AWAKELAB

Acceso a Bases de Datos con Python/Django

Implementar la capa de modelo de acceso a datos del aplicativo utilizando entidades con relaciones uno a uno, uno a muchos y muchos a muchos para dar solución a una problemática

Objetivo de la jornada

1. Define un modelo con entidades que tienen relaciones uno a uno para resolver un problema determinado acorde al framework Django
2. Define un modelo con entidades que tienen relaciones muchos a uno para resolver un problema determinado acorde al framework Django
3. Define un modelo con entidades que tienen relaciones muchos a muchos para resolver un problema determinado acorde al framework Django

Relaciones Muchos a Uno

Las bases de datos relacionales causan muchas angustias a los desarrolladores de aplicaciones en distintos lenguajes y/o frameworks. Usualmente la intención es solamente crear una aplicación web interesante. Sin embargo, el almacenamiento y tratamiento de datos requiere comúnmente la incorporación de una tecnología de bases de datos como hemos visto en apartados anteriores. Esta no es una situación particular de desarrollos en Django. Los desarrolladores deben trabajar con conceptos de bases de datos cualquiera sea la tecnología que estén usando para crear su aplicación, por ejemplo, Rails, Drupal, WordPress, y otros. Prácticamente todas las aplicaciones web modernas usan una base de datos para almacenar información.

La buena noticia es que no es necesario convertirse en un administrador de bases de datos experto para usar Django, pero hay algunos conceptos clave que debemos comprender para aprovechar el framework correctamente. Uno de estos son las diferentes formas en que los modelos de bases de datos pueden relacionarse entre sí.

Una de las virtudes del ORM de Django es que abstrae gran parte del trabajo de establecer estas relaciones. Por ejemplo, configurar una relación Muchos a Muchos en el nivel de la base de datos requeriría crear una tabla de búsqueda separada para realizar un seguimiento de las relaciones entre los Modelos A y B. Django se encarga de todo eso detrás de escena para que podamos concentrarnos en escribir nuestras consultas. El ORM no es una solución milagrosa. A veces las consultas que crea a nivel de base de datos no son las más eficientes, pero es una gran herramienta para ayudarnos a poner en funcionamiento rápidamente una aplicación sin perdernos en

los detalles de los esquemas de bases de datos.

Hay tres tipos de relaciones que pueden tener los modelos: Uno A Uno, Muchos A Uno y Muchos A Muchos. Revisaremos a continuación, brevemente, cada una de estas relaciones y luego mostraremos cómo implementarlas en una aplicación Django.

Qué es una relación Muchos A Uno

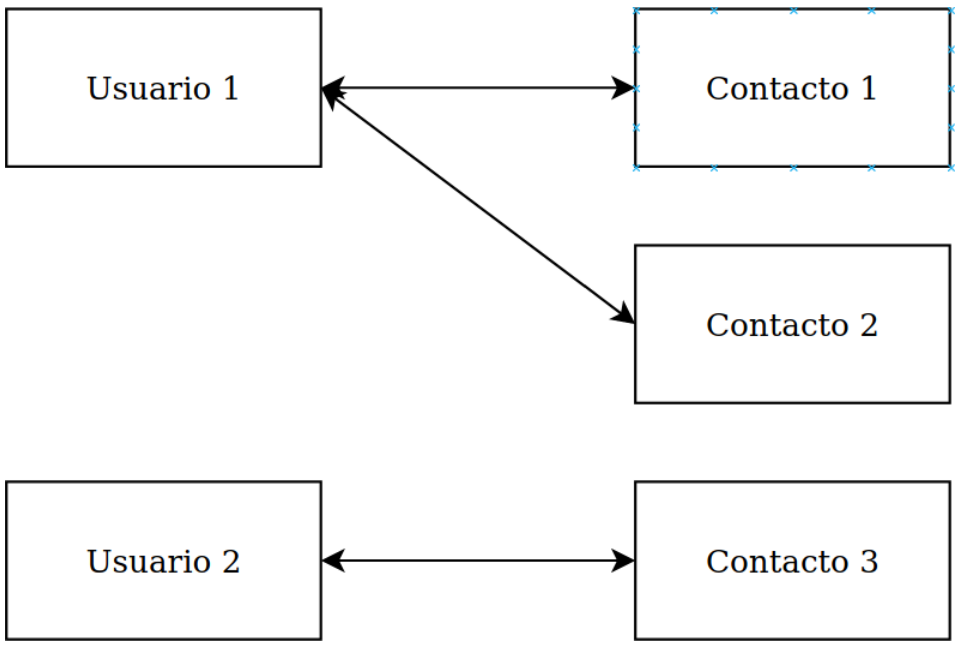
Para ilustrar Muchos A Uno en Django, hagamos otro ejemplo. Consideremos dos entidades: Usuario y Contacto. Para estas entidades decimos que:

