

THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Updated for
Django 1.1

Practical **django** Projects

*Write better web applications faster, and learn
how to build up your own reusable code library*

SECOND EDITION

James Bennett

Django Release Manager

apress®

Proyectos prácticos de Django

Segunda edición



james bennett

apress®

Proyectos prácticos de Django, segunda edición

Copyright © 2009 por James Bennett

Reservados todos los derechos. Ninguna parte de este trabajo puede reproducirse o transmitirse de ninguna forma o por ningún medio, electrónico o mecánico, incluidas las fotocopias, las grabaciones o cualquier sistema de almacenamiento o recuperación de información, sin el permiso previo por escrito del propietario de los derechos de autor y el editor.

ISBN-13 (pbk): 978-1-4302-1938-5

ISBN-13 (electrónico): 978-1-4302-1939-2

Impreso y encuadrado en los Estados Unidos de América 9 8 7 6 5 4 3 2 1

Los nombres comerciales pueden aparecer en este libro. En lugar de usar un símbolo de marca comercial cada vez que aparece un nombre de marca registrada, usamos los nombres solo de manera editorial y en beneficio del propietario de la marca comercial, sin intención de infringir la marca comercial.

Java y todas las marcas basadas en Java son marcas comerciales o marcas comerciales registradas de Sun Microsystems, Inc., en los Estados Unidos y otros países.

Apress, Inc., no está afiliada a Sun Microsystems, Inc., y este libro se escribió sin el respaldo de Sun Microsystems, Inc.

Editor principal: Duncan Parkes

Revisor técnico: Ben Ford

Consejo editorial: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell,

Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper,

Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Gerente senior de proyectos: Kylie Johnston

Editora de estilo: Nina Goldschlager Perry

Directora de producción asociada: Kari Brooks-Copony

Editora de producción senior: Laura Cheu

Compositor: Lynn L'Heureux

Corrector: Servicios de corrección e indexación BIM

Indexador: Ron Strauss

Diseño de portada: Kurt Krames

Director de Fabricación: Tom Debolski

Distribuido al comercio de libros en todo el mundo por Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Teléfono 1-800-SPRINGER, fax 201-348-4505, correo electrónico orders-ny@springer-sbm.com, o visite <http://www.springeronline.com>.

Para obtener información sobre traducciones, comuníquese directamente con Apress en 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Teléfono 510-549-5930, fax 510-549-5939, correo electrónico info@apress.com, o visite <http://www.apress.com>.

Los libros Apress y Friends of ED se pueden comprar al por mayor para uso académico, corporativo o promocional. Las versiones y licencias de libros electrónicos también están disponibles para la mayoría de los títulos. Para obtener más información, consulte nuestra página web de licencias de libros electrónicos de ventas al por mayor especiales en <http://www.apress.com/info/bulksales>.

La información de este libro se distribuye "tal cual", sin garantía. Aunque se han tomado todas las precauciones en la preparación de este trabajo, ni el(es) autor(es) ni Apress tendrán responsabilidad alguna ante ninguna persona o entidad con respecto a cualquier pérdida o daño causado o presuntamente causado directa o indirectamente por el información contenida en este trabajo.

El código fuente de este libro está disponible para los lectores en <http://www.apress.com>.

Contenido de un vistazo

Sobre el autor	xi
Acerca del revisor técnico	XIII
Introducción	XV
Capítulo 1 Bienvenido a Django	1
Capítulo 2 Su primer sitio Django: un CMS simple	9
Capítulo 3 Personalización del CMS Simple	23
Capítulo 4 Un weblog potenciado por Django	43
Capítulo 5 Expansión del Weblog	77
Capítulo 6 Plantillas para el Weblog	97
Capítulo 7 Terminando el Weblog	123
Capítulo 8 Un sitio social de código compartido	149
Capítulo 9 Procesamiento de Formularios en la Aplicación de Código Compartido	165
Capítulo 10 Finalización de la aplicación de código compartido	187
CAPÍTULO 11 Técnicas Prácticas de Desarrollo	205
Capítulo 12 Escritura de aplicaciones Django reutilizables	223
ÍNDICE	243

Contenido

Sobre el autor	xi
Acerca del revisor técnico	XIII
Introducción	XV
Capítulo 1 Bienvenido a Django	1
¿Qué es un marco web y por qué debería querer uno?	1 Saludando
a Django	2 Saludando a
Python	3 Instalación de
Django	4 Dando sus
primeros pasos con Django	5 Explorando su
proyecto Django	7 Mirando hacia el
futuro	8
Capítulo 2 Su primer sitio Django: un CMS simple	9
Configurando tu primer proyecto Django	9
Armando el CMS	12 Introducción al
sistema de plantillas de Django	18 Mirando hacia el
futuro	22
Capítulo 3 Personalización del CMS simple	23
Adición de edición de texto enriquecido	
23 Adición de un sistema de búsqueda al CMS	
26 Mejora de la vista de búsqueda	31
Mejora de la función de búsqueda con palabras clave	33
Mirando hacia el futuro	40

Capítulo 4 Un weblog potenciado por Django	43
Compilación de una lista de verificación de características	43 Escribir una aplicación
Django	44 Proyectos vs
Aplicaciones	44 Aplicaciones
independientes y acopladas	45 Creación de la aplicación
Weblog	45 Diseño de los
modelos	47 Construcción del modelo de entrada ..
Campos básicos	53
Slugs, valores predeterminados útiles y restricciones de unicidad	
54 Autores, comentarios y entradas destacadas	55
Diferentes tipos de entradas	57
Categorización y etiquetado de entradas	58
Escribir entradas sin escribir HTML	60 Toques
finales	61 Los modelos de
Weblog hasta ahora	62 Escritura de
las primeras vistas	65 Uso de las
vistas genéricas de Django	69
Desacoplamiento de las URL	
72 Mirando hacia el futuro	75
Capítulo 5 Expansión del Weblog	77
Escribir el modelo de enlace	77
Vistas para el modelo de enlace	83
Configuración de vistas para categorías	84
Uso de vistas genéricas (otra vez)	86
Vistas de etiquetas	87
Limpieza del módulo URLConf	88 Manejo
de entradas en vivo	93 Mirando
hacia el futuro	95

Capítulo 6 Plantillas para el Weblog	97
Tratando con Elementos Repetitivos: El Poder de la Herencia	97
Cómo funciona la herencia de plantillas	99
Límites de la herencia de plantillas	100
Definición de la plantilla base para el blog	100
Configuración de plantillas de sección	103
Visualización de archivos de entradas	104
Índice de entradas	104
Archivo Anual	105
Archivos mensuales y diarios	106
Detalle de entrada	107
Definición de plantillas para otros tipos de contenido	110
Ampliación del sistema de plantillas con etiquetas personalizadas	111
Cómo funciona una plantilla de Django	112
Una etiqueta personalizada sencilla	
113 Escribir una etiqueta más flexible con argumentos	115
Escritura de la función de compilación para la nueva etiqueta	
116 Escritura del LatestContentNode	119
Registro y uso de la nueva etiqueta	120
Mirando hacia el futuro	122
Capítulo 7 Terminando el Weblog	123
Comentarios y django.contrib.comments	123
Implementación de herencia de modelos y modelos abstractos	123
Instalación de la aplicación de comentarios	124
Realización de la configuración básica	
125 Recuperación de listas de comentarios para visualización	
127 Moderación de comentarios	
129 Uso de señales y Django Dispatcher	129
Creación del moderador automático de comentarios	130
Adición de soporte de Akismet	131
Envío de notificaciones por correo electrónico	135
Uso de las funciones de moderación de comentarios de Django	138

Adición de fuentes	140
Creación de la clase LatestEntriesFeed	140
Generación de entradas por categoría: un ejemplo de feed más complejo.	144
Mirando hacia el futuro	147
Capítulo 8 Un sitio social de código compartido	149
Compilación de una lista de verificación de características	149
Configuración de la aplicación	150
Construcción de los modelos iniciales	150
El modelo de lenguaje	151
fragmentos	153
Prueba de la aplicación	156
Creación de vistas iniciales para fragmentos e idiomas	156
CSS para pigmentos	
Resaltado de sintaxis	158
Vistas para idiomas	159
Vista avanzada: Autores principales	160
Mejorar la vista de los mejores autores	161
Adición de una vista top_languages	162
Mirando hacia el futuro	
Capítulo 9 Procesamiento de formularios en la aplicación de código compartido	165
Un breve recorrido por el sistema de formularios de Django	
165 Un ejemplo sencillo	165
Validación del nombre de usuario	167
Validación de la contraseña	168
Creación del nuevo usuario	168
Cómo funciona la validación de formularios	170
Tramitación del formulario	172
Escribir un formulario para agregar fragmentos de código	
174 Escribir una vista para procesar el formulario	
177 Escribir la plantilla para manejar la vista add_snippet	178
Generación automática del formulario a partir de una definición de modelo	
179 Plantillas simplificadas que muestran formularios	182
Edición de fragmentos	183
Mirando hacia el futuro	186

Capítulo 10 Finalización de la aplicación de código compartido	187
Fragmentos de marcadores	187
Adición de vistas básicas de marcadores	188
Creación de una nueva etiqueta de plantilla: {% if_bookmarked %}	191
Análisis anticipado en una plantilla de Django	192
Resolución de variables dentro de un nodo de plantilla	193
Uso de RequestContext para llenar automáticamente variables de plantilla ..	196
Adición del sistema de clasificación de usuarios	
198 Fragmentos de calificación	
199 Adición de una etiqueta de plantilla {% if_rated %}	
200 Recuperación de la calificación de un usuario	
201 Mirando hacia el futuro	202
CAPÍTULO 11 Técnicas Prácticas de Desarrollo	205
Uso de sistemas de control de versiones para rastrear su código	205
Un ejemplo sencillo	206
Herramientas de control de versiones y opciones de hospedaje	
208 Elección y uso de un VCS	208
Uso de entornos de Python aislados para administrar software	209
Herramientas de construcción	
212 Uso de una herramienta de implementación	
214 Simplificando su proceso de desarrollo de Django	215
Vivir sin proyectos	215
Uso de rutas relativas en la configuración	217
Manejo de configuraciones que cambian para diferentes entornos ..	218
Pruebas unitarias de sus aplicaciones	219
Mirando hacia adelante	222
Capítulo 12 Escritura de aplicaciones Django reutilizables	223
Una cosa a la vez	224
Mantenerse enfocado	224
Ventajas de las aplicaciones estrictamente enfocadas	225
Desarrollo de múltiples aplicaciones	226
Dibujar las líneas entre las aplicaciones	227
Dividir la aplicación de código compartido	228

Construyendo para la Flexibilidad	228
Gestión flexible de formularios	228
229 Gestión flexible de plantillas	230
Procesamiento posterior al formulario flexible	
231 Manejo flexible de URL	232
Aprovechando las API de Django	233
Mantenerse genérico	233
Distribución de aplicaciones Django	234
Herramientas de empaquetado de Python	
234 Escribir un script setup.py con distutils	235
Archivos estándar para incluir en un paquete	236
Documentación de una solicitud	237
Mirando hacia el futuro	241
ÍNDICE	243

Sobre el Autor



James Bennett es desarrollador web en *Lawrence Journal World* en Lawrence, Kansas, donde se desarrolló originalmente Django. Es colaborador habitual y administrador de versiones del proyecto Django de código abierto.

Acerca del revisor técnico

nBen Ford ha estado usando Django desde 2006, tanto en proyectos personales como en entornos más "empresariales". Django también lo ha puesto en el camino de aprender la magia más profunda de Python, incluida la metaprogramación, los decoradores y los descriptores. El viaje continúa.

Introducción

los Los últimos años han visto una explosión en el desarrollo de sitios web dinámicos basados en bases de datos. Mientras que muchos sitios alguna vez se construyeron usando nada más que HTML escrito a mano, o algunos scripts CGI o aplicaciones del lado del servidor, las aplicaciones web respaldadas por bases de datos de hoy en día se han convertido en la norma para todo, desde blogs personales hasta tiendas en línea y sitios de redes sociales que han revolucionado la forma en que muchas personas usan la Web.

Pero esto ha tenido un costo. Desarrollar estas aplicaciones, incluso para tareas relativamente sencillas utiliza, implica una cantidad significativa de trabajo complejo, y gran parte de ese trabajo termina repitiéndose para cada nueva aplicación. Aunque los desarrolladores web siempre han tenido acceso a bibliotecas de código que podrían automatizar ciertas tareas, como la creación de plantillas HTML o la consulta de bases de datos, el proceso de reunir todas las piezas necesarias para una aplicación completamente limpia sigue siendo en gran medida difícil y tedioso.

Este desafío ha llevado al reciente desarrollo y posterior popularidad de los "marcos web". Los frameworks web son colecciones reutilizables de componentes que manejan muchas de las tareas comunes y repetitivas del desarrollo de aplicaciones web de manera integrada. En lugar de pedirle que obtenga bibliotecas de código dispares y encuentre formas de hacer que funcionen juntas, los marcos web proporcionan todos los componentes necesarios en un solo paquete y se encargan del trabajo de integración por usted.

Django es uno de los marcos web más recientes, que surge de las necesidades de una operación de noticias en línea de ritmo rápido. Los desarrolladores originales de Django necesitaban un conjunto de herramientas que no solo los ayudara a desarrollar rápidamente aplicaciones web nuevas y altamente dinámicas en respuesta a los requisitos en rápida evolución de la industria de las noticias, sino que también les permitiera ahorrar tiempo y esfuerzo al reutilizar fragmentos de código e incluso códigos completos. aplicaciones, siempre que sea posible.

En este libro, verá cómo Django puede ayudarlo a lograr estos dos objetivos: rapidez desarrollo de aplicaciones y código flexible y reutilizable, tanto a través de las herramientas que le proporciona directamente como de las prácticas de desarrollo que hace posibles. Lo guiaré a través del desarrollo de varias aplicaciones de ejemplo y le mostraré cómo los diversos componentes y aplicaciones incluidos con Django pueden ayudarlo a escribir menos código en cada etapa del proceso de desarrollo. También verá de primera mano una serie de mejores prácticas para el código reutilizable y aprenderá cómo puede aplicarlas en sus propias aplicaciones. Además, aprenderá cómo integrar bibliotecas de terceros existentes en aplicaciones impulsadas por Django para minimizar la cantidad de código que necesitará escribir desde cero.

He escrito este libro desde un punto de vista pragmático. Todas las aplicaciones de muestra están destinadas a ser útiles en situaciones del mundo real, y una vez que haya trabajado con ellas, tendrá más que solo una comprensión técnica de Django y sus componentes. Tendrá una comprensión clara de cómo Django puede ayudarlo a convertirse en un desarrollador más productivo y efectivo.

Capítulo 1



Bienvenido a Django

Web el desarrollo es difícil, y no dejes que nadie te diga lo contrario. La creación de una aplicación web dinámica y completamente funcional con todas las características que los usuarios desean es una tarea abrumadora con una lista aparentemente interminable de cosas que debe hacer correctamente. Y antes de que pueda comenzar a pensar en la mayoría de ellos, debe hacer una gran cantidad de trabajo por adelantado: configurar una base de datos, crear todas las tablas para almacenar sus datos, planificar todas las relaciones y consultas, llegar a un solución para generar HTML de forma dinámica, descubra cómo asignar direcciones URL específicas a diferentes fragmentos de código, y más. Solo llegar al punto en el que puede agregar funciones que sus usuarios verán o que les interesarán es un trabajo enorme y en gran medida desagradecido.

Pero no tiene por qué ser así.

Este libro le enseñará cómo usar Django, un "marco web" que aliviará significativamente el dolor de embarcarse en nuevos proyectos de desarrollo. Podrás seguirlo mientras creas aplicaciones del mundo real, y en cada paso verás cómo Django está ahí para ayudarte.

Al final, te darás cuenta de algo maravilloso: que el desarrollo web vuelve a ser divertido.

¿Qué es un marco web y por qué debería querer uno?

El mayor inconveniente del desarrollo web es la gran cantidad de tedio que implica. Todas las tareas iniciales antes mencionadas más docenas más se esconden detrás de cada nueva aplicación que desarrolla, y absorben rápidamente toda la alegría incluso de los proyectos más emocionantes. Los trabajos de marcos web como Django tienen como objetivo eliminar todo ese tedio al proporcionar un conjunto organizado y reutilizable de bibliotecas y componentes comunes que pueden hacer el trabajo pesado, liberándolo para trabajar en las funciones que hacen que su proyecto sea único.

Esta idea de estandarizar un conjunto de bibliotecas comunes para hacer frente a tareas comunes está lejos de ser nueva. De hecho, esta estandarización es una práctica tan establecida en la mayoría de las áreas de programación que recibiría miradas extrañas si sugiriera que alguien debería comenzar a escribir código desde cero. Y en el desarrollo web empresarial, se han utilizado marcos de varios tipos durante años. La mayoría de las empresas que habitualmente necesitan desarrollar aplicaciones a gran escala dependen en gran medida de los marcos para proporcionar una funcionalidad común y acelerar sus procesos de desarrollo.

Pero en el mundo del desarrollo web, los marcos han sido tradicionalmente, casi por necesidad, tan pesados como las aplicaciones en las que se utilizan. Tienden a estar escritos en Java o C# y están dirigidos a grandes proyectos de desarrollo corporativo y, a veces, vienen con una etiqueta de precio que solo una compañía Fortune 500 podría amar. Django es parte de un nuevo

generación de marcos dirigidos a una audiencia más amplia: desarrolladores que no necesariamente tienen el peso de las necesidades de un conglomerado multinacional sobre sus hombros, pero que aún necesitan hacer las cosas rápidamente. En otras palabras, Django apunta a desarrolladores como tú y yo.

Los últimos dos años han visto el surgimiento de varios de estos nuevos marcos web que funcionan, escrito en y para lenguajes de programación que son mucho más accesibles para el desarrollador web promedio (y, lo que es igual de importante, para el servidor web promedio): PHP, Perl, Python y Ruby. Cada marco tiene una filosofía ligeramente diferente con respecto a la organización del código y la cantidad de "extras" que incluye, pero todos comparten un objetivo básico común: proporcionar un conjunto integrado y fácil de usar de componentes que manejan las tareas tediosas y repetitivas de desarrollo web con el menor alboroto posible.

Saludando a Django

Django comenzó su vida como un simple conjunto de herramientas utilizadas por el equipo web interno de una empresa de periódicos en una pequeña ciudad universitaria de Kansas. Como cualquiera que dedica suficiente tiempo al desarrollo web, se cansaron de escribir los mismos tipos de código una y otra vez: consultas de bases de datos, plantillas, los nueve metros completos. Se cansaron de esto rápidamente, de hecho, porque se vieron presionados para mantenerse al día con un horario apretado en la sala de redacción. La necesidad de un código personalizado para una gran historia o característica no era (y aún no lo es) inusual, y los plazos de desarrollo debían poder medirse en días, o incluso horas, para seguir el ritmo de las noticias.

En el espacio de un par de años, desarrollaron un conjunto de bibliotecas que funcionaron extremadamente bien. Juntos. Al automatizar o simplificar las tareas comunes del desarrollo web, las bibliotecas les ayudaron a realizar su trabajo de manera rápida y eficiente. En el verano de 2005, obtuvieron el permiso de los gerentes del periódico para lanzar esas bibliotecas al público, de forma gratuita, bajo una licencia de código abierto para que cualquiera pudiera usarlas y mejorarlas. También dieron a estas bibliotecas un nombre elegante, "Django", en honor al famoso guitarrista de jazz gitano Django Reinhardt.

Como corresponde a su herencia de sala de redacción, Django se anuncia a sí mismo como "el marco web para perfeccionistas con fechas límite". En esencia, es un conjunto de bibliotecas sólidas y bien probadas que cubren todos los aspectos repetitivos del desarrollo web:

- Un mapeador relacional de objetos, que es una biblioteca que sabe cómo se ve su base de datos cómo se ve su código y cómo cerrar la brecha entre ellos sin SQL escrito a mano repetitivo
- Un conjunto de bibliotecas HTTP que sabe cómo analizar las solicitudes web entrantes; cómo entregar entregárselas en un formato estándar y fácil de usar; y cómo convertir los resultados de su código en respuestas bien formadas
- Una biblioteca de enrutamiento de URL que le permite definir exactamente las URL que desea y asignarlas las partes apropiadas de su código
- Una biblioteca de validación que lo ayuda a mostrar formularios en páginas web y procesar la suscripción de usuarios datos proporcionados
- Un sistema de plantillas que permite que incluso los no programadores escriban HTML combinado con la generación de datos por su código y la cantidad justa de lógica de presentación

Y eso es solo rascar la superficie. Las bibliotecas centrales de Django incluyen una gran cantidad de otras funciones. turas que llegarás a amar. También se incluyen varias aplicaciones útiles que se basan en las funciones de Django y brindan soluciones listas para usar para necesidades específicas, como interfaces administrativas y autenticación de usuarios. En las aplicaciones de ejemplo utilizadas en este libro, verá todas estas características en acción. Así que vamos a sumergirnos.

Saludando a Python

Django está escrito en un lenguaje de programación llamado Python, por lo que las aplicaciones que desarrolles con él también estarán escritas en Python. Eso significa que necesitará tener Python instalado en su computadora antes de poder comenzar con Django. Puede descargar Python gratis desde <http://python.org/download/>; está disponible para todos los principales sistemas operativos. Mientras escribo esto, el

El lenguaje Python está en proceso de migrar de una serie de versiones principales (con números de versión de la forma "2.x") a otra (con números de versión de la forma "3.x"). Se espera que este proceso tome varios años, y la mayoría del software basado en Python, incluido Django, aún no ha comenzado a migrar a la nueva serie 3.x. Por lo tanto, es mejor instalar la última versión 2.x de Python:

Python 2.6.1 en el momento de escribir este artículo, para disfrutar de las últimas funciones y correcciones de errores para el lenguaje Python mientras usa Django.

Una vez que haya instalado Python, debería poder abrir un símbolo del sistema (Command Prompt en Windows, Terminal en Mac OS X o cualquier emulador de terminal en Linux) e inicie el intérprete interactivo de Python escribiendo el comando `python`. Normalmente, guardará su código de Python en archivos que se ejecutarán como parte de sus aplicaciones. Pero el intérprete interactivo le permitirá explorar Python, y Django, una vez que esté instalado, de una manera más libre: el intérprete le permite escribir el código de Python, una línea a la vez, y ver los resultados de inmediato. También puede usarlo para acceder e interactuar con el código en sus propios archivos de Python, código en las bibliotecas estándar de Python o código en cualquier biblioteca de terceros que haya instalado. Esta capacidad convierte al intérprete interactivo en una poderosa herramienta de aprendizaje y depuración.

Advertencia: aprendiendo Python

Si no sabes nada de Python, o incluso si nunca antes has hecho nada de programación, no te preocunes. Python es fácil de aprender y no necesita saber mucho para comenzar con Django. De hecho, muchos usuarios primerizos de Django aprenden Python y Django al mismo tiempo. (Cuando comencé con Python, aprendí los conceptos básicos en un fin de semana leyendo tutoriales en línea).

Llamaré la atención sobre conceptos importantes de Python cuando sea necesario, pero le recomiendo que consulte un tutorial de Python antes de profundizar en este libro. El índice de documentación de Python en <http://python.org/doc/> presenta una buena lista de tutoriales y libros (varios de los cuales están disponibles de forma gratuita en línea) para ayudarlo a aprender los conceptos básicos de Python. (Recomiendo saber al menos cómo funcionan las funciones y clases de Python). Podrá recoger el resto a medida que avanza.

Si está buscando una buena referencia para tener a mano mientras aprende Django, consulte *Beginning Python: From Novice to Professional, Second Edition* de Magnus Lie Hetland y *Dive Into Python* de Mark Pilgrim (ambos de Apress).

Cuando inicie por primera vez el intérprete de Python, verá algo como esto:

```
Python 2.6.1 (r261:67515, 2 de abril de 2009, 01:36:23)
[GCC 4.0.1 (Apple Computer, Inc. compilación 5488)] en darwin
Escriba "ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información.
>>>
```

El >>> es el símbolo del sistema de Python. Puede escribir una línea de código de Python y presionar Entrar, y si ese código devuelve un resultado, lo verá de inmediato. Para probar esto, pruebe con una línea simple que imprima algo de texto. Abra el intérprete de Python, escriba la siguiente línea en el indicador y luego presione la tecla Intro:

```
>>> imprimir "¡Hola, mundo!"
```

Verás que el resultado aparece en la siguiente línea:

```
¡Hola Mundo!
```

```
>>>
```

Cualquier cosa que pueda escribir en un archivo como parte de un programa Python puede escribirse directamente en el intérprete. También puede acceder al sistema de ayuda incorporado escribiendo help() y presionando Enter. Cuando esté listo para salir del intérprete de Python, presione Ctrl+D para cerrarlo.

