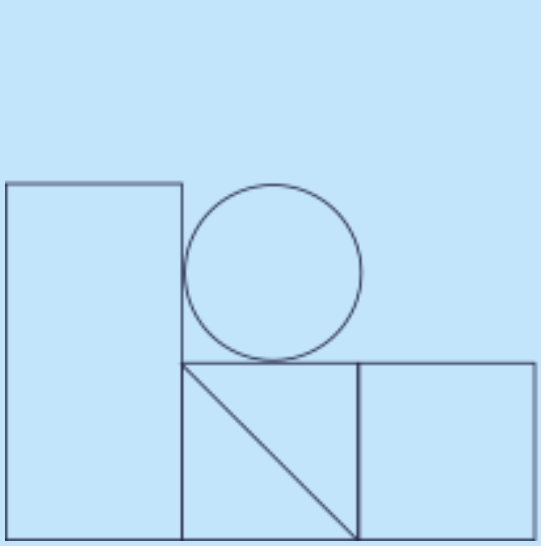
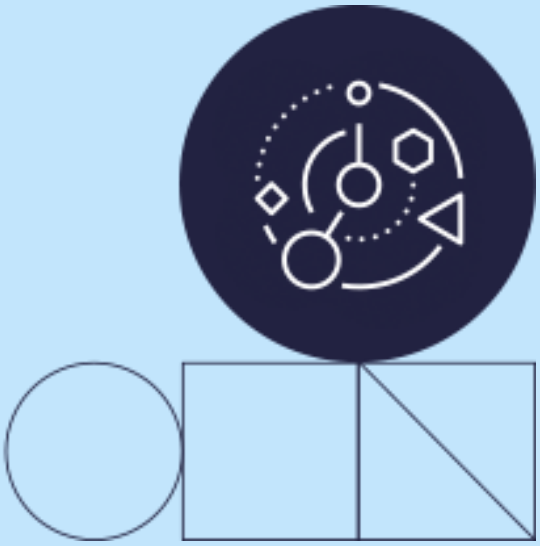
IBM SkillsBuild

IBM

Django

Qué es un ORM



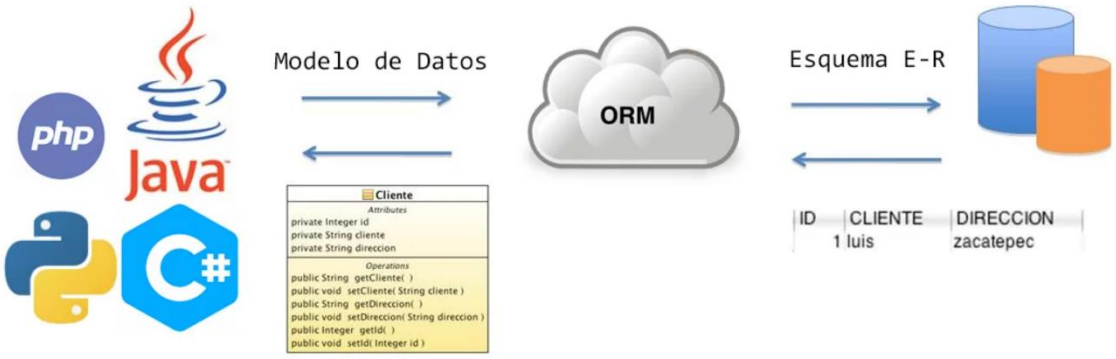


Índice

Introducción

Cuando se desarrolla una aplicación con acceso a base de datos empleando lenguajes tales como C#, Java, Python, PHP, VB, C++..., es necesario definir una estructura de clases que representan los datos que se almacenan en las bases de datos. Estas clases se denominan clases entidad y constituyen el modelo de datos de la aplicación.

En algunas ocasiones el modelo de datos se define a partir de esquema entidad-relación que define las tablas, campos y relaciones de la base de datos. En otras ocasiones el proceso es a la inversa y es el esquema entidad-relación el que se genera a partir del modelo de datos. En ambos casos, el proceso recibe el nombre de mapeo e implica un costo considerable de tiempo y recursos en el desarrollo de las aplicaciones.



Por lo tanto, un ORM (Object-Relational Mapping) es una técnica de programación que se utiliza para mapear entre los objetos de una aplicación y las tablas de una base de datos relacional. En lugar de escribir consultas SQL directamente para acceder a los datos de la base de datos, un ORM proporciona una capa de abstracción que permite a los desarrolladores trabajar con los objetos de la aplicación de manera más intuitiva.

