



FIG. 2.
PONY "MAGIC"

Sistemas Web Desconectados

Release 1

van Haaster, Diego Marcos; Defossé, Nahuel

August 12, 2009

Índice general

1. Tecnologías del servidor	3
1.1. CGI	3
1.2. WSGI	3
1.3. Lenguajes interpretados	3
1.4. Frameworks	3
1.5. Django	6
2. Glosario	13
3. Indices, glosario y tablas	15
A. Referencia de la definición de modelos	17
A.1. Campos	17
A.2. Opciones para todos los campos	22
A.3. Relaciones	25
A.4. Opciones de los Metadatos del Modelo	29
A.5. Managers	32
A.6. Métodos de Modelo	35
A.7. Opciones del Administrador	38
Índice	45

Índice:

Tecnologías del servidor

1.1 CGI

CGI es bla

1.2 WSGI

WSGI es CGI para Python.

1.3 Lenguajes interpretados

PHP Ruby Python

Python es un lenguaje interpretado.

1.4 Frameworks

Un framework web es una abstracción en la cual un código común que provee una funcionalidad genérica puede ser personalizada por el programador de manera selectiva para brindar una funcionalidad específica.

Se suele decir que los frameworks son similares a las bibliotecas de software (a veces llamadas librerías) dado que proveen abstracciones reusables de código a las cuales se accede mediante una API bien definida.

Sin embargo, podemos encontrar ciertas características que diferencian al framework de una librería o aplicaciones normales de usuario:

- **Inversion de control** Al contrario que las bibliotecas en las aplicaciones de usuario, en un framework, el flujo de control no es manejado por el llamador, sino por el framework. Es decir, cuando se utilizan bibliotecas o programas de usuario como soporte para brindar funcionalidad, estas son llamadas o invocados en el código de aplicación principal que es definido por el usuario. En un framework, el flujo de control principal está definido por el framework.

- **Comportamiento por defecto definido** Un framework tiene un comportamiento por defecto definido. En cada componente del framework, existe un comportamiento genérico con alguna utilidad, que puede ser redefinido con funcionalidad del usuario.
- **Extensibilidad** Un framework suele ser extendido por el usuario mediante redefinición o especialización para proveer una funcionalidad específica.
- **No modificabilidad del código del framework** En general no se permite la modificación del código del framework. Los programadores pueden extender el framework, pero no modificar su código.

Los diseñadores de frameworks tienen como objetivo facilitar el desarrollo de software, permitiendo a los programadores enfocarse en cumplimentar los requerimientos del análisis y diseño, en vez de dedicar tiempo a resolver los detalles comunes de bajo nivel. En general la utilización de un framework reduce el tiempo de desarrollo.

Por ejemplo, en un equipo donde se utiliza un framework web para desarrollar un sitio de banca electrónica, los desarrolladores pueden enfocarse en la lógica necesaria para realizar las extracciones de dinero, en vez de la mecánica para preservar el estado entre las peticiones del navegador.

Sin embargo, se suele argumentar que los frameworks pueden ser una carga, debido a la complejidad de sus APIs o la incertidumbre que genera la existencia de varios frameworks para un mismo tipo de aplicación. A pesar de tener como objetivo estandarizar y reducir el tiempo de desarrollo, el aprendizaje de un framework suele requerir tiempo extra en el desarrollo, aunque posteriores desarrollos pueden verse beneficiados de este aprendizaje inicial.

1.4.1 Model View Controller

Antes de profundizar en más código, tomémonos un momento para considerar el diseño global de una aplicación Web Django impulsada por bases de datos.

Como mencionamos en los capítulos anteriores, Django fue diseñado para promover el acoplamiento débil y la estricta separación entre las piezas de una aplicación. Si sigues esta filosofía, es fácil hacer cambios en un lugar particular de la aplicación sin afectar otras piezas. En las funciones de vista, por ejemplo, discutimos la importancia de separar la lógica de negocios de la lógica de presentación usando un sistema de plantillas. Con la capa de la base de datos, aplicamos esa misma filosofía para el acceso lógico a los datos.

Estas tres piezas juntas – la lógica de acceso a la base de datos, la lógica de negocios, y la lógica de presentación – comprenden un concepto que a veces es llamado el patrón de arquitectura de software *Modelo-Vista-Controlador* (MVC). En este patrón, el “Modelo” hace referencia al acceso a la capa de datos, la “Vista” se refiere a la parte del sistema que selecciona qué mostrar y cómo mostrarlo, y el “Controlador” implica la parte del sistema que decide qué vista usar, dependiendo de la entrada del usuario, accediendo al modelo si es necesario.

Django sigue el patrón MVC tan al pie de la letra que puede ser llamado un framework MVC. Someramente, la M, V y C se separan en Django de la siguiente manera:

- *M*, la porción de acceso a la base de datos, es manejada por la capa de la base de datos de Django, la cual describiremos en este capítulo.
- *V*, la porción que selecciona qué datos mostrar y cómo mostrarlos, es manejada por la vista y las plantillas.
- *C*, la porción que delega a la vista dependiendo de la entrada del usuario, es manejada por el framework mismo siguiendo tu `URLconf` y llamando a la función apropiada de Python para la URL obtenida.

Debido a que la “C” es manejada por el mismo framework y la parte más emocionante se produce en los modelos, las plantillas y las vistas, Django es conocido como un *Framework MTV*. En el patrón de diseño MTV,

- *M* significa “Model” (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.
- *T* significa “Template” (Plantilla), la capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web o otro tipo de documento.

- V significa “View” (Vista), la capa de la lógica de negocios. Esta capa contiene la lógica que accede al modelo y la delega a la plantilla apropiada; puedes pensar en esto como un puente entre el modelos y las plantillas.

Si estás familiarizado con otros frameworks de desarrollo web MVC, como Ruby on Rails, quizás consideres que las vistas de Django pueden ser el “controlador” y las plantillas de Django pueden ser la “vista”. Esto es una confusión desafortunada a raíz de las diferentes interpretaciones de MVC. En la interpretación de Django de MVC, la “vista” describe los datos que son presentados al usuario; no necesariamente el *cómo* se mostrarán, pero si *cuáles* datos son presentados. En contraste, Ruby on Rails y frameworks similares sugieren que el trabajo del controlador incluya la decisión de cuales datos son presentados al usuario, mientras que la vista sea estrictamente el *cómo* serán presentados y no *cuáles*.

Ninguna de las interpretaciones es más “correcta” que otras. Lo importante es entender los conceptos subyacentes.

1.4.2 Mapeador Objeto-Relacional

En las aplicaciones web modernas, la lógica arbitraria a menudo implica interactuar con una base de datos. Detrás de escena, un *sitio web impulsado por una base de datos* se conecta a un servidor de base de datos, recupera algunos datos de esta, y los muestra con un formato agradable en una página web. O, del mismo modo, el sitio puede proporcionar funcionalidad que permita a los visitantes del sitio poblar la base de datos por su propia cuenta.

Muchos sitios web más complejos proporcionan alguna combinación de las dos. Amazon.com, por ejemplo, es un gran ejemplo de un sitio que maneja una base de datos. Cada página de un producto es esencialmente una consulta a la base de datos de productos de Amazon formateada en HTML, y cuando envías una opinión de cliente (*customer review*), esta es insertada en la base de datos de opiniones.

Así como en el ‘Capítulo 3’_ detallamos la manera “tonta” de producir una salida con la vista (codificando *en duro*) el texto directamente dentro de la vista), hay una manera “tonta” de recuperar datos desde la base de datos en una vista. Esto es simple: sólo usa una biblioteca de Python existente para ejecutar una consulta SQL y haz algo con los resultados.

En este ejemplo de vista, usamos la biblioteca MySQLdb (disponible en <http://www.djangoproject.com/r/python-mysql/>) para conectarnos a una base de datos de MySQL, recuperar algunos registros, y alimentar con ellos una plantilla para mostrar una página web:

```
from django.shortcuts import render_to_response
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render_to_response('book_list.html', {'names': names})
```

Este enfoque funciona, pero deberían hacerse evidentes inmediatamente algunos problemas:

- Estamos codificando *en duro* (*hard-coding*) los parámetros de la conexión a la base de datos. Lo ideal sería que esos parámetros se guardasen en la configuración de Django.
- Tenemos que escribir una cantidad de código estereotípico: crear una conexión, un cursor, ejecutar una sentencia, y cerrar la conexión. Lo ideal sería que todo lo que tuviéramos que hacer fuera especificar los resultados que queremos.
- Nos ata a MySQL. Si, en el camino, cambiamos de MySQL a PostgreSQL, tenemos que usar un adaptador de base de datos diferente (por ej. *psycopg* en vez de *MySQLdb*), alterar los parámetros de conexión y – dependiendo de la naturaleza de las sentencia de SQL – posiblemente reescribir el SQL. La idea es que el

servidor de base de datos que usemos esté abstraído, entonces el pasarnos a otro servidor podría significar realizar un cambio en un único lugar.

Rails

Symfony

1.5 Django

Acá tenemos que justificar por que django

Django es un framework web escrito en Python el cual sigue vagamente el concepto de Modelo Vista Controlador. Ideado inicialmente como un administrador de contenido para varios sitios de noticias, los desarrolladores encontraron que su CMS era lo suficientemente genérico como para curbir un ámbito más aplio de aplicaciones.

En honor al músico Django Reinhart, fue liberado el código base bajo la licencia *BSD* en Julio del 2005 como Django Web Framework. El slogan del framework fue “Django, El framework para perfeccionistas con fechas límites”¹.

En junio del 2008 fue anunciada la cereación de la Django Software Foundation, la cual se hace cargo hasta la fecha del desarrollo y mantenimiento.

Los orígenes de Django en la administración de páginas de noticias son evidentes en su diseño, ya que proporciona una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos. Por ejemplo, en lugar de requerir que los desarrolladores escriban controladores y vistas para las áreas de administración de la página, Django proporciona una aplicación incorporada para administrar los contenidos que puede incluirse como parte de cualquier proyecto; la aplicación administrativa permite la creación, actualización y eliminación de objetos de contenido, llevando un registro de todas las acciones realizadas sobre cada uno (sistema de logging o bitácora), y proporciona una interfaz para administrar los usuarios y los grupos de usuarios (incluyendo una asignación detallada de permisos).

Con Django también se distribuyen aplicaciones que proporcionan un sistema de comentarios, herramientas para syndicar contenido via RSS y/o Atom, “páginas planas” que permiten gestionar páginas de contenido sin necesidad de escribir controladores o vistas para esas páginas, y un sistema de redirección de URLs.

Django como framework de desarrollo consiste en un conjunto de utilidades de consola que permiten crear y manipular proyectos y aplicaciones.

1.5.1 Estructuración de un proyecto en Django

Durante la instalación del framework en el sistema del desarrollador, se añade al PATH un comando con el nombre `django-admin.py`. Mediante este comando se crean proyectos y se los administra.