Instalando Django

Ahora que tiene Python instalado y funcionando, es hora de instalar Django y comenzar a explorar sus funciones. Puede obtener una copia del sitio web oficial de Django; visite www.djangoproject.com/download/ y siga las instrucciones para descargar la última versión oficial (que debería ser Django 1.1 en el momento de la publicación de este libro).

Advertencia: lanzamientos empaquetados frente a código de desarrollo

Django siempre se actualiza y mejora. Entonces, además del lanzamiento oficial, el código actual en desarrollo está disponible para descargar en forma de una "versión de desarrollo". El sitio web de Django ofrece instrucciones para instalar la versión de desarrollo en su computadora.

La ventaja de usar la versión de desarrollo es que puede usar inmediatamente las nuevas funciones tan pronto como se agregan, en lugar de esperar al próximo lanzamiento oficial. La desventaja, por supuesto, es que el código en desarrollo todavía está experimentando cambios y, por lo tanto, puede contener errores u otros problemas que aún no se han solucionado.

A lo largo de este libro, supondré que está utilizando la última versión oficial de Django. Una vez que eres un poco más cómodo con Django, sin embargo, debe sentirse libre de comenzar a explorar el código en desarrollo para tener una idea de las nuevas funciones que estarán disponibles en futuras versiones.

Una vez que haya descargado el código Django en su computadora, puede instalarlo escribiendo un solo comando. En Linux o Mac OS X, abra una terminal, navegue hasta el directorio donde se descargó Django y busque un archivo llamado `setup.py`. Escriba el siguiente comando e ingrese su contraseña cuando se le solicite:

```
sudo python setup.py instalar
```

En Windows, deberá abrir un símbolo del sistema con privilegios administrativos. Luego puede navegar al directorio de Django y escribir lo siguiente:

```
instalar python setup.py
```

El script `setup.py` es un procedimiento de instalación estándar para los módulos de Python y se encarga de instalar todo el código Django relevante en las ubicaciones correctas para su sistema operativo. Si tiene curiosidad, la Tabla 1-1 resume dónde terminará el código Django en varios sistemas.

Tabla 1-1. Ubicaciones de instalación de Django

Sistema operativo	Ubicación de Django
linux	/usr/local/lib/python2.6/site-packages/django
Mac OS X	/Biblioteca/Frameworks/Python.framework/Versions/2.6/lib/python2.5/sitio-paquetes/django
ventanas	C:\Python\paquetes de sitio\django

Dando sus primeros pasos con Django

Ahora debería poder verificar que Django se instaló correctamente en su computadora. A continuación, inicie el intérprete interactivo de Python y escriba lo siguiente:

```
>>> importar django  
>>> imprimir django.VERSION
```

La ejecución de estos comandos debería mostrar un conjunto de números entre paréntesis, que representa la versión de Django que está utilizando. La versión Django 1.1, por ejemplo, mostrará (1, 1, 0, 'final', 0). El software de Python generalmente usa una *tupla de versión* (una lista de números y/o palabras entre paréntesis y separados por comas) para representar los números de versión internamente, y Django no es diferente. (Esta tupla de versión facilita que los programas de Python analicen automáticamente números de versión complejos, como "1.0 beta 3" o "2.4 versión preliminar").

Ahora estás listo para crear tu primer proyecto Django. Un *proyecto* de Django es una especie de contenedor, que contiene una lista de una o más aplicaciones impulsadas por Django y la configuración que utilizan. Más adelante, cuando esté implementando sus aplicaciones Django detrás de un servidor web real, usará proyectos para configurarlas.

Para configurar su primer proyecto, cree un directorio en su computadora donde guardará sus proyectos de Django en progreso y luego navegue hasta él usando una terminal o un símbolo del sistema. Suele ser una buena idea tener un solo directorio donde guarde todo su propio código Python personalizado, así que elija un solo lugar lógico en su computadora para esto. Como verá un poco más adelante, hacerlo simplificará el proceso de decirle a Python cómo encontrar y usar ese código.

Ahora puede usar el script de administración incorporado de Django, django-admin.py, para crear su proyecto. django-admin.py vive en el subdirectorio bin/ del directorio en el que se instaló Django, y sabe cómo manejar varias tareas de administración que involucran proyectos de Django. El comando que le interesa se llama startproject, que creará un nuevo proyecto Django vacío. En el directorio donde desea crear su proyecto, escriba lo siguiente (consulte la Tabla 1-1 para conocer la ruta correcta para su sistema operativo):

```
/usr/local/lib/python2.6/site-packages/django/bin/django-admin.py startproject cms
```

Esto creará un nuevo subdirectorio llamado cms y lo llenará con los archivos básicos necesarios por cualquier proyecto de Django. (Verá por qué se llama cms en el próximo capítulo, cuando comience a trabajar con este proyecto).

Advertencia: errores de permisos

Si usa Linux o Mac OS X, es posible que vea un mensaje de error que dice "permiso denegado". Si esto sucede, debe informar a su sistema operativo que el script django-admin.py es seguro para ejecutarse como un programa.

Puede hacerlo navegando al directorio donde reside django-admin.py y escribiendo el comando chmod +x django-admin.py. Luego puede ejecutar el script django-admin.py como se mostró anteriormente.

En la siguiente sección, verá para qué sirve cada uno de los archivos en el directorio del proyecto, pero concéntrese en manage.py por ahora. Al igual que django-admin.py, el script manage.py se ocupa de las tareas comunes de gestión de proyectos y aplicaciones. Por ejemplo, puede iniciar un servidor web simple que alojará su proyecto con fines de prueba. Puede iniciar el script manage.py yendo al directorio de su proyecto y escribiendo lo siguiente:

```
python manage.py servidor de ejecución
```

Entonces debería poder abrir un navegador web y visitar la dirección http://127.0.0.1:8000/. De forma predeterminada, el servidor web de desarrollo se ejecuta en la dirección de red de "bucle invertido" local de su computadora, que siempre es 127.0.0.1, y se vincula al puerto 8000. Cuando visite esa dirección, debería ver una página simple que dice "¡Funcionó!" con algunas instrucciones básicas para personalizar su proyecto (vea la Figura 1-1).

Advertencia: cambiar la dirección y el puerto

Si algo más ya está usando el puerto 8000 en su computadora, si no puede ejecutar programas que se vinculan a ese puerto, o si desea ver las páginas servidas por el servidor de desarrollo de Django desde otra computadora, deberá especificar manualmente la dirección y el puerto que se usarán cuando inicie el servidor de desarrollo. Esto se logra utilizando la sintaxis python manage.py runserver ip_address:port_number.

Por ejemplo, para escuchar en todas las direcciones IP disponibles de su computadora (para otras computadoras puedan ver páginas del servidor de desarrollo) y enlazar al puerto 9000 en lugar de 8000, podría escribir python manage.py runserver 0.0.0.0:9000 (0.0 .0.0 es una dirección especial que significa "escuchar en todas las direcciones IP disponibles").

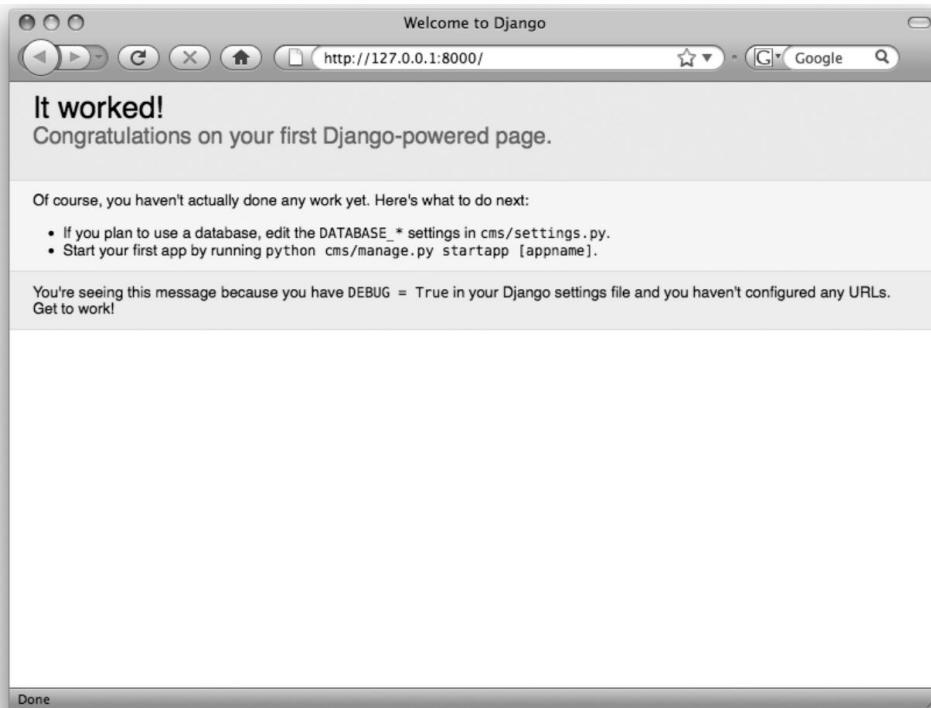


Figura 1-1. Pantalla de bienvenida de Django

Puede detener el servidor presionando Ctrl+C en el símbolo del sistema.

Explorando tu proyecto Django

El comando `startproject` de `django-admin.py` creó el directorio de su proyecto y automáticamente llenó algunos archivos. Aquí hay una introducción rápida a estos archivos, los cuales explicaré más adelante en capítulos futuros:

`__init__.py`: Este será un archivo vacío. Por ahora no necesita poner nada en él (y de hecho, la mayoría de las veces no necesitará hacerlo). Se usa para decirle a Python que su directorio contiene código ejecutable. Python puede tratar cualquier directorio que contenga un archivo `__init__.py` como un módulo de Python.

`manage.py`: Como expliqué anteriormente, este es un script de ayuda que sabe cómo manejar las tareas de administración comunes. Sabe cómo iniciar el servidor web de desarrollo incorporado, crear nuevos módulos de aplicaciones, configurar su base de datos y hacer muchas otras cosas que verá mientras construye sus primeras aplicaciones Django.

`settings.py`: este es un *módulo de configuración de Django*, que contiene la configuración de su proyecto Django. En los próximos capítulos, verá algunas de las configuraciones más comunes y cómo editarlas para adaptarlas a sus proyectos.

`urls.py`: este archivo contiene la configuración de la URL maestra de su proyecto. A diferencia de algunos lenguajes y marcos que simplemente imitan HTML al permitirle colocar el código en el directorio público del servidor web y acceder a él directamente por nombre de archivo, Django usa un archivo de configuración explícito para establecer qué URL apuntan a qué partes de su código. Este archivo define el conjunto de direcciones URL "raíz" para un proyecto completo.

Es posible que observe que después de iniciar el servidor web incorporado, aparecieron uno o más archivos nuevos en el directorio del proyecto con los mismos nombres que los de la lista anterior pero con una extensión `.pyc` en lugar de una extensión `.py`. Python puede leer el código directamente de sus archivos `.py`, pero también puede, y con frecuencia lo hace, compilar automáticamente el código en una forma que es más rápida de cargar cuando se inicia un programa. Este *código de bytes*, como se le llama, se almacena en archivos `.pyc` con el mismo nombre. Si el archivo original no ha cambiado desde la última vez que un programa lo usó, Python se cargará desde el archivo de código de bytes en lugar del archivo original para aumentar la velocidad.

Mirando hacia el futuro

En el próximo capítulo, verá cómo configurar su primer proyecto real de Django, que proporcionará un sistema de administración de contenido simple, o CMS. Si está listo para sumergirse, siga leyendo, pero también puede hacer una pausa y explorar Python o Django un poco más por su cuenta. Tanto los scripts `django-admin.py` como `manage.py` aceptan un comando de ayuda, que enumerará todas las cosas que pueden hacer. Además, el sistema de ayuda incorporado del intérprete de Python también puede extraer automáticamente la documentación de la mayoría de los módulos de Python en su computadora, incluidos los que se encuentran dentro de Django. También hay un comando de shell especial para `manage.py` que puede resultarle útil: utiliza el módulo de configuración de su proyecto para iniciar un intérprete de Python con un entorno Django totalmente configurado que puede explorar.

Si lo desea, también puede aprovechar esta oportunidad para configurar una base de datos para usar con Django. Si instaló Python 2.5 o cualquier versión posterior, no tendrá que hacerlo de inmediato. A partir de la versión 2.5, Python incluye directamente el sistema ligero de base de datos SQLite, que podrá utilizar a lo largo de este libro a medida que desarrolle sus primeras aplicaciones. Sin embargo, Django también admite bases de datos MySQL, PostgreSQL y Oracle, por lo que si prefiere trabajar con una de ellas, configúrela.

Capítulo 2



Su primer sitio de Django: Un CMS sencillo

Una tarea extremadamente común en el desarrollo web es crear un sistema de administración de contenido (CMS) simple, que permite a los usuarios crear y editar dinámicamente páginas en un sitio a través de una interfaz basada en la web. A veces llamados sitios de *folletos* porque tienden a usarse de la misma manera que los folletos impresos tradicionales que entregan las empresas, por lo general tienen características bastante simples, pero pueden ser tediosos de codificar una y otra vez.

En este capítulo, verá cómo Django hace que este tipo de sitio sea casi trivialmente fácil de construir. Lo guiaré a través de la configuración de un CMS simple, y luego, en el próximo capítulo, verá cómo agregar algunas funciones adicionales y brindar espacio para expandirlo en el futuro.

Configurando tu primer proyecto Django

En el último capítulo, creó un proyecto de Django llamado `cms`. Pero antes de que pueda hacer mucho con él, deberá realizar una configuración básica. Así que inicie su programa de edición de código favorito y utilícelo para abrir el archivo `settings.py` en su proyecto.

Advertencia: escribir Python

Desde aquí hasta el final de este libro, escribirá código Python y, ocasionalmente, una plantilla. Si aún no ha visto un tutorial de Python para tener una idea de los conceptos básicos, ahora sería un buen momento. Explicaré algunos de los conceptos más importantes a medida que avanzamos, pero debe consultar un tutorial de Python dedicado para explorarlos con más profundidad.

Y si no tiene un programa de edición adecuado para trabajar con código de programación, querrá obtener una. Casi todos los editores de programadores ofrecen soporte integrado para Python (y otros lenguajes populares), lo que simplificará el proceso de escritura de código.

No se desanime por el tamaño del archivo `settings.py` o la cantidad de configuraciones que encontrará en él. `django-admin.py` completó automáticamente los valores predeterminados para muchos de ellos y, por ahora, la mayoría de los valores predeterminados estarán bien. Cerca de la parte superior del archivo hay un grupo de configuraciones cuyos nombres comienzan todos

con BASE DE DATOS_. Estas configuraciones le dicen a Django qué tipo de base de datos usar y cómo conectarse a ella, y en este momento eso es todo lo que necesita completar.

Si instaló la última versión de Python, ya tendrá un módulo de adaptador de base de datos que puede comunicarse con bases de datos SQLite (Python 2.5 y versiones posteriores incluyen este módulo en la biblioteca estándar de Python). SQLite es un excelente sistema para usar cuando comienza a explorar Django porque almacena toda la base de datos en un solo archivo en su computadora, y no requiere ninguna configuración compleja de servidor o permisos de otros sistemas de base de datos.

Para usar SQLite, solo necesita cambiar dos configuraciones. Primero, busque la configuración DATABASE_ENGINE y cambiarlo de esto:

```
"  
MOTOR_BASE_DATOS = "
```

a esto:

```
MOTOR_BASE_DATOS = 'sqlite3'
```

Ahora necesita decirle a Django dónde encontrar el archivo de la base de datos SQLite; esta información va en la configuración DATABASE_NAME. Puede colocar el archivo en cualquier lugar del disco duro de su computadora donde tenga permiso para leer y escribir archivos. Incluso puede completar un nombre de archivo inexistente, y el motor de la base de datos SQLite creará el archivo automáticamente. Mantener el archivo de la base de datos dentro de la carpeta de su proyecto no es una mala idea en este caso, así que adelante y hágalo. Guardo todos mis proyectos de Django en una carpeta llamada django-projects dentro de mi directorio de inicio (en una computadora portátil con Mac OS X), así que lo llenaré así:

```
DATABASE_NAME = '/Usuarios/jbennett/django-projects/cms/cms.db'
```

Esta ruta se verá un poco diferente en otros sistemas operativos, por supuesto. En Windows podría ser C:\Documents and Settings\jbennett\django-projects\cms\cms.db, por ejemplo, mientras que en un sistema Linux podría ser /home/jbennett/django-projects/cms/cms.db.

Le digo a Django que el archivo de la base de datos SQLite debe vivir dentro del directorio del proyecto cms con un nombre de archivo de cms.db. No se requiere la extensión de archivo .db, pero me ayuda a recordar para qué es ese archivo, por lo que le recomiendo que use una convención de nomenclatura similar.

Advertencia: usar una base de datos diferente

Si desea configurar una base de datos MySQL, PostgreSQL u Oracle en lugar de usar SQLite, consulte la documentación de configuración de Django en línea en www.djangoproject.com/documentation/settings/ para ver los valores correctos para la configuración de la base de datos. Sin embargo, tenga en cuenta que también necesitará instalar un módulo adaptador de Python para la base de datos que está utilizando; a partir de Python 2.5, SQLite es el único sistema de base de datos compatible directamente con la biblioteca estándar de Python.

Si está utilizando una versión de Python anterior a la 2.5, debe instalar un módulo adaptador para su base de datos, independientemente de la base de datos que utilice. Para más detalles, consulte las instrucciones de instalación de Django en www.djangoproject.com/documentation/install/#get-your-database-running.

Finalmente, probablemente querrá cambiar la configuración TIME_ZONE, que le dice a Django qué zona horaria usar cuando muestra las fechas y horas de su base de datos. Su base de datos generalmente almacena fechas y horas como marcas de tiempo coordinadas (UTC) de hora universal (UTC es la zona horaria "base" anteriormente conocida como hora media de Greenwich o GMT). En lugar de utilizar un nombre de zona horaria específico del país (como la hora estándar central de EE. UU.) o una compensación UTC confusa (como UTC-6), la configuración TIME_ZONE utiliza nombres en formato zoneinfo . Este formato estándar, utilizado por muchos sistemas operativos de computadora, es fácil de leer para los humanos. La configuración predeterminada es

```
ZONA_HORARIA = "América/Chicago"
```

que es equivalente a la zona horaria del centro de EE. UU., seis horas menos que UTC. Las listas completas de nombres de zonas horarias de zoneinfo están disponibles en línea, y la documentación oficial de configuración de Django en www.djangoproject.com/documentation/settings/ incluye un enlace a una de esas listas. Debes cambiar tu configuración de TIME_ZONE a la zona en la que vives.

Advertencia: zonas horarias en Windows

Si está utilizando Microsoft Windows, tenga cuidado con la configuración de TIME_ZONE. Debido a las peculiaridades del entorno operativo de Windows, no es posible utilizar de forma fiable una zona horaria distinta de la que está utilizando actualmente la computadora. Entonces, para obtener los mejores resultados, especifique TIME_ZONE para que sea la misma que la zona horaria que usa Windows.

No necesitará cambiarlo todavía, pero busque una configuración llamada INSTALLED_APPS desplazándose hacia abajo hasta la parte inferior del archivo de configuración. Como se mencionó anteriormente, un proyecto de Django se compone de una o más aplicaciones impulsadas por Django, y esta configuración le dice a Django qué aplicaciones está usando su proyecto. El valor predeterminado se ve así:

```
APLICACIONES_INSTALADAS =  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
)
```

Cada uno de estos es una aplicación incluida con Django, y cada uno proporciona una pieza útil de funcionalidad común. django.contrib.auth, por ejemplo, proporciona un mecanismo para almacenar datos sobre los usuarios y para autenticarlos. django.contrib.sites le permite ejecutar múltiples sitios web desde un único proyecto Django y especificar qué elementos de su base de datos deben ser accesibles para cada sitio.

Con el tiempo, verá ejemplos de estas aplicaciones en acción, pero por ahora es mejor dejar los valores predeterminados como están. Proporcionan un "inicio rápido" para su proyecto al ocuparse de muchas tareas de inmediato, y pronto desarrollará su funcionalidad.

Ahora que le ha dado a Django algunos datos básicos de configuración, puede decirle que configure su base de datos. Abra una terminal o símbolo del sistema, navegue hasta el directorio de su proyecto y escriba este comando:

```
python administrar.py syncdb
```

Advertencia: lo que sucede durante syncdb

Cuando ejecutas `manage.py syncdb`, Django en realidad hace varias cosas en orden, y la salida en tu pantalla muestra cada paso. Primero, Django busca en cada módulo de aplicación listado en `INSTALLED_APPS` y encuentra los modelos de datos. Estas son clases de Python que definen los diferentes tipos de datos que usa la aplicación, y Django sabe cómo generar automáticamente declaraciones SQL `CREATE TABLE` apropiadas a partir de ellas. En el Capítulo 3, escribirá su primer modelo de datos y verá cómo Django genera el SQL para él.

Una vez que se han creado las tablas de la base de datos, Django encuentra y ejecuta cualquier inicialización específica de la aplicación. código para cada aplicación. En este caso, `django.contrib.auth` incluye un código que le solicita que cree una cuenta de usuario.

Finalmente, Django finaliza la configuración de la base de datos e instala cualquier dato inicial que haya proporcionado. El conjunto predeterminado de aplicaciones empaquetadas no utiliza esta función, pero más adelante verá cómo proporcionar un archivo de datos inicial que puede poner en marcha una aplicación brindándole datos para trabajar de inmediato. No proporcionará ningún dato inicial con esta aplicación CMS, pero algunas de las aplicaciones integradas de Django sí proporcionan datos que se insertarán en la base de datos cuando se instalen.

Este comando creará el archivo de la base de datos si es necesario y luego creará las tablas de la base de datos. para cada aplicación enumerada en la configuración `INSTALLED_APPS`. Primero verá varias líneas de salida desplazándose. Luego, debido a que se está instalando la aplicación de autenticación de usuario incluida, Django le preguntará si desea crear una cuenta de "superusuario" para la administración basada en web. Escriba `sí` y luego ingrese un nombre de usuario, una dirección de correo electrónico y una contraseña cuando se le solicite. Verá en breve cómo puede usar esta cuenta para iniciar sesión en una interfaz administrativa de Django.

Armando el CMS

La mayoría de las aplicaciones que creará con Django requerirán que escriba una buena cantidad de código por su cuenta. Django se encargará del trabajo pesado y de las tareas repetitivas, pero seguirá dependiendo de usted manejar las características únicas de cada aplicación específica. A veces, sin embargo, las funciones integradas en Django o las aplicaciones incluidas en él proporcionarán la mayor parte o la totalidad de lo que necesita. Las aplicaciones contrib incluidas con Django, por ejemplo, brindan una funcionalidad que probablemente reutilizará de un proyecto a otro.

Construirá su sencillo CMS de folletos confiando en gran medida en dos de las contribuciones de Django. aplicaciones: `django.contrib.flatpages` y `django.contrib.admin`.

El primero de estos, `django.contrib.flatpages`, proporciona un modelo de datos para una página simple, que incluye un título, contenido y algunas opciones configurables, como plantillas personalizadas o autenticación. La otra aplicación, `django.contrib.admin`, proporciona una potente interfaz administrativa que puede funcionar con cualquier modelo de datos de Django, lo que le permite crear una interfaz web más o menos "instantánea" para administrar un sitio.

El primer paso aquí es agregar estas aplicaciones a la configuración `INSTALLED_APPS`. Recuerda que Django colocó cuatro aplicaciones en la lista por defecto. Ahora puedes añadir dos más:

```
APLICACIONES_INSTALADAS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.flatpages',
)
```

Una vez que haya realizado ese cambio y haya guardado su archivo de configuración, vuelva a ejecutar syncdb:

```
python administrar.py syncdb
```

Verá que la salida se desplaza a medida que Django crea tablas de base de datos para los modelos de datos definidos en estas aplicaciones. Ahora abra el archivo urls.py de su proyecto, que, como vio en el capítulo anterior, contiene la configuración de la URL raíz de su proyecto. Para habilitar la aplicación administrativa, siga las instrucciones para "quitar el comentario" de las líneas en dos lugares de este archivo: dos líneas cerca de la parte superior del archivo que contiene declaraciones de importación y una línea cerca de la parte inferior, que cubriré en breve.

nNota Los comentarios de Python son líneas que comienzan con el carácter "#" y que no se ejecutan como código. Proporcionan información a una persona que lee el archivo o apuntan a un código que se ha desactivado temporalmente. (El autor podría haber deshabilitado el código porque era necesario desactivar alguna función momentáneamente o porque era necesario rastrear un error).

En cada uno de estos lugares en urls.py, descomente las líneas de código eliminando el comentario marcador al principio de la línea y el espacio que le sigue. (Eliminar el espacio es importante, porque Python interpreta los espacios como indicadores de la estructura del código). Luego guarde el archivo. Ahora el archivo urls.py de su proyecto importa el código necesario de la aplicación de administración e incluye las URL necesarias para que funcione.

