tabla de forma automática.

```
#Relación de muchos a muchos.
   author=models.ManyToManyField(Author)
   #No se necesita poner on delete, por defecto soporta los campos nulos o
vacíos, no es necesario colocarlo.
```

Internamente cuando Django detecta este tipo de relaciones, normaliza y crea una tercera tabla, digamos que parte la relación en tres tablas, las dos implicadas más la que sale de la relación mucho a mucho.

Vamos a dejar la tabla Book de ejempo con esta última relación únicamente, pues da más juego a posteriori. Es decir, no usamos por ahora la de OneToOne o la de uno a muchos.

Una vez tenemos nuestra clase/tabla creada hay que agregarla al admin.py, quedando nuestro código en el fichero admin.py de esta manera:

```
from django.contrib import admin

#Añadimos esta linea para comenzar con el registro.
from core.models import Author,Book #Modelo añadido a posteriori.

#Forma de registrar la tabla Author
admin.site.register(Author)
#Registramos las tabla Book
admin.site.register(Book)
```

Una vez registrado en admin, ejecutamos la orden de migrar

Python manage.py makemigrations

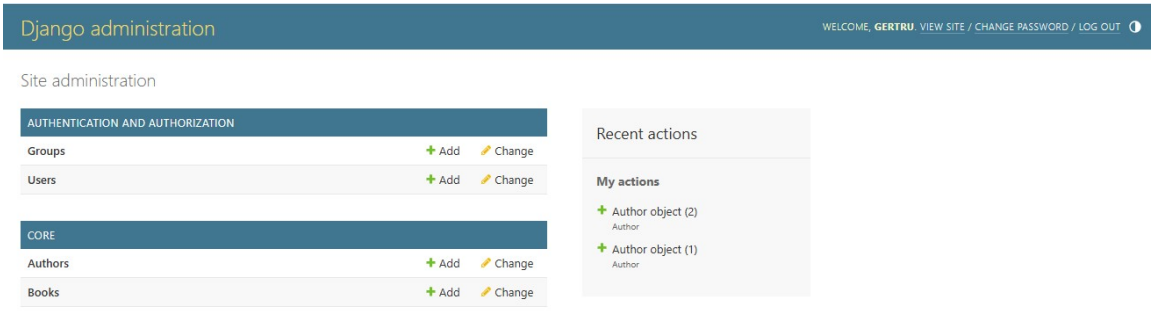
Se nos creará el nuevo fichero 0002_book.py en nuestra estructura:

```
PS C:\Users\gertr\workspace\unodjango> Python manage.py makemigrations
Migrations for 'core':
  core\migrations\0002_book.py
    - Create model Book
PS C:\Users\gertr\workspace\unodjango>
```

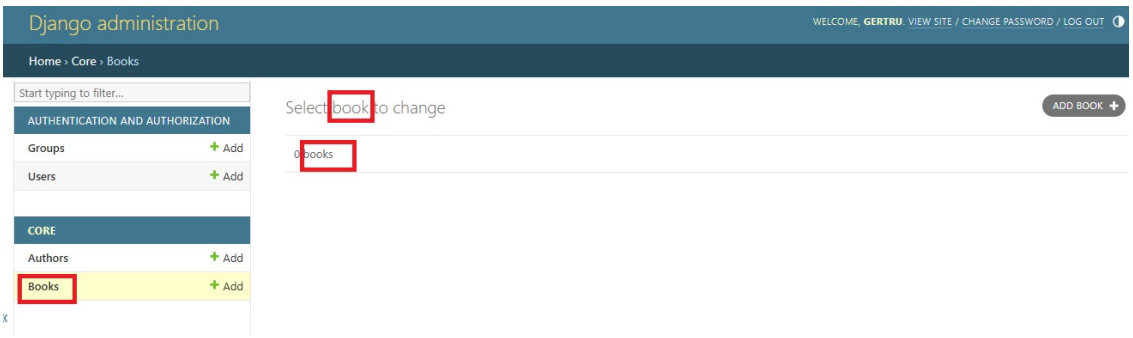
Ahora hay que aplicar la migración a la BD:

Python manage.py migrate

El servidor debe estar parado para poder ejecutar éstas instrucciones. Posteriormente si nos vamos al admin ya aparecerá nuestra tabla creada.



Si quisiéramos modificar estos valores marcados en rojo y que salen por defecto:



Hay que añadir la clase Meta, para configur éstos detalles . Esta clase Meta debe estar dentro de la clase Book.

Nos va a indicar con el verbose name , como se va a llamar nuestra tabla Book dentro del sitio de administración de Django. Así mismo cuando se llame en plural pues lo sustituirá por lo que pongamos en plural.

```
class Meta:
        verbose_name='Libro'
        verbose_name_plural='Libros'
    def __str__(self):
        return self.title
```

Como puede comprobarse, crear un libro, asignarle un usuario y la gestión (CRUDs) , es bastante sencillo desde el administrador de Django.

3.Cómo realizar las consultas a mi modelo

Para realizar consultas en Django a los modelos que tenemos, utilizaremos el ORM que lleva incorporado Django.

Las consultas a BD, como todos sabemos se realizan en SQL, pero Django está escrito en python. Para realizar esta "traducción", es decir llevar Python a SQL, existen los ORM. ORM, no son más que traductores.

La secuencia sería como sigue:

Python→ORM→SQL→Ejecución en BD

Como hemos dicho Django tiene un ORM integrado. Lo que tendremos que aprender es a escribir las sentencias de python que tomará este ORM.