- Cada Contacto tiene un Usuario
- Un Usuario puede tener muchos Contactos

Pensemos en un directorio o libreta de direcciones donde hay muchos usuarios identificados por un **nombre_usuario**, y cada usuario puede tener muchos contactos en su directorio propio:

- joel89 tiene abc@abc.dev, jules@jules.io, joe@joe.dev en su directorio.
- jules84 tiene def@abc.dev, vale@vale.io, john@john.dev en su directorio.

Si lo ilustramos gráficamente, sería:



Ahora, para crear estas entidades (modelos) en una aplicación de Django, podemos usar **django.db.models**. Creamos dos modelos, cada uno con los campos correspondientes:

```
from django.db import modelsclass Usuario(models.Model):
nombre_usuario = models.CharField(max_length=150)class
Contacto(models.Model):     email = models.EmailField()
```

La migración creará dos tablas, que aún no están conectadas según ninguna relación. Para esto agregaremos una **ForeignKey**.

FOREIGN KEYS

Para conectar las dos entidades de modo que muchos contactos estén conectados a un solo usuario, Django ofrece **ForeignKey**, como un campo que podemos agregar en los modelos:

```
from django.db import modelsclass Usuario(models.Model):    nombre_usuario =
models.CharField(max_length=150)class Contacto(models.Model):    email =
models.EmailField()    usuario = models.ForeignKey(to=Usuario,
on_delete=models.CASCADE)
```

Aquí agregamos una nueva columna denominada **usuario** que hace referencia al modelo de **Usuario** con una **ForeignKey**. Debemos ejecutar **python manage.py makemigrations** y **python manage.py migrate** para aplicar el cambio.

Es importante tener en cuenta que **ForeignKey** toma al menos dos argumentos:

```
usuario = models.ForeignKey(to=Usuario, on_delete=models.CASCADE)
```

to describe la entidad a la que queremos apuntar. **on_delete**, en cambio, describe cómo debe comportarse la base de datos cuando se elimina el lado "uno" de la relación. Cuando se elimina la entidad "uno", con CASCADE se eliminan también las entidades "muchos".

Otra cosa a tener en cuenta es que Django es un poco distinto a otros frameworks, donde el lado "muchos" se puede definir en el "uno". Ese es el caso de Laravel, por ejemplo.

Al final, el resultado es el mismo: la tabla "muchos" siempre estará conectada con una clave externa al "uno".

Ahora estamos listos para realizar consultas con el ORM de Django.

RELACIONANDO OBJETOS

Para probar lo que hemos realizado hasta el momento, procederemos de la siguiente manera, utilizando la consola de **Django** con **python manage.py shell**.

Importaremos los dos modelos, Usuario y Contacto:

```
>>> from directorio_app.models import Usuario, Contacto
```

A continuación, podemos ingresar información a la base de datos, en este caso un usuario:

```
>>> jules84 = Usuario.objects.create(nombre_usuario="jules84")
```

Además, podemos crear un grupo de contactos para este usuario:

```
>>> Contacto.objects.create(email="vale@vale.io",
usuario=jules84)<Contacto: Contacto object (1)>>>>
Contacto.objects.create(email="def@abc.dev",
usuario=jules84)<Contacto: Contacto object (2)>>>>
Contacto.objects.create(email="john@john.dev",
usuario=jules84)<Contacto: Contacto object (3)>>>>
```

Al crear un nuevo Contacto se pasa Usuario como argumento a **create**, con lo que unimos las dos entidades. Con estas entidades en su lugar, ahora estamos listos para realizar consultas.