Ahora podrá volver a iniciar el servidor web incorporado y ver la interfaz administrativa:

```
python manage.py servidor de ejecución
```

El patrón de URL para la aplicación de administración es ^admin/, lo que significa que si visita <http://127.0.0.1:8000/admin/> en su navegador web, verá la página de inicio de sesión. Ingrese el nombre de usuario y la contraseña que usó cuando syncdb le solicitó que creara una cuenta de usuario y verá la página de índice de administración principal (consulte la Figura 2-1). Pero tenga en cuenta que las URL que comienzan con admin/ son las únicas que funcionarán en este momento; aún no ha configurado ninguna otra URL.

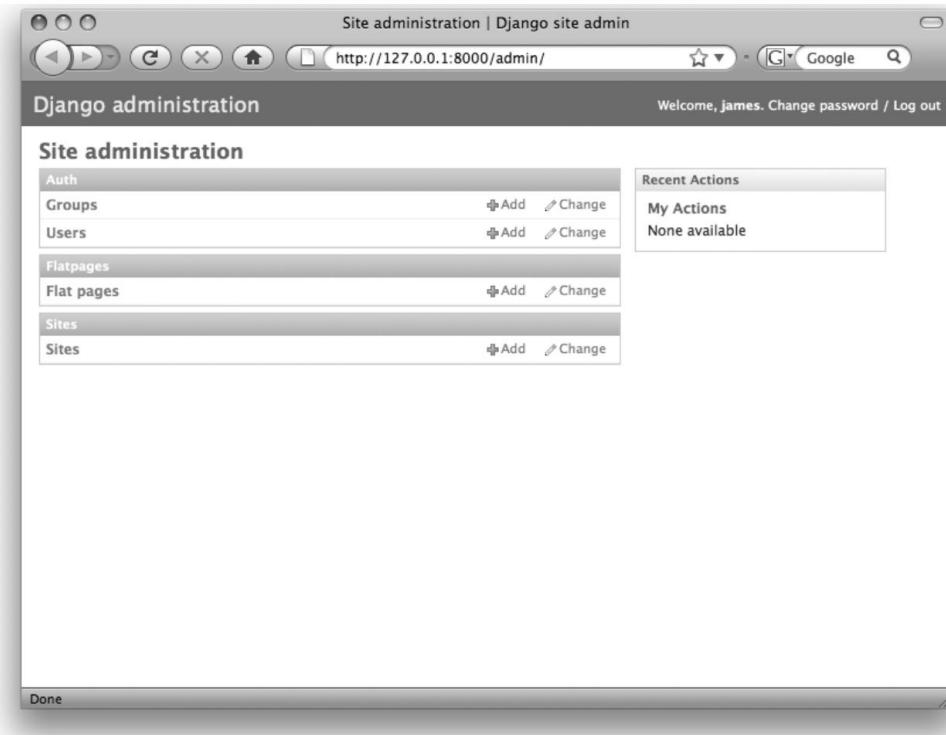


Figura 2-1. Página de inicio de la interfaz administrativa de Django

Advertencia: cómo funciona la configuración de URL de Django

Un archivo de configuración de URL de Django, o `URLConf`, define una lista de patrones de URL e indica cómo se asignan a partes de su código. Cada patrón de URL tiene al menos dos partes. La primera parte es una expresión regular que describe cómo se ve la URL. La segunda parte es una vista (una función de Python que puede responder a solicitudes HTTP) para asignar esa URL, o una inclusión, que apunta a un módulo `URLConf` diferente. La capacidad de incluir otros módulos de `URLConf` facilita la definición de conjuntos de URL reutilizables y "conectables", que se pueden colocar en cualquier punto de la jerarquía de URL de su proyecto.

Una expresión regular, en caso de que nunca haya encontrado ese término antes, es una forma común de representar un patrón particular de texto. La mayoría de los lenguajes de programación permiten verificar si un fragmento de texto determinado coincide con el patrón especificado en una expresión regular, y la mayoría de los libros de introducción a la programación cubren expresiones regulares. *Dive Into Python* de Mark Pilgrim (Apress, 2004) tiene un buen capítulo que cubre los conceptos básicos.

Además, tenga en cuenta que las expresiones regulares son bastante estrictas con respecto a la coincidencia. Normalmente, un servidor web será algo laxo y tratará a `/admin` y `/admin/` como la misma URL, por ejemplo, devolverá el mismo resultado de cualquier manera. Pero si especifica una expresión regular que incluye una barra al final, como lo hace su `urls.py` para el administrador, debe incluir la barra al final cuando visite esa dirección en su navegador. Si no lo hace, el patrón no coincidirá y obtendrá un error de "Página no encontrada".

Cada elemento enumerado en la página de índice corresponde a un modelo de datos en una de las aplicaciones instaladas. Los elementos se agrupan según la aplicación a la que pertenecen. La autorización aplicación, django.contrib.auth, proporciona modelos para usuarios y grupos; la aplicación de sitios, django.contrib.sites, proporciona un modelo para representar un sitio web; y la aplicación de páginas planas que acaba de instalar proporciona un modelo de "página plana". A la derecha de esta lista hay una barra lateral de Acciones recientes, que informa las acciones que ha realizado recientemente en la interfaz de administración. Ahora está vacío porque aún no ha hecho nada, pero mostrará un resumen de sus acciones tan pronto como comience a realizar cambios en el contenido del sitio. Como primer paso, haga clic en el vínculo Sitios. Verá una pantalla como la que se muestra en la Figura 2-2.

Como parte de su inicialización, django.contrib.sites creó un "objeto" de sitio de ejemplo para usted, que puede hacer clic para editar. Debido a que el servidor web incorporado se ejecuta en la interfaz de bucle invertido local de su computadora en el puerto 8000, cambie el campo Nombre de dominio a **127.0.0.1:8000** y cambie el campo Nombre para mostrar a **localhost**. Luego haga clic en el botón Guardar en la esquina inferior derecha para guardar los cambios en la base de datos. Si regresa al índice principal de la interfaz de administración, verá que la barra lateral Acciones recientes ahora tiene una entrada para ese sitio, lo que muestra que lo ha cambiado recientemente.

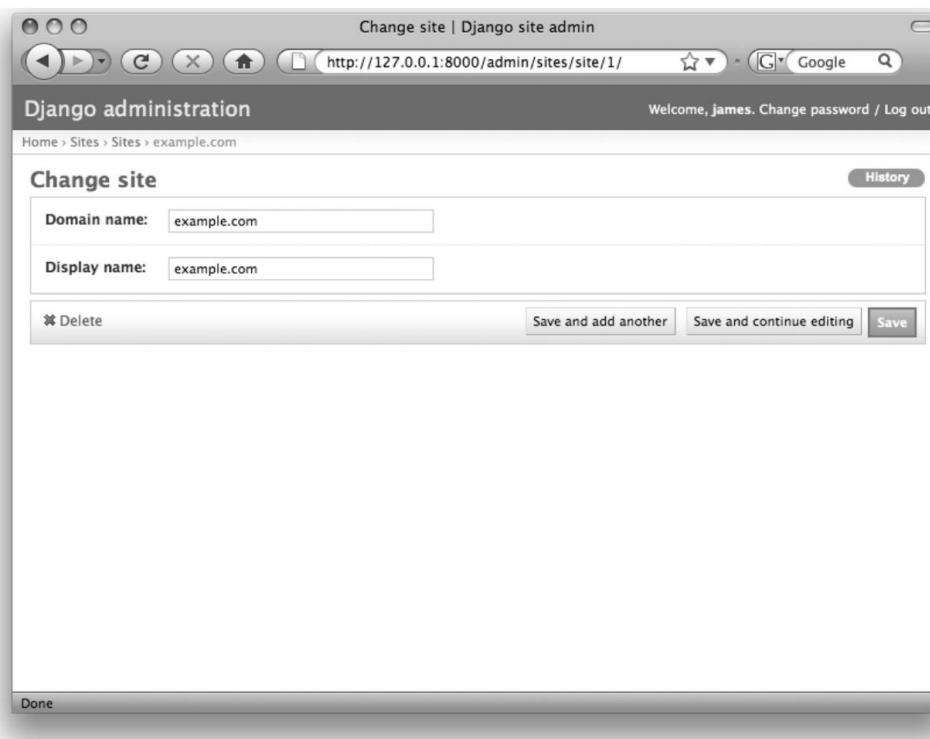


Figura 2-2. El objeto de sitio predeterminado creado por Django

Notará que la página de administración principal muestra un enlace Agregar y un enlace Cambiar junto a cada tipo de elemento (consulte la Figura 2-1). Agregue una nueva página plana haciendo clic en el enlace Agregar junto al enlace Páginas planas. Aparecerá un formulario en blanco, generado automáticamente a partir del modelo de datos apropiado. Introduzca los siguientes valores:

- En el campo URL, ingrese **/primera página/**.
- En el campo Título, ingrese **Mi primera página**.
- En el campo Contenido, ingrese **Esta es mi primera página plana de Django**.

Luego, desplácese hacia abajo y haga clic en el botón Guardar y continuar con la edición. Django guardará la nueva página plana en su base de datos y luego volverá a mostrar el formulario para que pueda editar la página. También notará que han aparecido dos botones sobre el formulario: Historial y Ver en el sitio. El botón Historial muestra un historial simplificado de esta página plana (en este momento, solo se ha creado la entrada inicial). El botón Ver en el sitio le permite ver la página plana en su URL pública. Hacer clic en el botón Ver en el sitio lo redirige a `http://127.0.0.1:8000/first-page/`, que, por el momento, mostrará un mensaje de error como el que se muestra en la Figura 2-3.



Figura 2-3. Un error de Django "Página no encontrada"

Este es un error 404 "Página no encontrada", pero con un giro: cada nuevo proyecto de Django comienza en modo de depuración, que muestra mensajes de error más útiles para ayudarlo a ponerse en marcha. En este caso, Django le muestra los patrones de URL que encontró en la URLConf de su proyecto y explica que la URL que intentó visitar no coincidía con ninguno de ellos. Esto tiene sentido porque aún no ha agregado nada que se parezca a la URL /primera página/. Así que arreglemos eso. Abra el archivo urls.py nuevamente y agregue la siguiente línea justo debajo del patrón de URL para la interfaz de administración:

```
(r'', include('django.contrib.flatpages.urls')),
```

La parte del patrón de esto es simplemente una cadena vacía (""), que en la sintaxis de expresión regular significa que realmente coincidirá con *cualquier* URL. Si quisiera, podría ingresar a urls.py y agregar una nueva línea cada vez que agregue una página plana. En su mayoría, definirá URL individuales en aplicaciones que desarrollará más adelante, pero debido a que django.contrib.flatpages le permite especificar cualquier cosa para la URL de una página, en este caso es más fácil simplemente colocar un patrón de URL "catch-all" para manejarlo.

Advertencia: orden de los patrones de URL

Cuando Django intenta hacer coincidir una URL, comienza en la parte superior de la lista de patrones de URL y avanza hacia abajo hasta que encuentra una coincidencia. Esto significa que es mejor tener patrones más específicos como la línea ^admin/ en primer lugar, y patrones más generales como el catch-all para páginas planas en último lugar; de lo contrario, algo como el catch-all podría coincidir con una URL antes de que Django llegue al patrón más específico que realmente querías.

El patrón de URL para el administrador simplemente especificó admin.site.root para manejar cualquier entrada Solicitud HTTP de una URL que coincide con su expresión regular. (admin.site.root es una *vista de Django*, que responde a una solicitud HTTP). Pero este nuevo patrón para páginas planas incluye una función que le dice a Django que use un módulo URLConf diferente (django.contrib.flatpages.urls) para solicitudes que coincidan con su expresión regular. El uso de include como este le permite "conectar" rápidamente diferentes conjuntos de URL cuando y donde los necesite.

Además, tenga en cuenta que en lugar de especificar URLConf a través de su ubicación en el disco (como django/contrib/flatpages/urls.py), la sintaxis lo especifica usando el mismo estilo que usa al importar código de Python: nombres de módulos y submódulos separados por puntos. Este es un patrón común en Python porque hay funciones que pueden realizar dinámicamente las mismas tareas que la declaración de importación. Encontrarás el patrón extremadamente útil.

Guarde su archivo urls.py y actualice la página en su navegador o navegue nuevamente a <http://127.0.0.1:8000/first-page/>. La página aún muestra un error, pero ahora está más cerca de tener el CMS simple funcionando (vea la Figura 2-4).

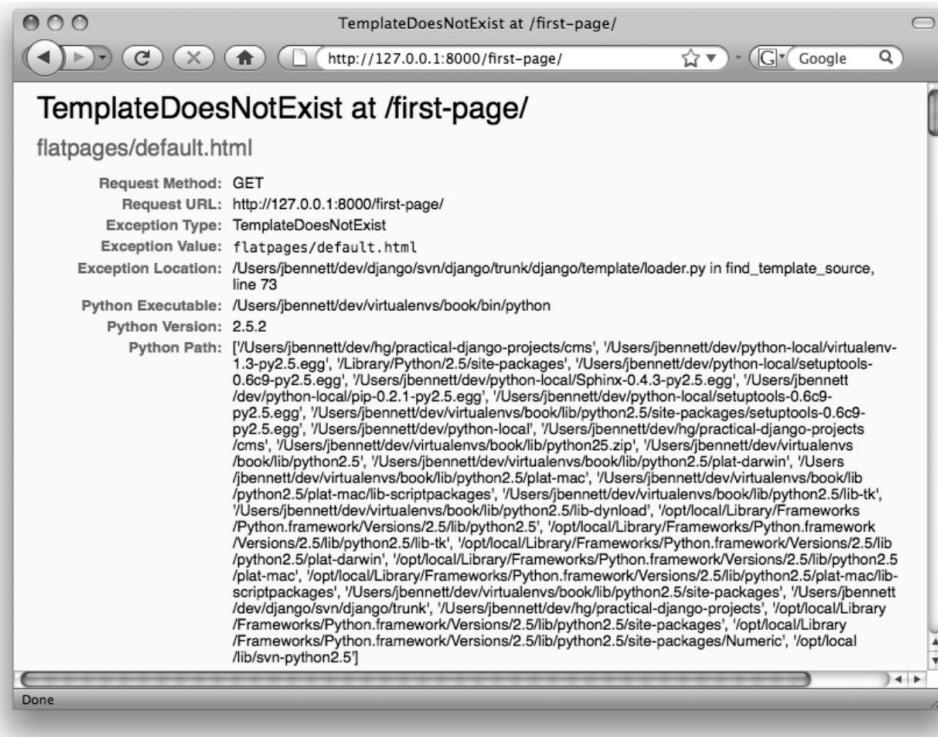


Figura 2-4. Una página de error del servidor Django

Esta página parece un poco aterradora, pero en realidad no lo es. Una vez más, el modo de depuración de Django intenta brindarle la mayor cantidad de información posible. La parte superior de la página muestra un breve resumen del error, seguido de información más detallada, incluido un *seguimiento completo de la pila* (una copia de todo lo que Python y Django estaban haciendo cuando ocurrió el error), una lista de la solicitud HTTP entrante y la configuración de su proyecto Django (con cualquier configuración confidencial, como las contraseñas de la base de datos, en blanco por razones de seguridad).

El problema aquí es que una página plana, como la mayoría de los resultados de Django, espera mostrarse a través de una plantilla que genera el HTML correcto. `django.contrib.flatpages`, por defecto, busca un archivo de plantilla llamado `flatpages/default.html`, y aún no lo ha creado. El formulario de edición en la interfaz de administración, si regresa y lo busca, también mostrará un campo donde puede ingresar un nombre de archivo de plantilla diferente por página. Así que hagamos una pausa por un momento y ocupémonos de eso.

Presentamos el sistema de plantillas de Django

Django incluye un sistema de plantillas que tiene dos objetivos principales de diseño:

- Proporcione una manera fácil de expresar la lógica necesaria para la presentación de su aplicación
- En la medida de lo posible, evite restringir los tipos de salida que puede generar

(Puedes encontrar el sistema de plantillas en el módulo django.template, si has estado explorando el código base de Django y quieres echarle un vistazo).

Algunos lenguajes de plantilla le permiten incrustar casi cualquier forma de código de programación directamente en las plantillas. Si bien esto puede ser útil, también crea una tendencia a que la lógica de programación principal de su aplicación migre lentamente de otras partes del código a las plantillas, que realmente deberían limitarse a los aspectos de presentación de la aplicación. Y algunos lenguajes de plantillas lo obligan a escribir XML u otros tipos específicos de marcado, incluso si lo que desea producir no es XML en absoluto. El sistema de plantillas de Django hace todo lo posible para evitar estos dos inconvenientes al mantener la programación permitida al mínimo y al no limitarlo a lenguajes de marcado específicos. (He usado el sistema de plantillas de Django para generar contenido para mensajes de correo electrónico e incluso hojas de cálculo de Excel, por ejemplo).

En última instancia, un archivo de plantilla de Django para una página web, en otras palabras, una plantilla cuya salida es HTML, no termina pareciendo tan diferente de una página web normal escrita a mano.

La mayor distinción está en dos características que proporciona el sistema de plantillas de Django:

- **Variables:** una variable se alimenta a la plantilla mediante una vista: la función real de Python que responde a una solicitud HTTP y está encerrado entre llaves dobles, así: {{ variable_name_here }}. Este marcador de posición simplemente se reemplaza con el valor real de la variable.
- **Etiquetas:** una etiqueta se envuelve entre llaves simples y signos de porcentaje, así: {% tag_name_aquí %}. Las etiquetas pueden hacer casi cualquier cosa y el efecto exacto depende de la etiqueta en particular. También puede escribir y usar sus propias etiquetas personalizadas en las plantillas de Django, por lo que si hay algo que necesita que no se proporciona de fábrica, puede agregarlo usted mismo.

Siempre que Django necesite un archivo de plantilla, puede buscar en cualquiera de varios lugares, definidos por módulos configurables llamados *cargadores de plantillas*. Por defecto, Django busca en los siguientes lugares:

- Dentro de cualquier directorio especificado en su módulo de configuración por la configuración TEMPLATE_DIRS
- Dentro de sus aplicaciones instaladas, si alguna de ellas incluye un directorio llamado templates/

Estos cargadores de plantillas le permiten proporcionar un conjunto de plantillas predeterminadas con cualquier aplicación determinada. pero también le da el poder de anularlos proyecto por proyecto enumerando directorios específicos en los que colocará plantillas personalizadas. La interfaz administrativa, por ejemplo, usa esto con gran efecto: django.contrib.admin contiene un directorio templates/ con las plantillas predeterminadas, pero puede agregar sus propias plantillas en un directorio de plantillas específico del proyecto si necesita personalizar la interfaz de administración. .

Continúe y elija un directorio donde le gustaría guardar las plantillas para la aplicación CMS simple. La ubicación exacta no importa, siempre que sea un lugar donde se le permita crear y leer archivos en su computadora. A continuación, abra el archivo settings.py de su proyecto, desplácese hacia abajo hasta que vea la configuración TEMPLATE_DIRS y agregue ese directorio a la lista. Aquí está el mío:

```
PLANTILLA_DIRS =  
    '/ Usuarios/jbennett/html/django-templates/cms/',  
)
```

Notarás que estoy especificando un directorio completamente diferente al que se guarda el código del proyecto. Esto suele ser una buena práctica porque refuerza la idea de que la presentación en particular, en forma de un conjunto de plantillas HTML, puede y debe desvincularse

del código back-end siempre que sea posible. También es una práctica útil para cualquier aplicación que pueda terminar reutilizando en varios sitios web. Obviamente, diferentes sitios tendrán diferentes conjuntos de plantillas, por lo que le resultará útil poder cambiarlas a voluntad sin necesidad de mover muchos archivos dentro y fuera de una ubicación específica del proyecto.

Advertencia: comas finales

Como ya habrás aprendido de un tutorial, Python ofrece dos formas simples de representar secuencias de elementos: listas y tuplas. Una tupla suele estar entre paréntesis, como has visto hasta ahora con INSTALLED_APPS y ahora la configuración de TEMPLATE_DIRS, que aceptan tuplas como valores legales. Pero las tuplas de Python requieren que los elementos se separen con comas, incluso si solo hay un elemento en la tupla. Omitir las comas es una molestia común para los usuarios que se están acostumbrando al lenguaje. he estado escribiendo Python durante varios años y todavía a veces me olvido de incluir las comas. En general, encuentro útil recordar que en Python, la coma, y no los paréntesis, que técnicamente no son necesarios, es lo que hace una tupla.

Ahora, dentro del directorio de plantillas que eligió, cree un subdirectorio llamado flatpages/, y en ese subdirectorío cree un nuevo archivo llamado default.html. Actualice la página plana en su navegador web y debería ver una página en blanco. Ahora tiene un directorio de plantilla especificado en su configuración y el archivo flatpages/default.html existe dentro de él, por lo que ya no hay ningún error. Pero el archivo de plantilla está vacío, por lo que no produce ningún resultado. Arreglemos eso abriendo el archivo default.html y agregando algo de contenido:

```
<html>
  <cabeza>
    <título>{{ página plana.título }}</título>
  </cabeza>
  <cuerpo>
    <h1>{{ página plana.título }}</h1>
    {{ página plana.contenido }}
  </cuerpo>
</html>
```

Ahora guarde el archivo y actualice la página en su navegador web nuevamente. deberías ver algunos como la pantalla que se muestra en la Figura 2-5.

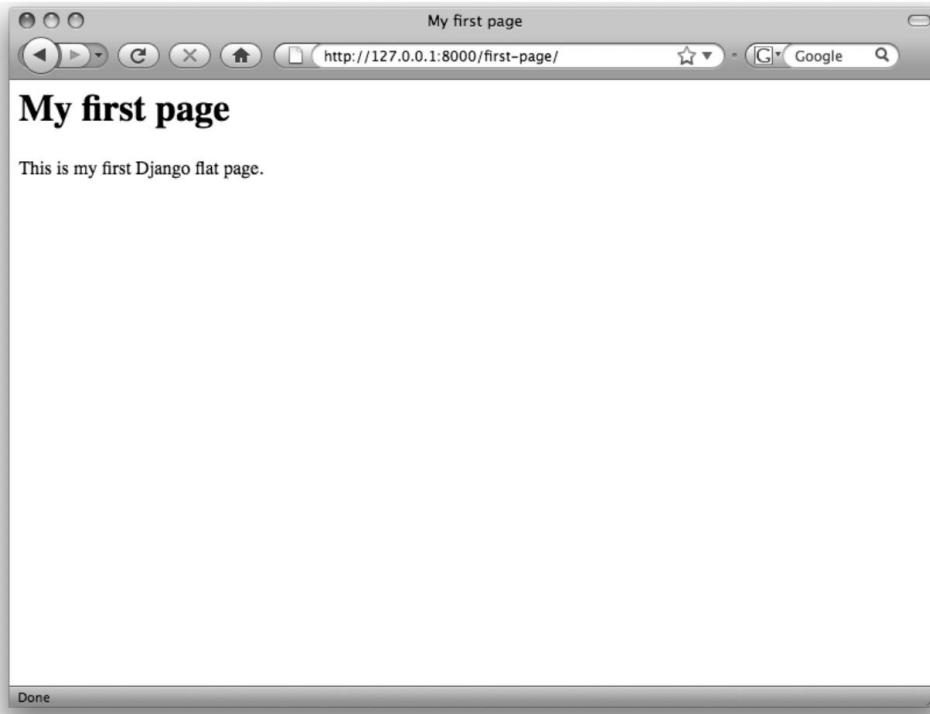


Figura 2-5. Tu primera página plana de Django

Verá que esta plantilla utiliza dos variables: `flatpage.title` y `flatpage.content`. y sin etiquetas. Esas variables en realidad provienen de una sola fuente: una página plana variable, que se pasó a la plantilla mediante una función de vista de Python definida dentro de las páginas `django.contrib.flat`. El valor de esta variable es un objeto `FlatPage`, una instancia del modelo de datos para páginas planas. Django creó este objeto consultando la base de datos en busca de una fila con una columna de URL que coincidiera con la URL `/primera página/`. Luego usó los datos de esa fila para crear un objeto de Python con atributos denominados título y contenido, coincidiendo con lo que ingresó en la interfaz de administración (junto con otros atributos, por ejemplo, URL, que no son tan importantes para el aspecto de presentación). de cosas).

Advertencia: ¿Cómo hizo eso Django?

Django incluye una biblioteca llamada mapeador relacional de objetos u ORM. El ORM comprende la estructura de sus modelos de datos (que se definen como clases simples de Python) y la estructura correspondiente de su base de datos. Proporciona una sintaxis sencilla para traducir entre filas y tablas en su base de datos y objetos de Python en vivo en su código, generalmente sin necesidad de que escriba sus propias consultas SQL. Además, una función de visualización en la aplicación de páginas planas incluidas de Django utiliza el ORM para buscar la página plana correcta y ponerla a disposición de la plantilla. (Escribirá su primera función de vista en el Capítulo 3.) A lo largo de este libro, verá ejemplos del ORM de Django en acción y tendrá una idea de todas sus características. También verá cómo puede omitirlo en situaciones en las que desea realizar su propia consulta a mano.