Básicamente, un ORM convierte las operaciones de lectura y escritura de datos en una base de datos relacional en operaciones de manipulación de objetos en el lenguaje de programación utilizado por la aplicación. Por ejemplo, en lugar de escribir una consulta SQL para recuperar todos los registros de una tabla, el desarrollador puede simplemente llamar a un método de la clase ORM correspondiente y obtener un conjunto de objetos que representan los datos de la tabla.

Los ORMs pueden acelerar el desarrollo de aplicaciones al simplificar el acceso a la base de datos y reducir la cantidad de código necesario para interactuar con la base de datos. Sin embargo, también es importante comprender los mecanismos subyacentes de la base de datos relacional y cómo se mapean a objetos para utilizar ORMs de manera efectiva.

¿Qué es un ORM?

Un ORM (Object Relational Mapping) es una herramienta de software que automatiza el mapeo entre el modelo de datos y el esquema entidad-relación. Esto conlleva dos ventajas básicas:

- Agiliza el desarrollo y reduce la cantidad de código al automatizar el mapeo permitiendo generar las clases entidad a partir de una base de datos existente, o viceversa.
- Independiza la aplicación del tipo de base de datos (SQL Server, Oracle, MySQL...), permitiendo la obtención de datos usando objetos y métodos sin escribir consultas SQL.

Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o base de datos virtual definida en el ORM, de tal modo que las acciones CRUD (Create, Read, Update, Delete) a ejecutar sobre la base de datos física se realizan de forma indirecta por medio del ORM.

La consecuencia más directa que se infiere del párrafo anterior es que, además de “mapear”, los ORMs tienden a “liberarnos” de la escritura o generación manual de código SQL (Structured Query Language) necesario para realizar las queries o consultas y gestionar la persistencia de datos en el RDBMS.

Así, los objetos o entidades de la base de datos virtual creada en nuestro ORM podrán ser manipulados por medio de algún lenguaje de nuestro interés según el tipo de ORM utilizado, por ejemplo, LINQ sobre Entity Framework de Microsoft. La interacción con el RDBMS quedará delegada en los métodos de actualización correspondientes proporcionados por el ORM. Los ORMs más completos ofrecen servicios para persistir todos los cambios en los estados de las entidades, previo seguimiento o tracking automático, sin escribir una sola línea de SQL.

¿Cuándo usar un ORM?

El uso del ORM’s tiene como contrapartida una penalización en el rendimiento al trabajarse con objetos en vez de directamente con los datos y usarse SQL generadas no optimizadas.

Es por ello que el uso de ORM’s está especialmente recomendado en aplicaciones con modelos de datos complejos donde el rendimiento no es crítico. El uso del ORM acelera el desarrollo evitando la escritura de código repetitivo para consultas, altas, bajas y modificaciones (CRUD), y dado que el rendimiento no es crítico tampoco es preciso el empleo de SQL’s optimizadas.

En el caso de aplicaciones con modelos de datos simples o donde el rendimiento es crítico no es recomendable el uso de ORM, debido a que la sobrecarga de código que añade el ORM para manejar los datos no compensa frente al acceso directo a los mismos y el empleo de consultas SQL optimizadas para acelerar las operaciones.

Existen diferentes ORM's en función del lenguaje y plataformas empleadas. Los más populares son los siguientes:

- C# / Visual Basic .NET -> Entity Framework Core / Entity Framework 6
- Java -> Hibernate
- Phyton -> SQLAlchemy
- PHP

ORM en Django

Uno de los principales objetivos de Django es crear contenido de manera rápida. Y una forma de conseguir esto es liberando al programador del acceso directo a la base de datos. En lugar de escribir consultas SQL a mano, Django nos proporciona un mapeo objeto-relacional (ORM por sus siglas en inglés) bastante práctico y sencillo de usar.

El modelo de datos

Django crea la asociación (o el mapeo) de las tablas de la base de datos a estructuras y tipos de datos de python usando el modelo de datos. Un modelo es una clase que corresponde al nombre de una tabla en la base de datos, y cada variable de clase es un campo de dicha tabla. En esta clase también se especifican los índices de la tabla, llaves foráneas, ordenamiento de los datos, reglas de validación, entre otras cosas. Cada app de nuestro sistema Django tiene un modelo asociado, y las tablas creadas son del tipo **app_nombremodelo**; así que si tengo una app **contabilidad** y mi modelo tiene la clase **Cliente**, la tabla se llamará **contabilidad_cliente**.

