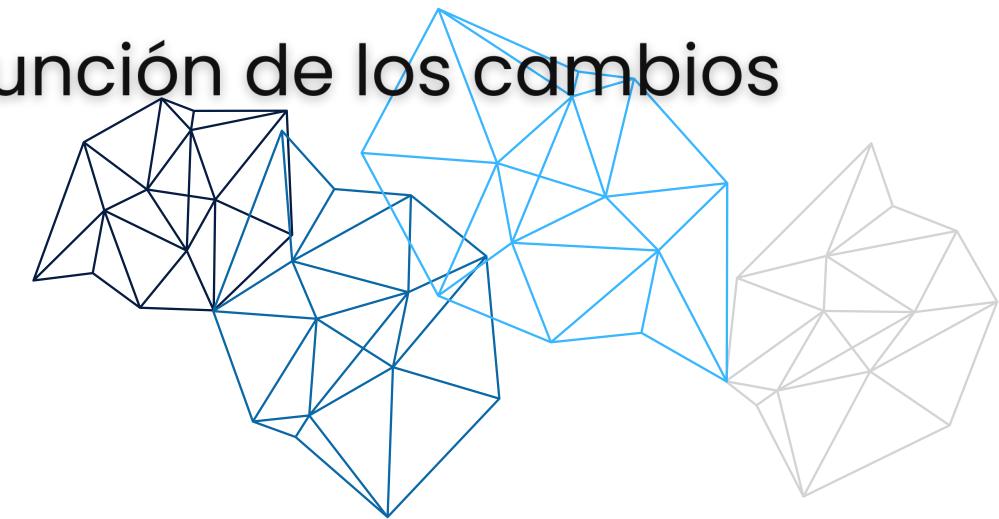


Modelos en Django

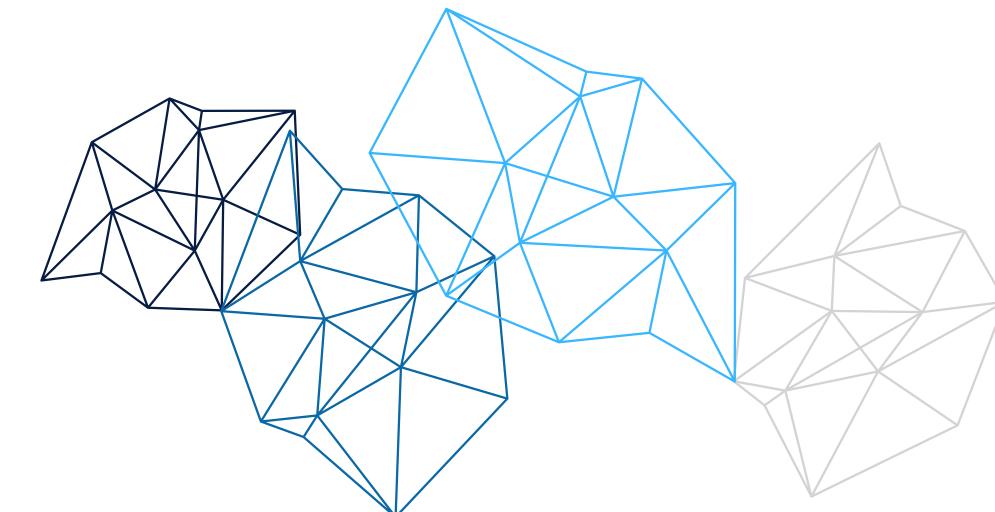
En Django, los modelos son clases de Python que definen la estructura y el comportamiento de las tablas de la base de datos. Las migraciones, por otro lado, son archivos generados automáticamente por Django que permiten mantener la consistencia entre los modelos y la base de datos a medida que se realizan cambios en el esquema de la base de datos.

Descripción general de cómo funcionan los modelos y las migraciones en Django:

- Definir modelos: En Django, los modelos se definen en archivos Python dentro de una aplicación. Cada modelo representa una tabla en la base de datos y define los campos y las relaciones que tendrá dicha tabla. Los modelos se definen como clases que heredan de la clase `django.db.models.Model`.
- Crear migraciones iniciales: Después de definir los modelos, se deben crear las migraciones iniciales. Una migración inicial es un archivo generado automáticamente que contiene instrucciones para crear todas las tablas correspondientes a los modelos definidos. Esto se logra ejecutando el comando **`python manage.py makemigrations`** en la terminal. Este comando examina los modelos y crea las migraciones necesarias en función de los cambios detectados.



