Curso de Django

Módulo 2: Modelos y Bases de Datos.

Lcdo. Diego Medardo Saavedra García. Mgtr.

2023-07-20

Tabla de contenidos

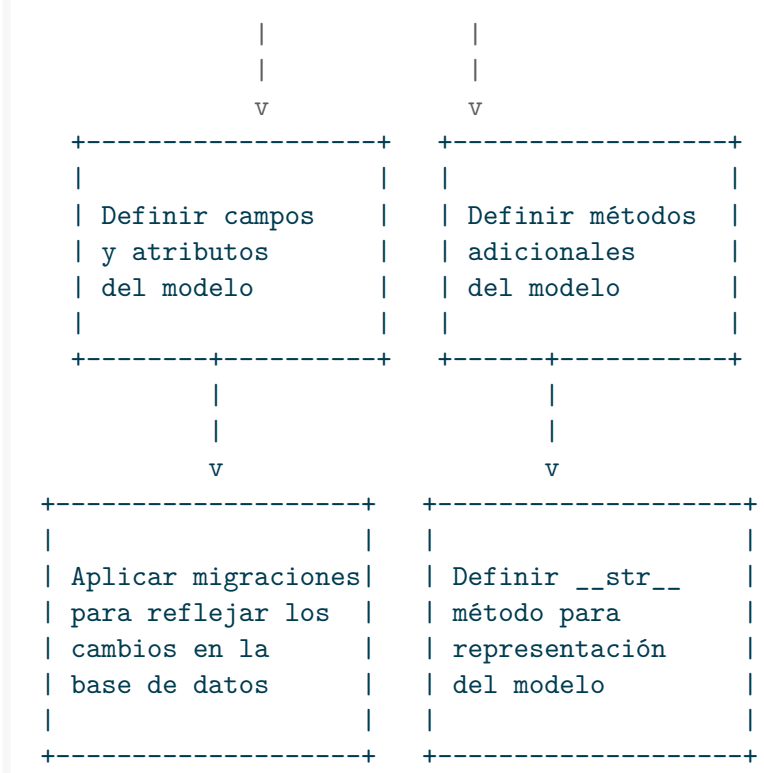
1 Módulo 2: Modelos y Bases de Datos.

1.1 Diseño de Modelos en Django

En este diagrama, se muestra el flujo de diseño de modelos en Django.

```
# Diseño de Modelos en Django

          +----------------------+
          |                      |
          | Definir el modelo    |
          | como una clase       |
          | Python que hereda de |
          | models.Model         |
          |                      |
          +----+------------+----+
```



1.2 1. Definimos el Modelo.

En Django, los modelos son la base para diseñar la estructura de la base de datos de nuestra aplicación web.

Cada modelo representa una tabla en la base de datos y define los campos que estarán presentes en dicha tabla.

Los modelos son definidos como clases Python que heredan de models.Model, lo que permite que Django maneje automáticamente la creación y gestión de la base de datos.

1.2.1 Creación de un Modelo.

Para crear un modelo en Django, primero definimos una clase Python que representa la tabla en la base de datos. Por ejemplo, si deseamos crear un modelo para representar las publicaciones en nuestro blog, podemos definirlo de la siguiente manera:

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

```
class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()
```

En este ejemplo, hemos definido el modelo Publicacion con tres campos: titulo,

contenido y fecha_publicacion.

Cada campo se representa mediante un atributo de la clase, donde: **models.CharField** representa un campo de texto, **models.TextField** representa un campo de texto más largo y **models.DateTimeField** representa una fecha y hora.

1.3 2. Definir Métodos del Modelo.

Además de los campos, también podemos definir métodos en el modelo para realizar acciones específicas o para dar formato a los datos. Por ejemplo, podríamos agregar un método que nos devuelva una representación más legible de la publicación:

```
from django.db import models

class Publicacion(models.Model):
    titulo = models.CharField(max_length=200)
    contenido = models.TextField()
    fecha_publicacion = models.DateTimeField()

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

En este caso, hemos definido el método **str** que se ejecutará cuando necesitemos obtener una representación de texto del objeto Publicacion. En este caso, hemos decidido que la representación será simplemente el título de la publicación.

Para poder probar los cambios que hemos realizado vamos a registrar nuestro modelo en el archivo **admin.py**

```
#  admin.py

from .models import Publicacion

admin.site.register(Publicacion)
```

1.4 3. Aplicar Migraciones.

Una vez que hemos definido nuestro modelo, necesitamos aplicar las migraciones para que los cambios se reflejen en la base de datos.