Migraciones

Cuando el modelo está definido, Django usa las migraciones para crear las tablas y campos necesarios en la base de datos, y si posteriormente hacemos modificaciones al modelo, las migraciones se encargan de replicar estos cambios.

Implementación del modelo

Todas las clases tienen que heredar de la clase **django.db.models.Model**, y las variables de clase corresponde a **django.db.models.Mi_TipoField()**, donde **Mi_Tipo** puede ser **Char**, **Date**, **Int**, **Bool**, **Email**, etc. A continuación expondremos un ejemplo de un modelo sencillo, el cual usaremos de base como ejemplo:

```
from django.db import models


class Cliente(models.Model):
    nombre =
models.CharField(max_length=64)
    apellidos =
models.CharField(max_length=64)
    rfc = models.CharField(max_length=15,
unique=True)
    fecha_nacimiento = models.DateField()
    activo =
models.BoolField(default=True)
```

Para crear la migración ejecutamos el comando:

```
python manage.py makemigrations
```

Este comando buscará cambios en todos los modelos de nuestras apps y creará las migraciones correspondientes. Y para ejecutarlas usamos el comando:

```
python manage.py migrate
```

En nuestro caso tendremos la tabla **contabilidad_cliente** con los campos **id**, **nombre**, **apellidos**, **rfc**, **fecha_nacimiento** y **activo**.

Nota: aunque no definimos el campo **id** en nuestro modelo, Django lo crea automáticamente por nosotros, y es de tipo entero auto-incremental. Este campo es la llave primaria de nuestra tabla.

Aunque podemos especificar la longitud de campo o valores default, depende de la base de datos que estemos usando el que estas opciones tengan efecto o no.

Llaves foráneas

Relación Uno a Muchos

Sin duda una de las relaciones más usadas. Voy a crear el modelo **Factura** para ligarlo al modelo **Cliente**, así podemos decir que un cliente tiene muchas facturas, pero una factura solo le pertenece a un cliente:

```
from django.db import models

class Cliente(models.Model):
    nombre =
models.CharField(max_length=64)
    apellidos =
models.CharField(max_length=64)
    rfc = models.CharField(max_length=15,
unique=True)
    fecha_nacimiento = models.DateField()
    activo =
models.BoolField(default=True)

class Factura(models.Model):
    cliente = models.ForeignKey(Cliente,
on_delete=models.CASCADE)
    importe =
models.DecimalField(max_digits=8,
decimal_digits=2)
    pagada =
models.BoolField(default=False)
```

Nota: en el caso de los campos que son llaves foráneas Django les agrega el sufijo **_id**, por lo que el campo cliente en la base de datos es **contabilidad_factura.cliente_id**.

Creación de registros

Ahora que ya sabemos cómo crear un modelo, el siguiente paso es crear instancias de estos modelos para almacenarlos en la base de datos. Para esto tenemos dos opciones: crear el registro y guardarlo en automático (usando el método **create()**), o crear el registro (una instancia), y guardarlo a posteriori (usando el método **save()**).

Usando el método save()

```
from contabilidad.models import Cliente,
Factura
import datetime

fecha_nacimiento = datetime.date(1980,
12, 5)
pedro = Cliente(
    nombre="Pedro",
    apellidos="Aguilar Ramírez",
    rfc="AGRM-801205-111",
    fecha_nacimiento=fecha_nacimiento,
    activo=True,
)
pedro.save()
```

```
factura = Factura(cliente=pedro,
importe=5690.12, pagada=False)
factura.save()
```

Cuando creamos la factura, tenemos que pasarle un registro de tipo Cliente por la relación uno a muchos, en este caso nuestro registro es pedro.

Usando el método create()

El método **MiModelo.objects.create()** en realidad es un wrapper a **save()**, permitiéndonos crear y guardar en un solo paso el modelo.

```
from contabilidad.models import Cliente,
Factura
import datetime

fecha_nacimiento = datetime.date(1980,
12, 5)
pedro = Cliente.objects.create(
    nombre="Pedro",
    apellidos="Aguilar Ramírez",
    rfc="AGRM-801205-111",
    fecha_nacimiento=fecha_nacimiento,
    activo=True,
)

factura =
Factura.objects.create(cliente=pedro,
importe=5690.12, pagada=False)
```

¿Cuál de los dos métodos es el correcto?

Depende de nuestro caso de uso. En la siguiente tabla resumiremos cuando usar cada método:

Caso de uso	save()	create()
Crear un registro (INSERT)	X	X
Actualizar un registro (UPDATE)	X	
Requiere una instancia del modelo	X	
Modificar un campo antes de grabarlo	X	
No necesita una instancia creada		X

Como podemos ver, **create()** solo se usa cuando vamos a crear un registro nuevo, para todo lo demás, usamos **save()**.