Con esta plantilla en su lugar, ahora tiene, literalmente, un CMS dinámico simple que le permitirá definir tantas páginas como desee, titularlas, completar el contenido y ubicarlas en cualquier URL (excepto las URL que comienzan con admin/ porque coincidirán con el patrón de URL de la interfaz de administración). Si quisiera, podría adornar la plantilla con un HTML más elegante y una buena hoja de estilo en cascada (CSS), crear algunas cuentas de usuario más a través de la interfaz administrativa e implementar la aplicación en un servidor web activo para su uso en el mundo real. . Pero hasta ahora, solo ha escrito un par de líneas de código real: el patrón de URL para las páginas en su urls.py

archivo, algunas configuraciones de Django y un poco de HTML.

Obviamente, poner en marcha una aplicación con Django no siempre será así fácil, pero espero que hayas visto que aprovechar los componentes de Django puede reducir significativamente la cantidad de trabajo que tienes que hacer.

Mirando hacia el futuro

Haga una pausa aquí por unos momentos para jugar con el CMS simple y explorar la interfaz administrativa de Django. Preste especial atención al enlace de documentación que aparece en la esquina superior derecha de cada página en el administrador. Proporciona documentación generada automáticamente para todos los modelos de datos, patrones de URL y etiquetas de plantilla disponibles en su proyecto Django. No todo será inmediatamente comprensible en este punto, pero haga clic en el área de documentación para tener una idea de lo que hay allí. Cuando esté desarrollando o trabajando con aplicaciones más complejas, el sistema de documentación del administrador será un recurso importante para conocer y comprender el código que está utilizando.

Cuando esté listo para volver al trabajo, el próximo capítulo lo estará esperando con una guía para personalizar este CMS simple y agregar algunas funciones útiles, incluida una función de búsqueda.

Capítulo 3



Personalización del CMS simple

los el CMS simple que armó en el último capítulo ya está en muy buena forma; es algo que a la mayoría de los desarrolladores no les importaría mostrar a los clientes como un prototipo inicial, por ejemplo. Pero hasta ahora, utiliza solo unas pocas aplicaciones estándar incluidas con Django y no ofrece ninguna característica adicional además de eso. En este capítulo, verá cómo tomar este proyecto simple como base y comenzar a agregar sus propias personalizaciones, como la edición de texto enriquecido en el administrador y un sistema de búsqueda para encontrar rápidamente páginas específicas.

Agregar edición de texto enriquecido

La interfaz administrativa predeterminada que proporciona Django para la aplicación de páginas planas ya tiene calidad de producción. Muchos sitios basados en Django ya lo usan tal como está para proporcionar una manera fácil de administrar la simple "página Acerca de" ocasional o para manejar tareas similares. Pero es posible que desee hacer que la interfaz administrativa basada en la web sea un poco más amigable al agregarle una interfaz de texto enriquecido para que los usuarios no tengan que escribir HTML sin formato.

Hay varios editores de texto enriquecido (RTE) basados en JavaScript, disponibles con diferentes características y configuraciones, pero usaré una llamada TinyMCE. Una de las opciones más populares, tiene aproximadamente el mejor soporte de navegador cruzado de cualquiera de los RTE existentes. (Debido a las diferencias en las API implementadas por los navegadores web, no existe un RTE multiplataforma verdaderamente consistente en este momento). TinyMCE también es gratuito y se publica bajo una licencia de código abierto. Puede descargar una copia de la última versión estable desde <http://tinymce.moxiecode.com/>.

Una vez que haya desempaquetado TinyMCE, verá que contiene un directorio `jscripts/`, dentro del cual hay un directorio `tiny_mce` que contiene todo el código de TinyMCE. Tome nota de dónde está ese directorio y vaya al archivo `urls.py` del proyecto. En `urls.py`, agregue una nueva línea para que tenga el siguiente aspecto:

desde django.conf.urls.defaults importar

*

```
# Quite el comentario de las siguientes dos líneas para habilitar el administrador:  
from django.contrib import admin  
admin.autodiscover()  
  
urlpatrones = patrones(),  
# Ejemplo:  
# (r'^cms/', include('cms.foo.urls')),
```

```
# Descomente la línea admin/doc a continuación y agregue 'django.contrib.admindocs'  
# a INSTALLED_APPS para habilitar la documentación del administrador:  
# (r'^admin/doc$', include('django.contrib.admindocs.urls')),  
  
# Descomente la siguiente línea para habilitar el administrador:  
(r'^admin/', include(admin.site.urls)),  
(r'^tiny_mce/(?P<ruta>.*$', 'django.views.static.serve',  
    { 'document_root': '/ruta/a/tiny_mce/ '}),  
(r'', include('django.contrib.flatpages.urls')),  
)
```

Reemplace la parte /path/to/tiny_mce con la ubicación real en su computadora del directorio tiny_mce. Por ejemplo, si el directorio reside en /Users/jbennett/javascript/TinyMCE/scripts/tiny_mce, usaría ese valor.

Advertencia: archivos multimedia en producción frente a desarrollo

En producción, por lo general querrá evitar que el mismo servidor web maneje Django y archivos de medios estáticos, como hojas de estilo o JavaScript. Debido a que el proceso del servidor web necesita mantener una copia del código de Django y sus aplicaciones en la memoria, es una pérdida de recursos usar ese mismo proceso para la simple tarea de servir un archivo fuera del disco.

Por ahora, estoy usando una función de ayuda integrada en Django que puede servir archivos estáticos, pero tenga en cuenta que debe usar esto solo para el desarrollo en su propia computadora. Usarlo en un sitio implementado en vivo afectará gravemente el rendimiento de su sitio. Cuando implemente una aplicación Django en un servidor web en vivo, consulte la documentación oficial de Django en <http://docs.djangoproject.com/> para ver las instrucciones para la configuración de su servidor específico.

Ahora solo necesita agregar las llamadas de JavaScript apropiadas a la plantilla utilizada para agregar y editar páginas planas. En el último capítulo, cuando completó la configuración TEMPLATE_DIRS, mencioné que Django también puede buscar plantillas directamente dentro de una aplicación y que esta capacidad le permite al autor de la aplicación proporcionar plantillas predeterminadas mientras permite que los proyectos individuales usen las suyas propias. Eso es precisamente lo que vas a aprovechar aquí. La aplicación de administración no solo está diseñada para usar sus propias plantillas como respaldo, sino que también le permite proporcionar las suyas propias si desea personalizarlas.

De forma predeterminada, la aplicación de administración buscará una plantilla en varios lugares, utilizando el primero uno que encuentra. Los nombres de plantilla que busca son los siguientes, en este orden:

1. admin/páginas planas/página plana/cambiar_formulario.html
2. admin/páginas planas/cambiar_formulario.html
3. admin/cambiar_formulario.html

Advertencia: elegir entre varias plantillas

Normalmente, cuando escribe una vista de Django, la función que realmente responde a una solicitud HTTP, la configurará para usar una sola plantilla para su salida. (Las aplicaciones que escribirá en este libro generalmente necesitarán especificar solo una plantilla para cada vista). Sin embargo, hay una función auxiliar, `django.template.loader.select_template`, que toma una lista de nombres de plantilla, busca archivos de plantilla que coincidan con esos nombres y usa el primero que encuentra. La aplicación de administración hace uso de esta función de ayuda para habilitar precisamente el tipo de personalización que estoy haciendo aquí. Si alguna vez está escribiendo una aplicación en la que necesita hacer lo mismo, tenga en cuenta esa función.

La aplicación de administración proporciona solo la última plantilla de esta lista: `admin/change_form.html`, y lo usa para agregar y editar elementos si no proporciona una plantilla personalizada. Pero como puede ver, hay un par de otras opciones. Mediante el uso de una lista de posibles nombres de plantillas, en lugar de una sola plantilla preconstruida, la aplicación de administración le permite anular la interfaz para una aplicación específica (en este caso, la aplicación de páginas planas, proporcionando la plantilla `admin.flatpages/change_form.html`) o para un modelo de datos específico (proporcionando la plantilla `admin.flatpages/flatpage/change_form.html`). En este momento, desea personalizar la interfaz solo para un modelo específico. Entonces, dentro de su directorio de plantillas, cree un administrador `subdirectorio`. Luego cree un subdirectorio `flatpages` dentro de `admin` y un subdirectorio `flatpage` dentro de `flatpages`. Finalmente, copie la plantilla `change_form` de `django/contrib/admin/templates/admin/change_form.html` en su copia de Django en `admin/flatpages/flatpage/` directorio que acaba de crear.

Ahora puede abrir la plantilla `change_form.html` en su directorio de plantillas y editarla para agregar el JavaScript apropiado para TinyMCE. Esta plantilla parecerá bastante compleja, y lo es, porque la aplicación de administración tiene que adaptarse para proporcionar formularios apropiados para cualquier modelo de datos, pero el cambio que realizará es bastante simple. En la línea 6 de la plantilla, verá lo siguiente:

```
 {{ medios }}
```

Inmediatamente debajo de eso, agregue lo siguiente:

```
<script type="text/javascript" src="/tiny_mce/tiny_mce.js"></script>
<script tipo="texto/javascript">
minúsculoMCE.init({
    modo: "áreas de texto",
    tema: "sencillo"
});
</script>
```

Esto hará uso de la URL que configuró para servir los archivos TinyMCE. Ahora guarde el archivo y vuelve a tu navegador web. El formulario que se muestra para agregar y editar páginas planas ahora tendrá el editor TinyMCE básico adjunto al área de texto para el contenido de la página, como se muestra en la Figura 3-1.

TinyMCE es extremadamente personalizable. Puede reorganizar la barra de herramientas de edición, elegir qué de los muchos controles incorporados deben aparecer en él, agregue sus propios controles y escriba nuevos temas para cambiar su apariencia. Y si desea utilizar otro RTE o realizar otras personalizaciones en la interfaz de administración, puede seguir el mismo proceso.

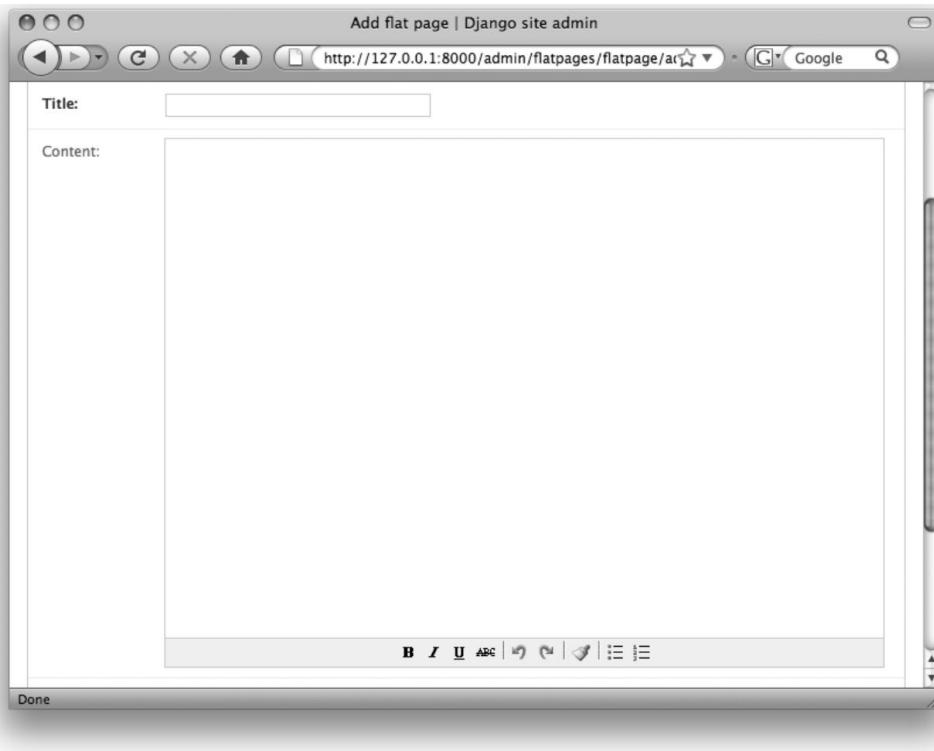


Figura 3-1. El formulario de administración de páginas planas con editor de texto enriquecido

Adición de un sistema de búsqueda al CMS

Hasta ahora, solo ha estado usando las aplicaciones incluidas con Django y ha realizado pequeñas personalizaciones en las plantillas que usan. Hasta ahora eso ha logrado mucho, pero para la mayoría de sus proyectos, escribirá sus propias aplicaciones en Python. Así que ahora agregará una nueva función, escrita en Python, al CMS simple: un sistema de búsqueda simple que permite a los usuarios escribir una consulta y obtener una lista de las páginas cuyos títulos o contenidos coinciden.

Sería posible agregar esto directamente a la aplicación de páginas planas incluida con Django, pero no es realmente una buena idea, por dos razones:

- Hace que actualizar Django sea una molestia. Tiene código de Python adicional que no viene con Django y el código debe conservarse durante la actualización.
- Una característica útil como un sistema de búsqueda podría necesitar expandirse más tarde para trabajar con otros tipos de contenido, en cuyo caso no tendría sentido que fuera parte de la aplicación flatpages.

Así que hagamos de esto su propia aplicación. Vaya al directorio de su proyecto y escriba el siguiente comando:

```
python manage.py startapp búsqueda
```

Así como el comando startproject para django-admin.py creó un nuevo directorio de proyecto vacío, el comando startapp para manage.py crea un módulo de aplicación nuevo y vacío.

Configurará el directorio search/ dentro de su proyecto y le agregará los siguientes archivos:

- `__init__.py`: Al igual que el del directorio del proyecto, este archivo `__init__.py` comienza vacío. Su trabajo es indicar que el directorio también es un módulo de Python.
- `models.py`: este archivo contendrá cualquier modelo de datos definido para la aplicación. Un poquito más adelante en este capítulo, escribirá su primer modelo en este archivo.
- `views.py`: este archivo contendrá las funciones de visualización, que responden a solicitudes HTTP y hacer la mayor parte del trabajo de interacción con el usuario.
- `tests.py`: aquí es donde puede colocar pruebas unitarias, que son funciones que le permiten verificar automáticamente que su aplicación funciona según lo previsto. Puede ignorar este archivo de forma segura por ahora. (Aprenderá más sobre las pruebas unitarias en el Capítulo 11).

Por ahora, solo escribirá una vista simple, así que abra el archivo `views.py`. El primer paso es importar las cosas que usarás. Parte de la filosofía de diseño de Python (y Django) es que debería poder ver claramente lo que sucede con la menor "magia" implícita posible.

Por lo tanto, cada archivo debe contener declaraciones de importación de Python para las cosas a las que quiere hacer referencia desde otros módulos de Python. Para comenzar, necesitará tres declaraciones de importación:

```
de django.http importar HttpResponseRedirect  
desde el cargador de importación django.template, Contexto  
desde django.contrib.flatpages.models importar FlatPage
```

Estas declaraciones le brindan una base sólida para escribir su vista de búsqueda:

- `HttpResponse` es la clase que utiliza Django para representar una respuesta HTTP. Cuando se proporciona una `HttpResponse` como valor de retorno de una vista, Django la convertirá automáticamente al formato de respuesta correcto para el servidor web en el que se ejecuta.
- El módulo cargador en `django.template` proporciona funciones para especificar el nombre de un archivo de plantilla, que se ubicará (asumiendo que está en un directorio especificado en `TEMPLATE_DIRS`), se leerá desde el disco y se analizará para renderizar.
- `Context` es una clase utilizada para representar las variables de una plantilla. Lo pasa a un diccionario de Python que contiene los nombres de las variables y sus valores. (Si está familiarizado con otros lenguajes de programación, un diccionario de Python es similar a lo que algunos lenguajes llaman *tabla hash* o *matriz asociativa*).
- `FlatPage` es la clase modelo que representa las páginas en el CMS.

Advertencia: estilo de nomenclatura de Python

Cada lenguaje de programación tiene un conjunto de convenciones estándar sobre cómo nombrar las cosas. Java, por ejemplo, tiende a usar mayúsculas y minúsculas, donde las cosas reciben nombres que se parecen a esto, mientras que PHP tiende a favorecer los guiones bajos o nombres que se parecen a esto.

La práctica estándar en Python es que las clases deben tener nombres en mayúsculas, por lo tanto Contexto, y use el estilo de mayúsculas y minúsculas para nombres de varias palabras como `HttpResponse` o `FlatPage`. Los módulos, funciones y variables normales usan nombres en minúsculas y guiones bajos para separar varias palabras en un nombre. Seguir esta convención ayudará a los programadores de Python, incluido usted, a comprender rápidamente una nueva pieza de código cuando la lea por primera vez.

Si está interesado en obtener más información sobre el estilo estándar de Python, puede leer la guía de estilo oficial de Python en línea en www.python.org/dev/peps/pep-0008/.

Ahora está listo para escribir una función de visualización que realizará una búsqueda básica. Aquí está la código, que irá a `views.py` debajo de las declaraciones de importación que agregó:

```
def buscar (solicitud):
    consulta = solicitud.GET['q']
    resultados = FlatPage.objects.filter(content__icontains=query)
    plantilla = loader.get_template('buscar/buscar.html')
    context = Context({ 'consulta': consulta, 'resultados': resultados })
    respuesta = template.render(context)
    devolver HttpResponse(respuesta)
```

Analicemos esto línea por línea. Primero, está definiendo una función de Python usando la clave definición de palabra El nombre de la función es `búsqueda` y toma un argumento llamado `solicitud`. Esta será una solicitud HTTP (una instancia de la clase `django.http.HttpRequest`), y Django se asegurará de que se pase a la función de visualización cuando sea necesario.

A continuación, mire la variable HTTP GET `q` para ver qué buscó el usuario. Django automáticamente analizó minuciosamente la URL, por lo que una URL como esta:

```
http://www.ejemplo.com/buscar?q=foo
```

da como resultado una `HttpRequest` cuyo atributo `GET` es un diccionario que contiene el nombre `q` y el valor `foo`. Luego puede leer ese valor tal como accedería a cualquier diccionario de Python.

La siguiente línea hace la búsqueda real. La clase `FlatPage`, como casi todos los mods de datos de Django `els`, tiene un atributo llamado `objetos` que se puede usar para realizar consultas en ese modelo. En este caso, desea filtrar a través de todas las páginas planas, buscando aquellas cuyo contenido contenga el término de búsqueda. Para hacer esto, usa el método de filtro y el contenido del argumento `__icontains=query`, almacenando los resultados en una variable llamada `resultados`. Esto proporcionará una lista de objetos `FlatPage` que coincidieron con la consulta.

Advertencia: sintaxis de búsqueda en la base de datos de Django

Como verá en breve, un modelo de datos de Django tiene atributos especiales llamados campos, que generalmente corresponden a los nombres de las columnas en la base de datos. Cuando utiliza el mapeador relacional de objetos (ORM) de Django para ejecutar una consulta, cada argumento de la consulta comprende una combinación de un nombre de campo y un operador de búsqueda, separados por guiones bajos dobles.

En este caso, el nombre del campo es contenido porque ese es el campo en el modelo FlatPage que representa el contenido de la página (cada FlatPage también tiene campos denominados título, URL, etc.). El operador de búsqueda es `icontains`, que verifica si el valor en esa columna contiene la cadena que le pasó. El `yo` en el frente significa que el operador realiza una búsqueda que no distingue entre mayúsculas y minúsculas, por lo que una consulta de `hola` coincidiría tanto con `hola` como con `Hola`, por ejemplo. Django ORM admite una gran cantidad de otros operadores de búsqueda, muchos de los cuales verá en acción a lo largo de este libro.

Ahora que tiene la consulta y los resultados, necesita generar algo de HTML y devolver una respuesta. Entonces, la siguiente línea usa la función `get_template` del módulo cargador que importó para cargar una plantilla llamada `search/search.html`. A continuación, debe proporcionar a la plantilla algunos datos con los que trabajar, así que cree un Contexto que contenga dos variables: `consulta` es la consulta de búsqueda y `resultados` contiene los resultados de la búsqueda.

Luego usa el método de representación de la plantilla, pasando el Contexto que creó, para generar el HTML para la respuesta. Y finalmente, devolverá un `HttpResponse` que contiene el HTML representado.

Ahora guarde el archivo `views.py`. Volverá a él en un momento y hará algunas mejoras, pero por ahora necesita crear una plantilla para que la vista de búsqueda pueda generar su HTML.

Vaya a su directorio de plantillas, cree un nuevo subdirectorio llamado `búsqueda` y dentro de eso cree un archivo llamado `búsqueda.html`. A continuación, abrirá el archivo `search.html` y le agregará lo siguiente:

```
<html>
  <cabeza>
    <título>Buscar</título>
  </cabeza>
  <cuerpo>
    <p>Has buscado "{{ consulta }}"; los resultados se enumeran a continuación.</p>
    <ul>
      {% para la página en los resultados%}
      <li><a href="{{ página.get_absolute_url }}">{{ página.título }}</a></li>
      {% endfor%}
    </ul>
  </cuerpo>
</html>
```

Esto hace uso de las dos variables que se le pasan. Utiliza `{{ consulta }}` para mostrar la consulta y recorre los resultados para mostrarlos en una lista desordenada. (Recuerde, puede generar variables directamente en las plantillas de Django envolviendo sus nombres entre llaves dobles).

Tenga en cuenta que también he usado una etiqueta de plantilla de Django, que le permite recorrer una secuencia de las cosas y hacer algo con cada una. La sintaxis es bastante simple. En efecto, dice, "para cada página en la variable de resultados, muestre el siguiente HTML, completado con los valores de esa página". Probablemente puedas adivinar que, dentro del ciclo for, {{ page.title }} se refiere al campo de título de la página actual en el ciclo, pero {{ page.get_absolute_url }} es nuevo. Es una práctica estándar para una clase de modelo de Django definir un método llamado get_absolute_url(), que generará una URL que se usará para hacer referencia al objeto, y el modelo FlatPage lo hace. (Su método get_absolute_url() simplemente devuelve el valor de su campo de URL; otros modelos pueden y tendrán formas más complejas de calcular sus URL).

Advertencia: llamar a los métodos de un objeto en una plantilla de Django

El sistema de plantillas de Django le permite acceder a métodos en objetos de Python de la misma manera que accede a cualquier otro atributo: usando un carácter de punto (.). Por ejemplo, {{ page.get_absolute_url }} usa un punto para llamar al método get_absolute_url(). Pero tenga en cuenta que en una plantilla no usa paréntesis cuando llama a un método, y no puede pasar argumentos a un método llamado de esta manera. Esto se remonta a la filosofía de Django de no permitir demasiada "programación" en las plantillas, algo que es lo suficientemente complejo como para necesitar que se le pasen argumentos probablemente no sea puramente de presentación. El sistema de plantillas de Django también prohíbe el acceso a métodos que alteran los datos en su base de datos. Las llamadas a esos métodos definitivamente pertenecen a una función de vista y no a una plantilla.

También puede acceder a los valores de un diccionario utilizando la misma sintaxis de puntos. Al igual que con la falta de paréntesis en las llamadas a métodos, esto es diferente de cómo lo haría en el código de Python (donde el acceso al diccionario usa corchetes, como en request.GET['q']), pero tiene la ventaja de hacer que la sintaxis de la plantilla de Django es extremadamente uniforme. La técnica también sirve como recordatorio de que las plantillas de Django no son simplemente código de Python y, por lo tanto, no ofrecen un lenguaje de programación completo.

Además, tenga en cuenta que la etiqueta for necesita una etiqueta endfor coincidente cuando haya terminado de decirle qué hacer dentro del ciclo. La mayoría de las etiquetas de plantilla de Django que abarcan una sección de la plantilla necesitarán una etiqueta final explícita para declarar cuando haya terminado con ellas.

Ahora abra su plantilla flatpages/default.html y en algún lugar coloque el siguiente HTML:

```
<método de formulario="obtener" acción="/buscar/">
<p><etiqueta para="id_q">Buscar:</etiqueta>
<tipo de entrada="texto" nombre="q" id="id_q" />
<tipo de entrada="enviar" valor="Enviar" /></p>
</formulario>
```

Este HTML agrega un cuadro de búsqueda que se enviará a la URL correcta con el GET correcto variable (q) para la consulta de búsqueda.

Finalmente, abra el urls.py de su proyecto. Después de las líneas para el administrador y el TinyMCE JavaScript, pero *antes* del patrón general para las páginas planas, agregue lo siguiente:

```
(r'^búsqueda/$', 'cms.búsqueda.vistas.búsqueda'),
```

Recuerde que debido a que esta expresión regular termina en una barra inclinada, deberá incluirla cuando escribe la dirección en su navegador. A diferencia de los patrones de URL que configuró anteriormente, que usaban la directiva de inclusión para extraer otros módulos de URLConf, este asigna la búsqueda de URL a una sola vista específica: la vista de búsqueda que acaba de escribir. Después de guardar el urls.py archivo, debería poder escribir una consulta de búsqueda en cualquier página de su CMS y obtener una lista de páginas coincidentes.