- Aplicar migraciones: Una vez que se han creado las migraciones iniciales, se deben aplicar a la base de datos para crear las tablas correspondientes. Esto se hace ejecutando el comando **python manage.py migrate**. Django leerá las migraciones pendientes y ejecutará los cambios necesarios en la base de datos.
- Modificar modelos: A medida que evoluciona una aplicación, es posible que se realicen cambios en los modelos existentes, como agregar campos, eliminar campos, modificar relaciones, etc. Cuando se realizan cambios en los modelos, se deben crear nuevas migraciones para reflejar esos cambios en la base de datos. Esto se hace nuevamente con el comando **python manage.py makemigrations**, que generará las migraciones necesarias.
- Aplicar migraciones posteriores: Una vez que se han creado las nuevas migraciones, se deben aplicar a la base de datos para actualizarla con los cambios en el esquema. Esto se hace ejecutando el comando **python manage.py migrate**.



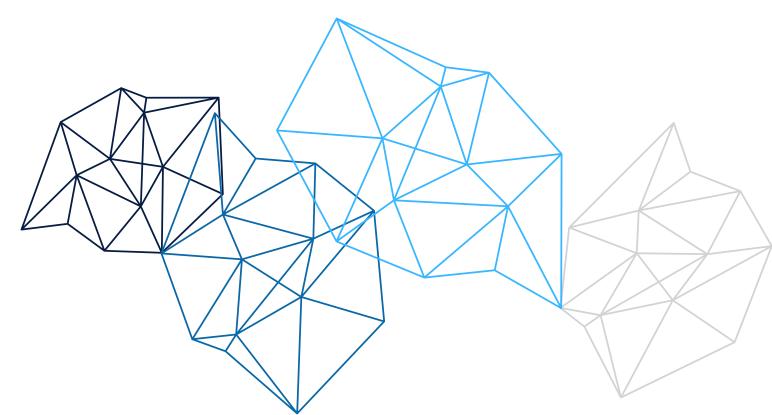
Django realiza un seguimiento de las migraciones aplicadas y las registra en una tabla especial en la base de datos. De esta manera, puede mantener un registro de los cambios realizados y garantizar que la base de datos esté sincronizada con los modelos definidos en la aplicación. Es importante tener en cuenta que las migraciones son esenciales para mantener la integridad de la base de datos a medida que evoluciona una aplicación Django. Además, las migraciones también facilitan la colaboración en equipo, ya que permiten a los desarrolladores compartir y aplicar cambios en la estructura de la base de datos de manera controlada y coherente.

Te estarás preguntando... **¿Cómo se definen los modelos?** Bueno, vamos a eso...

Los modelos se definen mediante la creación de clases en archivos Python. Cada clase representa una tabla en la base de datos y define los campos y las relaciones de esa tabla. Aquí tienes un ejemplo básico de cómo se definen los modelos en Django:

- Importa la clase **models** de Django:

```
from django.db import models
```

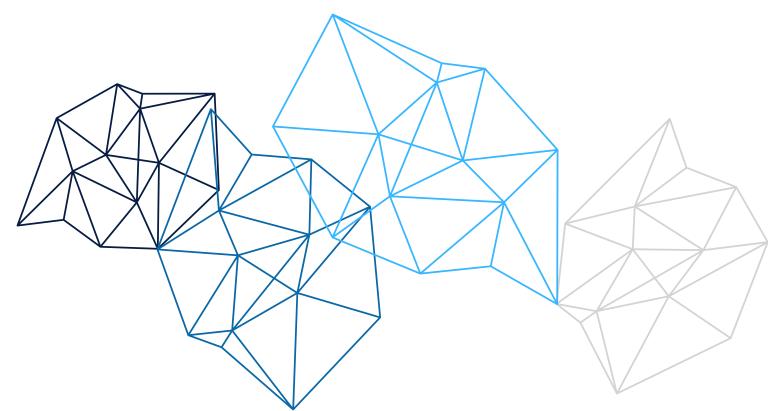


- Define una clase que herede de **models.Model** para representar una tabla en la base de datos:

```
class MiModelo(models.Model):  
    campo1 = models.CharField(max_length=100)  
    campo2 = models.IntegerField()  
    campo3 = models.DateTimeField()
```

En este ejemplo, hemos definido un modelo llamado MiModelo con tres campos: campo1, campo2 y campo3. Cada campo se define como una instancia de una clase de campo específica de Django, como CharField, IntegerField o DateTimeField. Estos campos determinan el tipo de datos que se almacenará en la base de datos y proporcionan opciones adicionales para configurar su comportamiento.

Una vez que has definido tus modelos, debes asegurarte de agregar tu aplicación y los modelos a la configuración de Django. Para hacer esto, agrega la aplicación a la lista INSTALLED_APPS en el archivo settings.py y ejecuta las migraciones correspondientes.



