

Capítulo 3
Modelos

1. Introducción a los modelos

1.1 Django y el modelo de datos

Django es un framework para el desarrollo de aplicaciones diseñadas sobre una base de datos relacional. Cada una de estas palabras es importante. Django está pensado para gestionar la parte técnica y facilitarle la creación rápida de una aplicación. Se basa en una base de datos relacional. Estas bases de datos son las más extendidas; existen desde hace mucho tiempo, son muy estables y ofrecen un gran rendimiento. Podemos citar PostgreSQL, apreciada por su rendimiento y su solidez, o MariaDB, valorada por su facilidad de uso y su popularidad. Ambas son extremadamente fiables.

Aunque Django es versátil y puede funcionar sobre MongoDB, por ejemplo, una base de datos orientada a documentos, está realmente diseñado y resulta eficaz para trabajar con una base de datos relacional.

Desarrollar una aplicación consiste, por tanto, simplemente en gestionar su modelo de datos y utilizarlo para gestionar funcionalidades o visualizaciones.

El diseño del modelo de datos es la principal preocupación del desarrollador de Django, muy por delante de saber cómo utilizar esos datos y tener que presentarlos en diversos formatos.

1.2 ¿Qué es un modelo?

En el modelado de bases de datos, se habla de entidades y relaciones. Una entidad puede ser, por ejemplo, un producto, un pedido, una entrega o incluso una persona, un rol o una misión. La relación permite vincular entidades. Podemos decidir que una persona tenga un solo rol, que una persona tenga entre 0 y varias misiones, pero que una misión sea realizada por una y solo una persona; que un pedido tenga entre uno y varios productos, pero que un producto pueda aparecer en entre 0 y n pedidos.

Es definiendo con precisión estas reglas como podremos modelar cómo se gestionarán los datos, cómo se validarán y cómo se presentarán.

Cada entidad puede tener varios campos que la describan. Cada campo de una entidad debe referirse a la propia entidad. Por ejemplo, una persona tiene un apellido, un nombre, una fecha de nacimiento… Para cada campo, se puede decidir si es obligatorio, si tiene límites (por ejemplo, un máximo de 50 letras para el nombre, una fecha de nacimiento comprendida entre 1950 y 2000, por ejemplo), etc.

Cada asociación también puede tener datos propios. Por ejemplo, la asociación entre el pedido y el producto puede incluir el número de productos pedidos. Pero también puede incluir el precio, si se decide que este puede variar de un pedido a otro de forma aleatoria.

También se podría optar por utilizar una entidad de lista de precios para gestionar precios diferentes según los tipos de clientes, o bien considerar que el precio es una propiedad del pedido o del producto y que siempre es fijo para un producto determinado.

Una vez más, para todas estas opciones, no hay una respuesta correcta a primera vista. Es necesario comprender cómo funcionan las cosas en el mundo real, allí donde se utilizará la aplicación, para diseñar un modelo de datos que permita representar la realidad.

Si bien comprender las herramientas de Django es relativamente sencillo –solo requiere práctica–, diseñar el modelo de datos de forma que se adapte realmente a las necesidades es una tarea muy importante a la que habrá que dedicar mucho tiempo.

© Editions ENI - All rights reserved

1.2.1 Ejemplo del tutorial de Django

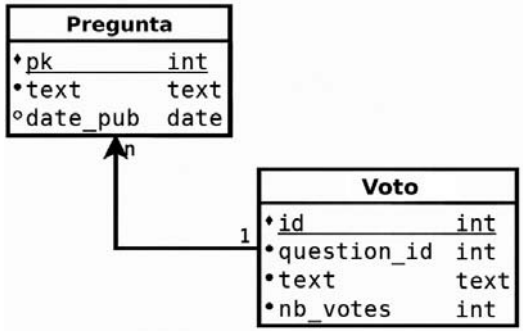
Por ejemplo, el tutorial de Django (que es excelente y que se recomienda realizar al menos una vez si se es principiante) tiene como objetivo describir una encuesta. Una encuesta es una lista de preguntas y cada pregunta puede tener varias opciones posibles.