Mejorar la vista de búsqueda

La vista de búsqueda funciona bastante bien para algo tan breve: solo se trata de media docena de líneas de código, más algunas instrucciones de importación. Pero puede hacerlo más corto, y es una buena idea hacerlo.

Notará que de las seis líneas de código real en la vista de búsqueda, cuatro están dedicadas a cargar la plantilla, crear un Contexto, representar el HTML y devolver la respuesta.

Esa es una serie de pasos que deberá seguir en casi todas las vistas que escriba, por lo que Django proporciona una función de acceso directo llamada django.shortcuts.render_to_response que maneja el proceso en un solo paso. Así que edite el archivo views.py para que se vea así:

```
de django.shortcuts import render_to_response
desde django.contrib.flatpages.models importar FlatPage

def buscar (solicitud):
    consulta = solicitud.GET['q']
    volver render_to_response('buscar/buscar.html',
        { 'consulta': consulta,
          'resultados': FlatPage.objects.filter( ª
            contenido__icontains=consulta ) })
```

La función render_to_response obtiene dos argumentos aquí:

1. El nombre del archivo de plantilla, search/search.html
2. El diccionario a usar para el contexto de la plantilla

Dada esa información, maneja todo el proceso de carga de la plantilla, representación de la salida y creación de HttpResponseRedirect. Observe también que ya no está usando una línea separada para obtener los resultados. Solo son necesarios para el contexto de la plantilla, por lo que puede hacer la consulta allí mismo dentro del diccionario, confiando en que su resultado se asignará correctamente a los resultados.

variable. También dividió los argumentos, incluido el diccionario, en varias líneas.

Python le permite hacer esto en cualquier momento que construya una lista o diccionario (así como en varias otras situaciones), y hace que el código sea mucho más fácil de leer que si estuviera todo extendido en una línea larga.

Guarde el archivo views.py y luego regrese y realice una búsqueda nuevamente. Notará que funciona exactamente de la misma manera, solo que ahora la vista de búsqueda es mucho más corta y simple. Y, lo que es más importante, no tiene el repetitivo "repetitivo" del proceso de carga y renderizado de plantillas. Habrá momentos en los que querrá hacerlo manualmente (por ejemplo, si desea insertar algún procesamiento adicional antes de devolver la respuesta), pero en general, debe usar el atajo render_to_response siempre que sea posible.

Otra mejora sencilla sería que la vista de búsqueda maneje situaciones en las que se accede directamente. En este momento, si solo visita la URL /buscar/ en lugar de acceder a ella a través del cuadro de búsqueda en otra página, verá un feo error quejándose de que la clave q no se encontró en el diccionario request.GET (porque la q variable proviene de realizar una búsqueda). Sería mucho más útil mostrar simplemente un formulario de búsqueda vacío, así que reescribamos la vista para hacer lo siguiente:

```
def búsqueda(solicitud):
    consulta = solicitud.GET.get('q', '') resultados =
    [] if consulta: resultados =
        FlatPage.objects.filter(content__icontains=query)
        return render_to_response('search/search.html', { 'consulta': consulta, 'resultados':
            resultados })
```

Ahora está utilizando request.GET.get('q', '') para leer la variable q. get(), un método disponible en cualquier diccionario de Python, le permite solicitar el valor de una clave en particular y especificar un valor predeterminado al que recurrir si la clave no existe (el valor predeterminado en este caso es solo una cadena vacía). Luego puede verificar el resultado para ver si hay una consulta de búsqueda. Si no lo hay, configura los resultados en una lista vacía y eso no se cambiará. Esto significa que puede reescribir la plantilla de esta manera:

```
<html>
<cabeza>
<título>Buscar</título>
</cabeza>
<body>
<form method="get" action="/search/"> <p><label
for="id_q">Buscar:</label> <input type="text" name="q"
id="id_q" value="{{ consulta }}"/> <input type="enviar" value="Enviar" /></p> </formulario>
{%
si los resultados son %}

<p>Has buscado "{{ consulta }}"; los resultados se enumeran a continuación.</p> <ul> {%
para página
en resultados %} <li><a href="{{ page.get_absolute_url }}"/>{{ page.title }}</a></li> {%
endfor %} </ul>
{%
else %} {%
if query %} <p>No se encontraron resultados.</p> {%
else %} <p>Escriba una
consulta de búsqueda en el cuadro de arriba y presione "Enviar" para buscar.</p> {%
endif %} {%
endif %} </body> </html>
```

Ahora la plantilla search.html mostrará el mismo cuadro de búsqueda que aparece en todas las demás páginas del CMS, y notará que también se ha agregado un atributo de valor al HTML para el cuadro de entrada de búsqueda. De esta manera, si hubo una consulta, se completará como un recordatorio de lo que buscó el usuario.

También estoy usando otra nueva etiqueta de plantilla: if. La etiqueta if funciona de manera similar a la declaración if en Python, lo que le permite probar si algo es cierto o no y le permite hacer algo en función del resultado. También requiere una cláusula else opcional, que estoy usando para mostrar un mensaje diferente si el usuario aún no ha buscado nada. Además, así como la etiqueta for necesita una etiqueta endfor, if necesita un endif. Y finalmente, observe que puede anidar la etiqueta if: dentro de la cláusula else, estoy usando otra etiqueta if para diferenciar entre los resultados vacíos porque no hubo una consulta y los resultados vacíos porque ninguna página coincidió con la consulta.

Advertencia: consideraciones de seguridad

Uno de los tipos más comunes de problemas de seguridad con las aplicaciones web es la vulnerabilidad a un ataque de secuencias de comandos entre sitios o XSS. Este tipo de vulnerabilidad ocurre cuando acepta ciegamente la entrada de un usuario y la muestra en una página de su sitio, como estoy haciendo con la consulta de búsqueda. El problema es que un pirata informático puede enviar una consulta de búsqueda que contiene HTML y JavaScript, y luego atraer a alguien para que visite una página para esa consulta. El JavaScript se ejecutará como si fuera parte de su sitio y podría usarse para secuestrar la cuenta de un usuario.

También existe el riesgo de otra forma de ataque, llamada inyección SQL, en la que un hacker confía en un sitio web para incluir la entrada del usuario directamente en una consulta de base de datos. Por ejemplo, un pirata informático podría enviar una consulta de búsqueda que contenga el texto "DROP DATABASE"; que podría, si se ejecuta a ciegas, eliminar toda la base de datos del sitio.

Sin embargo, Django proporciona cierta protección integrada contra este tipo de ataques. En primer lugar, las plantillas de Django "escapan" automáticamente del contenido de cualquier variable que muestres (de modo que, por ejemplo, el carácter < se convierte en <, eliminando la posibilidad de que una variable termine como HTML que un navegador web representa). En segundo lugar, Django construye cuidadosamente las consultas de la base de datos para que la inyección SQL no sea posible.

Sin embargo, no debe dejar que estos mecanismos lo adormezcan con una falsa sensación de invencibilidad. Cada vez que estés Al tratar con los datos enviados por los usuarios, debe asegurarse cuidadosamente de que está tomando las medidas adecuadas para preservar la seguridad de su sitio.

Mejorar la función de búsqueda con palabras clave

La función de búsqueda que acabas de agregar al CMS es bastante útil, pero puedes mejorarla un poco agregando la capacidad de reconocer palabras clave específicas y abrir automáticamente páginas específicas como respuesta. Esto permitirá que los administradores del sitio brinden sugerencias útiles para los usuarios que realizan búsquedas y también creará metadatos útiles que quizás deseas aprovechar más adelante.

Para agregar esta función, deberás crear un modelo de datos de Django; los modelos van en models.py archivo, así que ábrelo. Verás que ya tiene una declaración de importación en la parte superior:

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

Esta declaración importa el módulo que contiene todas las clases necesarias para crear modelos de datos de Django, y el comando startapp lo agregó automáticamente al archivo models.py para ayudarlo a comenzar. Debajo de esa línea, agrega lo siguiente:

```
desde django.contrib.flatpages.models importar FlatPage
```

```
clase SearchKeyword(modelos.Modelo):
    palabra clave = modelos.CharField(max_length=50)
    página = modelos.ForeignKey(FlatPage)

    def __unicode__(uno mismo):
        volver auto.palabra clave
```

Este es un modelo simple de Django con dos campos:

- keyword: Este es un CharField, lo que significa que aceptará cadenas cortas. He especificado un max_length de 50, lo que significa que pueden entrar hasta 50 caracteres en este campo. En la base de datos, Django convertirá esto en una columna declarada como VARCHAR(50).
- página: esta es una clave externa que apunta al modelo FlatPage, lo que significa que cada palabra clave de búsqueda está vinculada a una página específica. Django convertirá esto en una columna de clave externa que hace referencia a la tabla en la que se almacenan las páginas planas.

Finalmente, hay un método en este modelo: `__unicode__()`. Este es un método estándar que todas las clases de modelos de Django deben definir, y se usa siempre que se necesita una representación de cadena (Unicode) de una palabra clave de búsqueda. Si alguna vez ha trabajado con Java, esto es como el método `toString()` en una clase de Java. El método `__unicode__()` debe devolver algo que se pueda usar con sensatez como una representación de la palabra clave de búsqueda, por lo que está definido para devolver el valor del campo de palabra clave.

Advertencia: los dos tipos de cadenas de Python

Python en realidad tiene dos clases diferentes que representan cadenas: str y unicode. (También hay una clase principal, basestring, que no se puede instanciar directamente pero proporciona una forma útil de verificar si algo es un tipo de cadena). Las instancias de str a veces se denominan cadenas de bytes porque cada una corresponde a una serie específica de bytes en una codificación de caracteres específica. (El valor predeterminado para Python es ASCII, pero puede crear fácilmente cadenas en otras codificaciones). Mientras tanto, las instancias de Unicode son cadenas de caracteres Unicode y deben convertirse a una codificación basada en bytes, como UTF-8 o UTF_16 antes de salir. (Unicode en sí mismo no es una "codificación").

Debido a esto, las clases de Python pueden definir cualquiera de dos métodos con nombres especiales, `__str__()` o `__unicode__()`, para proporcionar representaciones de cadenas de caracteres de sí mismos o, si es necesario, pueden definir ambos. Todas las funciones internas de Django están diseñadas para funcionar con cadenas Unicode, por lo que es mejor simplemente definir `__unicode__()`. Las cadenas almacenadas por los modelos de Django se convertirán en cadenas Unicode cuando se recuperen de su base de datos, y Django se convertirá automáticamente en cadenas de bytes codificadas apropiadamente cuando produzca una salida para una respuesta HTTP.

Tenga en cuenta que no todo el software de Python está escrito para manejar cadenas Unicode (o incluso cadenas de bytes no codificadas en ASCII) correctamente. Cuando escribe aplicaciones que se basan en software de terceros, a veces tendrá que solucionar esto convirtiendo manualmente una cadena. Django proporciona un conjunto de funciones de utilidad para hacer esto más fácil, y en capítulos posteriores las verá en acción.

Guarde el archivo `models.py`, luego abra el archivo `settings.py` del proyecto y desplácese hacia abajo hasta la configuración `INSTALLED_APPS`. Agregue `cms.search` a la lista y guarde el archivo. Esto le dirá a Django que la aplicación de búsqueda dentro del directorio del proyecto cms ahora es parte del proyecto y que se debe instalar su modelo de datos. A continuación, ejecute `python manage.py syncdb` y Django creará la nueva tabla para el modelo `SearchKeyword`.

Advertencia: ¿Por qué funcionaba antes la vista de búsqueda?

Probablemente haya notado que ya usé la vista de búsqueda sin agregar la aplicación de búsqueda a `INSTALLED_APPS`. Esto funcionó porque puede aprovechar cualquier código de Python en su computadora al enrutar URL para ver funciones, independientemente de si están en una aplicación que figura en `INSTALLED_APPLICACIONES` o no. De hecho, no tienen que ser parte de un módulo de aplicación de Django. Esto significa que, si realmente lo desea o lo necesita, puede mantener bibliotecas de código independientes en su computadora y llamarlas desde sus proyectos de Django.

Sin embargo, Django necesita saber exactamente para qué aplicaciones instalar modelos de datos. Entonces, ahora que tiene un modelo, es necesario agregar la aplicación de búsqueda a `INSTALLED_APPS` para que Django cree la tabla de base de datos para él. Hay algunas otras funciones que requieren que tengas una aplicación y la incluyas en `INSTALLED_APPS`. La mayoría de las veces querrá hacer eso, independientemente de si es estrictamente necesario (aunque solo sea para proporcionar un recordatorio rápido de lo que está usando su proyecto), pero a veces es útil para saber qué requiere esto y qué no.

Si se conecta manualmente a su base de datos y observa el diseño de la tabla (consulte la documentación del sistema de base de datos específico que está utilizando para ver cómo hacerlo), verá que la nueva tabla se creó con dos columnas correspondientes a los campos de la palabra clave de búsqueda modelo. La tabla también tiene una tercera columna, `id`, que se declara como la clave principal y es un entero de incremento automático. Si no marca explícitamente ninguno de los campos en un modelo para que sirva como clave principal, Django lo hará por usted automáticamente.

A continuación, querrá habilitar la interfaz administrativa para el nuevo modelo. Para hacer esto, cree un nuevo archivo llamado `admin.py` y coloque el siguiente código dentro:

desde `django.contrib.admin` importar `ModelAdmin`

de `cms.search.models` importar `SearchKeyword`

clase `SearchKeywordAdmin(admin.ModelAdmin)`:

pasar

`admin.site.register(SearchKeyword, SearchKeywordAdmin)`

Este código define una subclase de `django.contrib.admin.ModelAdmin` llamada `SearchKeywordAdmin`.

La declaración de aprobación significa que no desea personalizar nada en esta subclase (aunque en un momento verá cómo realizar algunos cambios en este tipo de clase).

Luego, la función `admin.site.register` le dice a la interfaz administrativa de Django que asocie esta subclase `ModelAdmin` con el modelo `SearchKeyword`.

Ahora puede volver a iniciar el servidor web de desarrollo y verá que el nuevo modelo aparece en el índice. Puede agregar y editar palabras clave al igual que puede agregar y editar instancias de cualquiera de los modelos de las otras aplicaciones instaladas. Desafortunadamente, esta interfaz es un poco torpe: las palabras clave se agregan en una página separada y debe elegir explícitamente con qué página asociar cada palabra clave, como se muestra en la Figura 3-2.

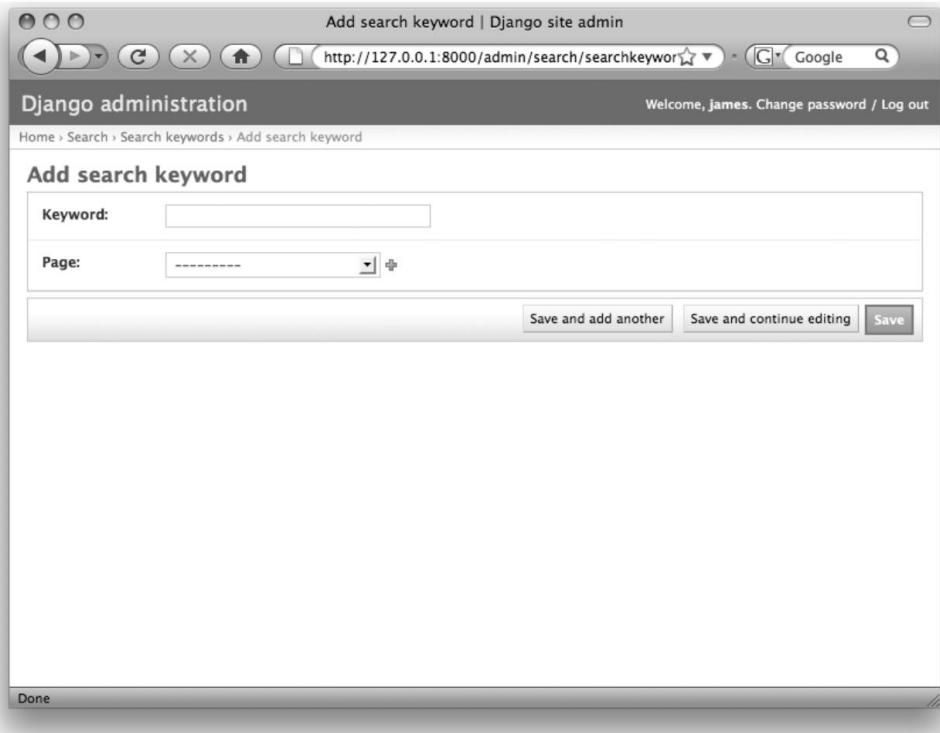


Figura 3-2. El formulario de administración predeterminado para una palabra clave de búsqueda

Lo que realmente le gustaría es que la interfaz para las palabras clave de búsqueda aparezca en la misma página que el formulario para agregar y editar páginas. Puede hacerlo haciendo algunos pequeños cambios en la clase `SearchKeyword` para que se vea así:

```
desde django.contrib.admin importar admin
desde django.contrib.flatpages.admin importar FlatPageAdmin
desde django.contrib.flatpages.models importar FlatPage

de cms.search.models importar SearchKeyword
```

```
clase SearchKeywordInline(admin.StackedInline):
    modelo = palabra clave de búsqueda
```

```
clase FlatPageAdminWithKeywords(FlatPageAdmin):
```

```
    en línea = [Buscar palabra clave en línea]
```

```
admin.site.unregister(FlatPage)
```

```
admin.site.register(FlatPage, FlatPageAdminWithKeywords)
```

Este código está haciendo varias cosas. Primero, define un nuevo tipo de clase: una subclase de django.contrib.admin.StackedInline. Esta clase permite que un formulario para agregar o editar un tipo de modelo se incruste dentro del formulario para agregar o editar un modelo con el que está relacionado. (También hay otra clase para esto, llamada TabularInline; la diferencia entre estas clases está en la forma en que se verá el formulario cuando se incruste). En este caso, se le dice a la clase que su modelo es SearchKeyword, lo que significa que incrustará un formulario para agregar o editar palabras clave de búsqueda.

A continuación, la clase de administrador existente para el modelo FlatPage se importa y subclasifica, y se le agrega una nueva opción: la declaración en línea, que debería ser una lista de clases en línea para usar. Esto solo enumera la clase SearchKeywordInline que ha definido. Finalmente, la función admin.site.unregister elimina la definición de administrador existente que las páginas planas aplicación proporcionada, y una llamada a admin.site.register lo reemplaza con la nueva definición que acaba de escribir.

Una vez que haya guardado este archivo, puede volver a la interfaz de administración en su navegador y vea que cada página plana ahora tiene varios formularios en línea para buscar palabras clave (vea la Figura 3-3).

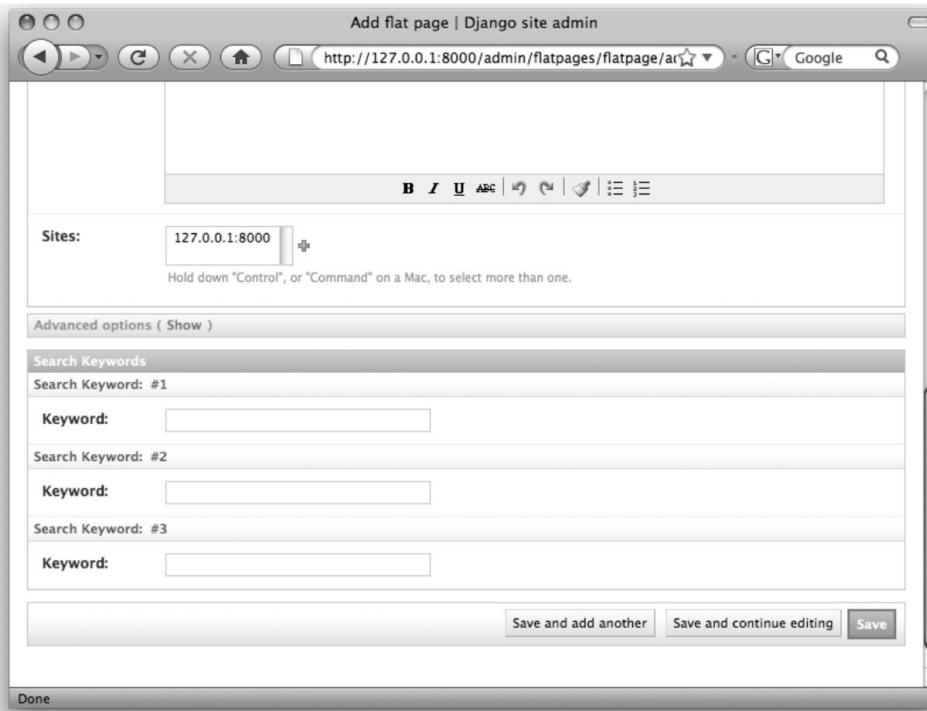


Figura 3-3. Las palabras clave de búsqueda se pueden agregar y editar en línea, junto con una página plana.

Continúe y agregue algunas palabras clave a las páginas de su base de datos; querrá que estén disponibles cuando pruebe la búsqueda mejorada basada en palabras clave.

Agregar soporte para palabras clave en la vista de búsqueda es bastante fácil. Simplemente edite la vista para que se parece a lo siguiente:

```
def buscar (solicitud):
    consulta = solicitud.GET.get('q', '')
    resultados_palabra_clave = []
    resultados = []
    si consulta:
        keyword_results = FlatPage.objects.filter( ÿ
searchkeyword__keyword__in=query.split()).distinct()
        resultados = FlatPage.objects.filter(content__icontains=query)
    volver render_to_response('buscar/buscar.html',
        { 'consulta': consulta,
        'resultados_palabra_clave': resultados_palabra_clave,
        'resultados': resultados })
```

Ha agregado una segunda consulta en el código anterior, que busca páginas cuyas palabras clave de búsqueda asociadas coincidan con la consulta. Aunque puede parecer desalentador al principio, en realidad es bastante simple.

Primero, está utilizando una llamada para filtrar, al igual que en la otra consulta. Este, sin embargo, es interesante. En realidad, está alcanzando "a través" de la clave externa del modelo SearchKeyword y buscando en el campo de palabra clave allí. Cada vez que tenga una relación como esta entre modelos, puede encadenar búsquedas en la relación usando guiones bajos dobles: searchkeyword__keyword

se traduce como "el campo de palabra clave en el modelo de palabra clave de búsqueda relacionado". El operador de búsqueda aquí es __in, que toma una lista de cosas con las que comparar. Lo estás alimentando query.split().

En este punto, la variable de consulta es una cadena y Python proporciona un método split() que, de forma predeterminada, se divide en espacios. Esto es exactamente lo que desea: poder gestionar consultas que contengan varias palabras.

A continuación, la llamada a filtrar es seguida por distinct(). La naturaleza de esta consulta significa que, si una sola página tiene varias palabras clave que coinciden con la búsqueda, aparecerán varias copias de esa página en los resultados. Desea solo una copia de cada página, por lo que usa el distinto ()

que agrega la palabra clave SQL DISTINCT a la consulta de la base de datos.

Finalmente, agrega keyword_results al contexto que usará con la plantilla. La plantilla tendrá que actualizarse. Aunque se está volviendo un poco más complejo debido a los múltiples casos que tiene que manejar, sigue siendo bastante sencillo de seguir:

```
<html>
<cabeza>
<título>Buscar</título>
</cabeza>
<cuerpo>
<método de formulario="obtener" acción="/buscar/">
<p><etiqueta para="id_q">Buscar:</etiqueta>
<tipo de entrada="texto" nombre="q" id="id_q" valor="{{ consulta }}" />
<tipo de entrada="enviar" valor="Enviar" /></p>
</formulario>
```

```
{% si keyword_results o resultados %}  
    <p>Has buscado "{{ consulta }}".</p>  
    {% si palabra clave_resultados %}  
        <p>Páginas recomendadas:</p>  
        <ul>  
            {% para página en keyword_results %}  
                <li><a href="{{ página.get_absolute_url }}">{{ página.título }}</a></li>  
            {% endfor %}  
        </ul>  
    {% terminara si %}  
    {% si resulta %}  
        <p>Resultados de búsqueda:</p>  
        <ul>  
            {% para la página en los resultados%}  
                <li><a href="{{ página.get_absolute_url }}">{{ página.título }}</a></li>  
            {% endfor %}  
        </ul>  
    {% terminara si %}  
    {% terminara si %}  
    {% si consulta y no palabra clave_resultados y no resultados %}  
        <p>No se encontraron resultados.</p>  
    {% más %}  
        <p>Escriba una consulta de búsqueda en el cuadro de arriba y presione "Enviar"  
            para buscar.</p>  
    {% terminara si %}  
</cuerpo>  
</html>
```

La complejidad realmente proviene de las etiquetas if anidadas para tratar los distintos casos, pero esas etiquetas anidadas le permiten cubrir todas las posibilidades. Además, observe la línea que dice {% si palabra clave_resultados o resultados %}: la etiqueta if le permite hacer una lógica simple para probar si se cumple alguna o todas las condiciones. En este caso, proporciona una manera fácil de manejar la situación en la que hay *algún* tipo de resultado y luego aborda los diferentes casos individualmente, según sea necesario. Si ha agregado algunas palabras clave a las páginas de su base de datos, intente buscar esas palabras clave ahora y verá que aparecen las páginas adecuadas en los resultados de búsqueda.