Un proyecto se crea mediante la siguiente orden:

```
$ django-admin.py startproject mi_proyecto # Crea el proyecto mi_proyecto
```

Un proyecto es un paquete Python que contiene 3 módulos:

- **manage.py** Interfase de consola para la ejecución de comandos
- **urls.py** Mapeo de URLs en vistas (funciones)
- **settings.py** Configuración de la base de datos, directorios de plantillas, etc.

En el ejemplo anterior, un listado gerárquico del sistema de archivos mostraría la siguiente estructura:

¹ Del ingles “The Web framework for perfectionists with deadlines”

```
mi_proyecto
|-- __init__.py
|-- manage.py
|-- settings.py
'-- urls.py
```

El proyecto funciona como un contenedor de aplicaciones que se rigen bajo la misma base de datos, los mismos templates, las mismas clases de middleware entre otros parámetros.

Analicemos a continuación la función de cada uno de estos 3 módulos.

Módulo settings

Este módulo define la configuración del proyecto, siendo sus atributos principales la configuración de la base de datos a utilizar, la ruta en la cual se encuentran los medios estáticos, cuál es el nombre del archivo raíz de urls (generalmente `urls.py`). Otros atributos son las clases middleware, las rutas de los templates, el idioma para las aplicaciones que soportan *i18n*, etc.

Al ser un módulo del lenguaje python, la configuración se puede editar muy fácilmente a diferencia de configuraciones realizadas en XML, además de contar con la ventaja de poder configurar en caliente algunos parámetros que así lo requieran.

Un parámetro fundamental es la lista denominada `INSTALLED_APPS` que contiene los nombres de las aplicaciones instaladas en el proyecto.

Módulo manage

Esta es la interfase con el framework. Este módulo es un script ejecutable, que recibe como primer argumento un nombre de comando de django.

Los comandos de django permiten, permiten entre otras cosas:

- **startapp** <nombre de aplicación> Crear una aplicación
- **runserver** Correr el proyecto en un servidor de desarrollo.
- **syncdb** Generar las tablas en la base de datos de las aplicaciones instaladas

El resultado de el comando **startapp** en el ejemplo anterior genera el siguiente resultado:

```
mi_proyecto
|-- mi_aplicacion
|   |-- __init__.py
|   |-- models.py
|   |-- tests.py
|   '-- views.py
|-- __init__.py
|-- manage.py
|-- settings.py
'-- urls.py
```

Módulo urls

Este nombre de módulo aparece a nivel proyecto, pero también puede aparecer a nivel aplicación. Su misión es definir las asociaciones entre URLs y vistas, de manera de que el framework sepa que vista utilizar en función de la URL que está requiriendo el cliente. Las URLs se escriben mediante expresiones regulares. Se suele aprovechar la posibilidad

del módulo de expresiones regulares del lenguaje python, que permite recuperar grupos nombrados (en contraposición al enfoque ordinal tradicional).

La asociación url-vistas se define en el módulo bajo el nombre *urlpatterns*. También es posible derivar el tratado de una parte de la expresión regular a otro módulo de urls. Generalmente esto ocurre cuando se desea delegar el tratado de las urls a una aplicación particular.

Ej: Derivar el tratado de todo lo que comience con la cadena *personas* a al módulo de urls de la aplicación *personas*.

```
(r'^personas', include('mi_proyecto.personas.urls'))
```

1.5.2 Estructura de una aplicación Django

Una aplicación es un paquete python que consta de un módulo *models* y un módulo *views*.

Módulo *models*

Cada vez que se crea una aplicación, se genera un módulo *models.py*, en el cual se le permite al programador definir modelos de objetos, que luego son transformados en tablas relacionales ².

Módulo *views*

Cada aplicación posee un módulo *views*, donde se definen las funciones que atienden al cliente y son activadas gracias a el mapeo definido en el módulo *urls* del proyecto o de la aplicación.

Las funciones que trabajan como vistas deben recibir como primer parámetro el request y opcionalmente parámetros que pueden ser recuperados del mapeo de urls.

Dentro del módulo de urls

```
# Tras un mapeo como el siguiente
(r'^persona/(?P<id_persona>\d)/$', mi_vista)
# la vista se define como
def mi_vista(request, id_persona):
    persona = Personas.objects.get(id = id_persona)
    datos = {'persona': persona, }
    return render_to_response('plantilla.html', datos)
```

El ciclo de una petición

Cada vez que un browser realiza una petición a un proyecto desarrollado en django, la petición HTTP pasa por varias capas.

Inicialmente atraviesa los Middlewares, en la cual, el middleware de Request, empaqueta las variables del request en una instancia de la clase Request.

Luego de atravesar los middlewares de request, mediante las definiciones de URLs, se selecciona la vista a ser ejecutada.

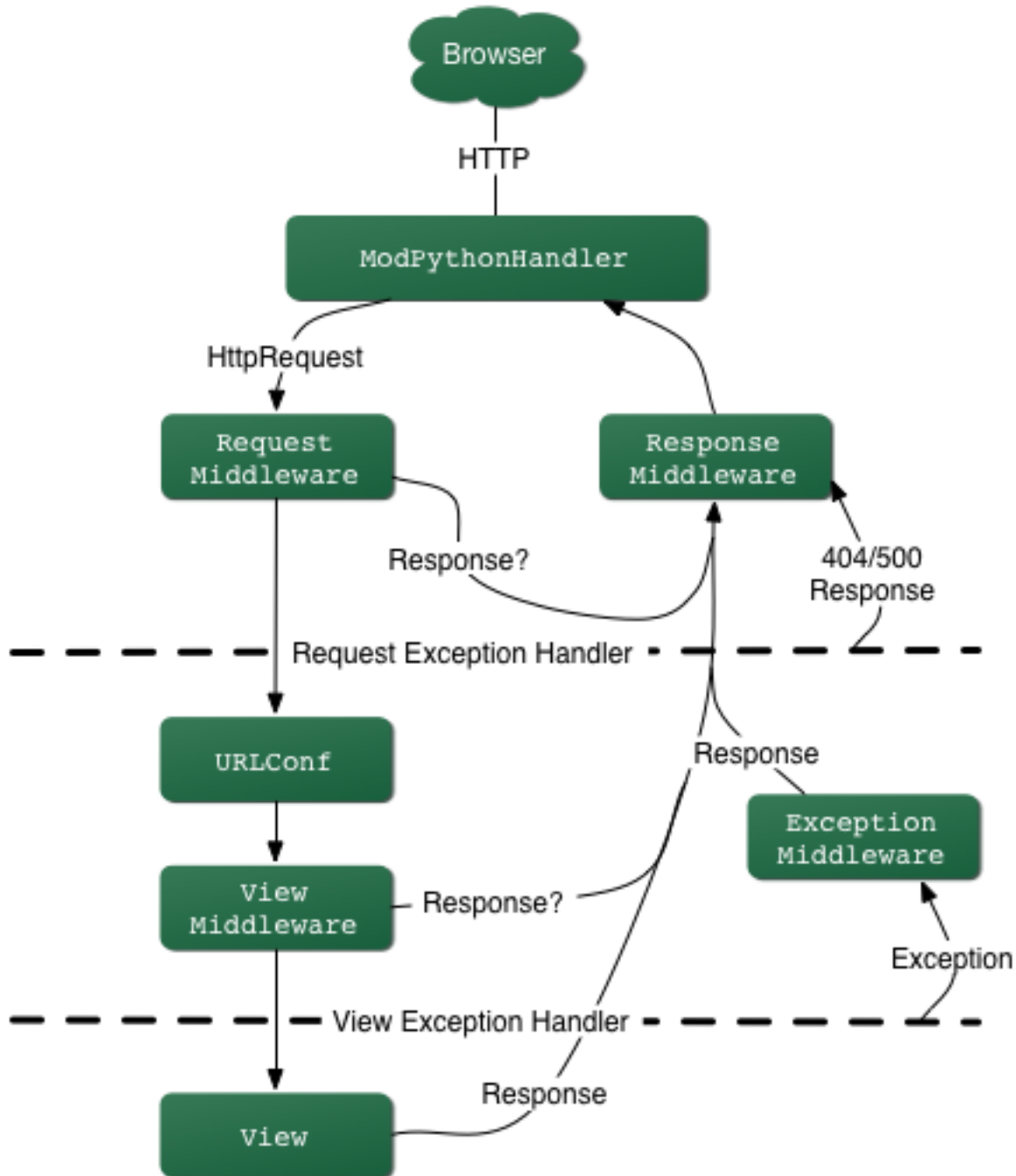
Una vista es una función que recibe como primer argumento el request y opcionalmente una serie de parámetros que puede recuperar de la propia URL.

² Mediante el comando *syncdb* del módulo *manage* del proyecto

Dentro de la vista se suelen hacer llamadas al ORM, para realizar consultas sobre la base de datos. Una vez que la vista a completado la lógica, genera un mapeo que es transferido a la capa de templates.

El template rellena sus comodines en función de los valores del mapeo que le entrega la vista. Un template puede poseer lógica muy básica (bifurcaciones, bucles de repetición, formateo de datos, etc).

El template se entrega como un HttpResponse. La responsabilidad de la vista es entregar una instancia de esta clase.



Interactuar con una base de datos

Primero, necesitamos tener en cuenta algunas configuraciones iniciales: necesitamos indicarle a Django qué servidor de base de datos usar y cómo conectarse con el mismo.

Asumimos que haz configurado un servidor de base de datos, lo has activado, y has creado una base de datos en este (por ej. usando la sentencia `CREATE DATABASE`). SQLite es un caso especial; es este caso, no hay que crear una base de datos, porque SQLite usa un archivo autónomo sobre el sistema de archivos para guardar los datos.

Como con `TEMPLATE_DIRS` en los capítulos anteriores, la configuración de la base de datos se encuentra en el archivo de configuración de Django, llamado, por omisión, `settings.py`. Edita este archivo y busca las opciones de la base de datos:

Hay pocas reglas estrictas sobre cómo encajar el código Django en este esquema; es flexible. Si estás construyendo un sitio web simple, quizás uses una sola aplicación. Si estás construyendo un sitio web complejo con varias piezas que no se relacionan entre sí, tales como un sistema de comercio electrónico o un foro, probablemente quieras dividir esto en aplicaciones para que te sea posible reusar estas individualmente en un futuro.

Es más, no necesariamente debes crear aplicaciones en absoluto, como lo hace evidente la función de la vista del ejemplo que creamos antes en este libro. En estos casos, simplemente creamos un archivo llamado `views.py`, llenamos este con una función de vista, y apuntamos nuestra URLconf a esa función. No se necesitan “aplicaciones”.

No obstante, existe un requisito respecto a la convención de la aplicación: si estás usando la capa de base de datos de Django (modelos), debes crear una aplicación de Django. Los modelos deben vivir dentro de aplicaciones.