OBTENIENDO OBJETOS

Para acceder a un Usuario desde nuestra base de datos podemos ejecutar:

```
>>> Usuario.objects.get(nombre_usuario="jules84")<Usuario: Usuario
object (1)>>>>
```

Si solo hay un usuario como en nuestro ejemplo, podemos ejecutar:

```
>>> Usuario.objects.first()<Usuario: Usuario object (1)>>>>
```

Ahora, podríamos desear buscar todos los contactos relacionados con ese usuario. Con el ORM de Django podemos usar lo que se llama un **lookup** que posee el formato
.relacionado_set donde **relacionado** es el nombre de la entidad a la que queremos acceder.

Entonces, para buscar todos los Contactos relacionados a nuestro Usuario (el usuario tiene muchos contactos) podemos ejecutar:

```
>>>
Usuario.objects.get(nombre_usuario="jules84").contacto_set.all().valu
es()<QuerySet [{'id': 3, 'email': 'john@john.dev', 'usuario_id': 1},
{'id': 2, 'email': 'def@abc.dev', 'usuario_id': 1}, {'id': 1,
'email': 'vale@vale.io', 'usuario_id': 1}]>>>>
```

También hay una forma de hacer lo contrario: desde un Contacto podemos volver al Usuario relacionado (el Contacto tiene un Usuario):

```
>>> Contacto.objects.first()._dict_{'_state':
<django.db.models.base.ModelState object at 0x7f2a179109d0>, 'id': 1,
'email': 'vale@vale.io', 'usuario_id': 1}>>>
```

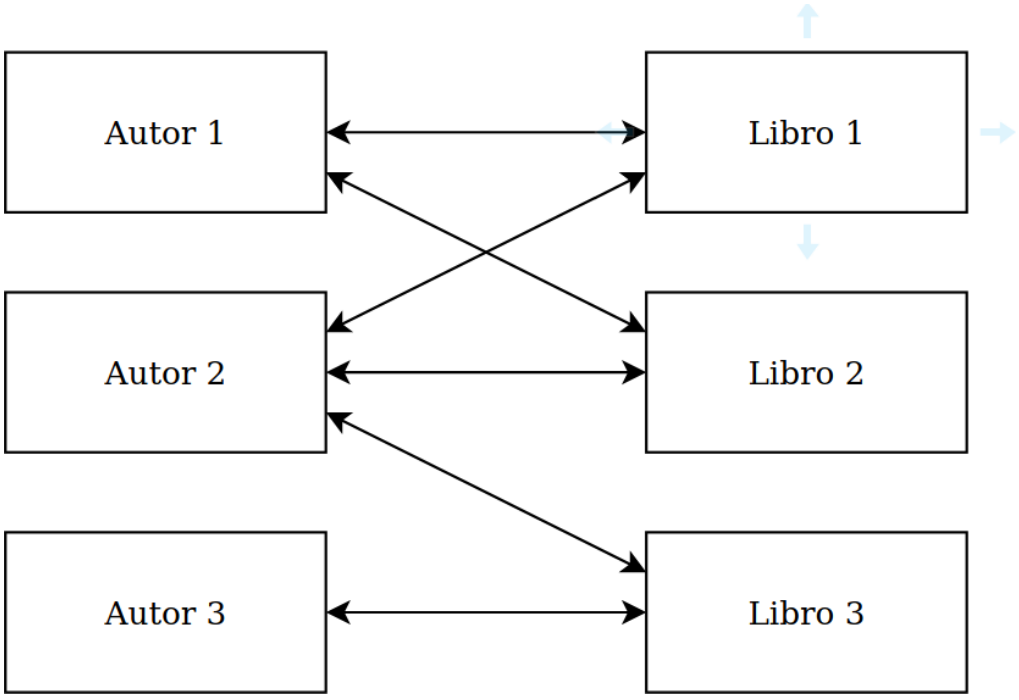
Relaciones Muchos a Muchos

Las relaciones Muchos A Muchos son el tipo de relación más complejo. El modelo A puede vincularse a muchos elementos del modelo B. Del mismo modo, el modelo B puede vincularse a muchos elementos del modelo A. Veamos algunos ejemplos:

- **Autores (A) y libros (B)**: un autor puede escribir muchos libros. Los libros también pueden tener varios autores.