```
# Ejecutar en la terminal o consola
python manage.py makemigrations
python manage.py migrate
```

Con estos pasos, hemos diseñado nuestro modelo de Publicaciones en Django y aplicado las migraciones para crear la tabla correspondiente en la base de datos. Ahora estamos listos para utilizar nuestro modelo y almacenar datos en la base de datos.

Finalmente creamos un superusuario para acceder a la administración de nuestro proyecto.

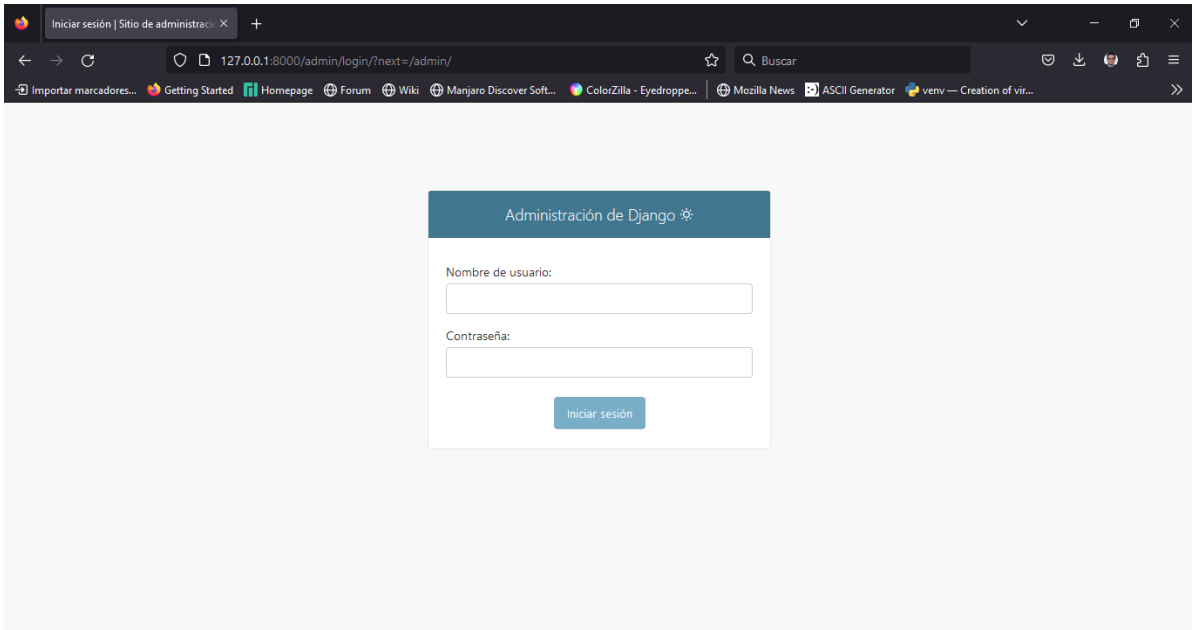
```
python manage.py createsuperuser
```

Llenamos un pequeño formulario que nos pide: nombre de usuario, correo electrónico (no obligatorio), password, repeat again password.

Y listo para poder acceder a la administración de nuestro proyecto nos dirigimos a la siguiente url

http://127.0.0.1:8000/admin

De forma gráfica ingresamos nuestro usuario y contraseña creado.



1.5 Ejemplo de las clases “Blog” y “Comentario” en Django:

```python
# En el archivo models.py de la aplicación "blog"

from django.db import models

class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    pub_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