Un ejemplo clásico sería obtener todos los registros de una tabla:

Vamos a utilizar el atributo objects contenido en la clase Models, de la cual heredan todas nuestras tablas/modelos.

```
#Devolver todos los registros de la tabla Author
Author.objects.all()
```

Para poder comprobar que esto que he escrito me devuelve efectivamente todos los registros de la tabla, vamos a ejecutar la siguiente instrucción que nos va proporcionar una consola interactiva.

Python manage.py shell

PS C:\Users\gertr\workspace\unodjango> python manage.py shell

>>> from core.models import Author

>>> Author.objects.all()

<QuerySet [<Author: Pepito Perez>, <Author: Pepa Pere>]>

>>>

QuerySet es el tipo de datos que devuelve la consulta.

Si queremos saber cómo es el SQL que el ORM ha traducido después de realizar la consulta de los registro de Author, en el mismo Shell que antes, podemos ejecutar crear una variable con el contenido del resultado de la consulta y luego imprimir ese contenido, así sabremos la consulta que el ORM ha traducido.

>>> resultadoconsulta=Author.objects.all()

>>> print(resultadoconsulta.query)

SELECT "core_author"."id", "core_author"."name", "core_author"."last_name", "core_author"."age" FROM "core_author"

>>>

Esta select es una select *

Si quisiéramos hacer la select pero con un filtro por ejemplo en el apellido de los Authores. Igualmente en el Shell ejecutamos el siguiente código introducido en nuestro models.py

```
#Devolver todos los registros de la tabla Author
Author.objects.all()

#SELECT "core_author"."id", "core_author"."name", "core_author"."last_name",
"core_author"."age" FROM "core_author"
#Si quisieramos hacer un select de aquellos autores que tengan apellido
'perez’ hay que realizar un filtro
Author.objects.filter(last_name='Perez')
```

>>> Author.objects.filter(last_name='Perez')

<QuerySet [<Author: Pepito Perez>]>

>>>

```
#Si quisiéramos que sólo traiga el primero de los registros.
Author.objects.filter(last_name='Perez').first()
```

>>> Author.objects.filter(last_name='Perez').first()

<Author: Pepito Perez>

>>>

Hay una pequeña diferencia entra una consulta y otra. En la primera, con el filter, el resultado sigue siendo un conjunto de datos, un queryset. En la segunda , nos va a traer sólo un registro, que va a ser de tipo Author, no es un queryset.

Si realizo esta consulta en el Shell:

>>> resul=Author.objects.filter(last_name='Perez').first()

>>> print(resul.query)

Traceback (most recent call last):

File "<console>", line 1, in <module>

AttributeError: 'Author' object has no attribute 'query'

>>>

Nos va a dar error, pues el resultado del first, no tiene el atributo query.

Existe una alternativa al first() sería la siguiente instrucción

```
#Esta instrucción sería más rápida pero si no hay ningún registro con el criterio buscado nos va a devolver error y hay que tenerlo en cuenta porque habría que capturar la excepción.
Author.objects.get(last_name='Perez')
```

4.Cómo realizar las inserciones en mi modelo

```
#Con la instrucción siguiente creamos un registro en BD.
Author.objects.create() #Así tal cual nos va a dar un error pues la edad no puede ser vacio, así lo hemos definido, sin default. Entonces habría que sustituirla por esta otra:
Author.objects.create(age=22)
```

Si ejecutamos esto en el Shell

>>> Author.objects.create(age=22)

<Author: >

>>>

Nos devuelve que Author, no tiene nombre, pues no se lo hemos metido.

Si hacemos una consulta en la BD como sigue:

>>> Author.objects.all()

<QuerySet [<Author: Pepito Perez>, <Author: Pepa Pere>, <Author: >]>

>>>

Nos devuelve los que había y el nuevo, al cual no le hemos metido nada en nombre y apellido. Me devuelve nombre y apellidos de esa forma porque le puse el __str__ dentro de la clase de la siguiente manera:

```
def __str__(self):
      return f'{self.name} {self.last_name}'
```

Si queremos que me devuelve también edad, modificamos ese def y el resultado sería:

```
 def __str__(self):
      return f'{self.name} {self.last_name} {self.age}'
```

Si queremos insertar directamente todos los datos del Author.

Author.objects.create(name='Juan',last_name='Perez',age='33)

Realizar la consulta a ver el resultado.

La idea del ORM, es no pensar en cómo hacer la consulta. Aún así el ORM de Django permite ejecutar sentencias Sql directamente.

Ejemplos de consultas:

Si quisiéramos consultar en Book, desde el Shell y tenemos la relación OneToOne creada(ojo, si hay que cambiar la relación, hay que volver a migrar los cambios a la BD):

>>> from core.models import Book

>>> Book.objects.all()

<QuerySet [<Book: 222 La gran princesa>]>

>>> Book.objects.create(title='Yo',cod='2')

<Book: 2 Yo>

```
>>> Book.objects.all()
<QuerySet [<Book: 222 La gran princesa>, <Book: 2 Yo>]>
>>> Book.objects.all().first()
<Book: 222 La gran princesa>
>>> res=Book.objects.all().first()
>>> res
<Book: 222 La gran princesa>
>>> res.title
'La gran princesa'
>>> res.cod
'222'
>>>
```

Con la instrucción siguiente podríamos borrar el libro.

```
book_to_delete = Book.objects.get(title="La gran princesa")
book_to_delete.delete()
```