Por lo tanto, tenemos dos modelos: el modelo **pregunta** y el modelo **opción**.

En el modelo **pregunta** habrá simplemente un texto que contenga la pregunta, seguido de toda una serie de propiedades que se pueden añadir y que están relacionadas exclusivamente con la pregunta. En el ejemplo del tutorial, una fecha de publicación.

En el modelo de **opción**, tenemos el texto que describe la opción, pero también el enlace al modelo de **pregunta**, ya que esa opción es una de las respuestas posibles a la pregunta. En efecto, cada opción está vinculada a una y solo una pregunta, pero una pregunta puede tener técnicamente entre cero y varias opciones (y, funcionalmente, al menos dos).

También hay un campo «votes» que permite contar el número de votos para la opción en cuestión.



Ejemplo de esquema de base de datos posible para el tutorial de Django

Sería posible evitar que hubiera dos preguntas idénticas añadiendo una restricción de unicidad en el campo `text`. También sería posible evitar que hubiera dos opciones idénticas para una misma pregunta añadiendo una restricción de unicidad en el par de campos (`question_id`, `text`).

En función de las necesidades de la aplicación, se podría haber concebido algo más complejo, por ejemplo, gestionar los votos en un modelo aparte, para realizar un seguimiento de los votos, vincularlos a una cuenta de usuario e impedir que se vote dos veces, o simplemente para guardar el momento en que se realizó el voto y así poder disponer de estadísticas sobre la evolución de los votos a lo largo del tiempo.

La idea que hay que recordar es que cada detalle de un objeto debe describirse mediante su modelo y que las funcionalidades que se vayan a implementar determinarán el número de modelos que hay que crear, su complejidad y los campos que deben incluirse en ellos.

1.2.2 Ejemplo de una biblioteca de libros

Tomemos otro ejemplo: queremos gestionar una biblioteca de libros. Podríamos imaginar gestionar todo con un único modelo **libro** que contuviera el título del libro, el nombre de su autor, pero también campos de texto para indicar su formato (bolsillo, cómic…) o incluso el género (novela negra, fantasía…).

Si hacemos esto, a la hora de introducir los datos, habrá que diseñar un formulario con solo campos de texto. Si queremos responder a preguntas como: ¿Qué libros ha escrito tal autor? o ¿Qué libros pertenecen a tal género?, es posible, pero no está estructurado.

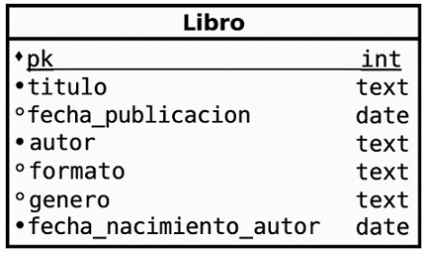
En efecto, repetir el nombre del género, por ejemplo, en cada entrada es problemático, ya que se repite la misma información muchas veces y se corre el riesgo de cometer un error al introducirla.

Además, cuando se empieza, a menudo se tiene la tentación de poner toda la información en el mismo sitio. Por ejemplo, se podría añadir un campo `fecha_nacimiento_autor` en el modelo «libro» para disponer también de esta información, ya que se desea mostrarla en la página que presenta el libro.

© Editions ENI - All rights reserved

Es una mala idea, ya que esta información pertenece al autor y no al libro en sí. Por lo tanto, no hay que dudar en crear más modelos, cada uno con su propia coherencia de datos, en lugar de intentar crear pocos modelos pero muy recargados.