class Comentario(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    author = models.CharField(max_length=50)
    content = models.TextField()
    pub_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comentario de {self.author} en {self.blog}"
```

En este ejemplo, hemos definido dos clases:

La clase “Blog”: Representa una publicación en el blog y tiene tres campos:

- “title” (título de la publicación),
- “content” (contenido de la publicación) y
- “pub_date” (fecha de publicación).

La fecha de publicación se establece automáticamente utilizando la función “auto_now_add=True”.

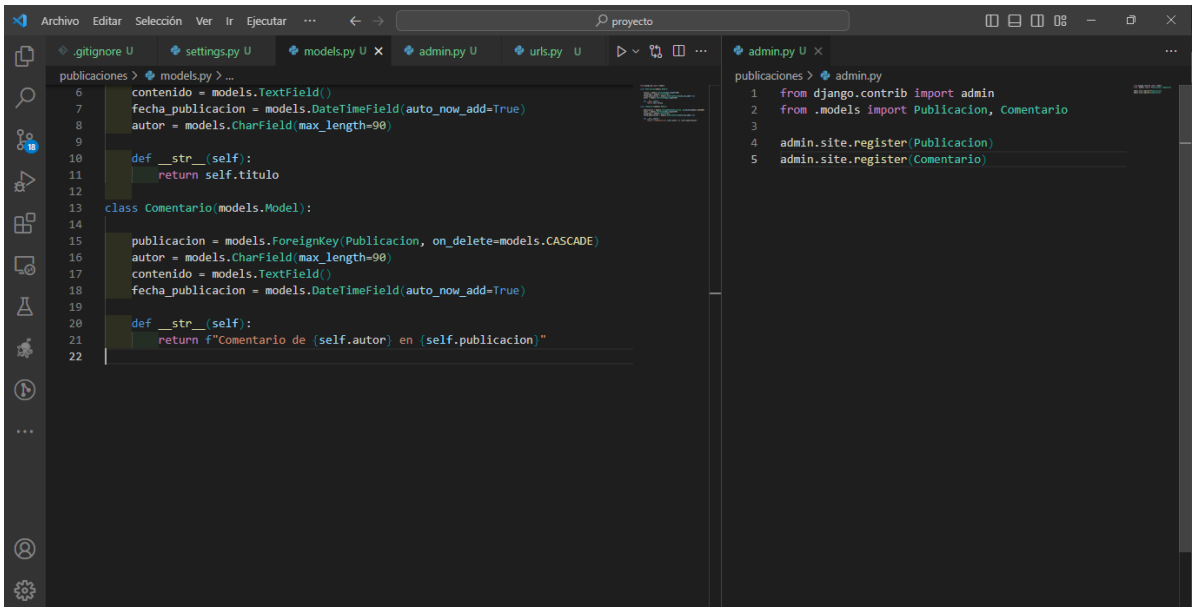
También hemos definido un método “str” para que al imprimir una instancia de la clase, se muestre el título de la publicación.

La clase “Comentario”: Representa un comentario en una publicación de blog específica y tiene cuatro campos:

- “blog” (clave externa que se relaciona con el blog al que pertenece el comentario),
- “author” (nombre del autor del comentario),
- “content” (contenido del comentario) y

- “pub_date” (fecha de publicación del comentario).

Al igual que en la clase “Blog”, hemos definido un método “str” para mostrar información útil al imprimir una instancia de la clase.



1.6 Migraciones de la Base de Datos

Las migraciones en Django son una forma de gestionar los cambios en la estructura de la base de datos de manera controlada y consistente. Representan los cambios en la estructura de la base de datos en forma de archivos Python y se utilizan para crear, modificar o eliminar tablas y campos.

Cuando definimos nuestros modelos en Django (como se mostró en el ejemplo de la clase “Blog” y “Comentario”), estamos describiendo la estructura de nuestras tablas en la base de datos. Sin embargo, antes de que estos modelos se puedan utilizar, Django necesita traducirlos en el lenguaje específico del motor de base de datos que estamos utilizando (por ejemplo, PostgreSQL, MySQL, SQLite, etc.).

Es aquí donde entran en juego las migraciones. Cuando creamos o modificamos modelos, Django genera automáticamente archivos de migración que contienen instrucciones para aplicar los cambios necesarios en la base de datos. Cada migración representa un paso en la evolución de la estructura de la base de datos.

Comandos para crear y aplicar migraciones:

```
python manage.py makemigrations
```

Este comando se utiliza para crear una nueva migración a partir de los cambios detectados en los modelos. Cuando ejecutamos este comando, Django analiza los modelos definidos en nuestra aplicación y compara la estructura actual con la estructura de la última migración aplicada. Luego, genera una nueva migración que contiene las instrucciones para llevar la base de datos a su estado actual.

```
python manage.py migrate
```

Una vez que hemos creado una o varias migraciones, utilizamos este comando para aplicar esas migraciones pendientes y modificar la base de datos de acuerdo con los cambios en los modelos. Django realiza las operaciones necesarias en la base de datos para reflejar la estructura actual de los modelos definidos en nuestra aplicación.

Es importante ejecutar estos comandos cada vez que realizamos cambios en los modelos para mantener la coherencia entre la estructura de la base de datos y la estructura definida en los modelos de Django.