Dentro del directorio del proyecto `mysite` que creaste en el ‘**Capítulo 2**’, escribe este comando para crear una nueva aplicación llamada `books`:

```
python manage.py startapp books
```

Este comando no produce ninguna salida, pero crea un directorio `books` dentro del directorio `mysite`. Echemos un vistazo al contenido:

```
books/
  __init__.py
  models.py
  views.py
```

Esos archivos contendrán los modelos y las vistas para esta aplicación.

Echa un vistazo a `models.py` y `views.py` en tu editor de texto favorito. Ambos archivos están vacíos, excepto por la importación en `models.py`. Este es el espacio disponible para ser creativo con tu aplicación de Django.

1.5.3 Modelos

Un modelo de Django es una descripción de los datos en la base de datos, representada como código de Python. Esta es tu capa de datos – lo equivalente de tu sentencia `SQL CREATE TABLE` – excepto que están en Python en vez de SQL, e incluye más que sólo definición de columnas de la base de datos. Django usa un modelo para ejecutar código SQL detrás de las escenas y retornar estructuras de datos convenientes en Python representando las filas de tus tablas de la base de datos. Django también usa modelos para representar conceptos de alto nivel que no necesariamente pueden ser manejados por SQL.

Si estás familiarizado con base de datos, inmediatamente podría pensar, “¿No es redundante definir modelos de datos en Python y en SQL?” Django trabaja de este modo por varias razones:

- La introspección requiere ***overhead*** y es imperfecta. Con el objetivo de proveer una API conveniente de acceso a los datos, Django necesita conocer *de alguna forma* la capa de la base de datos, y hay dos formas de

lograr esto. La primera sería describir explícitamente los datos en Python, y la segunda sería la introspección de la base de datos en tiempo de ejecución para determinar el modelo de la base de datos.

La segunda forma parece clara, porque los metadatos sobre tus tablas vive en un único lugar, pero introduce algunos problemas. Primero, introspeccionar una base de datos en tiempo de ejecución obviamente requiere overhead. Si el framework tuviera que introspeccionar la base de datos cada vez que se procese una petición, o incluso cuando el servidor web sea inicializado, esto podría provocar un nivel de overhead inaceptable. (Mientras algunos creen que el nivel de overhead es aceptable, los desarrolladores de Django apuntan a quitar del framework tanto overhead como sea posible, y esta aproximación hace que Django sea más rápido que los frameworks competidores de alto nivel en mediciones de desempeño). Segundo, algunas bases de datos, notablemente viejas versiones de MySQL, no guardan suficiente metadatos para asegurarse una completa introspección.

- Escribir Python es divertido, y dejar todo en Python limita el número de veces que tu cerebro tiene que realizar un “cambio de contexto”. Si te mantienes en un solo entorno/mentalidad de programación tanto tiempo como sea posible, ayuda para la productividad. Teniendo que escribir SQL, luego Python, y luego SQL otra vez es perjudicial.
- Tener modelos de datos guardados como código en vez de en tu base de datos hace fácil dejar tus modelos bajo un control de versiones. De esta forma, puedes fácilmente dejar rastro de los cambios a tu capa de modelos.
- SQL permite sólo un cierto nivel de metadatos acerca de un **layout** de datos. La mayoría de sistemas de base de datos, por ejemplo, no provee un tipo de datos especializado para representar una dirección web o de email. Los modelos de Django sí. La ventaja de un tipo de datos de alto nivel es la alta productividad y la reusabilidad de código.
- SQL es inconsistente a través de distintas plataformas. Si estás redistribuyendo una aplicación web, por ejemplo, es mucho más pragmático distribuir un módulo de Python que describa tu capa de datos que separar conjuntos de sentencias `CREATE TABLE` para MySQL, PostgreSQL y SQLite.

Una contra de esta aproximación, sin embargo, es que es posible que el código Python quede fuera de sincronía respecto a lo que hay actualmente en la base. Si haces cambios en un modelo Django, necesitarás hacer los mismos cambios dentro de tu base de datos para mantenerla consistente con el modelo. Detallaremos algunas estrategias para manejar este problema más adelante en este capítulo.

Finalmente, Django incluye una utilidad que puede generar modelos haciendo introspección sobre una base de datos existente. Esto es útil para comenzar a trabajar rápidamente sobre datos heredados.

Los modelos son la fuente de información sobre los datos de la aplicación. Esencialmente están compuestos de campos y comportamiento propio de los datos almacenados. Generalmente, un modelo se corresponde con una tabla en la base de datos.

Dentro de un proyecto los modelos se definen por aplicación en el módulo `models.py`.

Un modelo es una clase Python que hereda de `django.db.models.Model` y cada atributo representa un campo requerido por el modelo de datos de la aplicación. Con esta información Django genera automáticamente una *API* de acceso a los datos en la base.

Este modelo de ejemplo define una `Persona` que encapsula los datos correspondientes al nombre y el apellido.

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length = 30)
    apellido = models.CharField(max_length = 30)
```

nombre y apellido son atributos de clase

```
CREATE TABLE miapp_persona (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
```

```
"apellido" varchar(30) NOT NULL  
);
```

1.5.4 Consultas

bala

1.5.5 Administradores de consultas

Estos objetos representan la interfase de comunicacion con la base de datos. Cada modelo tiene por lo menos un administrador para acceder a los datos almacenados.

Glosario

API [Application-Programming-Interface](#); conjunto de funciones y procedimientos (o métodos, si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

DOM [Document-Object-Model](#); interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos.

JSON [JavaScript-Object-Notation](#); formato ligero para el intercambio de datos.

RPC [Remote-Procedure-Call](#); es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.

field An attribute on a [model](#); a given field usually maps directly to a single database column.

generic view A higher-order [view](#) function that abstracts common idioms and patterns found in view development and abstracts them.

model Models store your application's data.

MTV hola

MVC [Model-view-controller](#); a software pattern.

project A Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.

property Also known as “managed attributes”, and a feature of Python since version 2.2. From [the property documentation](#):

Properties are a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls. [...] You could only do this by overriding `__getattr__` and `__setattr__`; but overriding `__setattr__` slows down all attribute assignments considerably, and overriding `__getattr__` is always a bit tricky to get right. Properties let you do this painlessly, without having to override `__getattr__` or `__setattr__`.

queryset An object representing some set of rows to be fetched from the database.

slug A short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs. For example, in a typical blog entry URL:

`http://www.djangoproject.com/weblog/2008/apr/12/spring/`

the last bit (`spring`) is the slug.

template A chunk of text that separates the presentation of a document from its data.

view A function responsible for rendering a page.

BSD ve ese de

i18n La internacionalización es el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin la necesidad de realizar cambios de ingeniería ni en el código. La localización es el proceso de adaptar el software para una región específica mediante la adición de componentes específicos de un locale y la traducción de los textos, por lo que también se le puede denominar regionalización. No obstante la traducción literal del inglés es la más extendida.

Indices, glosario y tablas

- *Índice*
- *Índice de Módulos*
- *Glosario*

Referencia de la definición de modelos

El *doff-modelos* se explica lo básico de la definición de modelos. Existe un enorme rango de opciones disponibles que no se han cubierto en otro lado. Este apéndice explica toda opción disponible en la definición de modelos.

A.1 Campos

La parte más importante de un modelo – y la única requerida – es la lista de campos de la base de datos que define.

Restricciones en el nombre de los campos

Existen solo dos restricciones en el nombre de los campos:

1. Un nombre de campo no puede ser una palabra reservada, porque eso ocasionaría un error de sintaxis, por ejemplo:

```
var Ejemplo = type('Ejemplo', [ models.Model ], {  
    var = new models.IntegerField() // 'var' es una palabra reservada!  
});
```

2. Un nombre de campo no puede contener dos o más guiones bajos consecutivos, debido a la forma en que trabaja la sintaxis de las consultas de búsqueda, por ejemplo:

```
var Ejemplo = type('Ejemplo', [models.Model], {  
    foo__bar = new models.IntegerField() // 'foo__bar' tiene dos guiones bajos!  
});
```

Estas limitaciones se pueden manejar sin mayores problemas, dado que el nombre del campo no necesariamente tiene que coincidir con el nombre de la columna en la base de datos. Ver *db_column*, más abajo.

Las palabras reservadas de SQL, como `join`, `where`, o `select`, son permitidas como nombres de campo, dado que se “escapan” todos los nombres de tabla y columna de la base de datos en cada consulta SQL subyacente.

Cada campo en el modelo debe ser una instancia del tipo de campo apropiado. Los tipos de `Field` son utilizados para determinar algunas cosas:

- El tipo de columna de la base de datos (ej., `INTEGER`, `VARCHAR`).
- El widget a usar en la generación de formularios (ej., `<input type="text">`, `<select>`).
- Los requerimientos mínimos de validación.

A continuación, una lista completa de los campos, ordenados alfabéticamente. Los campos de relación (`ForeignKey`, etc.) se tratan en la siguiente sección.

A.1.1 AutoField

Un `IntegerField` que se incrementa automáticamente de acuerdo con los IDs disponibles. Normalmente no es necesario utilizarlos directamente ya que se agrega un campo de clave primaria automáticamente al modelo si no se especifica una clave primaria.

A.1.2 BooleanField

Un campo Verdadero/Falso.

A.1.3 CharField

Un campo string, para cadenas cortas o largas. Para grandes cantidades de texto, usar `TextField`.

`CharField` requiere un argumento extra, `max_length`, que es la longitud máxima (en caracteres) del campo. Esta longitud máxima es reforzada a nivel de la base de datos y en la validación.

A.1.4 DateField

Un campo de fecha. `DateField` tiene algunos argumentos opcionales extra, como se muestra en la [Tabla](#).

Cuadro A.1: Argumentos opcionales extra de `DateField`

Argu-mento	Descripción
<code>auto_now</code>	Asigna automáticamente al campo un valor igual al momento en que se salva el objeto. Es útil para las marcas de tiempo “última modificación”. Observar que <i>siempre</i> se usa la fecha actual; no es un valor por omisión que se pueda sobrescribir.
<code>auto_now_add</code>	Asigna automáticamente al campo un valor igual al momento en que se crea el objeto. Es útil para la creación de marcas de tiempo. Observar que <i>siempre</i> se usa la fecha actual; no es un valor por omisión que se pueda sobrescribir.

A.1.5 DateTimeField

Un campo de fecha y hora. Tiene las mismas opciones extra que `DateField`.

A.1.6 EmailField

Un `CharField` que chequea que el valor sea una dirección de email válida. No acepta `max_length`; su `max_length` se establece automáticamente en 75.

A.1.7 FileField

Advertencia: No este implementado actualmente ver bien la doc.

Un campo de captura de archivos. Tiene un argumento *requerido*, como se ve en la [Tabla](#).

Cuadro A.2: Opciones extra de FileField

Argu-mento	Descripción
upload_to	Una ruta del sistema de archivos local que se agregará a la configuración de MEDIA_ROOT para determinar el resultado de la función de ayuda <code>get_<fieldname>_url()</code> .

Esta ruta puede contener formato `strftime`, que será reemplazada por la fecha y hora de la captura del archivo (de manera que los archivos capturados no llenen una ruta dada).

El uso de un `FileField` o un `ImageField` en un modelo requiere algunos pasos:

1. En el archivo de configuración (settings), es necesario definir `MEDIA_ROOT` con la ruta completa al directorio donde quieras que Django almacene los archivos subidos. (Por performance, estos archivos no se almacenan en la base de datos.) Definir `MEDIA_URL` con la URL pública base de ese directorio. Asegurarse de que la cuenta del usuario del servidor web tenga permiso de escritura en este directorio.
2. Agregar el `FileField` o `ImageField` al modelo, asegurándose de definir la opción `upload_to` para decirle a Django a cual subdirectorio de `MEDIA_ROOT` debe subir los archivos.
3. Todo lo que se va a almacenar en la base de datos es la ruta al archivo (relativa a `MEDIA_ROOT`). Seguramente preferirás usar la facilidad de la función `get_<fieldname>_url` provista por Django. Por ejemplo, si tu `ImageField` se llama `mug_shot`, puedes obtener la URL absoluta a tu image en un plantilla con `{{object.get_mug_shot_url}}`.

Por ejemplo, digamos que tu `MEDIA_ROOT` es `'/home/media'`, y `upload_to` es `'photos/%Y/%m/%d'`. La parte `'%Y/%m/%d'` de `upload_to` es formato `strftime`; `'%Y'` es el año en cuatro dígitos, `'%m'` es el mes en dos dígitos, y `'%d'` es el día en dos dígitos. Si subes un archivo el 15 de enero de 2007, será guardado en `/home/media/photos/2007/01/15`.

Si quieres recuperar el nombre en disco del archivo subido, o una URL que se refiera a ese archivo, o el tamaño del archivo, puedes usar los métodos `get_FIELD_filename()`, `get_FIELD_url()`, y `get_FIELD_size()`. Ver el Apéndice C para una explicación completa de estos métodos.

Nota: Cualquiera sea la forma en que manejes tus archivos subidos, tienes que prestar mucha atención a donde los estás subiendo y que tipo de archivos son, para evitar huecos en la seguridad. *Valida todos los archivos subidos* para asegurarte que esos archivos son lo que piensas que son.

Por ejemplo, si dejas que cualquiera suba archivos ciegamente, sin validación, a un directorio que está dentro de la raíz de documentos (*document root*) de tu servidor web, alguien podría subir un script CGI o PHP y ejecutarlo visitando su URL en tu sitio. ¡No permitas que pase!

A.1.8 FilePathField

Un campo cuyas opciones están limitadas a los nombres de archivo en una cierta ruta en el sistema de archivos. Tiene tres argumentos especiales, que se muestran en la [Tabla](#).

Cuadro A.3: Opciones extra de FilePathField

Argu-mento	Descripción
path	<i>Requerido</i> ; la ruta absoluta en el sistema de archivos hacia el directorio del cual este FilePathField debe tomar sus opciones (ej.: "/home/images").
match	Opcional; una expresión regular como string, que FilePathField usará para filtrar los nombres de archivo. Observar que la regex será aplicada al nombre de archivo base, no a la ruta completa (ej.: "foo.*\..txt^", va a matchear con un archivo llamado foo23.txt, pero no con bar.txt o foo23.gif).
recursive	Opcional; true o false. El valor por omisión es false. Especifica si deben incluirse todos los subdirectorios de path.

Por supuesto, estos argumentos pueden usarse juntos.

El único ‘gotcha’ potencial es que match se aplica sobre el nombre de archivo base, no la ruta completa. De esta manera, este ejemplo:

```
FilePathField({path:"/home/images", match:"foo.*", recursive:true})
```

va a matchear con /home/images/foo.gif pero no con /home/images/foo/bar.gif porque el match se aplica al nombre de archivo base (foo.gif y bar.gif).

A.1.9 FloatField

Un numero de punto flotante, representado en JavaScript por una instancia de Number. Tiene dos argumentos requeridos, que se muestran en la *Tabla*.

Cuadro A.4: Opciones extra de FloatField

Argumento	Descripción
max_digits	La cantidad máximo de dígitos permitidos en el número.
decimal_places	La cantidad de posiciones decimales a almacenar con el número.

Por ejemplo, para almacenar números hasta 999 con una resolución de dos decimales, hay que usar:

```
models.FloatField(..., max_digits=5, decimal_places=2)
```

Y para almacenar números hasta aproximadamente mil millones con una resolución de diez dígitos decimales, hay que usar:

```
models.FloatField(..., max_digits=19, decimal_places=10)
```

A.1.10 ImageField

Advertencia: No este implementado actualmente ver bien la doc.

Similar a FileField, pero valida que el objeto capturado sea una imagen válida. Tiene dos argumentos opcionales extra, height_field y width_field, que si se utilizan, serán auto-rellenados con la altura y el ancho de la imagen cada vez que se guarde una instancia del modelo.

Además de los métodos especiales get_FIELD_* que están disponibles para FileField, un ImageField tiene también los métodos get_FIELD_height() y get_FIELD_width(). Éstos están documentados en el *apendices-doff-dbapi*.

A.1.11 IntegerField

Un entero.

A.1.12 IPAddressField

Una dirección IP, en formato string (ej.: "24.124.1.30").

A.1.13 NullBooleanField

Similar a `BooleanField`, pero permite `NULL` como opción. Usar éste en lugar de un `BooleanField` con `null = true`.

A.1.14 PositiveIntegerField

Similar a `IntegerField`, pero debe ser positivo.

A.1.15 PositiveSmallIntegerField

Similar a `PositiveIntegerField`, pero solo permite valores por debajo de un límite. El valor máximo permitido para estos campos depende de la base de datos, pero como las bases de datos tienen un tipo entero corto de 2 bytes, el valor máximo positivo usualmente es 65,535.

A.1.16 SlugField

“Slug” es un término de la prensa. Un *slug* es una etiqueta corta para algo, que contiene solo letras, números, guiones bajos o simples. Generalmente se usan en URLs.

De igual forma que en `CharField`, puedes especificar `max_length`. Si `max_length` no está especificado, el valor por omisión es de 50.

Un `SlugField` implica `db_index=true` debido a que son los se usan principalmente para búsquedas en la base de datos.

A.1.17 SmallIntegerField

Similar a `IntegerField`, pero solo permite valores en un cierto rango dependiente de la base de datos (usualmente -32,768 a +32,767).

A.1.18 TextField

Un campo de texto de longitud ilimitada.

A.1.19 TimeField

Un campo de hora. Acepta las mismas opciones de autocompletación de `DateField` y `DateTimeField`.

A.1.20 URLField

Un campo para una URL. Si la opción `verify_exists` es `true` (valor por omisión), se chequea la existencia de la URL dada (la URL carga y no da una respuesta 404).

Como los otros campos de caracteres, `URLField` toma el argumento `max_length`. Si no se especifica, el valor por omisión es 200.

A.2 Opciones para todos los campos

Los siguientes argumentos están disponibles para todos los tipos de campo. Todos son opcionales.

A.2.1 null

Si está en `true`, se almacenaran valores vacíos como `NULL` en la base de datos. El valor por omisión es `false`.

Los valores de string nulo siempre se almacenan como strings vacíos, no como `NULL`. `null=true` se debe utilizar solo para campos no-string, como enteros, booleanos y fechas. En los dos casos, también es necesario establecer `blank=true` si se desea permitir valores vacíos en los formularios, ya que el parámetro `null` solo afecta el almacenamiento en la base de datos (ver la siguiente sección, titulada *blank*).

Se debe evitar utilizar `null` en campos basados en string como `CharField` y `TextField` salvo que se tenga una excelente razón para hacerlo. Si un campo basado en string tiene `null=true`, eso significa que tiene dos valores posibles para “sin datos”: `NULL` y el string vacío. En la mayoría de los casos, esto es redundante; la convención es usar el string vacío, no `NULL`.

A.2.2 blank

Si está en `true`, está permitido que el campo esté en blanco. El valor por omisión es `false`.

Este es diferente de `null`. `null` solo se relaciona con la base de datos, mientras que `blank` está relacionado con la validación. Si un campo tiene `blank=true`, la validación permitirá la entrada de un valor vacío. Si un campo tiene `blank=false`, es un campo requerido.

A.2.3 choices

Un arreglo conteniendo tuplas para usar como opciones para este campo.

Si esto está dado, el sistema de formularios utilizará un cuadro de selección en lugar del campo de texto estándar, y limitará las opciones a las dadas.

Una lista de opciones se ve así:

```
var YEAR_IN_SCHOOL_CHOICES = [
    ['FR', 'Freshman'],
    ['SO', 'Sophomore'],
    ['JR', 'Junior'],
    ['SR', 'Senior'],
    ['GR', 'Graduate'],
]
```

El primer elemento de cada tupla es el valor actual a ser almacenado. El segundo elemento es el nombre legible por humanos para la opción.

La lista de opciones puede ser definida también como parte del modelo:

```
var Foo = type('Foo', [ models.Model ], {
  GENDER_CHOICES: [
    ['M', 'Male'],
    ['F', 'Female']
  ],
  gender: new models.CharField({ max_length:1, choices:GENDER_CHOICES}),
});
```

o fuera del modelo:

```
var GENDER_CHOICES: [
  ['M', 'Male'],
  ['F', 'Female']
];
var Foo = type ('Foo', [models.Model], {
  gender: new models.CharField({ max_length:1, choices:GENDER_CHOICES}),
});
```

Para cada campo del modelo que tenga establecidas `choices`, Se agregará un método para recuperar el nombre legible por humanos para el valor actual del campo. Ver *apendices-doff-dbapi* para más detalles.

A.2.4 db_column

El nombre de la columna de la base de datos a usar para este campo. De no estar definido, se utilizará el nombre del campo. Esto es útil cuando se está definiendo un modelo sobre una base de datos existente.

Si el nombre de columna de la base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en un nombre de variable, no hay problema. Los nombres de columna y tabla son escapeados por comillas detrás de la escena.

A.2.5 db_index

Si está en `true`, Un índice es creado en la base de datos para esta columna cuando cree la tabla.

A.2.6 default

El valor por omisión del campo.

A.2.7 editable

Si es `false`, el campo no será editable en el procesamiento de formularios. El valor por omisión es `true`.

A.2.8 help_text

Texto de ayuda extra a ser mostrado bajo el campo en el formulario. Es útil como documentación aunque el objeto no termine siendo representado en un formulario.

A.2.9 primary_key

Si es `true`, este campo es la clave primaria del modelo.

Si no se especifica `primary_key=true` para ningún campo del modelo, se agregará automáticamente este campo:

```
id = new models.AutoField('ID', { primary_key: true });
```

Por lo tanto, no es necesario establecer `primary_key=true` en ningún campo, salvo que se quiera sobrescribir el comportamiento por omisión de la clave primaria.

`primary_key=true` implica `blank=false`, `null=false`, y `unique=true`. Solo se permite una clave primaria en un objeto.

A.2.10 radio_admin

Por omisión, la generación de formularios usa una interfaz de cuadro de selección (`<select>`) para campos que son `ForeignKey` o tienen `choices`. Si `radio_admin` es `true`, un `radio-button` es utilizado en su lugar.

No utilice esto para un campo que no sea `ForeignKey` o no tenga `choices`.

A.2.11 unique

Si es `true`, el valor para este campo debe ser único en la tabla.

A.2.12 unique_for_date

Asignar como valor el nombre de un `DateTimeField` o `DateTimeField` para requerir que este campo sea único para el valor del campo tipo fecha, por ejemplo:

```
var Story = type('Story', [ models.Model ] {
  pub_date: new models.DateTimeField(),
  slug: new models.SlugField({unique_for_date:"pub_date"}),
  ...
});
```

En este código, no se permite la creación de dos historias con el mismo slug publicados en la misma fecha. Esto difiere de usar la restricción `unique_together` en que solo toma en cuenta la fecha del campo `pub_date`; la hora no importa.

A.2.13 unique_for_month

Similar a `unique_for_date`, pero requiere que el campo sea único con respecto al mes del campo dado.

A.2.14 unique_for_year

Similar a `unique_for_date` y `unique_for_month`, pero para el año.

A.2.15 verbose_name

Cada tipo de campo, excepto `ForeignKey`, `ManyToManyField`, y `OneToOneField`, toma un primer argumento posicional opcional – un nombre descriptivo –. Si el nombre descriptivo no está dado, se crea automáticamente usando el nombre de atributo del campo, convirtiendo guiones bajos en espacios.

En este ejemplo, el nombre descriptivo es "Person's first name":

```
first_name = new models.CharField("Person's first name", { max_length: 30 })
```

En este ejemplo, el nombre descriptivo es "first name":

```
first_name = new models.CharField({maxlength: 30})
```

`ForeignKey`, `ManyToManyField`, y `OneToOneField` requieren que el primer argumento sea una clase del modelo, en este caso hay que usar `verbose_name` como argumento con nombre:

```
poll = new models.ForeignKey(Poll, {verbose_name: "the related poll"})
sites = new models.ManyToManyField(Site, {verbose_name: "list of sites"})
place = new models.OneToOneField(Place, {verbose_name: "related place"})
```

La convención es no capitalizar la primera letra del `verbose_name` estas son pasadas a mayúscula automáticamente cuando sea necesario.

A.3 Relaciones

Es claro que el poder de las bases de datos se basa en relacionar tablas entre sí. Los tres tipos de relaciones más comunes en las bases de datos estan soportadas: muchos-a-uno, muchos-a-muchos, y uno-a-uno (utilizada indirectamente en la herencia).

A.3.1 Relaciones Muchos-a-Uno

El campo `ForeignKey` define las relaciones muchos-a-uno. Se usa como cualquier otro tipo `Field`: incluyéndolo como un atributo en el modelo.

`ForeignKey` requiere un argumento posicional: el tipo al cual se relaciona el modelo.

Por ejemplo, si un modelo `Car` tiene un `Manufacturer` – es decir, un `Manufacturer` fabrica múltiples autos pero cada `Car` tiene solo un `Manufacturer` – la definición es:

```
var Manufacturer = type('Manufacturer', [ models.Model ], {
  ...
});

var Car = type('Car', [ models.Model ],
  manufacturer: new models.ForeignKey(Manufacturer),
  ...
);
```

Para crear una relación *recursiva* – un objeto que tiene una relación muchos-a-uno con él mismo – `models.ForeignKey('this')`:

```
var Employee = type('Employee', [ models.Model ], {
  manager: new models.ForeignKey('this'),
  ...
});
```

Si se necesita crear una relación con un modelo que aún no se ha definido, el nombre del modelo puede ser utilizado en lugar del objeto modelo:

```
var Car = type('Car', [ models.Model ], {
  manufacturer: new models.ForeignKey('Manufacturer'),
  ...
});

var Manufacturer = type('Manufacturer', [ models.Model ], {
  ...
});
```

Observar que de todas formas solo se pueden usar strings para hacer referencia a modelos dentro del mismo archivo `models.js` – no se pueden usar strings para hacer referencias a un modelo en una aplicación diferente, o hacer referencia a un modelo que ha sido requerido de cualquier otro lado.

Detrás de la escena, `"_id"` es agregado al nombre de campo para crear su nombre de columna en la base de datos. En el ejemplo anterior, la tabla de la base de datos correspondiente al modelo `Car`, tendrá una columna `manufacturer_id`. (Esto puede se puede cambiar explícitamente especificando `db_column`; ver más arriba en la sección [db_column](#).) De todas formas, el código nunca debe utilizar el nombre de la columna de la base de datos, salvo que escribas SQL. Siempre se utilizaran los nombres de campo del modelo.

Se sugiere, pero no es requerido, que el nombre de un campo `ForeignKey` (`manufacturer` en el ejemplo) sea el nombre del modelo en minúsculas. Igualmente se puede poner cualquier nombre. Por ejemplo:

```
var Car = type('Car', [ models.Model ], {
  company_that_makes_it: new models.ForeignKey(Manufacturer),
  // ...
});
```

Los campos `ForeignKey` reciben algunos argumentos extra para definir como debe trabajar la relación (ver [Tabla](#)). Todos son opcionales.

Advertencia: ver esta tabla hay opciones que no estan.

Cuadro A.5: Opciones de ForeignKey

Argumento	Descripción
<code>edit_inline</code>	Si no es <code>false</code> , este objeto relacionado se edita “inline” en la página del objeto relacionado. Esto significa que el objeto no tendrá su propia interfaz de administración. Usa <code>models.TABULAR</code> o <code>models.STACKED</code> , que designan si los objetos editables inline se muestran como una tabla o como una pila de conjuntos de campos, respectivamente.
<code>limit_choices_to</code>	Un diccionario para buscar argumentos y valores (ver el Apéndice C) que limita las opciones de administración disponibles para este objeto. Usa esto con funciones del módulo <code>datetime</code> de Python para limitar las opciones de fecha de los objetos. Por ejemplo: <pre>limit_choices_to: {'pub_date__lte': datetime.now}</pre> sólo permite la elección de objetos relacionados con <code>pub_date</code> anterior a la fecha/hora actual. En lugar de un diccionario, esto puede ser un objeto <code>Q</code> (ver Apéndice C) para consultas más complejas. No es compatible con <code>edit_inline</code> .
<code>max_num_in_admin</code>	Para objetos editados inline, este es el número máximo de objetos relacionados a mostrar en la interfaz de administración. Por lo tanto, si una pizza puede tener como máximo diez ingredientes, <code>max_num_in_admin=10</code> asegurará que un usuario nunca ingresará más de diez ingredientes. Observar que esto no asegura que no se puedan crear más de diez ingredientes relacionados. Simplemente controla la interfaz de administración; no fortalece cosas a nivel de Python API o base de datos.
<code>min_num_in_admin</code>	La cantidad mínima de objetos relacionados que se muestran en la interfaz de administración. Normalmente, en el momento de la creación se muestran <code>num_in_admin</code> objetos inline, y en el momento de edición se muestran <code>num_extra_on_change</code> objetos en blanco además de todos los objetos relacionados preexistentes. De todas formas, nunca se mostrarán menos de <code>min_num_in_admin</code> objetos relacionados.
<code>num_extra_on_change</code>	La cantidad de campos en blanco extra de objetos relacionados a mostrar en el momento de realizar cambios.
<code>num_in_admin</code>	El valor por omisión de la cantidad de objetos inline a mostrar en la página del objeto en el momento de agregar.
<code>raw_id_admin</code>	Solo muestra un campo para ingresar un entero en lugar de un menú desplegable. Esto es útil cuando se relaciona con un tipo de objeto que tiene demasiadas filas para que sea práctico utilizar una caja de selección.
<code>related_name</code>	No es utilizado con <code>edit_inline</code> . El nombre a utilizar para la relación desde el objeto relacionado de hacia éste objeto. Para más información, ver el Apéndice C.
<code>to_field</code>	El campo en el objeto relacionado con el cual se establece la relación. Por omisión, Django usa la clave primaria del objeto relacionado.

A.3.2 Relaciones Muchos-a-Muchos

Para definir una relación muchos-a-muchos, `ManyToManyField` es el campo. Al igual que `ForeignKey`, `ManyToManyField` requiere un argumento posicional: el tipo al cual se relaciona el modelo.

Por ejemplo, si una `Pizza` tiene múltiples objetos `Topping` – es decir, un `Topping` puede estar en múltiples pizzas y cada `Pizza` tiene múltiples ingredientes (toppings) – debe representarse así:

```
var Topping = type('Topping', [ models.Model ], {
  ...
});

var Pizza = type('Pizza', [ models.Model ], {
  toppings: new models.ManyToManyField(Topping),
  ...
});
```

Como sucede con `ForeignKey`, una relación de un objeto con sí mismo puede definirse usando el string `'this'` en lugar del nombre del modelo, y se pueden hacer referencias a modelos que todavía no se definieron usando un string que contenga el nombre del modelo. De todas formas solo se pueden usar strings para hacer referencia a modelos dentro del mismo archivo `models.js` – no se puede usar un string para hacer referencia a un modelo en una aplicación diferente, o hacer referencia a un modelo que ha sido importado de cualquier otro lado.

Se sugiere, pero no es requerido, que el nombre de un campo `ManyToManyField` (`toppings`, en el ejemplo) sea un término en plural que describa al conjunto de objetos relacionados con el modelo.

Detrás de la escena, se crea una tabla join intermedia para representar la relación muchos-a-muchos.

No importa cual de los modelos tiene el `ManyToManyField`, pero es necesario que esté en uno de los modelos – no en los dos.

Los objetos `ManyToManyField` toman algunos argumentos extra para definir como debe trabajar la relación (ver [Tabla](#)). Todos son opcionales.

Advertencia: ver esta tabla hay opciones que no estan.

Cuadro A.6: Opciones de ManyToManyField

Argumento	Descripción
<code>related_name</code>	El nombre a utilizar para la relación desde el objeto relacionado hacia este objeto. Ver Apéndice C para más información..
<code>filter_intel</code>	Usa una interfaz de “filtro” JavaScript agradable y discreta en lugar de la menos cómoda <code><select multiple></code> en el formulario administrativo de este objeto. El valor debe ser <code>models.HORIZONTAL</code> o <code>models.VERTICAL</code> (es decir, la interfaz debe apilarse horizontal o verticalmente).
<code>limit_choices_to</code>	Ver la descripción en <code>ForeignKey</code> .
<code>symmetrical</code>	Solo utilizado en la definición de <code>ManyToManyField</code> sobre sí mismo. Considera el siguiente modelo: <pre>var Person = type('Person', [models.Model], { friends: new models.ManyToManyField("this") });</pre> <p>Cuando Django procesa este modelo, identifica que tiene un <code>ManyToManyField</code> sobre sí mismo, y como resultado, no agrega un atributo <code>person_set</code> a la clase <code>Person</code>. En lugar de eso, se asumen que el <code>ManyToManyField</code> es simétrico – esto es, si yo soy tu amigo, entonces tu eres mi amigo.</p> <p>Si no deseas la simetría en las relaciones <code>ManyToMany</code> con <code>self</code>, establece <code>symmetrical</code> en <code>false</code>. Esto forzará a Django a agregar el descriptor para la relación inversa, permitiendo que las relaciones <code>ManyToMany</code> sean asimétricas.</p>
<code>db_table</code>	El nombre de la tabla a crear para almacenar los datos de la relación muchos-a-muchos. Si no se provee, Django asumirá un nombre por omisión basado en los nombres de las dos tablas a ser vinculadas.

A.4 Opciones de los Metadatos del Modelo

Los metadatos específicos de un modelo viven en un `Object Meta` definido en el cuerpo del modelo:

```
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({max_length:100}),

    Meta: {
        // model metadata options go here
        ...
    }
});
```

Los metadatos del modelo son “cualquier cosa que no sea un campo”, como opciones de ordenamiento, etc.

Las secciones que siguen presentan una lista de todas las posibles `Meta` opciones. Ninguna de estas opciones es requerida. Agregar `Meta` a un modelo es completamente opcional.

A.4.1 db_table

El nombre de la tabla de la base de datos a usar para el modelo.

Si no se define el nombre de la tabla de la base de datos es derivado automáticamente a partir del nombre del modelo y la aplicación que lo contiene. Un nombre de tabla de base de datos de un modelo se construye uniendo la etiqueta de la aplicación del modelo – el nombre que tiene la aplicación – con el nombre del modelo, con un guión bajo entre ellos.

Por ejemplo, para la aplicación `books`, un modelo definido como `Book` tendrá una tabla en la base de datos llamada `book_books`.

Para sobrescribir el nombre de la tabla de la base de datos, se debe usar el parámetro `db_table` dentro de `Meta`:

```
var Book = type('Book', [ models.Model ], {
    ...

    Meta: {
        db_table: 'things_to_read'
    }
});
```

Si el nombre de tabla de base de datos es una palabra reservada de SQL, o contiene caracteres que no están permitidos en los nombres de variable, no hay problema. Los nombres de tabla y de columna son escapeados con comillas al generar el SQL.

A.4.2 get_latest_by

El nombre de un `DateTimeField` o `DateTimeField` del modelo. Esto especifica el campo a utilizar por omisión en el método `latest()` del `Manager` del modelo.

Aquí hay un ejemplo:

```
var CustomerOrder = type('CustomerOrder' , [ models.Model ], {
    order_date: new models.DateTimeField(),
    ...

    Meta: {
        get_latest_by: "order_date"
    }
});
```

Ver *apendices-doff-dbapi* para más información sobre el método `latest()`.

A.4.3 order_with_respect_to

Marca este objeto como “ordenable” con respecto al campo dado. Esto se utiliza casi siempre con objetos relacionados para permitir que puedan ser ordenados respecto a un objeto padre. Por ejemplo, si una `Answer` se relaciona a un objeto `Question`, y una pregunta tiene más de una respuesta, y el orden de las respuestas importa, harás esto:

```
var Answer = type('Answer', [ models.Model ], {
    question: new models.ForeignKey(Question),
    ...

    Meta: {
        order_with_respect_to: 'question'
    }
});
```

A.4.4 ordering

El ordenamiento por omisión del objeto, utilizado cuando se obtienen listas de objetos:

```
var Book = type('Book', [ models.Model ], {
    title: new models.CharField({maxlength: 100}),

    Meta: {
        ordering: ['title']
    }
});
```

Esto es una arreglo de strings. Cada string es un nombre de campo con un prefijo opcional `-`, que indica orden descendente. Los campos sin un `-` precedente se ordenarán en forma ascendente. Use el string `"?"` para ordenar al azar.

Por ejemplo, para ordenar por un campo `title` en orden ascendente:

```
ordering: ['title']
```

Para ordenar por `title` en orden descendente:

```
ordering: ['-title']
```

Para ordenar por `title` en orden descendente, y luego por `author` en orden ascendente:

```
ordering: ['-title', 'author']
```

A.4.5 `unique_together`

Conjuntos de nombres de campo que tomados juntos deben ser únicos:

```
var Employee = type('Employee', [ models.Model ], {
    department: new models.ForeignKey(Department),
    extension: new models.CharField({max_length: 10}),
    ...

    Meta: {
        unique_together: [["department", "extension"]]
    }
});
```

Esto es un arreglo de arreglos de campos que deben ser únicos cuando se consideran juntos. Es usado en la validacion de formularios y se refuerza a nivel de base de datos (esto es, se incluyen las sentencias `UNIQUE` apropiadas en la sentencia `CREATE TABLE`).

A.4.6 `verbose_name`

Un nombre legible por humanos para el objeto, en singular:

```
var CustomerOrder = type('CustomerOrder', [ models.Model ], {
    order_date: new models.DateTimeField(),
    ...

    Meta: {
        verbose_name: "order"
    }
});
```

```
    }  
  });
```

Si no se define, se utilizará una versión adaptada del nombre del modelo, en la cual `CamelCase` se convierte en `camel case`.

A.4.7 `verbose_name_plural`

El nombre del objeto en plural:

```
var Sphynx = type('Sphynx', [ models.Model ], {  
    ...  
  
    Meta: {  
        verbose_name_plural: "sphynges"  
    }  
});
```

Si no se define, se agregará una “s” al final del `verbose_name`.

A.5 Managers

Un `Manager` es la interfaz a través de la cual se proveen las operaciones de consulta de la base de datos a los modelos de Django. Existe al menos un `Manager` para cada modelo en una aplicación Django.

La forma en que trabajan las clases `Manager` está documentada en el Apéndice C. Esta sección trata específicamente las opciones del modelo que personaliza el comportamiento del `Manager`.

A.5.1 Nombres de Manager

Por omisión, Django agrega un `Manager` llamado `objects` a cada clase modelo de Django. De todas formas, si tu quieres usar `objects` como nombre de campo, o quieres usar un nombre distinto de `objects` para el `Manager`, puedes renombrarlo en cada uno de los modelos. Para renombrar el `Manager` para una clase dada, define un atributo de clase de tipo `models.Manager()` en ese modelo, por ejemplo:

```
from django.db import models  
  
class Person(models.Model):  
    ...  
  
    people = models.Manager()
```

Usando este modelo de ejemplo, `Person.objects` generará una excepción `AttributeError` (dado que `Person` no tiene un atributo `objects`), pero `Person.people.all()` devolverá una lista de todos los objetos `Person`.

A.5.2 Managers Personalizados

Puedes utilizar un `Manager` personalizado en un modelo en particular extendiendo la clase base `Manager` e instanciando tu `Manager` personalizado en tu modelo.

Hay dos razones por las que puedes querer personalizar un Manager: para agregar métodos extra al Manager, y/o para modificar el QuerySet inicial que devuelve el Manager.

Agregando Métodos Extra al Manager

Agregar métodos extra al Manager es la forma preferida de agregar funcionalidad a nivel de tabla a tus modelos. (Para funcionalidad a nivel de registro – esto es, funciones que actúan sobre una instancia simple de un objeto modelo – usa métodos modelo (ver más abajo), no métodos de Manager personalizados.)

Un método Manager personalizado puede retornar cualquier cosa que necesites. No tiene que retornar un QuerySet.

Por ejemplo, este Manager personalizado ofrece un método `with_counts()`, que retorna una lista de todos los objetos `OpinionPoll`, cada uno con un atributo extra `num_responses` que es el resultado de una consulta agregada:

```
from django.db import connection

class PollManager(models.Manager):

    def with_counts(self):
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY 1, 2, 3
            ORDER BY 3 DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(maxlength=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(Poll)
    person_name = models.CharField(maxlength=50)
    response = models.TextField()
```

En este ejemplo, puedes usar `OpinionPoll.objects.with_counts()` para retornar la lista de objetos `OpinionPoll` con el atributo `num_responses`.

Otra cosa a observar en este ejemplo es que los métodos de un Manager pueden acceder a `self.model` para obtener la clase modelo a la cual están anexados.

Modificando los QuerySets iniciales del Manager

Un QuerySet base de un Manager devuelve todos los objetos en el sistema. Por ejemplo, usando este modelo:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    author = models.CharField(maxlength=50)
```

la sentencia `Book.objects.all()` retornará todos los libros de la base de datos.

Puedes sobrescribir el `QuerySet` base, sobrescribiendo el método `Manager.get_query_set()`. `get_query_set()` debe retornar un `QuerySet` con las propiedades que tu requieres.

Por ejemplo, el siguiente modelo tiene *dos* managers – uno que devuelve todos los objetos, y otro que retorna solo los libros de Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_query_set(self):
        return super(DahlBookManager, self).get_query_set().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(maxlength=100)
    author = models.CharField(maxlength=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

Con este modelo de ejemplo, `Book.objects.all()` retornará todos los libros de la base de datos, pero `Book.dahl_objects.all()` solo retornará aquellos escritos por Roald Dahl.

Por supuesto, como `get_query_set()` devuelve un objeto `QuerySet`, puedes usar `filter()`, `exclude()`, y todos los otro métodos de `QuerySet` sobre él. Por lo tanto, estas sentencias son todas legales:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

Este ejemplo también señala otra técnica interesante: usar varios managers en el mismo modelo. Puedes agregar tantas instancias de `Manager()` como quieras. Esta es una manera fácil de definir “filters” comunes para tus modelos. Aquí hay un ejemplo:

```
class MaleManager(models.Manager):
    def get_query_set(self):
        return super(MaleManager, self).get_query_set().filter(sex='M')

class FemaleManager(models.Manager):
    def get_query_set(self):
        return super(FemaleManager, self).get_query_set().filter(sex='F')

class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    last_name = models.CharField(maxlength=50)
    sex = models.CharField(maxlength=1, choices= (('M', 'Male'), ('F', 'Female')))
    people = models.Manager()
    men = MaleManager()
    women = FemaleManager()
```

Este ejemplo te permite consultar `Person.men.all()`, `Person.women.all()`, y `Person.people.all()`, con los resultados predecibles.