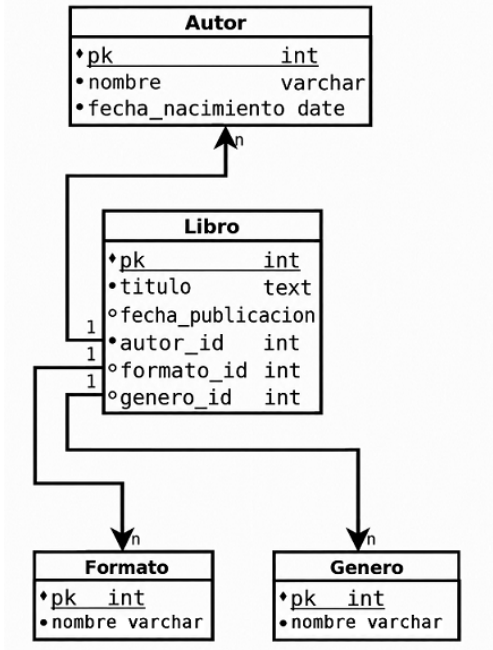
En efecto, con este tipo de modelos, se multiplica la información idéntica y se corre el riesgo de tener dos fechas de nacimiento diferentes para un mismo autor.



Ejemplo de modelo mal diseñado

Disponer de un modelo específico para gestionar los datos propios del autor permite evitar estos escollos, y es algo importante que hay que hacer, aunque el modelo no contenga nada más que el nombre del autor.

Disponer de un modelo para gestionar los formatos y otro para los géneros permitirá predefinir estos datos, con lo que, al introducir un libro, ya no se tratará de un campo libre, sino de una lista desplegable, lo que elimina cualquier ambigüedad y cualquier riesgo de error al introducir los datos.



Ejemplo de un modelo mejor diseñado

1.3 Concepto de app en Django

En la semántica de Django, el proyecto es el conjunto del código que va a crear. Contiene el paquete Python del proyecto que hemos detallado, en el que se encuentra un paquete con el mismo nombre que el proyecto y que contiene, entre otros, el archivo settings.py. El proyecto también contiene el directorio static, en el que se almacenarán todos los archivos de imagen, las hojas de estilo o el JavaScript que vaya a utilizar.

El código propiamente dicho se encontrará en aplicaciones (o apps de Django). Por lo tanto, Django le propone dividir su código en varias aplicaciones para facilitar su mantenimiento. Depende de usted dividirlas de forma adecuada e intentar que ninguna aplicación sea demasiado grande.

© Editions ENI - All rights reserved

Para crear una nueva aplicación, hay que abrir, en el terminal, una consola bash dentro del contenedor Docker de Django:

```
$ make bash
```

Una vez dentro del contenedor, puede escribir el siguiente comando:

```
$ poetry run python ./manage.py --help
```

Esto permite ver las diferentes acciones que se pueden realizar desde el punto de entrada de Django. En nuestro caso, queremos utilizar `startapp`. A continuación le mostramos cómo ver las opciones disponibles de este comando:

```
$ poetry run python ./manage.py startapp --help
```

Por nuestra parte, vamos a utilizar su forma más básica:

```
$ poetry run python ./manage.py startapp app
```

Acabamos de crear, por tanto, una aplicación llamada **app** que contiene varios elementos. Siguiendo un orden cronológico, el primer elemento es el archivo models.py, en el que vamos a escribir nuestros modelos. Este será el tema de este capítulo, pero también del capítulo Conceptos avanzados sobre los modelos, donde profundizaremos en él.

A continuación, tenemos el archivo admin.py, en el que vamos a declarar nuestra interfaz de administración. Abordaremos este tema en los capítulos CRUD y Gestión avanzada de la interfaz de administración.

El archivo views.py contendrá todas las vistas, que se tratan en los capítulos Introducción a las vistas y Conceptos avanzados sobre las vistas, así como los endpoints de la API, que se tratan en el capítulo REST y Swagger.

Por último, el archivo tests.py, que contendrá todas las pruebas unitarias, que se tratan en el capítulo Pruebas.

También existe un archivo apps.py que permite configurar los metadatos de la aplicación de Django. Este se puede personalizar para adaptar la visualización del nombre de la aplicación y hacer que sea traducible.

El archivo tendrá entonces este aspecto:

```
from django.apps import AppConfig

from django.utils.translation import gettext_lazy as gettext


class AppConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'app'
    verbose_name = gettext("Mi aplicación")
```

Para terminar, cabe destacar el directorio «migrations», que contendrá todas las instrucciones que se deben dar a la base de datos para pasar de un estado a otro (y así seguir los cambios que se vayan realizando en el modelo, creando, modificando o eliminando tablas o columnas, por ejemplo, para que la base de datos se ajuste al modelo).