Consultando el modelo

Ahora que ya tenemos creado nuestro modelo y hemos creado algunos registros, tenemos que consultar los datos. Podemos obtener todos los datos de un modelo (tabla), o solo unos cuantos registros. También podemos ver todos los campos del modelo, o solo unos cuantos. Todos los métodos de esta sección regresan un **QuerySet** con los registros consultados, y podemos iterar sobre este **QuerySet** para procesar cada registro, y si no encontró registro alguno, entonces tendremos un **QuerySet** vacío. Veamos unos ejemplos.

Recuperando todos los registros del modelo

Este es el caso más simple, solo tenemos que usar el método **all()** de nuestro modelo:

```
from contabilidad.models import Cliente,
Factura

clientes = Cliente.objects.all()
for cliente in clientes:
    print(f"Nombre: {cliente.nombre},
Apellidos: {cliente.apellidos}")
```

Y en el caso de los modelos con relaciones, también podemos acceder a los datos del modelo relacionado, usando el nombre del campo que tiene la relación, seguido de un doble guion bajo (__) y por último el campo del otro modelo:

```
from contabilidad.models import Cliente,
Factura

facturas = Factura.objects.all()
for factura in facturas:
    print(f"Pagada: {factura.pagada},
Importe: {factura.importe}, RFC Cliente:
{factura.cliente__rfc}")
```

Como se puede observar al final de la última línea, para acceder al campo **rfc** desde el modelo **Factura**, usamos **factura.cliente__rfc**.

Filtrando los registros

Podemos usar el método **filter()** como un símil del **WHERE** de SQL, y así obtener un conjunto de datos limitado por condiciones. Los parámetros de **filter()** corresponden a los nombres de las variables del modelo, y se pueden agregar modificadores con el sufijo __.

```
from contabilidad.models import Cliente,
Factura
import datetime

# obtenemos exclusivamente a los clientes
activos
clientes =
Cliente.objects.filter(activo=True)

# buscamos a los clientes que su fecha de
nacimiento > 1980-03-01
# y que no estén activos. El modificaror
'__gt' significa greater than, o >
fecha = datetime.date(1980, 3, 1)
clientes =
Cliente.objects.filter(activo=False,
fecha_nacimiento__gt=fecha)
# si queremos un filtro inclusivo usamos
'__gte' (greater than equal)
clientes =
Cliente.objects.filter(activo=False,
fecha_nacimiento__gte=fecha)

# para buscar un rango de fechas y todos
los que se llamen Carlos:
inicio = datetime.date(1980, 3, 1)
final = datetime.date(2000, 1, 1)
clientes = Clientes.objects.filter(
    fecha_nacimiento__range=(inicio,
final),
    nombre="Carlos")

# podemos buscar por los valores de una
lista
clientes =
Clientes.objects.filter(nombre__in=["Juan
", "María", "Gabriela"])
```

Cuando tenemos modelos con relaciones el procedimiento es muy similar, solo tenemos que usar __ para acceder al modelo referenciado, y si queremos usar un modificador usamos el sufijo __.

Busquemos las facturas pagadas de los clientes nacidos en la década de los 80's:

```
from contabilidad.models import Cliente,
Factura
import datetime


inicio = datetime.date(1980, 1, 1)
final = datetime.date(1989, 12, 31)
facturas = Factura.objects.filter(
    pagada=True,
    cliente__fecha_nacimiento__range=(ini
cio, final))
```

Nota: podemos restringir el número de registros regresados usando la notación de slices nativa de Python, y esto se traduce al equivalente **LIMIT** de SQL. Para regresar los primeros diez registros: **Factura.objects.all()[:10]** o si queremos regresar los registros del 5 al 20 usamos **Factura.objects.all()[5:20]**.

Obteniendo un solo registro

En ocasiones solo necesitamos un registro en particular, y para esto usamos el método **get()**. Los parámetros de este método son los nombres de los campos, los cuales nos servirán como condiciones SQL para obtener el registro. El caso más común es filtrar por el id del registro, por lo que si queremos obtener al Cliente con **id** número 5 hacemos lo siguiente:

```
from contabilidad.models import Cliente

pedro = Cliente.objects.get(id=5)
print(pedro.nombre)
```

En este caso no obtenemos un **QuerySet**, pero sí obtenemos un objeto con la información de nuestro registro.