Antes de terminar, agreguemos una función más útil a la vista de búsqueda. Si solo hay un resultado que coincide con precisión con una palabra clave, lo redirigirá directamente a esa página y le ahorrará al usuario un clic del mouse. Puede lograr esto utilizando `HttpResponseRedirect`, una subclase de la clase `HttpResponse` que emite una redirección HTTP a una URL que especifique. Abra `views.py` y agregue la siguiente línea en la parte superior:

```
de django.http importar HttpResponseRedirect
```

Esto es necesario porque, de nuevo, Python requiere que importes explícitamente cualquier cosa que planea usar. Ahora edite la vista de búsqueda de esta manera:

```
def buscar (solicitud):  
    consulta = solicitud.GET.get('q', '')  
    keyword_results = resultados = []
```

```
si consulta:  
    keyword_results = FlatPage.objects.filter(ÿ  
        searchkeyword__keyword__in=query.split()).distinct()  
    si keyword_results.count() == 1:  
        devolver HttpResponseRedirect(keyword_results[0].get_absolute_url())  
    resultados = FlatPage.objects.filter(content__icontains=query)  
    volver render_to_response('buscar/buscar.html',  
        { 'consulta': consulta,  
        'resultados_palabra_clave': resultados_palabra_clave,  
        'resultados': resultados })
```

Hasta ahora, ha estado tratando los resultados de las consultas de la base de datos como listas normales de Python, y, aunque se pueden usar así, en realidad forman un tipo especial de objeto llamado QuerySet. QuerySet es una clase que utiliza Django para representar una consulta de base de datos. Cada QuerySet tiene los métodos que ha visto hasta ahora (`filter()` y `distinct()`), además de varios otros, que puede "encadenar" para crear una consulta cada vez más compleja. Un QuerySet también tiene un conteo () método, que le dirá cuántas filas en la base de datos coincidieron con la consulta. (Hace un SELECT COUNT para averiguarlo, aunque por razones de eficiencia, también puede aprovechar otros métodos que no requieren una consulta adicional).

Advertencia: ¿Cuándo ejecuta Django la consulta?

La característica más importante de QuerySet es que es "perezoso". Inicialmente, no hace nada excepto tomar nota de qué consulta se supone que debe ejecutar eventualmente en la base de datos, por lo que puede seguir encadenando cosas adicionales para agregar filtrado, una cláusula DISTINCT u otras condiciones. La consulta de la base de datos real no se ejecutará hasta que haga algo que obligue a que suceda, como (en este caso) contar o recorrer los resultados.

Al usar `count()`, puede ver si una búsqueda de palabra clave arrojó exactamente un resultado y luego emita una dirección. La URL a la que redirige es `keyword_results[0].get_absolute_url()`; este fragmento de código extrae la primera (y, en este caso, la única) página de resultados y llama a su método `get_absolute_url()` para obtener la URL.

Adelante, prueba esto. Agregue una nueva palabra clave de búsqueda que sea exclusiva de una página y luego búsqüela. Si configuró la vista como se describió anteriormente, será redirigido inmediatamente a esa página.

Mirando hacia el futuro

En los últimos dos capítulos, ha pasado literalmente de *nada* a un CMS útil y funcional con una sencilla interfaz administrativa basada en web. Agregó la edición de texto enriquecido para evitar que los usuarios tengan que escribir HTML sin procesar y agregó un sistema de búsqueda que permite a los administradores configurar resultados basados en palabras clave. A lo largo del camino, ha escrito menos de cien líneas de texto real.

código. Django hizo la mayor parte del trabajo pesado y usted solo proporcionó las plantillas y un poco de código para habilitar la función de búsqueda.

Lo mejor de todo es que ahora tiene una solución simple y reutilizable para una tarea común de desarrollo web: un CMS estilo folleto. Cada vez que necesite volver a crearlo, puede configurar Django y seguir estos mismos pasos sencillos (o incluso simplemente hacer una copia del proyecto, cambiando la configuración adecuada en el proceso). Hacer esto te ahorrará tiempo y te liberará del tedio de una situación bastante repetitiva.

Síntase libre de pasar un tiempo jugando con el CMS: agregue un poco de estilo a las plantillas, personalice un poco más las páginas de administración o, si se siente realmente aventurero, incluso intente agregar algunas funciones propias. Si desea algún tipo de tarea, consulte la documentación de la API de la base de datos de Django (en línea en www.djangoproject.com/documentation/db-api/) y vea si puede averiguar cómo agregar una vista de índice que enumere todas las páginas en la base de datos.

Cuando esté listo para un nuevo proyecto, comience a leer el siguiente capítulo, donde comenzará ing en su primera aplicación desde cero: un weblog impulsado por Django.

Capítulo 4



Un weblog potenciado por Django

los El CMS simple que creó en los últimos dos capítulos fue un buen ejemplo de cómo las aplicaciones integradas de Django pueden ayudarlo a poner en marcha un proyecto rápidamente y sin mucho código. Pero la mayor parte del tiempo, probablemente estarás desarrollando cosas que no están cubiertas tan claramente por las aplicaciones precompiladas incluidas con el mismo Django. Django todavía tiene mucho que ofrecer en estas situaciones, principalmente quitándote la mayor parte del trabajo repetitivo de tus hombros. Durante el resto de este libro, escribirá aplicaciones desde cero y verá cómo los componentes de Django pueden hacer que el proceso sea mucho más fácil y mucho menos doloroso. Comencemos con algo que se está convirtiendo rápidamente en una necesidad para cualquier organización que se conecta a Internet: un weblog.

Compilación de una lista de verificación de características

Las aplicaciones del mundo real generalmente comienzan con al menos una especificación aproximada de lo que deberán hacer, y aquí seguiré el mismo proceso. Antes de sentarse y escribir la aplicación de blog, deberá decidir por adelantado qué quiere que haga. Cuando escribí una aplicación de weblog para mi uso personal, esta era la lista de características que tenía en mente:

- Tiene que proporcionar una manera fácil de agregar y editar entradas sin escribir en bruto HTML.
- Debe ser compatible con varios autores y proporcionar una forma de separar las entradas de acuerdo con autor.
- Cada entrada debe permitir que se muestre un breve extracto opcional cuando se realiza un resumen. necesario.
- Los autores del weblog deberían poder crear categorías y asignarles entradas.
- Los autores deben poder decidir qué entradas se mostrarán públicamente y cuáles no lo hará (para, por ejemplo, marcar una entrada sin terminar como borrador y volver a ella más tarde).
- Las entradas deben poder ser "destacadas", y estas entradas deben ser fácilmente recuperables (para mostrar en la página de inicio del weblog, por ejemplo).
- También se debe proporcionar un registro de enlaces para permitir la publicación de enlaces interesantes o notables.
- Tanto las entradas como los enlaces deben admitir el etiquetado: agregar palabras descriptivas arbitrarias para proporcionar metadatos u organización adicionales.

- El registro de enlaces debe integrarse con Delicious (un sitio de marcadores sociales en <http://delicious.com/>) u otros servicios populares para compartir enlaces, de modo que los enlaces publicados en el weblog también se muestren automáticamente en el servicio.
- Los visitantes deben poder buscar entradas y enlaces por fecha, por etiqueta o (en el caso de entradas) por categoría.
- Los visitantes del blog deberían poder dejar comentarios en las entradas y enlaces.
- Los comentarios deben estar sujetos a algún tipo de moderación para evitar comentarios.

correo no deseado.

Hay más funciones que podría agregar aquí, pero esta lista es suficiente para mantenerlo ocupado por un tiempo; hará uso de una amplia gama de características de Django. Entonces empecemos.

Escribir una aplicación Django

En el último capítulo, cuando agregó la función de búsqueda y el modelo SearchKeyword al CMS simple, creó una aplicación Django simple, inicialmente creada con la aplicación de inicio `manage.py` orden—detenerlos. En ese momento, no dediqué mucho tiempo a detallar lo que sucede en una aplicación Django. Sin embargo, ahora que va a comenzar a hacer cosas más complejas, vale la pena detenerse un momento para repasarlo, para comprender cómo las aplicaciones individuales de Django difieren de un proyecto de Django.

Proyectos vs Aplicaciones

Como ya ha visto, configura un *proyecto* Django a través de su módulo de configuración, que—entre otras cosas, especifica la base de datos a la que se conectará y la lista de aplicaciones que utiliza. En cierto modo, la cualidad definitoria de un proyecto es que es la "cosa" que contiene la configuración. (incluyendo tanto el módulo de configuración como el módulo URLConf raíz, que especifica la configuración de URL base del proyecto).

Un proyecto también puede contener otro código si tiene sentido que ese código sea parte del proyecto directamente, pero la necesidad de esto es bastante rara. Generalmente, un proyecto existe para proporcionar un "contenedor" para que un conjunto de aplicaciones de Django funcionen juntas, y la mayoría de los proyectos nunca lo harán. necesita algo más allá de los archivos iniciales creados por `django-admin.py startproject`.

Una aplicación Django , por otro lado, es responsable de proporcionar alguna funcionalidad y debe tratar de enfocarse en esa funcionalidad tanto como sea posible. Una aplicación no tiene un módulo de configuración, ese es el trabajo de cualquier proyecto que lo use, pero proporciona varias otras cosas:

- Una aplicación puede (ya menudo lo hace) proporcionar uno o más modelos de datos.
- Una aplicación generalmente proporciona una o más funciones de vista, a menudo relacionadas de alguna manera a sus modelos de datos.
- Una aplicación puede proporcionar bibliotecas de etiquetas de plantillas personalizadas, que amplían las funciones de Django. sistema de plantillas con funciones adicionales específicas de la aplicación.
- Una aplicación puede (y generalmente debería) proporcionar un módulo URLConf adecuado para ser "conectado" a un proyecto (a través de la directiva `include`, como ya ha visto en el caso de la interfaz administrativa y la aplicación de páginas planas incluidas con Django).

Y, por supuesto, una aplicación también debe proporcionar cualquier código de "utilidad" adicional necesario para respaldarse a sí misma, o debe tener dependencias claras en otras aplicaciones o en módulos de Python de terceros que brindan ese soporte.

Aplicaciones independientes y acopladas

Es importante ser consciente de la distinción entre dos formas diferentes de desarrollar aplicaciones Django. Un método, que usé en el último capítulo, usa la aplicación de inicio `manage.py` comando para crear un módulo de aplicación dentro del directorio del proyecto. Si bien esto es fácil y conveniente, tiene algunos inconvenientes, sobre todo en el hecho de que "acopla" el aplicación al proyecto. Cualquier otro código de Python que quiera acceder a esa aplicación necesita saber que "vive" dentro de ese proyecto en particular. (Por ejemplo, para importar la palabra clave de búsqueda modelo de una pieza de código separada, tendría que importarlo desde `cms.search.models` en lugar de solo `search.models`). Cada vez que desee reutilizar la aplicación, debe hacer una copia del proyecto o crear un conjunto de directorios vacíos para emular el directorio del proyecto estructura.

La alternativa es desarrollar una aplicación independiente, que actúe como una aplicación independiente y autónoma. contiene el módulo de Python y no es necesario mantenerlo dentro de un directorio de proyecto para que funcione correctamente. Una aplicación independiente es mucho más fácil de reutilizar y distribuir, pero configurarla up implica un poco más de trabajo inicial: el comando `manage.py startapp` no puede crear cosas automáticamente para usted a menos que esté desarrollando una aplicación que esté acoplada a un proyecto en particular.

Hay casos en los que desarrollará aplicaciones únicas que no necesitan ser reutilizables o distribuibles. (En esos casos, está perfectamente bien desarrollarlos dentro de un proyecto en particular y junto con él; solo tenga cuidado con el hecho de que muchas piezas de código supuestamente "únicas" como esto eventualmente necesita ser reutilizado en otro lugar). Pero en general, obtendrá más beneficios al desarrollar aplicaciones independientes que pueden reutilizarse en muchos proyectos diferentes. Así es como trabajará durante el resto de este libro.

Creación de la aplicación de blog

Debido a que esta será una aplicación independiente, deberá crear un módulo de Python manualmente en lugar de depender de la aplicación de inicio de `manage.py`, pero eso no es demasiado difícil. Tal vez recuerde que todo lo que hizo realmente el comando `startapp` fue crear un directorio y poner tres archivos en él, y eso es todo lo que necesita hacer para comenzar.

Solo hay dos cosas de las que debe preocuparse al configurar manualmente una nueva aplicación módulo de cationes: cómo llamarlo y dónde ponerlo. Puede llamar a una aplicación por cualquier nombre que sea legal para un módulo de Python: Python permite que los nombres de los módulos consten de cualquier combinación de letras y números y, opcionalmente, guiones bajos para separar palabras (aunque el nombre debe comenzar con una letra). Debido a que Django lleva el nombre de un músico de jazz, a algunos desarrolladores les gusta continuar con el patrón al nombrar aplicaciones con nombres de figuras famosas del jazz. (Por ejemplo, el empresa para la que trabajo vende un CMS llamado Ellington, llamado así por Duke Ellington, y hay un popular aplicación de comercio electrónico de código abierto llamada Satchmo en honor a Louis Armstrong). Esto no es obligatorio, pero es algo que me gusta hacer cuando no hay un nombre más obvio. Entonces, cuando escribí mi propia aplicación de blog, la nombré Coltrane en honor a John Coltrane. Eso parecía apropiado, dado que Coltrane era conocido por la composición y la improvisación, dos habilidades que también hacen a un buen bloguero.

Dónde poner el código de la aplicación es una pregunta un poco más difícil de responder. Hasta ahora no te has encontrado con este problema porque el script `manage.py` de Django, para facilitar la configuración inicial y el desarrollo, oscurece un requisito importante para el código de Python: debe colocarse en un directorio que esté en la *ruta de Python*. La ruta de Python es simplemente una lista de directorios donde Python buscará cada vez que encuentre una declaración de importación. Por lo tanto, el código que debe importarse (como lo será su aplicación, para poder usarse como parte de un proyecto de Django) debe estar en la ruta de Python.

Cuando instaló Python, se configuró una ruta predeterminada de Python e incluía un directorio llamado `sitename-packages`. Cuando instaló Django, el script del instalador `setup.py` colocó todo el código de Django dentro de ese directorio. Puede colocar su propio código en los paquetes del sitio si lo desea, pero generalmente no es una buena idea hacerlo. El directorio de paquetes del sitio casi siempre se configura en una parte del sistema de archivos de su computadora que requiere acceso administrativo para escribir, y no se divertirá mucho saltando constantemente a través del aro de autenticación para colocar cosas allí. En cambio, la mayoría de los programadores de Python crean un directorio donde guardarán su propio código y lo agregarán a la ruta de Python, así que hágámoslo. Debido a que ya ha creado un directorio para contener sus proyectos de Django, continúe y agréguelo a su ruta de Python y coloque sus aplicaciones independientes también. De esta manera, deberá agregar solo un directorio a la ruta de Python y no estará dispersando el código en varias ubicaciones en su computadora.

Advertencia: cómo cambiar su ruta de Python

En Mac OS X, así como en la mayoría de los demás sistemas basados en UNIX o Linux, cambiar la ruta de Python es fácil.

Puede escribir un comando como el siguiente para agregar directorios a la ruta:

```
export PYTHONPATH=/home/myuser/my-python-code:$PYTHONPATH
```

Para evitar escribir eso una y otra vez, generalmente puede agregarlo a un archivo llamado `.profile` o `.bash_perfil` en su directorio de inicio. De esa manera, se ejecutará cada vez que abra una línea de comando (aunque es posible que también deba agregarlo a un archivo `.shrc` o `.bashrc`).

En Windows, la configuración es un poco más complicada. Esto se debe en gran medida a que Windows, a diferencia de los sistemas basados en UNIX, no es tan amigable con los programas basados en la línea de comandos. En el área Sistema del Panel de control, en la pestaña Avanzado, puede establecer variables de entorno. La variable `PYTHONPATH` ya debe estar configurada con el valor inicial que proporcionó Python, y puede agregarle nuevos directorios (los directorios en la lista deben estar separados por punto y coma).

Ahora, en el mismo directorio donde creó el proyecto `cms` (en otras palabras, junto `cms`, no dentro de `cms`), cree un nuevo directorio llamado `coltrane`. Dentro de eso, crea cuatro archivos vacíos:

- `__init__.py`
- `modelos.py`
- `vistas.py`
- `admin.py`

Esto es todo lo que necesitará por ahora: el archivo `__init__.py` le indicará a Python que el directorio `coltrane` es un módulo de Python, y los archivos `models.py` y `views.py` contendrán el código inicial para la aplicación de blog. Finalmente, `admin.py` le permitirá configurar la interfaz administrativa de Django para el weblog.

Diseñando los modelos

Necesitará varios modelos para implementar todas las características de su lista, y un par de ellos serán moderadamente complejos. Sin embargo, puede comenzar con uno simple: el modelo que representará las categorías a las que se asignarán las entradas. Abre el archivo `models.py` de la aplicación `weblog`.

archivo y agregue lo siguiente:

```
desde modelos de importación django.db
```

```
categoría de clase (modelos.Modelo):
```

```
    título = modelos.CharField(max_length=250)
    slug = modelos.SlugField(único=True)
    descripción = modelos.TextField()
```

```
def __unicode__(uno mismo):
    volver self.title
```

La mayor parte de esto debería ser familiar después de su primera incursión en los modelos de Django en el último capítulo. La declaración de importación extrae el módulo de modelos de Django, que incluye el Modelo base clase y definiciones para los diferentes tipos de campos para representar datos. Ya ha visto `CharField` (este tiene una longitud máxima más larga para permitir nombres de categoría largos) y el método `__unicode__()` (que, para este modelo, devuelve el valor del campo de título). Pero aquí hay dos nuevos tipos de campos: `SlugField` y `TextField`.

El significado de `TextField` es bastante intuitivo. Está destinado a almacenar una mayor cantidad de texto (en la base de datos, se convertirá en una columna de TEXTO), y se utilizará aquí para proporcionar una descripción útil de la categoría.

`SlugField` es un poco más interesante. Está destinado a almacenar una *babosa*: una pieza corta y significativa de texto, compuesto enteramente de caracteres que son seguros de usar en una URL. Utiliza `SlugField` cuando genera la URL para un objeto en particular. Esto significa, por ejemplo, que en lugar de tener una URL como `/categories?category_id=47`, podría tener `/categories/programming/`. Esto es útil para los visitantes de su sitio (porque hace que la URL sea significativa y más fácil de recordar) y para la indexación de motores de búsqueda. Las URL que contienen una palabra relevante a menudo ocupan un lugar más alto en Google y otros motores de búsqueda que las URL que no la contienen. El término *slug*, como corresponde a la herencia de Django, proviene de la industria periodística, donde se usa en la producción de preprints y, a veces, en formatos de cable como un identificador más corto para una noticia.

Tenga en cuenta que he agregado un argumento adicional a `SlugField`: `unique=True`. Debido a que el `slug` se usará en la URL y la misma URL no puede hacer referencia a dos categorías diferentes, debe ser única. La interfaz administrativa de Django hará cumplir la unicidad para este campo, y `manage.py syncdb` creará la tabla de la base de datos con una restricción ÚNICA para esa columna.

También querrá poder administrar categorías a través de la interfaz administrativa de Django, entonces en el archivo admin.py agregue lo siguiente:

```
desde django.contrib.admin importador de importación
de coltrane.models categoría de importación
```

```
clase CategoryAdmin(admin.ModelAdmin):
    pasar
```

```
admin.site.register(Categoría, CategoryAdmin)
```

Al desarrollar una aplicación, es útil detenerse de vez en cuando y probarla. Así que regrese al proyecto cms, abra su archivo de configuración y agregue coltrane, la nueva aplicación de weblog, a su configuración INSTALLED_APPS:

```
APLICACIONES_INSTALADAS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.flatpages',
    'cms.buscar',
    'coltrano',
)
```

Debido a que está directamente en la ruta de Python, simplemente agregar coltrane funcionará. A continuación, ejecute python manage.py syncdb para instalar la tabla para el modelo de categoría e iniciar el servidor de desarrollo. La página de índice de administración se verá como la que se muestra en la Figura 4-1.

Puede ver que aparece el modelo Categoría, pero está etiquetado como "Categorías". eso no es bien. La interfaz de administración de Django genera esa etiqueta a partir del nombre de la clase del modelo e intenta pluralizarla agregando una "s", lo que funciona la mayor parte del tiempo. Aunque no siempre funciona, y cuando no lo hace, Django le permite especificar el nombre plural correcto. Vuelva al archivo models.py del blog y edite la clase de modelo Categoría para que se vea como lo siguiente:

```
categoría de clase (modelos.Modelo):
    título = modelos.CharField(max_length=250)
    slug = modelos.SlugField(único=True)
    descripción = modelos.TextField()
```

```
metaclase:
    verbose_name_plural = "Categorías"
```

```
def __unicode__(uno mismo):
    volver self.title
```

Una vez que guarde el archivo y actualice la página de índice de administración en su navegador, debería ver algo similar a lo que se muestra en la Figura 4-2.

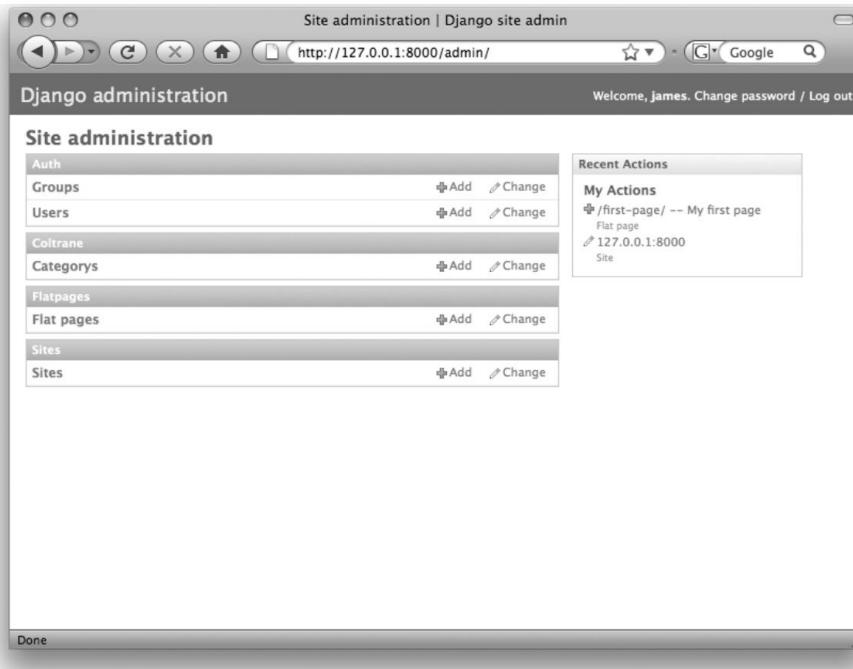


Figura 4-1. La interfaz de administración de Django con el modelo de Categoría

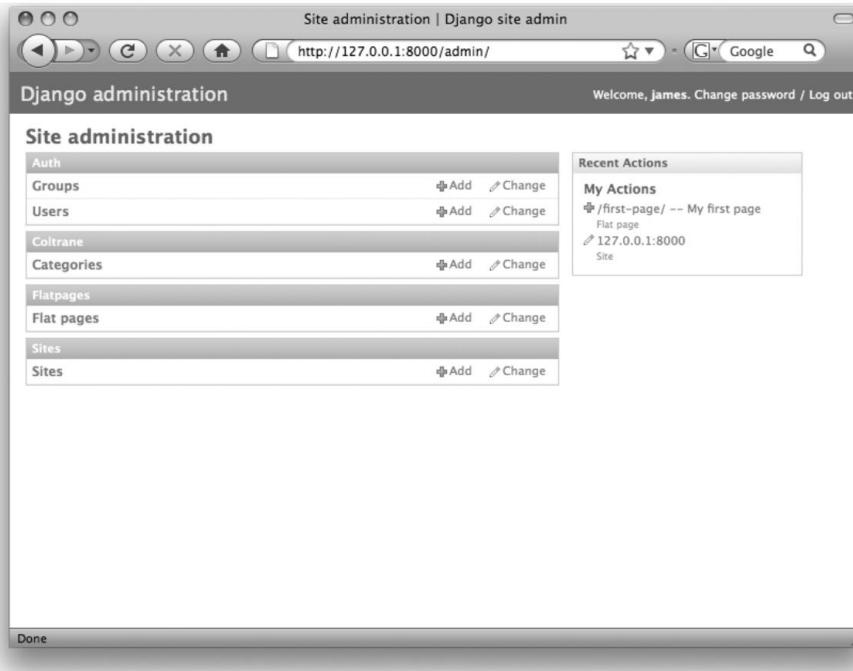


Figura 4-2. La pluralización correcta del modelo Categoría

Debido a que a menudo necesita proporcionar metainformación adicional sobre un modelo, Django le permite agregar una clase interna llamada Meta, que puede especificar una gran cantidad de opciones comunes. En este caso, está utilizando una opción llamada verbose_name_plural, que devolverá un nombre en plural para la clase modelo siempre que sea necesario. (También hay una opción verbose_name, que puede especificar una versión singular si difiere significativamente del nombre de la clase, pero no la necesita aquí). Verá una serie de otras opciones útiles para la clase Meta interna a medida que avanza. los modelos del weblog.