Si usas objetos `Manager` personalizados, toma nota que el primer `Manager` que encuentre Django (en el orden en el que están definidos en el modelo) tiene un status especial. Django interpreta el primer `Manager` definido en una clase como el `Manager` por omisión. Ciertas operaciones – como las del sitio de administración de Django – usan el `Manager` por omisión para obtener listas de objetos, por lo que generalmente es una buena idea que el primer `Manager` esté relativamente sin filtrar. En el último ejemplo, el `manager people` está definido primero – por lo cual es el `Manager` por omisión.

A.6 Métodos de Modelo

Define métodos personalizados en un modelo para agregar funcionalidad personalizada a nivel de registro para tus objetos. Mientras que los métodos `Manager` están pensados para hacer cosas a nivel de tabla, los métodos de modelo deben actual en una instancia particular del modelo.

Esta es una técnica valiosa para mantener la lógica del negocio en un sólo lugar: el modelo. Por ejemplo, este modelo tiene algunos métodos personalizados:

```
class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    last_name = models.CharField(maxlength=50)
    birth_date = models.DateField()
    address = models.CharField(maxlength=100)
    city = models.CharField(maxlength=50)
    state = models.USStateField() # Yes, this is America-centric...

    def baby_boomer_status(self):
        """Returns the person's baby-boomer status."""
        import datetime
        if datetime.date(1945, 8, 1) <= self.birth_date <= datetime.date(1964, 12, 31):
            return "Baby boomer"
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        return "Post-boomer"

    def is_midwestern(self):
        """Returns true if this person is from the Midwest."""
        return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')

    @property
    def full_name(self):
        """Returns the person's full name."""
        return '%s %s' % (self.first_name, self.last_name)
```

El último método en este ejemplo es una *propiedad* – un atributo implementado por código getter/setter personalizado. Las propiedades son un truco ingenioso agregado en Python 2.2; puedes leer más acerca de ellas en <http://www.python.org/download/releases/2.2/descrintro/#property>.

Existen también un puñado de métodos de modelo que tienen un significado “especial” para Python o Django. Estos métodos se describen en las secciones que siguen.

A.6.1 `__str__`

`__str__()` es un “método mágico” de Protopy que define lo que debe ser devuelto si llamas a `string()` sobre el objeto. Django usa `string(obj)` (o la función relacionada `unicode(obj)`, que se describe más abajo) en varios lugares, particularmente como el valor mostrado para hacer el render de un objeto en el sitio de administración de

Django y como el valor insertado en un plantilla cuando muestra un objeto. Por eso, siempre debes retornar un string agradable y legible por humanos en el `__str__` de un objeto. A pesar de que esto no es requerido, es altamente recomendado.

Aquí hay un ejemplo:

```
class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    last_name = models.CharField(maxlength=50)

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

A.6.2 get_absolute_url

Define un método `get_absolute_url()` para decirle a Django cómo calcular la URL de un objeto, por ejemplo:

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

Django usa esto en la interfaz de administración. Si un objeto define `get_absolute_url()`, la página de edición del objeto tendrá un enlace “View on site”, que te llevará directamente a la vista pública del objeto, según `get_absolute_url()`.

También un par de otras partes de Django, como el framework de sindicación de feeds, usan `get_absolute_url()` como facilidad para recompensar a las personas que han definido el método.

Es una buena práctica usar `get_absolute_url()` en plantillas, en lugar de codificar en duro las URL de tus objetos. Por ejemplo, este código de plantilla es *mal*:

```
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

Pero este es bueno:

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

El problema con la forma en que simplemente escribimos `get_absolute_url()` es que viola levemente el principio DRY: la URL de este objeto se define dos veces, en el archivo `URLconf` y en el modelo.

Además puedes desacoplar tus modelos de el `URLconf` usando el decorator `permalink`. A este decorator se le pasa función de view, una lista de parámetros posicionales, y (opcionalmente) un diccionario de parámetros por nombre. Django calcula la URL completa correspondiente usando el `URLconf`, sustituyendo los parámetros que le has pasado en la URL. Por ejemplo, si tu `URLconf` contiene una línea como ésta:

```
(r'^people/(\d+)/$', 'people.views.details'),
```

tu modelo puede tener un método `get_absolute_url` como éste:

```
@models.permalink
def get_absolute_url(self):
    return ('people.views.details', [str(self.id)])
```

En forma similar, si tienes una entrada en `URLconf` que se ve como esta:

```
(r'/archive/(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<day>\d{1,2})/$', archive_view)
```


puedes hacer referencia a la misma usando `permalink()` como sigue:

```
@models.permalink
def get_absolute_url(self):
    return ('archive_view', (), {
        'year': self.created.year,
        'month': self.created.month,
        'day': self.created.day})
```

Observar que especificamos una secuencia vacía para el segundo argumento en este caso, porque sólo queremos pasar argumentos por clave, no argumentos por nombre.

De esta forma, estás ligando la URL absoluta del modelo a la vista que se utiliza para mostrarla, sin repetir la información de la URL en ningún lado. Aún puedes usar el método `get_absolute_url` en plantillas, como antes.

A.6.3 Ejecutando SQL personalizado

Siéntete libre de escribir sentencias SQL personalizadas en métodos personalizados de modelo y métodos a nivel de módulo. El objeto `django.db.connection` representa la conexión actual a la base de datos. Para usarla, invoca `connection.cursor()` para obtener un objeto cursor. Después, llama a `cursor.execute(sql, [params])` para ejecutar la SQL, y `cursor.fetchone()` o `cursor.fetchall()` para devolver las filas resultantes:

```
def my_custom_sql(self):
    from django.db import connection
    cursor = connection.cursor()
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()
    return row
```

`connection` y `cursor` implementan en su mayor parte la DB-API estándar de Python (<http://www.python.org/peps/pep-0249.html>). Si no estás familiarizado con la DB-API de Python, observa que la sentencia SQL en `cursor.execute()` usa placeholders, `"%s"`, en lugar de agregar los parámetros directamente dentro de la SQL. Si usas esta técnica, la biblioteca subyacente de base de datos automáticamente agregará comillas y secuencias de escape a tus parámetros según sea necesario. (Observar también que Django espera el placeholder `"%s"`, no el placeholder `"?"`, que es utilizado por los enlaces Python a SQLite. Python bindings. Esto es por consistencia y salud mental).

Una nota final: Si todo lo que quieres hacer es usar una cláusula `WHERE` personalizada, puedes usar los argumentos `where`, `tables`, y `params` de la API estándar de búsqueda. Ver Apéndice C.

A.6.4 Sobreescribiendo los Métodos por omisión del Modelo

Como se explica en el Apéndice C, cada modelo obtiene algunos métodos automáticamente – los más notables son `save()` y `delete()`. Puedes sobreescribir estos métodos para alterar el comportamiento.

Un caso de uso clásico de sobreescritura de los métodos incorporados es cuando necesitas que suceda algo cuando guardas un objeto, por ejemplo:

```
class Blog(models.Model):
    name = models.CharField(maxlength=100)
    tagline = models.TextField()

    def save(self):
        do_something()
```

```
super(Blog, self).save() # Call the "real" save() method.  
do_something_else()
```

También puedes evitar el guardado:

```
class Blog(models.Model):  
    name = models.CharField(maxlength=100)  
    tagline = models.TextField()  
  
    def save(self):  
        if self.name == "Yoko Ono's blog":  
            return # Yoko shall never have her own blog!  
        else:  
            super(Blog, self).save() # Call the "real" save() method
```

A.7 Opciones del Administrador

La clase `Admin` le dice a Django cómo mostrar el modelo en el sitio de administración.

Las siguientes secciones presentan una lista de todas las opciones posibles de `Admin`. Ninguna de estas opciones es requerida. Para utilizar una interfaz de administración sin especificar ninguna opción, use `pass`, como en:

```
class Admin:  
    pass
```

Agregar `class Admin` a un modelo es completamente opcional.

A.7.1 `date_hierarchy`

Establece `date_hierarchy` con el nombre de un `DateField` o `DateTimeField` en tu modelo, y la página de la lista de cambios incluirá una navegación basada en la fecha usando ese campo.

Aquí hay un ejemplo:

```
class CustomerOrder(models.Model):  
    order_date = models.DateTimeField()  
    ...  
  
class Admin:  
    date_hierarchy = "order_date"
```

A.7.2 `fields`

Establece `fields` para controlar la disposición de las páginas “agregar” y “modificar” de la interfaz de administración.

`fields` es una estructura anidada bastante compleja que se demuestra mejor con un ejemplo. Lo siguiente está tomado del modelo `FlatPage` que es parte de `django.contrib.flatpages`:

```
class FlatPage(models.Model):
    ...

    class Admin:
        fields = (
            (None, {
                'fields': ('url', 'title', 'content', 'sites')
            }),
            ('Advanced options', {
                'classes': 'collapse',
                'fields': ('enable_comments', 'registration_required', 'template_name')
            }),
        )
```

Formalmente, `fields` es una lista de tuplas dobles, en la que cada tupla doble representa un `<fieldset>` en el formulario de la página de administración. (Un `<fieldset>` es una “sección” del formulario.)

Las tuplas dobles son de la forma `(name, field_options)`, donde `name` es un string que representa el título del conjunto de campos, y `field_options` es un diccionario de información acerca del conjunto de campos, incluyendo una lista de los campos a mostrar en él.

Si `fields` no está definido, Django mostrará por omisión cada campo que no sea un `AutoField` y tenga `editable=True`, en un conjunto de campos simple, en el mismo orden en que aparecen los campos definidos en el modelo.

El diccionario `field_options` puede tener las clave que se describen en la siguiente sección.

fields

Una tupla de nombres de campo a mostrar en el conjunto de campos. Esta clave es requerida.

Para mostrar múltiples campos en la misma línea, encierra esos campos en su propia tupla. En este ejemplo, los campos `first_name` y `last_name` se mostrarán en la misma línea:

```
'fields': (('first_name', 'last_name'), 'address', 'city', 'state'),
```

classes

Un string conteniendo clases extra CSS para aplicar al conjunto de campos.

Aplica múltiples clases separándolas con espacios:

```
'classes': 'wide extrapretty',
```

Dos clases útiles definidas por la hoja de estilo del sitio de administración por omisión son `collapse` y `wide`. Los conjuntos de campos con el estilo `collapse` serán colapsados inicialmente en el sitio de administración y reemplazados por un pequeño enlace “click to expand”. Los conjuntos de campos con el estilo `wide` tendrán espacio horizontal extra.

description

Un string de texto extra opcional para mostrar encima de cada conjunto de campos, bajo el encabezado del mismo. Se usa tal cual es, de manera que puedes usar cualquier HTML, y debes crear las secuencias de escape correspondientes para cualquier carácter especial HTML (como los ampersands).

A.7.3 js

Una lista de strings representando URLs de archivos JavaScript a vincular en la pantalla de administración mediante etiquetas `<script src="">`. Esto puede ser usado para ajustar un tipo determinado de página de administración en JavaScript o para proveer “quick links” para llenar valores por omisión para ciertos campos.

