Django-Modelos

Jorge Barón Abad

Modelos

Lo primero que vamos a hacer en una aplicación es definir las clases y tablas de la base de datos que vamos a usar en la aplicación.

En Django (y gran mayoría de frameworks) se utilizan unas clases que se llaman **Modelos (Models)** , que tienen una característica especial, estas clases están asociadas a una tabla de una base de datos.

Por lo tanto los atributos de las clases, hacen referencia a una columna de la tabla correspondiente en la base de datos.

Cada registro sería como una instancia(objeto) de una clase.

Modelos

En un resumen sencillo, por ejemplo si tenemos la clase Persona en python:

```
class Persona:

    nombre = “”
    edad = 0

    def __str__(self):
        return self.title
```

Dicha clase se convertirá en una tabla en la base de datos, que se llama Persona con las columnas nombre y edad, donde guardaremos las personas que acceden a la aplicación.

Nombre	Edad
Pepito	45
Juanito	30

Modelos

Ejemplo del tutorial Django Girls:

```
from django.conf import settings
from django.db import models
from django.utils import timezone


class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)
    published_date = models.DateTimeField(blank=True, null=True)

    def publish(self):
        self.published_date = timezone.now()
        self.save()

    def __str__(self):
        return self.title
```

Modelos

Explicamos este modelo de Django Girls:

- `class Post(models.Model):` -> Siempre deben heredar de la clase padre Model para indicar que es un modelo
- **models.ForeignKey** -> Indicamos que es una clave foránea que hace referencia a otro modelo
- **models.CharField(max_length=200)** -> Indicamos que es de tipo char con un máximo de 200 caracteres
- **models.TextField()** -> Indicamos que es de tipo Texto
- **models.DateTimeField(default=timezone.now)** -> Indicamos que es de tipo DateTime

Tipos Campos

Ya sabemos crear un Modelo sencillo, ahora vamos a ver que tipo de atributos podemos definir para crear las tablas correspondientes:

Para ello tenemos el siguiente enlace de la documentación de Django dónde aparecen todos los tipos que tenemos:

- https://docs.djangoproject.com/en/5.1/ref/models/fields/#field-types

Tipos Campos

Existen muchos tipos de campo, os indico los más comunes:

- **CharField**: Este campo se utiliza para almacenar cadenas de texto de longitud variable.
- **TextField**:Este campo se utiliza para almacenar cadenas de texto de longitud variable.
- **IntegerField**: Este campo se utiliza para almacenar números enteros.
- **FloatField**: Este campo se utiliza para almacenar números decimales.
- **BooleanField**: Este campo se utiliza para almacenar valores booleanos (True o False).
- **DateTimeField**: Este campo se utiliza para almacenar fechas y horas.
- **DateField**: Este campo se utiliza para almacenar fechas.
- **TimeField**: Este campo se utiliza para almacenar horas.

Tipos Campos: Parámetros

En Django, los tipos de campos se crean con funciones que se llaman a partir de la clase models. Por ejemplo:

```
models.CharField(max_length= 200)
```

En este caso a la función se le ha pasado un parámetro que significa el tamaño máximo que puede tener ese campo en la base de datos.

Pero existen más parámetros. Se incluye el enlace a la documentación, para que podáis acceder a todos:

https://docs.djangoproject.com/en/5.1/ref/models/fields/

Tipos Campos: Parámetros

A continuación os indico los más comunes:

- **null**: Indica si queremos que se guarde como nulo el valor si no se introduce nada. Por defecto: false.
- **blank**: Permitimos que el campo puede estar vacío
- **choices**: Permite elegir de una serie de valores.
- **db_column**: El nombre de la columna en la base de datos. Por defecto el del atributo
- **primary_key**: Para indicar que ese campo es la clave primaria. Si no se especifica ninguna clave primaria en los atributos, por defecto Django la crea, pero nos da la oportunidad de especificarlo nosotros.
- **unique**: Si el valor es único. Por defecto es falso

Procedimientos para crear Modelos

Una vez que conocemos un modo más avanzado crear modelos. Sería conveniente indicar cuál es el procedimiento para crear las clases, crear las migraciones y luego crear las tablas:

1. En el archivo **models.py** creamos los modelos
2. Creamos las migraciones con el comando : **python manage.py makemigrations**
3. Ejecutamos la migración para que se cree en la Base de Datos: **python manage.py migrate**

IMPORTANTE: Si hacemos cambios en el archivo **models.py**, hay que volver a realizar este procedimiento, para actualizar la base de datos.