Si hace clic en la interfaz de administración para agregar una categoría, verá los campos apropiados en una forma agradable: título, slug y descripción. Pero agregar una categoría de esta manera revelará otra deficiencia. La mayoría de las veces, el valor del campo slug probablemente será similar o incluso idéntico al valor del campo de título (por ejemplo, una categoría de Programación probablemente debería tener un slug como "programación"). Escribir manualmente el slug cada vez sería tedioso, así que ¿por qué no generarlo automáticamente a partir del título y dejar que el usuario lo cambie manualmente si es necesario? Esto es bastante fácil de hacer. En el archivo admin.py, cambie la clase CategoryAdmin para que se vea así:

```
clase CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = { 'slug': ['título'] }
```

Luego guarde el archivo admin.py y agregue una categoría. El argumento prepopulated_fields activará una pieza útil de JavaScript en la interfaz administrativa de Django, y automáticamente completará un slug sugerido a medida que escribe un valor en el campo de título. Tenga en cuenta que prepopulated_fields obtiene una lista: esto significa que puede especificar varios campos desde los cuales dibujar el valor del slug, lo cual no es común pero a veces es útil. El JavaScript que genera slugs también es lo suficientemente inteligente como para reconocer y omitir palabras como "a", "an", "the", etc. Estas se denominan *palabras vacías* y, por lo general, no son útiles en un slug.

Además, tenga en cuenta que cuando Django crea la tabla de base de datos para este modelo, agregará un índice a la columna de babosas. Puede decirle manualmente a Django que haga esto con cualquier campo (usando la opción db_index=True para el campo), pero SlugField obtendrá el índice automáticamente. Esto proporciona un impulso de rendimiento en el caso común de usar un slug de una URL para realizar consultas a la base de datos.

Advertencia: Slugs y Normalización

Si está familiarizado con las teorías de la normalización de bases de datos (directrices para diseñar bases de datos relacionales para evitar información duplicada), es posible que se pregunte por qué el slug tiene su propia columna si solo se va a generar a partir del título. Esto huele sospechosamente a duplicación innecesaria, ¿no?

El slug tiene su propia columna principalmente porque no depende necesariamente del título. Para algunos títulos de categorías largas, por ejemplo, el slug puede diferir significativamente para permanecer corto y memorable. Además, las tablas normalizadas no son una regla absoluta. La desnormalización deliberada, siempre que se haga con cuidado, a menudo puede generar importantes mejoras de rendimiento, como verá cuando escriba el modelo para las entradas.

Mientras observa las categorías en la interfaz de administración, hagamos una pausa y agreguemos otra función útil: sugerencias útiles que brindan a los usuarios de la aplicación de weblog más información a medida que completan los datos. Así que edite la definición del campo de título en models.py de esta manera:

```
title = models.CharField(max_length=250, help_text='Máximo 250 caracteres.')
```

A continuación, guarde el archivo `models.py` y vuelva a mirar el formulario de administración (consulte la Figura 4-3).

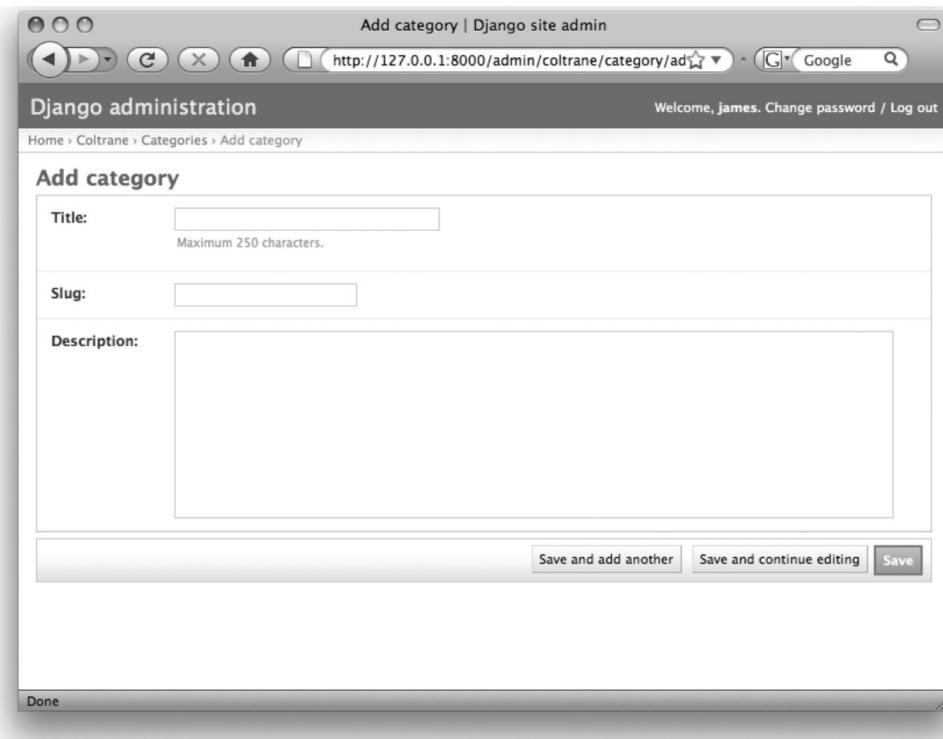


Figura 4-3. El formulario de administración para agregar una categoría

La cadena proporcionada en el argumento `help_text` aparece debajo del cuadro de texto del campo de título, lo que brinda una pista útil sobre lo que se puede ingresar allí. Puede agregar texto de ayuda a cualquier campo de su modelo y, por lo general, es una buena idea hacerlo siempre que haya algo que los usuarios deban saber al ingresar datos. Así que vamos a agregarlo también para el campo `slug`:

```
slug = modelos.SlugField(help_text="ÿ  
Valor sugerido generado automáticamente a partir del título. Debe ser único.")
```

A continuación, guarde el archivo `models.py` y vuelva a actualizar el formulario de administración. Verá que aparece texto debajo del cuadro de texto del campo `slug`, notificando a los usuarios que se completará un valor sugerido y recordándoles que el `slug` debe ser único.

Antes de continuar, agreguemos una mejora más. Si intenta agregar un par de categorías, puede notar que la página de administración, que enumera todas las categorías, no necesariamente las mantiene en ningún orden. Sería bueno que se muestren en una lista alfabética para que un usuario pueda escanearlos rápidamente. Una vez más, esto es bastante fácil de hacer. La clase `Meta` interna acepta una opción para especificar un orden predeterminado para el modelo:

metaclase:

```
ordenar = ['título']  
verbose_name_plural = "Categorías"
```

Guarde el archivo `models.py` después de insertar ese código. Cuando actualice la página de administración, verá que las categorías están ordenadas alfabéticamente. A menos que lo anule específicamente por consulta, Django ahora agregará la cláusula `ORDER BY title ASC` a cualquier consulta de base de datos para la tabla de categorías, lo que devolverá las categorías en el orden alfabético correcto. Observe que el valor para ordenar es una lista. Puede especificar varios campos aquí y se colocarán correctamente en una cláusula `ORDER BY` para la mayoría de las consultas. (La aplicación de administración usa solo el primer campo en la opción de pedido al recuperar listas de objetos).

Otra cosa útil que puede agregar es un método especial llamado `get_absolute_url()`. En el Capítulo 2, vio que esta es la práctica estándar para un modelo de Django que quiere especificar su propia URL, y cada modelo que se pretenda usar en una vista pública debe tener un método `get_absolute_url()`. Así que agreguemos uno:

```
def get_absolute_url(auto):
    devuelve "/categorías/%s/" % self.slug
```

Por ahora, solo coloque este método en la parte inferior de la clase `Categoría`; recuerde que debe estar sangrado para ser parte de la clase. Verá un poco más adelante cómo mantener organizadas todas las partes de una clase modelo de Django.

Este método devolverá una cadena con el valor del campo `slug` de la categoría interpolado en el lugar correcto. Agregar este método también hará que la interfaz de administración muestre una Vista en el botón del sitio para cada categoría, aunque por ahora no será muy útil porque aún no ha configurado ninguna URL o vista para mostrarlas.

Advertencia: formato de cadenas de Python

Si bien es posible crear una cadena por concatenación, construyendo las piezas una a la vez y usando el signo más (+) para unirlas, eso se vuelve extremadamente tedioso si necesita incluir múltiples variables o valores generados en el resultado final. . Por lo tanto, la mayoría de los lenguajes, incluido Python, brindan una forma más sencilla de interpolar variables y valores en una cadena utilizando caracteres de formato especial.

Los caracteres de formato (y, en muchos idiomas, los nombres de funciones que construyen cadenas en este fashion) provienen de la familia de funciones `printf` en la biblioteca estándar del lenguaje de programación C. Pero Python no usa una función para esto. En su lugar, simplemente escriba la cadena con los caracteres apropiados para matizar, luego siga con un signo de porcentaje (%) y los valores que se interpolarán en el resultado.

La especificación completa de la sintaxis de formato de cadena de Python, incluida una lista de los caracteres de formato, está disponible en la documentación de Python en línea en <http://docs.python.org/library/stdtypes.html#operaciones-de-formato-de-cadenas>.

Construcción del modelo de entrada

Ahora que tiene categorías para asignar entradas, es hora de crear el modelo para las entradas del weblog. Debido a que realmente será el centro de atención de esta aplicación, también será el modelo más complejo que necesitará construir, así que analicemos las cosas poco a poco.

Campos básicos

En primer lugar, debe tener algunos campos básicos para contener el título de la entrada, el extracto opcional, el texto de la entrada y la fecha en que se publicó la entrada. Así que empecemos con esos. Abra el archivo `models.py` y, debajo de la clase de modelo Categoría, comience a agregar el nuevo modelo Entrada.

(No ejecute `manage.py syncdb` todavía. Agregarás más campos a este modelo, y es mejor esperar hasta que termine antes de que Django cree las tablas de la base de datos). Agrega estas líneas primero:

Entrada de clase (`modelos.Modelo`):

```
título = modelos.CharField(max_length=250)
extracto = modelos.TextField(en blanco=True)
cuerpo = modelos.TextField()
pub_date = modelos.DateTimeField()
```

Además, continúe y configure una definición de administrador básica para este modelo en `admin.py`. querrás para cambiar la línea que importa el modelo Categoría para importar también Entrada:

de `coltrane.models` categoría de importación, Entrada

Y luego agregue la nueva clase de administrador para el modelo de entrada:

```
clase EntryAdmin(admin.ModelAdmin):
    pasar
```

```
admin.site.register(Entrada, EntryAdmin)
```

Los primeros tres campos en este nuevo modelo (título, extracto y cuerpo) son todos de tipos que ha visto antes. Pero el campo `pub_date` tiene un nuevo tipo de campo llamado `DateTimeField`. Representará la fecha de publicación de la entrada. En comparación con los tipos de campo que ha visto hasta ahora, `DateTimeField` es único en varios aspectos:

- Cuando almacene entradas o las recupere de la base de datos, este campo tendrá como su valor un objeto de fecha y hora de Python (la clase de fecha y hora se encuentra en el módulo, que es una parte estándar de Python), independientemente de cómo se almacene realmente en la base de datos (diferentes bases de datos, internamente, lo manejarán de maneras ligeramente diferentes).
Django también proporciona tipos de campos separados, que almacenan solo una fecha o solo una hora, pero `DateTimeField` maneja ambos. Esto significa que puede realizar un seguimiento no solo de la fecha en que se publicó la entrada, sino también de la hora (para que eventualmente pueda mostrar algo como "Publicado el 7 de octubre a las 22:00 horas").
- El tipo exacto de columna de la base de datos creada para este campo variará de una base de datos a otra. base de datos. Hasta ahora, ha visto campos que constantemente se convierten en el mismo tipo de columna (`VARCHAR` para `CharField`, por ejemplo) sin importar qué tipo de base de datos esté usando. Sin embargo, debido a las variaciones en los tipos de columnas, Django utilizará diferentes opciones según corresponda. Por ejemplo, `DateTimeField` se convertirá en una columna `DATETIME` en `SQLite` y una columna `TIMESTAMP` en `PostgreSQL`.
- Hasta ahora, cada tipo de campo con el que ha trabajado se ha traducido directamente en una entrada de formulario en la interfaz administrativa, generalmente un cuadro de texto. Sin embargo, un `DateTimeField` se convierte en dos entradas de formulario: una para la fecha y otra para la hora. Verá esto cuando comience a trabajar con entradas en la interfaz administrativa.

También hay una opción en el campo de extracto que no ha visto antes: en blanco = Verdadero. Así que hasta ahora, la cuestión de los campos obligatorios no ha surgido realmente. Ha estado trabajando con modelos simples en los que no es necesario que algunas cosas sean opcionales, por lo que el comportamiento predeterminado de Django: hacer que el campo sea obligatorio al ingresar datos a través de un formulario en la interfaz de administración y crear una columna NOT NULL en la base de datos, ha estado bien. En este caso, sin embargo, debe hacer que el campo de extracto sea opcional, y la opción en blanco = Verdadero le dice a Django que está bien no ingresar nada para este campo. Puede agregar blank=True a cualquier tipo de campo en un modelo de Django.

Advertencia: campos en blanco frente a campos nulos

Django en realidad usa dos opciones separadas para manejar los campos requeridos y no requeridos en los modelos: en blanco y nulo. La opción en blanco afecta solo a los formularios que se muestran a los usuarios de una aplicación impulsada por Django y evita que el formulario muestre un error de validación si no se ingresa ningún valor. La opción nula, por otro lado, configurará la base de datos para aceptar un valor NULL. Si necesita permitir que los usuarios dejen un campo en blanco y tengan un NULL insertado en su columna en la base de datos, deberá especificar ambas opciones.

Si esto le parece extraño, tenga en cuenta que hay casos muy comunes en los que querrá permitir que un usuario para dejar un campo en blanco en un formulario (o incluso ocultar un campo por completo) pero aún evitar que un valor NULL vaya a la base de datos (al generar un valor para ese campo si el usuario no proporciona uno). Verá un ejemplo más adelante en este capítulo.

Además, es importante tener en cuenta que para los tipos de campo basados en texto (CharField, TextField y otros), Django nunca insertará un NULL. Para estos tipos de campos, se insertará un valor en blanco como una cadena vacía. Esto es para evitar una situación en la que potencialmente haya dos valores en blanco diferentes para el campo (ya sea una cadena vacía o NULL) y para garantizar que el código que verifica los valores en blanco se mantenga simple. Debido a esto, generalmente debe evitar especificar null=True en los tipos de campo basados en texto.

Slugs, valores predeterminados útiles y restricciones de unicidad

Así como agregó un slug para las categorías, es una buena idea agregar uno para las entradas y configurarlo para completar un valor predeterminado desde el título de la entrada. Así que agregue lo siguiente al modelo Entry:

```
slug = modelos.SlugField()
```

Luego cambie la clase EntryAdmin para completar automáticamente el slug:

```
clase EntryAdmin(admin.ModelAdmin):
    prepopulated_fields = { 'slug': ['título'] }
```

Con el modelo Categoría, agregó unique=True para forzar que el slug sea único, pero para las entradas sería bueno tener algo ligeramente diferente. La mayoría de los buenos programas de weblog crean direcciones URL que incluyen las fechas de publicación de las entradas (para que se vean como /2007/10/09/entry-title/), lo que significa que todo lo que realmente necesita es que la combinación del slug y la fecha de publicación sea única. Django proporciona una manera fácil de especificar esto, a través de una opción llamada unique_for_date:

```
slug = modelos.SlugField(unique_for_date='pub_date')
```

Esto le indicará a Django que permita que un slug en particular se use solo una vez en cada fecha. La restricción `unique_for_date` es una de las tres restricciones basadas en fechas admitidas por Django. Los otros dos son `unique_for_month` y `unique_for_year`. Mientras que `unique_for_date` permite que un valor dado se use solo una vez al día, los otros dos valores restringen el uso una vez al mes y una vez al año, respectivamente.

También sería bueno proporcionar un valor predeterminado razonable para el campo `pub_date`. La mayoría de las veces, las entradas se "publicarán" el mismo día en que se ingresan, por lo que el valor predeterminado es el actual. La fecha y la hora serían convenientes para los autores del weblog. Django le permite especificar un valor predeterminado para cualquier tipo de campo usando la opción predeterminada. La única pregunta es cómo especificar un valor predeterminado de "ahora mismo".

La respuesta se encuentra en el módulo de fecha y hora estándar de Python. Esto proporciona una función, `fecha y hora`. `datetime.now()`, para obtener la fecha y la hora actuales y devuelve el tipo correcto de objeto (una fecha y hora de Python, como se describió anteriormente) para completar un `DateTimeField`. Entonces, en la parte superior del archivo `models.py`, agregue una declaración de importación para que el módulo de fecha y hora esté disponible:

```
importar fecha y hora
```

y luego edite el campo `pub_date` para agregar el valor predeterminado:

```
pub_date = modelos.DateTimeField(predeterminado=datetime.datetime.now)
```

Observe que no hay paréntesis allí: es `datetime.datetime.now`, *no* `datetime.datetime.now()`. Cuando especifica un valor predeterminado, Django le permite proporcionar un valor apropiado o una función, que generará el valor apropiado a pedido. En este caso, está proporcionando una función y Django la llamará siempre que se necesite el valor predeterminado. Esto garantiza que se genere la fecha y hora actual correcta cada vez.

Advertencia: funciones frente a valores devueltos

Python le permite referirse a funciones directamente y pasárselas como objetos de "primera clase" de la misma manera que puede pasar cualquier otro tipo de valor. La diferencia es simplemente que omite los paréntesis, como lo ha hecho con el valor predeterminado para el campo `pub_date`. Comprender la diferencia entre la función y el valor devuelto al llamar a la función es fundamental para usar muchas partes de Django de manera efectiva. En este caso, si el valor predeterminado se hubiera especificado como `datetime.datetime.now()`, se habría llamado una vez, cuando se cargó el modelo por primera vez, y nunca más, creando un valor predeterminado aparentemente invariable.

En general, los programadores de Python se refieren a esto como pasar un invocable, un valor que puede llamarse como una función (aunque en algunos usos avanzados de Python, puede encontrar cosas que son invocables pero que en realidad no son funciones).

Hay algunos otros casos, algunos de los cuales verá más adelante en este libro, donde esta distinción es importante. Puede conducir a errores inesperados y sutiles en sus aplicaciones, por lo tanto, siempre tenga cuidado de omitir los paréntesis en una situación en la que desea pasar una función y hacer que se llame repetidamente.

Autores, comentarios y entradas destacadas

Debido a que el weblog debe ser compatible con varios autores, necesita una forma de marcar el autor de cada entrada. En el último capítulo, cuando implementó palabras clave de búsqueda, vio que Django

proporcionó el campo ForeignKey para relacionar un modelo con otro (y lo traduce a una clave externa en la base de datos). La solución obvia es tener un modelo que represente a los autores y una clave externa en cada entrada que la vincule a un autor.

Este es un caso en el que Django te ayudará inmensamente. La aplicación incluida django.contrib.auth proporciona un modelo de usuario. (Esta es la cuenta de usuario que creó cuando ejecutó ning manage.py syncdb por primera vez, que se almacena en la base de datos como una instancia del modelo de usuario). Este modelo vive en el módulo django.contrib.auth.models, por lo que deberá agregar una declaración de importación en el archivo models.py del blog. Desde django.contrib.auth.models, importe Usuario y luego agregue la clave externa al modelo de Entrada:

```
autor = modelos.ForeignKey(Usuario)
```

Advertencia: ¿Por qué no especificar el usuario actual como predeterminado?

Después de tomarse la molestia de configurar slugs para que se completen automáticamente y el campo pub_date de forma predeterminada en la fecha y hora actuales, es posible que se pregunte por qué no estoy usando un valor predeterminado aquí para completar el usuario actual cuando se escribe una entrada. . La razón principal es que, en la interfaz administrativa, Django asume que otorgará acceso solo a las personas en las que confía y, por lo tanto, completarán este tipo de campo correctamente. Puede, si conoce bien la interfaz administrativa de Django, configurarlo para que el campo se llene automáticamente (y aplicar otras restricciones, como permitir que los usuarios editen solo sus propias entradas). Pero, en general, es mejor no usar la interfaz de administración para situaciones en las que no confies completamente en alguien. En su lugar, debe configurar su propia vista que pueda aplicar cualquier seguridad u otro comportamiento que desee (en el Capítulo 9 verá un ejemplo de cómo hacerlo).

Otra característica que es fácil de agregar es una forma por entrada de permitir o rechazar comentarios. Todavía no ha visto el código que realmente manejará los comentarios enviados por los usuarios (eso vendrá un poco más tarde); sin embargo, necesitará algo en el modelo de entrada que le permita verificar si se deben permitir comentarios. Así que agreguemos un campo para ello:

```
enable_comments = modelos.BooleanField(predeterminado=True)
```

Un campo booleano tiene solo dos valores posibles, verdadero o falso, y en los formularios basados en la web ser representado por una casilla de verificación. Le doy un valor predeterminado de Verdadero porque la mayoría de las personas probablemente querrán comentarios de forma predeterminada, pero el autor de una entrada podrá desmarcar la casilla en la interfaz de administración para deshabilitar los comentarios.

Mientras observa BooleanField, recuerde que una de las características de su lista es el
posibilidad de marcar las entradas como "destacadas" para que puedan destacarse para una presentación especial.
Eso también es fácil de hacer con un BooleanField:

```
destacado = modelos.BooleanField(predeterminado=False)
```

Esta vez, establezca el valor predeterminado en Falso, porque solo se deben mostrar algunas entradas específicas.

Diferentes tipos de entradas

También debe admitir las entradas que están marcadas como "borradores", que no están destinadas a mostrarse en público. Esto significa que necesitará alguna forma de registrar el estado de una entrada. Una forma sería usar otro BooleanField, con un nombre como `is_draft` o, quizás `is_public`. Luego, podría simplemente consultar las entradas con el valor apropiado, y los autores podrían marcar o desmarcar la casilla para controlar si una entrada se muestra públicamente.

Pero sería mejor tener algo que puedas extender más adelante. Si alguna vez se necesita un solo valor posible más, BooleanField no funcionará. La solución ideal sería alguna forma de especificar una lista de opciones y permitir que el usuario seleccione entre ellas; luego, si alguna vez necesita más opciones, simplemente puede agregarlas a la lista. Django proporciona una manera fácil de hacer esto a través de una opción llamada `elecciones`. Así es como lo implementará:

```
ESTADO_OPCIONES = (
    (1, 'En vivo'),
    (2, 'Borrador'),
)
estado = modelos.IntegerField(opciones=ESTADO_OPCIONES, predeterminado=1)
```

Aquí está utilizando `IntegerField`, que, como su nombre lo indica, almacena un número, un entero. en la base de datos Pero usó la opción de opciones y definió un conjunto de opciones para ella. El valor pasado a la opción de opciones debe ser una lista o una tupla, y cada elemento que contiene también debe ser una lista o una tupla con los siguientes dos elementos:

- El valor real para almacenar en la base de datos
- Un nombre legible por humanos para representar la elección

También especificó un valor predeterminado: el valor asociado con el estado En vivo, que denotará las entradas del weblog que se mostrarán en vivo en el sitio.

Puede usar opciones con cualquiera de los tipos de campo de modelo de Django, pero generalmente es más útil con `IntegerField` (donde puede usarlo para proporcionar nombres significativos para una lista de opciones numéricas) y `CharField` (donde, por ejemplo, puede usarlo para almacenar abreviaturas cortas en la base de datos, pero aún así realizar un seguimiento de las palabras o frases completas). representar).

Si ha usado otros lenguajes de programación que admiten enumeraciones, este es un concepto similar. De hecho, podrías (y probablemente deberías) hacer que se vea un poco más similar. Editar el Modelo de entrada para que empiece así:

Entrada de clase (`modelos.Modelo`):

```
ESTADO_ENVIVO = 1
BORRADOR_ESTADO = 2
ESTADO_OPCIONES = (
    (LIVE_STATUS, 'En vivo'),
    (DRAFT_STATUS, 'Borrador'),
)
```

Ahora, en lugar de codificar los valores enteros en cualquier lugar donde esté haciendo consultas para tipos específicos de entradas, puede consultar Entry.LIVE_STATUS o Entry.DRAFT_STATUS y saber que será el valor correcto. El campo de estado también se puede actualizar:

```
estado = modelos.IntegerField(opciones=ESTADO_OPCIONES, predeterminado=ESTADO_EN VIVO)
```

Y, solo para mostrar lo fácil que es agregar nuevas opciones, agreguemos una tercera opción: oculto.