- **Estudiantes (A) y cursos (B)**: un estudiante se inscribe en varios cursos. Cada curso tiene varios estudiantes inscritos.

- **Personas (A) y direcciones (B)**: una persona puede tener varias direcciones asociadas. Del mismo modo, una dirección se puede asociar con varias personas diferentes.



Debemos tener en cuenta que la relación que elijamos para los modelos depende completamente del contexto en el que lo estemos utilizando. Por ejemplo, en un contexto de automóvil / conductor, podemos considerarlo como una relación Uno a Uno. Sin embargo, si no miramos un automóvil que se conduce actualmente, podríamos representar esto como una relación Muchos A Muchos: los conductores pueden conducir muchos automóviles y los automóviles pueden tener muchos conductores.

Además, si bien una relación casas / direcciones sería Uno A Uno, si habláramos de departamentos en un gran edificio, podríamos modelarlo como Uno A Muchos (una dirección enlaza con muchos departamentos). Es crucial pensar en cómo se relacionan los modelos en el contexto de nuestra aplicación.

Las relaciones de Muchos A Muchos utilizan **ManyToManyField**. Veamos cómo se podría implementar el ejemplo de Autores y libros:

```
from django.db import modelsclass Libro(models.Model): titulo =
models.CharField(max_length=50)class Autor(models.Model): nombre =
models.CharField(max_length=50) libros =
models.ManyToManyField(Libro)
```

Y usemos las relaciones:

```
# Obtener todos los libros para un autor llamado `a`libros = a.libros#
Obteber todos los autores para un libro llamado `b`autores
= b.autor_set.all()
```

De manera similar al caso de Uno A muchos, podemos hacer referencia a los libros de un autor usando los libros de campo que hemos establecido en el modelo de autor, pero para usar la relación en la otra dirección tenemos que usar la sintaxis **author_set.all()**.