Podemos eliminar los archivos de migraciones existentes. Dentro de la carpeta migrations eliminar todos los archivos que empiecen por 0000_. **NO BORRAR EL ARCHIVO __init__.py**

Si tenemos problemas con la aplicación a la hora de crear registros, debemos proceder a eliminar la base de datos. Eliminamos el archivo **db.sqlite3**, al hacer el migrate volverá a crearse.

Por último accedemos a la parte de administración para comprobar que se hayan creado las tablas.Acordaos de añadir los modelos al archivo **admin.py** y crear el **superusuario** para acceder a admin.

Ejemplos de Modelos

```python
from django.db import models

# Create your models here.

clas
    (variable) direccion: TextField[str]
    direccion = models.TextField()

class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    apellidos = models.CharField(max_length=200,blank=True)
    edad = models.IntegerField(null = True)

class Libro(models.Model):
    IDIOMAS = [
        ("ES", "Español"),
        ("EN", "Inglés"),
        ("FR", "Francés"),
        ("IT", "Italiano"),
    ]

    nombre = models.CharField(max_length=200)
    tipo = models.CharField(
        max_length=2,
        choices=IDIOMAS,
        default="ES",
    )

    descripcion = models.TextField()
    fecha_publicacion = models.DateField()
```

```python
class Cliente(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.CharField(max_length=200,unique=True)
    puntos = models.FloatField(default=5.0,db_column = "puntos_biblioteca")

class DatosCliente(models.Model):
    direccion = models.TextField()
    gustos = models.TextField()
    telefono = models.IntegerField()
```

Tipos de Campos: Relacionales

Ahora toca relacionar tablas unas con otras, para ello existen diferentes tipos de relaciones entre una tabla y otra:

- One to One -> Relación uno a uno
- Many to One -> Relación muchos a uno
- Many to Many -> Relación muchos a muchos

One to One

Por ejemplo tenemos los clientes de nuestra tabla cliente y creamos una tabla con los datos del perfil del cliente que almacenará teléfono, dirección, gustos,etc…

Por lo tanto cada registro de la tabla **datosClientes**, tendrá que tener un cliente asociado. ¿Cómo hacemos esto en nuestros modelos?

Creamos un atributo en el modelo **DatosCliente** de la siguiente forma

cliente = models.OneToOneField(Cliente, on_delete = models.CASCADE)

Destacar las partes:

- OneToOneField-> Es para indicar que es una relación una a una
- Cliente -> Es para indicar el modelo con el que está relacionado
- on_delete = models.CASCADE -> Es para indicar cómo tratar la eliminación de un registro asociado

One to One

Un ejemplo de como quedaría el modelo es el siguiente:

```
class Cliente(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.CharField(max_length=200,unique=True)
    puntos = models.FloatField(default=5.0,db_column = "puntos_biblioteca")

class DatosCliente(models.Model):
    cliente = models.OneToOneField(Cliente,
                                   on_delete = models.CASCADE)
    direccion = models.TextField()
    gustos = models.TextField()
    telefono = models.IntegerField()
```

Many to One

Ahora queremos representar la relación que tiene una biblioteca, con los libros que tiene.

En este caso una biblioteca va a tener muchos libros, pero los libros solo van a pertenecer a una biblioteca

En este caso en el modelo Libro, tendremos un atributo que representa la biblioteca a la que pertenece, de la siguiente forma:

```
biblioteca = models.ForeignKey(Biblioteca, on_delete = models.CASCADE)
```

Many to One

El código quedaría así:

```python
class Biblioteca(models.Model):
    nombre = models.CharField(max_length=100)
    direccion = models.TextField()

class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    apellidos = models.CharField(max_length=200,blank=True)
    edad = models.IntegerField(null = True)

class Libro(models.Model):
    IDIOMAS = [
        ("ES", "Español"),
        ("EN", "Inglés"),
        ("FR", "Francés"),
        ("IT", "Italiano"),
    ]

    nombre = models.CharField(max_length=200)
    tipo = models.CharField(
        max_length=2,
        choices=IDIOMAS,
        default="ES",
    )

    descripcion = models.TextField()
    fecha_publicacion = models.DateField()
    biblioteca = models.ForeignKey(Biblioteca, on_delete = models.CASCADE)
```

Many to Many

Ahora queremos representar la relación que tiene un libro con su autor o autores.