En Django, hay varios tipos de campos que puedes utilizar al definir modelos. A continuación, te mostramos algunos de los tipos de campos más comunes disponibles en el módulo `django.db.models`:

- **CharField**: Almacena cadenas de texto de longitud limitada.

```
campo = models.CharField(max_length=100)
```

- **TextField**: Almacena cadenas de texto de longitud variable.

```
campo = models.TextField()
```

- **IntegerField**: Almacena números enteros.

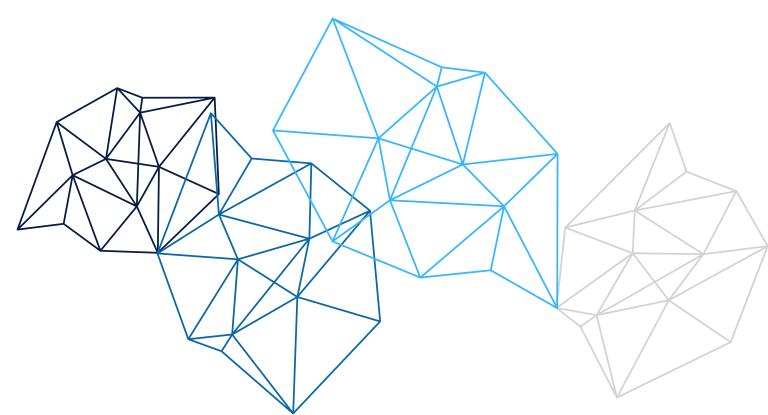
```
campo = models.IntegerField()
```

- **FloatField**: Almacena números de punto flotante.

```
campo = models.FloatField()
```

- **BooleanField**: Almacena valores booleanos (True o False).

```
campo = models.BooleanField()
```



- **DateTimeField**: Almacena fecha y hora.

```
campo = models.DateTimeField()
```

- **DateField**: Almacena solo la fecha.

```
campo = models.DateField()
```

- **TimeField**: Almacena solo la hora.

```
campo = models.TimeField()
```

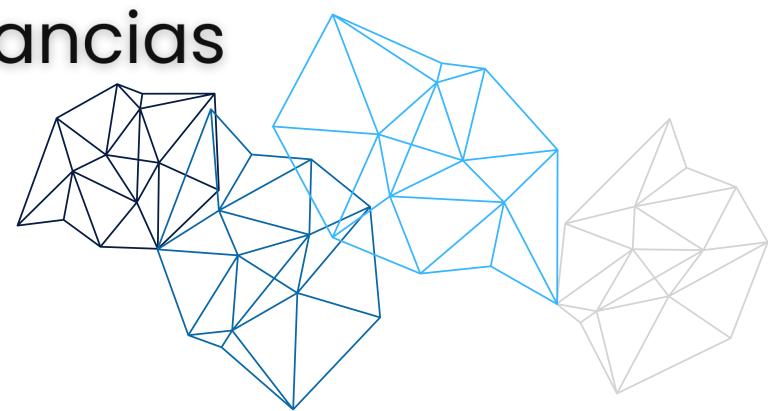
- **ForeignKey**: Define una relación de muchos a uno con otro modelo.

```
otro_modelo = models.ForeignKey(OtroModelo, on_delete=models.CASCADE)
```

- **ManyToManyField**: Define una relación de muchos a muchos con otro modelo.

```
otro_modelo = models.ManyToManyField(OtroModelo)
```

En Django, el parámetro **on_delete** se utiliza para especificar el comportamiento que se debe tomar cuando se elimina el objeto relacionado en una relación de clave externa (**ForeignKey**) o de muchos a muchos (**ManyToManyField**). Indica qué debe hacerse con las instancias relacionadas cuando se elimina la instancia principal.



Estos son solo algunos ejemplos de los tipos de campos más comunes en Django. Además de estos, existen otros campos especializados, como **EmailField**, **ImageField**, **FileField**, **URLField**, entre otros. También puedes crear tus propios campos personalizados si los necesitas.

Algunos de los parámetros comunes adicionales que se pueden utilizar en los campos de Django:

- **null**: Indica si el campo puede tener un valor **NULL** en la base de datos. El valor predeterminado es **False**.

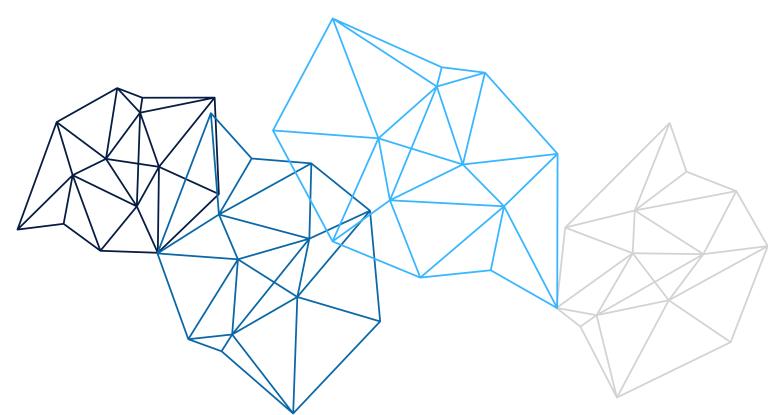
```
campo = models.CharField(max_length=100, null=True)
```