Para terminar nuestro trabajo, ahora hay que modificar la configuración del proyecto para añadir la aplicación que acabamos de crear. Para ello, hay que abrir el archivo settings.py e identificar la constante `INSTALLED_APPS` para añadirle lo siguiente:

```
INSTALLED_APPS = [
    'app.apps.AppConfig',
    'django.contrib.admin',
    ...
]
```

Para ver el código correspondiente a esta sección: git diff v1.0.6..v1.1.0.

1.4 Creación de un modelo

Para crear un modelo, hay que ir al archivo models.py y crear una clase que herede del modelo base de Django.

En el ejemplo principal de este libro, vamos a gestionar juegos.

```
class Game(models.Model):
    pass
```

© Editions ENI - All rights reserved

En este momento, hemos creado un modelo que representa un juego. Este solo tiene un campo (heredado de la clase padre), que es su clave primaria (denominada `id`).

Por defecto, esto creará una tabla llamada `app_game` y el modelo tomará el nombre `game`, que se pluralizará automáticamente añadiendo una `s` al final.

Es posible cambiar esto modificando los metadatos de nuestro modelo:

```
from django.utils.translation import gettext_lazy as gettext
class Game(models.Model):
    class Meta:
        verbose_name = gettext("game")
        verbose_name_plural = gettext("games")
```

Aquí hemos definido nuestro modelo; ahora solo queda definir su contenido.

1.5 Migraciones

Ya tenemos definido nuestro modelo, pero aún no se ha creado nada en la base de datos. Para ello, hay que generar migraciones.

```
$ make bash
$ poetry run python ./manage.py makemigrations
```

Esto creará un nuevo archivo en el directorio «migrations», llamado «0001_initial.py», que contiene la descripción de las operaciones que hay que realizar para crear una tabla que se corresponda con nuestro modelo.

Para aplicar esta migración, ahora hay que utilizar el siguiente comando:

```
$ poetry run python ./manage.py migrate
```

La base de datos ya se ajusta a los modelos que se han declarado.

Cabe señalar que cada vez que se cree un nuevo modelo, se modifique uno o se elimine, será necesario crear nuevas migraciones. Estas se numerarán automáticamente y, en ocasiones, llevarán un sufijo con la fecha.

La adición de un sufijo con la fecha solo se produce cuando se crea una migración vacía:

```
$ poetry run python ./manage.py makemigrations --empty app_name
```

Crear una migración de este tipo permite añadirle elementos manualmente, como RunPython o RunSQL.

Cuando se realizan muchos cambios, es fácil generar un gran número de migraciones. Existen herramientas que permiten agrupar las migraciones (*squash*).

```
$ poetry run python ./manage.py squashmigrations
$ poetry run python ./manage.py delete_squashed_migrations
```

El primer comando forma parte de los comandos de Django, mientras que el segundo pertenece a Django Extensions.

Le recomendamos que utilice estos comandos para generar migraciones tras cada modificación de los modelos y que las agrupe periódicamente, con el fin de probarlas y comprobar los cambios introducidos en los archivos de migración.

Para ver el código relacionado con esta sección: git diff v1.1.0..v1.1.1.

1.6 ¿Qué se puede hacer con un modelo?

La parte que vamos a descubrir aquí es el ORM de Django, donde ORM significa object-relational mapping (mapeo objeto-relacional), en concreto su parte de alto nivel.

La idea es crear clases de Python que describan entidades SQL y, a continuación, crear instancias de ellas.

La función del ORM, además de permitirnos escribir estas clases, es convertir las instrucciones de Python relativas a los objetos en consultas SQL. En otras palabras, manipular instancias de modelo generará consultas SQL automáticamente, lo que permitirá guardar los cambios de forma persistente en la base de datos.

Aprender a manipular estas instancias será el objetivo del siguiente capítulo.

© Editions ENI - All rights reserved