Es decir un libro puede ser escrito por un Autor o por varios. Por lo tanto un Autor puede haber escrito varios libros y a la vez esos libros pueden pertenecer a más de un autor.

Este caso es una relación Mucho a Muchos(Many to Many). Para ello debemos hacer lo siguiente:

autores = models.ManyToManyField(Autor)

Sólo tenemos que indicar en uno de los modelos la relación ManyToMany en el campo concreto.

Automáticamente se creará la tabla intermedia(**libro_autores**), pero no es necesario que nosotros la creemos.

Many to Many

```python
class Autor(models.Model):
    nombre = models.CharField(max_length=100)
    apellidos = models.CharField(max_length=200,blank=True)
    edad = models.IntegerField(null = True)

class Libro(models.Model):
    IDIOMAS = [
        ("ES", "Español"),
        ("EN", "Inglés"),
        ("FR", "Francés"),
        ("IT", "Italiano"),
    ]

    nombre = models.CharField(max_length=200)
    tipo = models.CharField(
        max_length=2,
        choices=IDIOMAS,
        default="ES",
    )

    descripcion = models.TextField()
    fecha_publicacion = models.DateField()
    biblioteca = models.ForeignKey(Biblioteca, on_delete = models.CASCADE)
    autores =   models.ManyToManyField(Autor)
```

Many to Many

Pero…¿Qué pasaría si quiero gestionar los préstamos de los libros con los clientes de la biblioteca?

Por un lado debemos saber los libros que ha sacado cada cliente y por otro los clientes que han usado los libros.

Además debemos tener claro cuando se hizo ese préstamo.

En este caso esta relación mucho a muchos, necesita una tabla intermedia para gestionarla.

En nuestro caso esta clase(Modelo) se llamará préstamos y se relaciona con cliente y libro de la siguiente forma.

Many to Many

Lo primero es incluir en el modelo **Cliente** la relación mucho a mucho con el modelo **Libro**, pero debemos añadir un parámetro extra: **through,** que sirve para identificar y especificar la tabla intermedia:

libros = models.ManyToManyField(Libro, through='Prestamos')

Y ahora creamos la tabla prestamos para que se relacione con ambas tablas:

```
class Prestamos(models.Model):
    cliente = models.ForeignKey(Cliente, on_delete=models.CASCADE)
    libro = models.ForeignKey(Libro, on_delete=models.CASCADE)
    fecha_prestamo = models.DateTimeField(default=timezone.now)
```

Many to Many: related name

En el caso que queramos añadir varias relaciones desde un modelo hacia el mismo modelo. Debemos incluir un parámetro a la hora de crear las relaciones, para darle un nombre a estas relaciones y evitar que django se confunda a la hora de realizar las migraciones.

Este parámetro se llama **related_name**. Importante este valor no puede repetirse en la aplicación, tiene que ser distinto para toda la aplicación(más adelante veremos el otro uso de este parámetro). Os pongo un ejemplo del modelo **Cliente**:

```
libros = models.ManyToManyField(Libro,
                                through='Prestamo',
                                related_name="libros")
libros_preferidos = models.ForeignKey(Libro,
                                      on_delete = models.CASCADE,
                                      related_name="favoritos")
```

Seeders

A continuación explicaré una herramienta que nos va a ser muy útil para crear datos en nuestra base de datos por defectos y además crear datos falsos y evitar tener que rellenarlos nosotros a mano.

A este proceso se le llama **Seeding,** y en muchos Frameworks se le llama **seeder** a la acción de crear con datos la base de datos preestablecidos.

Cuando se incluyen datos falsos se le llama **Faker**.

Seeders

Para ello debemos incluir en nuestro archivo **requirements.txt** lo siguiente:

```
django-seed~=0.3.1
```

```
psycopg2-binary~=2.9.9
```

Esto son los dos programas que vamos a utilizar, y a continuación para instalar dichos programas en nuestro entorno de python, lanzamos el siguiente comando desde el directorio donde esta el archivo y desde el entorno de python:

pip install -r requirements.txt

Seeders

A continuación en el archivo **mysite/settings.py** modificamos el array de **INSTALLED_APPS,** he incluimos lo siguiente:

```
INSTALLED_APPS = [

        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        …
        …
        'django_seed',
]
```

Seeders

Y ahora sólo tenemos que lanzar el siguiente comando para generar datos aleatorios en nuestra base de datos:

python manage.py seed <nombre_app> --number=15

Dónde number es el número de elementos que queremos que se cree en cada tabla.