- **blank**: Indica si el campo es obligatorio en los formularios. El valor predeterminado es **False**. Si **blank=True**, el campo puede estar vacío.

```
campo = models.CharField(max_length=100, blank=True)
```

- **default**: Establece un valor predeterminado para el campo si no se proporciona ningún valor. Puede ser un valor o una función que se evalúa al crear un nuevo objeto.

```
campo = models.IntegerField(default=0)
```



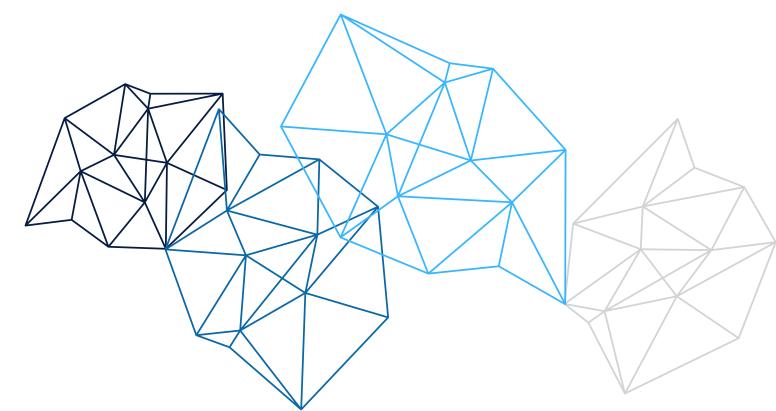
- **choices**: Permite definir opciones predefinidas para el campo, especificando una lista de tuplas con los valores y las etiquetas correspondientes.

```
OPCIONES = (
    ('opcion1', 'Opción 1'),
    ('opcion2', 'Opción 2'),
)

campo = models.CharField(max_length=100, choices=OPCIONES)
```

- **verbose_name**: Permite especificar un nombre legible por humanos para el campo. Este nombre se utiliza en la representación textual del campo, como en los formularios generados automáticamente.

```
campo = models.CharField(max_length=100, verbose_name='Nombre del campo')
```



- **help_text**: Proporciona una ayuda o descripción adicional para el campo, que se muestra en los formularios generados automáticamente.

```
campo = models.CharField(max_length=100, help_text='Ingrese su nombre completo')
```

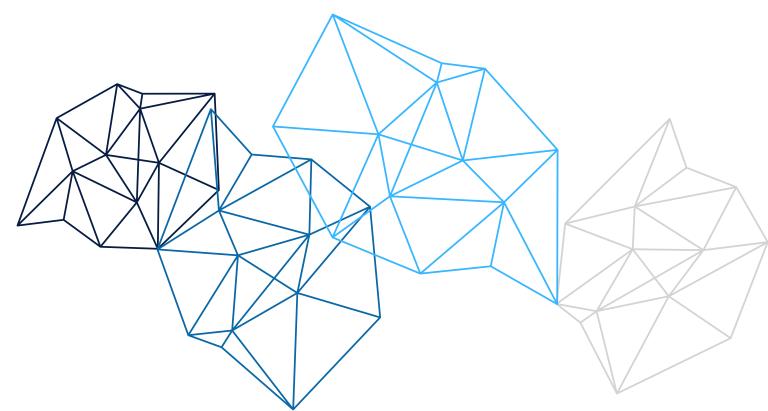
- **unique**: Indica si el campo debe ser único en la base de datos.

```
campo = models.CharField(max_length=100, unique=True)
```

- **auto_now**: Actualiza automáticamente el valor del campo con la fecha y hora actuales cada vez que se guarda el objeto en la base de datos.

```
campo = models.DateTimeField(auto_now=True)
```

Estos son algunos ejemplos pero existen muchas más opciones, podes verlas en la documentación de Django



Uso de filtros:

Formateo de fechas:

```
<p>Fecha actual: {{ fecha_actual|date:"d/m/Y" }}</p>
```

Ordenamiento de listas:

```
<ul>
    {% for item in lista|sort %}
        <li>{{ item }}</li>
    {% endfor %}
</ul>
```

Conversión a mayúsculas o minúsculas:

```
<p>Texto en mayúsculas: {{ texto|upper }}</p>
<p>Texto en minúsculas: {{ texto|lower }}</p>
```

Existen muchos más, te invitamos a investigar y probar todos estos códigos!