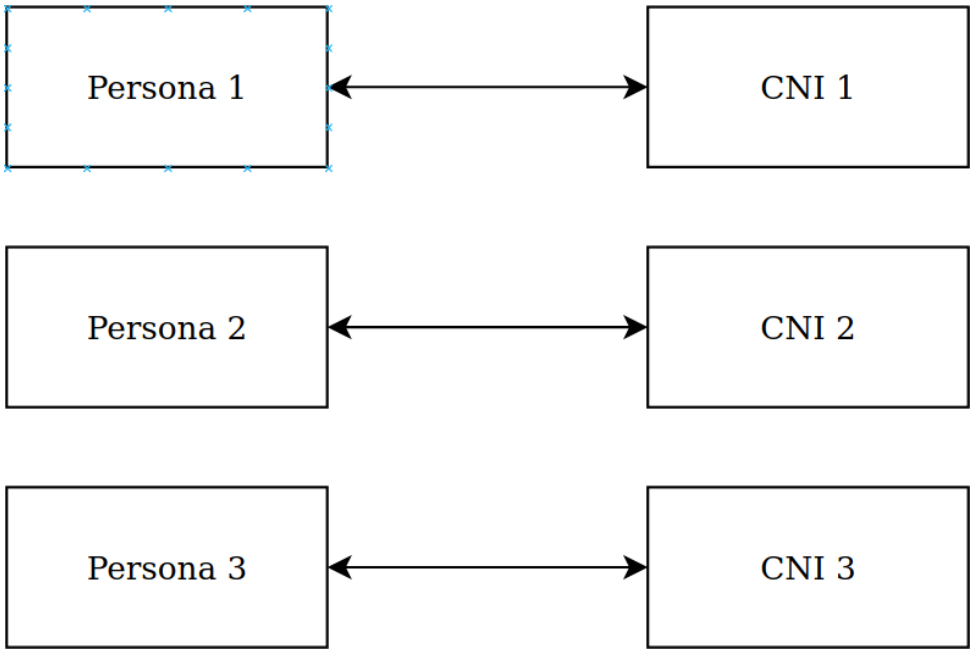
Relaciones Uno a Uno

Las relaciones Uno A Uno no son tan comunes como las Uno A Muchos. Se establecen cuando el Modelo A puede vincularse a un solo elemento del Modelo B, y el Modelo B puede vincularse a un solo elemento del Modelo A. Se utilizan relaciones Uno A Uno cuando existe el concepto de un emparejamiento estricto. Algunos ejemplos de la naturaleza para este caso son:

- **Personas (A) a Cédula Nacional de Identidad (B):** una persona solo puede tener una CNI. Una CNI solo puede representar a una única persona.

- **Automóviles (A) y conductores (B):** si estamos mirando un automóvil en particular mientras se conduce, ese automóvil solo puede tener un único conductor. El conductor de ese automóvil solo puede conducir ese automóvil específico.

- **Casas (A) y Direcciones (B):** Una casa solo puede tener una dirección. Una dirección solo puede referirse a una casa.



Las relaciones Uno A Uno se definen mediante **OneToOneField**. Nuestro ejemplo de Persona / CNI podría definirse así:

```
from django.db import modelsclass Persona(models.Model):      nombre =
models.CharField(max_length=50)      apellido =
models.CharField(max_length=50) class CNI(models.Model):      cni =
models.CharField(max_length=9)      persona =
models.OneToOneField(Persona,on_delete=models.CASCADE)
```

Aquí, la relación se define con la línea **persona = models.OneToOneField(Persona, on_delete=models.CASCADE)**. Al igual que **ForeignKey**, **OneToOneField** requiere dos argumentos posicionales: el nombre de la clase con la que está relacionado el modelo y el comportamiento de eliminación especificado con **on_delete**.

Hacer referencia a sus objetos relacionados es bastante simple:

```
# Obtener CNI para una Persona llamada `p`cni = p.CNI# Obtener la
persona para una número particular de CNI llamado `s`persona =
s.persona
```

En este caso no hay un **_set** porque la relación solo apuntará a un único elemento en el otro modelo.

OPCIÓN `on_delete`

La opción **on_delete** está disponible para los tres tipos de datos del modelo de relación y admite los siguientes valores:

- **on_delete = models.CASCADE (default)** .- Elimina automáticamente los registros relacionados cuando se elimina la instancia relacionada.

- **on_delete = models.PROTECT**.- Evita que se elimine una instancia relacionada.

- **on_delete = models.SET_NULL**.- Asigna NULL a los registros relacionados cuando se elimina la instancia relacionada, tenga en cuenta que esto requiere que el campo también use la opción null = True.

- **on_delete = models.SET_DEFAULT**.- Asigna un valor predeterminado a los registros relacionados cuando se elimina la instancia relacionada, tenga en cuenta que esto requiere que el campo también use un valor de opción predeterminado.

- **on_delete = models.SET**.- Asigna un valor establecido a través de un `callable` a registros relacionados cuando se elimina la instancia relacionada.

- **on_delete = modelos.DO_NOTHING**.- No se realiza ninguna acción cuando se eliminan registros relacionados. Esta es generalmente una mala práctica de base de datos relacional, por lo que, de manera predeterminada, las bases de datos generarán un error ya que está dejando registros huérfanos sin valor, nulo o de otro tipo. Si usa este valor, debe asegurarse de que la tabla de la base de datos no aplique la integridad referencial.

Referencias

[1] Django Documentation
https://docs.djangoproject.com/en/3.1/
[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.
[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.
[4] Understanding many to one in Django
https://www.valentinog.com/blog/many-to-one/

[5] The 3 types of database model relationships (and how to use them in Django)
https://betterprogramming.pub/how-to-design-relationships-between-your-django-models-caa01bc17a5c