Nota: lo más común es usar **get()** con el **id**, pero si creamos manualmente la llave primaria de nuestro modelo, le podemos poner el nombre que nosotros consideremos más adecuado. Para facilitarnos la vida, Django pone a nuestra disposición la variable **pk**, que apunta a la llave primaria. Es por esto que se recomienda usar **pk** con **get(): Cliente.objects.get(pk=5)**.

A diferencia de **filter()** y **all()**, si **get()** no encuentra ningún registro lanzará la excepción **modelo.DoesNotExist**, por lo que hay que manejarla siempre que usemos **get()**:

```
from contabilidad.models import Cliente

cliente_id=5
try:
    pedro =
Cliente.objects.get(pk=cliente_id)
    print(pedro.nombre)
except Cliente.DoesNotExist:
    print(f"No existe el registro con
id={cliente_id}")
```

Muchas veces necesitamos crear el usuario que estamos buscando si es que no existe, de esta forma:

```
from contabilidad.models import Cliente
import datetime


cliente_id=5
fecha_nacimiento = datetime.date(1980,
12, 5)
try:
    pedro = Cliente.objects.get(
        nombre="Pedro",
        apellidos="Aguilar Ramírez",
        rfc="AGRM-801205-111",
        fecha_nacimiento=fecha_nacimiento
)
except Cliente.DoesNotExist:
    pedro = Cliente.objects.create(
        nombre="Pedro",
```

```
        apellidos="Aguilar Ramírez",
        rfc="AGRM-801205-111",
        fecha_nacimiento=fecha_nacimiento
)
```

Esto además de engorroso rompe con el principio DRY. Afortunadamente para nosotros, Django tiene un atajo (este es solo uno de muchos que nos hacen la vida más fácil):

```
from contabilidad.models import Cliente
import datetime


cliente_id=5
fecha_nacimiento = datetime.date(1980,
12, 5)
pedro = Cliente.objects.get_or_create(
    nombre="Pedro",
    apellidos="Aguilar Ramírez",
    rfc="AGRM-801205-111",
    fecha_nacimiento=fecha_nacimiento)
```

Solo tenemos que hacer la búsqueda usando todos los campos que no tengan valores **default**, para que el registro pueda ser creado si no existe.

Ordenando el resultado

Así como en SQL usamos **ORDER BY** para ordenar el resultado, el ORM de Django tiene el método **order_by()**. Usarlo es muy sencillo, solo tenemos que especificar los nombres de los campos a ordenar, y si queremos un ordenamiento descendente anteponemos un guion al nombre del campo:

```
from contabilidad.models import Cliente

# Buscamos todos los clientes con
apellido paterno 'García' y
# ordenamos ascendentemente por
'apellidos', y después
# descendentemente por 'fecha_nacimiento'
clientes =
Cliente.objects.filter(apellidos__startsw
ith="García").order_by(
    "apellidos", "-fecha_nacimiento")
```

Funciones de agregación (Agregates)

También es posible usar funciones como **Sum**, **Avg**, **Count** (entre otras), sólo que en lugar de usar un **group_by()** usamos **annotate()**. Veamos un ejemplo más complejo con varios conceptos que hemos visto:

```
import datetime
from django.db import models
from contabilidad import Factura

inicio = datetime.date(2020, 1, 1)
final = datetime.date(2020, 12, 31)
Factura.objects.filter(
    cliente_id=3, fecha__range=(inicio,
final), importe__lte=50000
    ).annotate(Sum("importe")
    ).order_by("fecha", "importe")
```

Esta consulta nos dará el total por día de todas las facturas con un importe menor o igual a 50,000, ordenando primero por fecha y después por el importe, para el cliente con un **id** de 3.

Pro-Tip: mirar el SQL generado

Aunque no estemos trabajando con SQL directamente, al final Django convierte las consultas del ORM a código SQL. En ocasiones es muy útil inspeccionar este código para validar que nuestra consulta sea la correcta. Para esto solo tenemos que revisar el contenido de la variable **query**:

```
from contabilidad.models import Cliente

clientes =
Cliente.objects.filter(apellidos__startsw
ith="García").order_by(
    "apellidos", "-fecha_nacimiento")
print(clientes.query)
```

Conclusión

El ORM de Django es muy sencillo de usar y potente al mismo tiempo. Sin embargo, habrá situaciones donde una consulta SQL no la podamos expresar con el ORM, pero Django nos permite ejecutar raw queries en estos casos. También es posible extender el funcionamiento del ORM creando nuestros propios filtros o funciones, manteniendo toda la lógica del modelo en un mismo lugar.