Si os da error con el atributo DateTime por el timezone, en el settings.py del proyecto debéis especificar lo siguiente:

USE_TZ = False

En el siguiente enlace tenéis más información sobre esta herramienta:

https://github.com/Brobin/django-seed

Fixtures

El plugin django-seed, nos generará muchos datos irreales que no nos interesa. Por esa razón es más cómodo usar los Fixtures, que nos permitirán exportar e importar datos desde nuestra base de datos.

De esta forma podemos destruir nuestra base de datos tantas veces como queramos, que podremos recuperar los datos cuando queramos.

Fixtures

Lo primero es crear una carpeta fixtures dentro de nuestra aplicación.

A continuación lanzamos el siguiente comando:

python manage.py dumpdata --indent 4 > biblioteca/fixtures/datos.json

De esta forma guardamos en un archivo todos los datos de nuestra base de datos en .json

A continuación si borramos la base de datos y lanzamos los siguientes comandos:

python manage.py migrate

python manage.py loaddata biblioteca/fixtures/datos.json

Tendremos nuestra base de datos exactamente igual que como lo dejamos antes.

Fixtures

Podemos hacer fixtures de modelos en concretos (esto sería más correcto que toda la base de datos):

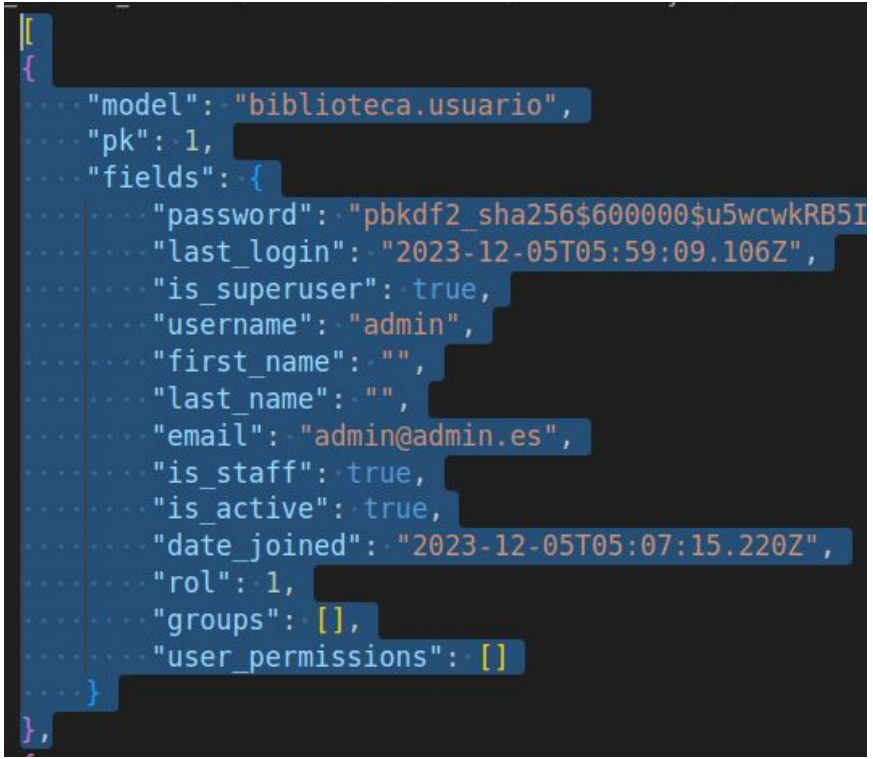
python manage.py dumpdata biblioteca.Usuario biblioteca.Cliente biblioteca.Bibliotecario --indent 4 > biblioteca/fixtures/usuarios.json

Y después podemos exportar sólo esos modelos con el comando:

python manage.py loaddata biblioteca/fixtures/usuarios.json

Fixtures

También podemos modificar a mano estos archivos .json. Que tendrían una forma parecida a la siguiente:



Fixtures

Destacar un detalle importante.

Es aconsejable que los fixtures con datos importantes, como contraseñas, nunca se suban a GIT, por lo tanto deberían añadirse a .gitignore