Si usas URLs relativas – esto es, URLs que no empiezan con `http://` o `/` – entonces el sitio de administración prefijará automáticamente estos enlaces con `settings.ADMIN_MEDIA_PREFIX`.

A.7.4 list_display

Establece `list_display` para controlar que campos se muestran en la página de la lista de del administrador.

Si no se define `list_display`, el sitio de administración mostrará una columna simple con la representación `__str__()` de cada objeto.

Aquí hay algunos casos especiales a observar acerca de `list_display`:

- Si el campo es una `ForeignKey`, Django mostrará el `__str__()` del objeto relacionado.
- No se admiten los campos `ManyToManyField`, porque eso implicaría la ejecución de una sentencia SQL separada para cada fila en la tabla. Si de todas formas quieres hacer esto, dale a tu modelo un método personalizado, y agrega el nombre de ese método a `list_display`. (Más información sobre métodos personalizados en `list_display` en breve.)
- Si el campo es un `BooleanField` o `NullBooleanField`, Django mostrará unos bonitos iconos “on” o “off” en lugar de `true` o `false`.
- Si el string dado es un método del modelo, Django lo invocará y mostrará la salida. Este método debe tener un atributo de función `short_description` para usar como encabezado del campo.

Aquí está un modelo de ejemplo completo:

```
class Person(models.Model):
    name = models.CharField(maxlength=50)
    birthday = models.DateField()

    class Admin:
        list_display = ('name', 'decade_born_in')

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "0's"
    decade_born_in.short_description = 'Birth decade'
```

- Si el string dado es un método del modelo, Django hará un HTML-escape de la salida por omisión. Si no quieres ‘escapear’ la salida del método, dale al método un atributo `allow_tags` con el valor en `true`.

Aquí está un modelo de ejemplo completo:

```
class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    last_name = models.CharField(maxlength=50)
    color_code = models.CharField(maxlength=6)

    class Admin:
        list_display = ('first_name', 'last_name', 'colored_name')

    def colored_name(self):
```

```

    return '<span style="color: # %s;">%s </span>' % (self.color_code, self.first_name, se
    colored_name.allow_tags = True

```

- Si el string dado es un método del modelo que retorna true o false, Django mostrará un bonito icono “on” o “off” si le das al método un atributo boolean con valor en true.

Aquí está un modelo de ejemplo completo:

```

class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    birthday = models.DateField()

    class Admin:
        list_display = ('name', 'born_in_fifties')

    def born_in_fifties(self):
        return self.birthday.strftime('%Y')[2:3] == 5
    born_in_fifties.boolean = True

```

- Los métodos `__str__()` son tan válidos en `list_display` como cualquiera otro método del modelo, por lo cual está perfectamente OK hacer esto:

```
list_display = ('__str__', 'some_other_field')
```

- Usualmente, los elementos de `list_display` que no son campos de la base de datos no pueden ser utilizados en ordenamientos (porque Django hace todo el ordenamiento a nivel de base de datos).

De todas formas, si un elemento de `list_display` representa cierto campo de la base de datos, puedes indicar este hecho estableciendo el atributo `admin_order_field` del ítem, por ejemplo:

```

class Person(models.Model):
    first_name = models.CharField(maxlength=50)
    color_code = models.CharField(maxlength=6)

    class Admin:
        list_display = ('first_name', 'colored_first_name')

    def colored_first_name(self):
        return '<span style="color: # %s;">%s</span>' % (self.color_code, self.first_name)
    colored_first_name.allow_tags = True
    colored_first_name.admin_order_field = 'first_name'

```

El código precedente le dirá a Django que ordene según el campo `first_name` cuando trate de ordenar por `colored_first_name` en el sitio de administración.

A.7.5 list_display_links

Establece `list_display_links` para controlar cuales campos de `list_display` deben ser vinculados a la pagina de cambios de un objeto.

Por omisión, la página de la lista de cambios vinculará la primera columna – el primer campo especificado en `list_display` – a la página de cambios de cada ítem. Pero `list_display_links` te permite cambiar cuáles columnas se vinculan. Establece `list_display_links` a una lista o tupla de nombres de campo (en el mismo formato que `list_display`) para vincularlos.

`list_display_links` puede especificar uno o varios nombres de campo. Mientras los nombres de campo aparezcan en `list_display`, a Django no le preocupa si los campos vinculados son muchos o pocos. El único requerimiento es que si quieres usar “`list_display_links`”, debes definir `list_display`.

En este ejemplo, los campos `first_name` y `last_name` serán vinculados a la página de la lista de cambios:

```
class Person(models.Model):
    ...

    class Admin:
        list_display = ('first_name', 'last_name', 'birthday')
        list_display_links = ('first_name', 'last_name')
```

Finalmente, observa que para usar `list_display_links`, debes definir también `list_display`.

A.7.6 list_filter

Establece `list_filter` para activar los filtros en la barra lateral derecha de la página de la lista de cambios en la interfaz de administración. Debe ser una lista de nombres de campo, y cada campo especificado debe ser de alguno de los tipos `BooleanField`, `DateField`, `DateTimeField`, o `ForeignKey`.

Este ejemplo, tomado del modelo `django.contrib.auth.models.User` muestra como trabajan ambos, `list_display` y `list_filter`:

```
class User(models.Model):
    ...

    class Admin:
        list_display = ('username', 'email', 'first_name', 'last_name', 'is_staff')
        list_filter = ('is_staff', 'is_superuser')
```

A.7.7 list_per_page

Establece `list_per_page` para controlar cuantos items aparecen en cada página de la lista de cambios del administrador. Por omisión, este valor se establece en 100.

A.7.8 list_select_related

Establece `list_select_related` para indicarle a Django que use `select_related()` al recuperar la lista de objetos de la página de la lista de cambios del administrador. Esto puede ahorrarte una cantidad de consultas a la base de datos si estás utilizando objetos relacionados en la lista de cambios que muestra el administrador.

El valor debe ser `true` o `false`. Por omisión es `false`, salvo que uno de los campos `list_display` sea una `ForeignKey`.

Para más detalles sobre `select_related()`, ver Apéndice C.

A.7.9 ordering

Establece `ordering` para especificar como deben ordenarse los objetos en la página de la lista de cambios del administrador. Esto debe ser una lista o tupla en el mismo formato que el parámetro `ordering` del modelo.

Si no está definido, la interfaz de administración de Django usará el ordenamiento por omisión del modelo.

A.7.10 save_as

Establece `save_as` a `true` para habilitar la característica “save as” en los formularios de cambios del administrador. Normalmente, los objetos tienen tres opciones al guardar: “Save”, “Save and continue editing” y “Save and add another”. Si `save_as` es `true`, “Save and add another” será reemplazado por un botón “Save as”.

“Save as” significa que el objeto será guardado como un objeto nuevo (con un identificador nuevo), en lugar del objeto viejo.

Por omisión, `save_as` es `false`.

A.7.11 save_on_top

Establece `save_on_top` para agregar botones de guardado a lo largo del encabezado de tus formularios de cambios del administrador.

Normalmente, los botones de guardado aparecen solamente al pie de los formularios. Si estableces `save_on_top`, los botones aparecerán en el encabezado y al pie del formulario.

Por omisión, `save_on_top` es `false`.

A.7.12 search_fields

Establece `search_fields` para habilitar un cuadro de búsqueda en la página de la lista de cambios del administrador. Debe ser una lista de nombres de campo que se utilizará para la búsqueda cuando alguien envíe una consulta en ese cuadro de texto.

Estos campos deben ser de alguna tipo de campo de texto, como `CharField` o `TextField`. También puedes realizar una búsqueda relacionada sobre una `ForeignKey` con la notación de búsqueda de la API:

```
class Employee(models.Model):
    department = models.ForeignKey(Department)
    ...

    class Admin:
        search_fields = ['department__name']
```

Cuando alguien hace una búsqueda en el cuadro de búsqueda del administrador, Django divide la consulta de búsqueda en palabras y retorna todos los objetos que contengan alguna de las palabras, sin distinguir mayúsculas y minúsculas, donde cada palabra debe estar en al menos uno de los `search_fields`. Por ejemplo, si `search_fields` es `['first_name', 'last_name']` y un usuario busca `john lennon`, Django hará el equivalente a esta cláusula `WHERE` en SQL:

```
WHERE (first_name ILIKE '%john%' OR last_name ILIKE '%john%')
AND (first_name ILIKE '%lennon%' OR last_name ILIKE '%lennon%')
```

Para búsquedas más rápidas y/o más restrictivas, agrega como prefijo al nombre de campo un operador como se muestra en la Tabla B-7.

Cuadro A.7: Tabla B-7. Operadores Permitidos en search_fields

Operador	Significado
^	<p>Matchea el principio del campo. Por ejemplo, si <code>search_fields</code> es <code>['^first_name', '^last_name']</code>, y un usuario busca <code>john lennon</code>, Django hará el equivalente a esta cláusula WHERE en SQL:</p> <pre>WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%' AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%'))</pre> <p>Esta consulta es más eficiente que la consulta <code>'%john%'</code>, dado que la base de datos solo necesita examinar el principio de una columna de datos, en lugar de buscar a través de todos los datos de la columna. Además, si la columna tiene un índice, algunas bases de datos pueden permitir el uso del índice para esta consulta, a pesar de que sea una consulta LIKE.</p>
=	<p>Matchea exactamente, sin distinguir mayúsculas y minúsculas. Por ejemplo, si <code>search_fields</code> es <code>['=first_name', '=last_name']</code> y un usuario busca <code>john lennon</code>, Django hará el equivalente a esta cláusula WHERE en SQL:</p> <pre>WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john' AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon'))</pre> <p>Observar que la entrada de la consulta se divide por los espacios, por lo cual actualmente no es posible hacer una búsqueda de todos los registros en los cuales <code>first_name</code> es exactamente <code>'john winston'</code> (con un espacio en el medio).</p>
@	<p>Realiza una búsqueda en todo el texto. Es similar al método de búsqueda predeterminado, pero usa un índice. Actualmente solo está disponible para MySQL.</p>

Índice

A

API, [13](#)

B

BSD, [14](#)

D

DOM, [13](#)

E

extend() (built-in function), [20](#)

F

field, [13](#)

G

generic view, [13](#)

H

HTMLElement.enable() (built-in function), [27](#)

I

i18n, [14](#)

instance() (built-in function), [21](#)

isubclass() (built-in function), [21](#)

isundefined() (built-in function), [20](#)

J

JSON, [13](#)

M

model, [13](#)

MTV, [13](#)

MVC, [13](#)

P

project, [13](#)

property, [13](#)

publish() (built-in function), [18](#)

Q

queryset, [13](#)

R

require() (built-in function), [18](#)

RPC, [13](#)

S

slug, [13](#)

super() (built-in function), [20](#)

T

template, [14](#)

type() (built-in function), [18](#)

V

view, [14](#)