Esta opción común que ofrecen los paquetes populares de registro web cubre situaciones en las que una entrada no es realmente un borrador, pero tampoco debe mostrarse públicamente. Ahora la parte relevante de la Entrada modelo se ve así:

```
ESTADO_ENVIVO = 1
BORRADOR_ESTADO = 2
ESTADO_OCULTO = 3
ESTADO_OPCIONES = (
    (LIVE_STATUS, 'En vivo'),
    (DRAFT_STATUS, 'Borrador'),
    (ESTADO_OCULTO, 'Oculto'),
)
```

Y así como puede hacer referencia a Entry.LIVE_STATUS y Entry.DRAFT_STATUS, ahora también puede consultar Entry.HIDDEN_STATUS.

Advertencia: tenga cuidado con los "números mágicos"

En general, cada vez que se encuentra escribiendo código que se basa en un valor fijo específico, como los valores de estado para la clase Entry, es una buena idea crear una variable que lo contenga y hacer referencia a esa variable. (Esto a veces se denomina constante, aunque Python no tiene una semántica especial para tal cosa).

Luego, si el valor (muchos programadores llaman a este tipo de valores "números mágicos") alguna vez necesita actualizarse, solo tendrá que hacer un único cambio en su código.

Es convencional en Python (y en muchos otros lenguajes de programación) que este tipo de constantes tengan nombres completamente en mayúsculas para indicar que tienen un significado diferente al de otras variables.

(Ya has visto que todas las configuraciones de Django usan nombres en mayúsculas; esta es la razón).

Categorización y etiquetado de entradas

Recordará que su lista de funciones requiere dos tipos de grupos de entrada: categorías (para las que ya ha sentado algunas bases en la forma del modelo de Categoría) y etiquetas. Configurar el modelo Entry para usar categorías es fácil:

```
categorías = modelos.ManyToManyField(Categoría)
```

ManyToManyField es otra forma de relacionar dos modelos entre sí. Mientras que una clave externa le permite relacionarse solo con un objeto específico de la otra clase de modelo, un ManyToManyField le permite relacionarse con tantos de ellos como desee. En la interfaz de administración, esto se representará como una lista de categorías presentadas en un elemento HTML <select multiple>.

Advertencia: cómo funcionan las relaciones de muchos a muchos

En el nivel de la base de datos, ManyToManyField en realidad está representado por una tabla de combinación separada. Cada fila de esa tabla consta de dos claves externas: una a cada lado de la relación. En este caso, la tabla se llamará `coltrane_entry_categories`, y cada fila tendrá una clave externa que apunta a la tabla de entradas y otra que apunta a la tabla de categorías.

Probablemente nunca necesitará referirse a esta tabla de unión explícitamente. Sin embargo, es una buena idea saber que está ahí y tener una idea de cómo funciona, aunque solo sea para tener un recordatorio de que seleccionar o filtrar aspectos de una relación de muchos a muchos siempre implicará unirse a la tabla adicional. (Por otro lado, las consultas basadas en una clave externa, según los parámetros exactos que esté utilizando para realizar la consulta, a veces no necesitan realizar una combinación).

El etiquetado es un poco más complicado porque, en última instancia, las etiquetas deben aplicarse a dos modelos diferentes: el modelo de entrada que está escribiendo ahora y el modelo de enlace que escribirá (en el próximo capítulo) para representar un registro de enlace. Puede definir dos modelos de etiquetas, uno para entradas y otro para enlaces, o configurar múltiples relaciones de muchos a muchos para permitir que un solo modelo de etiqueta sea suficiente para ambos, pero Django proporciona una solución más simple en forma de una *relación genérica*.

Las relaciones genéricas en realidad involucran dos tipos de campos especiales, `GenericForeignKey` y `GenericRelation`, que permiten que un modelo tenga relaciones con cualquier otro modelo instalado en su proyecto. Debido a la complejidad necesaria para que esto funcione, pueden ser un poco complicados de configurar y usar. Tienes suerte en este caso particular: hay una aplicación Django de código abierto que implementa etiquetas a través de relaciones genéricas y que ya ha hecho todo el trabajo duro.

La aplicación se llama `django-tagging` y puede descargarla desde <http://code.google.com/p/django-tagging/>. Tome una copia y descomprímala para que el módulo de etiquetado que proporciona esté en su ruta de Python, luego agregue el etiquetado a su configuración `INSTALLED_APPS`. Para agregar etiquetas a su modelo de entrada, deberá importar un tipo de campo personalizado definido en `django-tagging`, así que agregue la siguiente declaración de importación en el archivo `models.py` del weblog:

```
de tagging.fields importar TagField
```

A continuación, agregue lo siguiente al modelo de entrada:

```
etiquetas = TagField()
```

Esto puede parecer un poco extraño, pero en realidad es la forma correcta de manejar el etiquetado, por dos razones:

- Django proporciona muchos tipos de campos incorporados que puede agregar a sus modelos, pero hay de ninguna manera podría cubrir todo lo que podría necesitar para representar en una clase modelo. Entonces, además de los campos incorporados, Django también proporciona una API para escribir su propia tipos de campo. El TagField proporcionado por django-tagging es simplemente un ejemplo de esto.
- Encapsular tipos comunes de funcionalidad en aplicaciones reutilizables y "conectables" es precisamente lo que Django trata de fomentar. El hecho de que, en este caso, la aplicación haya sido escrita por otra persona y no esté incluida en `django.contrib` no debería ser un factor disuasorio. A medida que trabaje más con Django, probablemente aprovechará el gran ecosistema de aplicaciones de terceros que le evitan tener que reinventar la rueda con sus propias implementaciones de muchas funciones comunes.

Advertencia: aprender más sobre las relaciones genéricas

He omitido intencionalmente los detalles de cómo funcionan las relaciones genéricas porque son algo complejas y requieren una comprensión un poco más profunda de Django que la que has desarrollado hasta ahora. Si desea obtener más información sobre ellos, el código correspondiente se encuentra en la aplicación django.contrib.contenttypes incluida con Django, y los detalles completos están disponibles en la documentación oficial de Django en línea en <http://docs.djangoproject.com/en/dev/ref/contrib/contenttypes/>.

Escribir entradas sin escribir HTML

La última característica importante del modelo Entry es la capacidad de escribir entradas sin tener que redactarlas en HTML sin formato. Las aplicaciones de blogs más populares permiten a los usuarios escribir entradas usando una sintaxis más simple que se convertirá automáticamente a HTML según sea necesario.

Hay una serie de sistemas ampliamente utilizados que pueden tomar texto sin formato con un poco de sintaxis especial y realizar la conversión. Textile, Markdown, BBCode y reStructuredText son los más populares.

Una forma de manejar esto es con filtros de plantilla. Como viste en el último capítulo, el sistema de plantillas de Django le permite aplicar filtros a las variables en sus plantillas (como lo hizo cuando usó el filtro de escape para evitar ataques de secuencias de comandos entre sitios). Django incluye filtros de plantilla ya preparados para aplicar Textile, Markdown y reStructuredText a cualquier parte de texto en una plantilla, y esa sería una solución fácil. Desafortunadamente, también es *caro* solución. Ejecutar un convertidor de texto a HTML cada vez que muestre una entrada consumirá innecesariamente ciclos de CPU en su servidor, especialmente porque el HTML resultante será el mismo cada vez. Una mejor solución sería generar el HTML una vez, cuando la entrada se guarde en la base de datos—y luego recuperéla directamente para mostrarla.

Simplemente podría almacenar el HTML generado en el cuerpo y extraer los campos, pero eso eliminaría el beneficio de usar una sintaxis más simple para escribir entradas. Tan pronto como volviste a editar una entrada, se le presentará el HTML en lugar del texto sin formato a partir del cual se generó. Entonces, lo que realmente necesita es un par de campos separados que almacenarán el HTML y un poco de código para generarlo cada vez que se guarde una entrada. Si antes estaba preocupado por la normalización de la base de datos (el principio de que la información no debe duplicarse innecesariamente), este es un buen ejemplo de dónde es útil la desnormalización deliberada. En la mayoría de los alojamientos web para consumidores, el espacio en disco es mucho más abundante que el tiempo del procesador, por lo que aceptar un poco de redundancia en la base de datos a cambio de menos procesamiento en cada vista de página es una buena compensación.

Primero, agreguemos los campos:

```
extracto_html = modelos.TextField(editable=False, en blanco=True)
body_html = modelos.TextField(editable=False, en blanco=True)
```

Al igual que sus contrapartes de texto sin formato, ambos usan TextField. Ambos también usan la opción en blanco porque no desea que los usuarios tengan que ingresar nada en estos campos. También agregan la opción editable=False. Esto le dice a Django que no se moleste en mostrar estos campos cuando genera formularios para el modelo Entry, porque generará automáticamente el HTML para ponerlos.

Generar el HTML cada vez que se guarda una entrada es bastante fácil. El modelo base La clase de la que heredan todos los modelos de Django define un método llamado `save()`, y los modelos individuales pueden anular ese método para proporcionar un comportamiento personalizado. La única parte difícil es elegir un convertidor de texto a HTML para usar. Me gusta Markdown, así que me quedo con eso. Hay un convertidor Python Markdown de código abierto disponible, que puedes descargar en <https://sourceforge.net/proyectos/python-markdown/>. Proporciona un módulo llamado `Markdown`, que contiene la función `Markdown` para realizar la conversión de texto a HTML. Esto significa que usas una importación más declaración:

de descuento de descuento de importación

El método `save()` real dentro del modelo `Entry` es bastante corto:

```
def save(self, force_insert=False, force_update=False):
    self.cuerpo_html = markdown(self.cuerpo)
    si auto.extracto:
        self.extracto_html = rebaja(self.extracto)
    super(Entrada, self).save(force_insert, force_update)
```

Este ejecuta Markdown sobre el campo del cuerpo y almacena el HTML resultante en `body_HTML`. También realiza una conversión similar para el campo de extracto (después de verificar si se ingresó un extracto; recuerde que es opcional), y luego guarda la entrada. Tenga en cuenta que el método `save()` acepta un par de argumentos adicionales. Django los usa internamente para forzar ciertos tipos de consultas al guardar en su base de datos. (En algunos casos, es necesario forzar una consulta INSERTAR o ACTUALIZAR. Normalmente, Django simplemente elige uno u otro en función de si está guardando un objeto nuevo o actualizando un objeto existente). El método `save()` debe aceptar estos argumentos y pasárselos a la implementación base.

Advertencia: Usar `super`

Los lenguajes orientados a objetos que usan subclases generalmente necesitan proporcionar una forma de acceder a las funciones de una clase principal, incluso si esas funciones se anulan. Las convenciones para esto varían de un idioma a otro, pero en Python la práctica estándar es usar `super`, como se muestra en el código anterior.

Últimos retoques

Ahora tiene todos los campos que necesitará para manejar su lista de características para las entradas. Ha tomado un poco de tiempo cubrir la lista completa, pero si observa el modelo `Entry`, notará que solo tiene alrededor de 30 líneas de código real. Django logra empaquetar mucha funcionalidad en una cantidad muy pequeña de código. Sin embargo, antes de continuar, agreguemos algunos toques adicionales a este modelo para que sea un poco más fácil trabajar con él.

Ya ha visto con el modelo Categoría que Django intentará pluralizar el nombre del modelo cuando lo muestra en la interfaz de administración, a veces con resultados incorrectos. Así que agreguemos un nombre plural para el modelo `Entry` también:

metaclasa:

```
verbose_name_plural = "Entradas"
```

Mientras lo hace, también puede agregar un orden predeterminado para el modelo. En este caso, desea las entradas ordenadas por fecha con las entradas más recientes primero, por lo que agregará un orden opción dentro de la clase Meta interna:

```
pedido = ['-pub_date']
```

Ahora Django usará ORDER BY pub_date DESC al recuperar listas de entradas.

Avancemos también y agreguemos un método `__unicode__()` para que pueda obtener una representación de cadena simple de una entrada:

```
def __unicode__(uno mismo):
    volver self.title
```

También es una buena idea agregar texto de ayuda a la mayoría de los campos. Use su criterio para decidir qué campos lo necesitan, pero siéntase libre de comparar y tomar prestado de la versión completa de la Entrada modelo incluido en este libro.

Finalmente, agreguemos un método más: `get_absolute_url()`. Recuerde del Capítulo 2 que es una convención estándar en Django que un modelo especifique su propia URL. En este caso, devolverá una URL que incluye la fecha de publicación de la entrada y su slug:

```
def get_absolute_url(auto):
    devolver "/weblog/%s/%s/" % y
        (self.pub_date.strftime("%Y/%b/%d").lower(), self.slug)
```

Una vez más, está utilizando el formato de cadena estándar de Python. En este caso, está interpolando dos valores: la fecha de publicación de la entrada (con un poco de formato adicional proporcionado por `strftime()` método disponible en los objetos de fecha y hora de Python) y el slug de la entrada. Esta cadena de formato particular dará como resultado una URL como /weblog/2007/oct/09/my-entry/. El carácter %b en `strftime()` produce una abreviatura de tres letras del mes (que se fuerza a escribir en minúsculas con el método `lower()` para garantizar direcciones URL en minúsculas constantes). En general, prefiero esa abreviatura a una representación de mes numérico porque es un poco más legible. Si prefiere que el mes se represente numéricamente, use %m en lugar de %b.

Los modelos de blogs hasta ahora

Ya tienes dos de los tres modelos que necesitarás. Solo queda por escribir el modelo Link, del que se ocupará en el próximo capítulo. El resto de este capítulo cubrirá las vistas y URL de las entradas en el weblog. Pero antes de pasar a eso, hagamos una pausa para organizar el archivo `models.py` para que sea más fácil de entender y editar más adelante.

Mencioné anteriormente que Python tiene una guía de estilo oficial. es una buena idea seguir que cada vez que esté escribiendo código Python porque hará que su código sea más claro y más comprensible para cualquier persona que necesite leerlo (incluido usted). También hay una guía de estilo (mucho más corta) para Django, que también proporciona algunas convenciones útiles para mantener el código legible. La pauta para las clases modelo es disponerlas en este orden:

1. Cualquier constante y/o lista de opciones
2. La lista completa de campos
3. La clase Meta, si está presente

4. El método `__unicode__()`
5. El método `save()`, si se anula
6. El método `get_absolute_url()`, si está presente
7. Cualquier método personalizado adicional

Para modelos complejos, también me gusta dividir la lista de campos en grupos lógicos, con un breve comentario que explique qué es cada grupo. En general, es más fácil encontrar cosas si mantiene los nombres de los campos y las opciones en orden alfabético siempre que sea posible. Entonces, con eso en mente, aquí está el archivo `models.py` completo hasta el momento, organizado y formateado para que sea claro y legible:

```
importar fecha y hora
```

```
de django.contrib.auth.models importar Usuario de
django.db importar modelos
```

```
from markdown import markdown
from tagging.fields import TagField
```

```
categoría de clase (modelos.Modelo):
    título = modelos.CharField (max_length = 250,
                                help_text='Máximo 250 caracteres.') slug =
        modelos.SlugField(unique=True, help_text="¡ Valor sugerido generado
automáticamente a partir del título. Debe ser único")
    descripción = modelos.TextField()
```

metaclase:

```
pedido = ["título"]
verbose_name_plural = "Categorías"
```

```
def __unicode__(self):
    return self.title

def get_absolute_url(self): return "/%
categories/%s/" % self.slug
```

```
Entrada de clase (modelos.Modelo):
```

```
LIVE_STATUS = 1
DRAFT_STATUS = 2
HIDDEN_STATUS = 3
STATUS_CHOICES =
    ( (LIVE_STATUS, 'Live'),
    (DRAFT_STATUS, 'Draft'),
    (HIDDEN_STATUS, 'Hidden'),
)
```

```
# Campos
básicos. título = modelos.CharField(max_length=250,
                                      help_text="Máximo 250 caracteres")
extracto = models.TextField(blank=True,
                           help_text="Un breve resumen de la entrada. Opcional.")
cuerpo = modelos.TextField()
pub_date = modelos.DateTimeField(predeterminado=fechahora.fechahora.ahora)

# Campos para almacenar HTML
generado. extracto_html = modelos.TextFieldeditable=False, en
blanco=Verdadero) body_html = modelos.TextFieldeditable=False, en blanco=Verdadero)

# Metadatos.
autor = modelos.ForeignKey(Usuario)
enable_comments = modelos.BooleanField(Verdadero)
destacado = modelos.BooleanField(predeterminado=False)
slug = modelos.SlugField(unique_for_date='pub_date',
                          help_text="Valor sugerido generado automáticamente ѕ del título.
Debe ser único.")
status = models.IntegerField(choices=STATUS_CHOICES, default=LIVE_STATUS,
                             help_text="Solo las entradas con estado activo ѕ se
mostrarán públicamente").

# Categorización.
categorías = modelos.ManyToManyField(Categoría)
etiquetas = TagField(help_text="Separar etiquetas con espacios.")

metaclase:
pedido = ['-pub_date']
verbose_name_plural = "Entradas"

def __unicode__(self):
    return self.title

def save(self, force_insert=False, force_update=False): self.body_html
    = markdown(self.body) if self.excerpt: self.excerpt_html =
    markdown(self.excerpt) super(Entry, self).save(force_insert,
    forzar_actualización)

def get_absolute_url(auto):
    devolver "/weblog/%s/%s/" % (self.pub_date.strftime("%Y/%b/%d").lower(), self.slug)
```

Continúe y ejecute manage.py syncdb en el directorio del proyecto. Agregará el nuevo modo de entrada la tabla de el (y la tabla de unión por su relación de muchos a muchos con el modelo de Categoría), además de un par de tablas para modelos de la aplicación de etiquetado que está utilizando. Luego, use la interfaz administrativa para agregar un par de entradas de prueba al weblog; está a punto de comenzar a escribir vistas para ellos, por lo que necesitará algunas entradas con las que trabajar.

Escribir las primeras vistas

Abra el archivo views.py que creó dentro del directorio coltrane y agregue un par de import declaraciones en la parte superior para incluir cosas que necesitará para estas vistas:

```
de django.shortcuts import render_to_response  
desde coltrane.models entrada de importación
```

La primera línea que ya has visto: render_to_response() es la función de atajos que maneja la carga y renderización de una plantilla, así como la devolución de una HttpResponseRedirect. La segunda línea importa el modelo de entrada que acaba de crear, por lo que podrá recuperar entradas de la base de datos para mostrarlas.

Para su primera vista, comience con un índice simple que muestre todas las entradas "en vivo". Aquí está el código:

```
def índice_entradas(solicitud):  
    devolver render_to_response('coltrane/entry_index.html',  
                                { 'lista_de_entradas': Entrada.objetos.todos() })
```

A continuación, cree un directorio coltrane en su directorio de plantillas (el directorio que configuró para las plantillas del proyecto cms) y coloque un archivo entry_index.html en él. Agregue el siguiente HTML al archivo:

```
<html>  
    <cabeza>  
        <title>Índice de entradas</title>  
    </cabeza>  
    <cuerpo>  
        <h1>Índice de entradas</h1>  
        {% para entrada en lista_entrada %}  
            <h2>{{ entrada.título }}</h2>  
            <p>Publicado el {{ entry.pub_date|date:"F j, Y" }}</p>  
            {% si entrada.extracto_html%}  
                {{entrada.extracto_html|seguro}}  
            {% más %}  
                {{entrada.body_html|truncatewords_html:"50"|seguro}}  
            {% terminara si %}  
            <p><a href="{{ entry.get_absolute_url }}>Leer entrada completa</a></p>  
        {% endfor%}  
    </cuerpo>  
</html>
```

Tenga en cuenta que está utilizando un filtro para mostrar el extracto aquí. Recordarás que Django's El sistema de plantillas "escapa" automáticamente el contenido de las variables para evitar el cruce de sitios. ataques de secuencias de comandos. Si bien desea tener esa protección la mayor parte del tiempo, sabe que el contenido de estas variables es seguro porque provienen de datos que un usuario de confianza ingresó en la interfaz de administración. El filtro seguro le permite decirle a Django que confía en una variable en particular y que no necesita ningún escape.

Finalmente, deberá configurar una URL. Abra el archivo urls.py en el directorio cms y, en la lista de patrones de URL, agregue el siguiente patrón antes del patrón general para las páginas planas:

```
(r'^weblog/$', 'coltrane.views.entries_index'),
```

En ese momento, debería poder visitar <http://127.0.0.1:8000/weblog/>. Verás todos los las entradas que ha creado hasta ahora, que se muestran con la plantilla que acaba de crear. Hay algunas cosas que vale la pena señalar sobre la plantilla:

- Está utilizando un nuevo filtro: fecha. Es el primero que has visto que requiere una discusión, en este caso, una cadena de formato que describe cómo presentar una fecha. La sintaxis para esto es similar a la sintaxis para el método strftime(), excepto que no usa signos de porcentaje para marcar caracteres de formato. "10 de octubre de 2007" es un ejemplo de un resultado producido por esta cadena de formato.
- Está utilizando la etiqueta if para probar si hay un extracto en cada entrada. Si lo hay, entonces se muestra. Si no lo hay, se mostrarán las primeras 50 palabras del cuerpo de la entrada.
- Cuando no hay extracto, el cuerpo de la entrada se corta a través del archivo truncatewords_html filter El argumento de este filtro le dice cuántas palabras permitir. Cuando se alcanza el límite, el filtro finaliza el fragmento de texto con puntos suspensivos (...), indicando al lector que hay más texto en la entrada completa. Como su nombre lo indica, el truncatewords_html filter sabe cómo reconocer etiquetas HTML y no las cuenta como palabras. También hará un seguimiento de las etiquetas abiertas y las cerrará si corta el texto antes de una etiqueta de cierre. (Un filtro separado, palabras truncadas, simplemente corta en el número especificado de palabras y no presta atención a HTML).

Mostrar un índice de todas las entradas es un buen primer paso, pero es solo el comienzo. También deberá poder mostrar entradas individuales y deberá consultarlas en función de la información que pueda leer de la URL. En este caso, el método get_absolute_url() en el modelo Entry dará una URL que contiene la fecha de publicación (formateada) y el slug de la entrada.

Antes de escribir la vista que recupera la entrada, echemos un vistazo al patrón de URL correspondiente. Esto da una pista de cómo obtendrá esa información de la URL:

```
(r'^weblog/(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(P?<slug>[-\w]+)$',  
 'coltrane.views.entry_detail'),
```

Esto es un poco más complicado que los patrones de URL que has visto hasta ahora. La expresión regular busca varias cosas e incluye la extraña construcción ?P varias veces. Así que vamos a recorrerlo paso a paso.

En primer lugar, en la sintaxis de expresiones regulares de Python, un conjunto de paréntesis cuyo contenido comienza con ?P, seguido de un nombre entre paréntesis y un patrón, coincide con un "grupo con nombre". Que es decir, cualquier texto que coincida con una de estas partes de la URL irá a un diccionario, donde las claves son los nombres entre paréntesis y los valores son las partes del texto que coinciden. Entonces, esta URL busca cuatro grupos con nombre: año, mes, día y slug.

Los patrones reales utilizados en estos grupos nombrados son bastante simples una vez que se supera ese obstáculo. borrado:

- El \d{4} del año coincidirá con cuatro dígitos consecutivos.
- El \w{3} del mes coincidirá con tres letras consecutivas: el formateador %b que usó en el método get_absolute_url() devolverá el mes como una cadena de tres letras como "oct" o "junio".
- El \d{2} para el día coincidirá con dos dígitos consecutivos.
- El [-\w]+ para slug es algo complicado. Coincidirá con cualquier secuencia de caracteres consecutivos donde cada carácter sea una letra, un número o un guión. Este es precisamente el mismo conjunto de caracteres que permite Django en un SlugField.

Cuando una URL coincide con este patrón, Django pasará los grupos nombrados a la función de vista especificada como argumentos de palabras clave. Esto significa que la vista entry_detail recibirá argumentos de palabras clave llamados año, mes, día y slug, lo que simplificará mucho el proceso de búsqueda de la entrada. Veamos cómo funciona escribiendo la vista entry_detail:

```
def entry_detail(solicitud, año, mes, día, slug):  
    importar fecha y hora, hora  
    sello_de_fecha = hora.strptime(año+mes+día, "%Y%b%d")  
    fecha_publicación = fechahora.fecha["sello_fecha[:3]"]  
    volver render_to_response('coltrane/entry_detail.html',  
        { 'entrada': Entrada.objetos.get(pub_date__year=ÿ  
            pub_fecha.año,  
            fecha_publicación__mes=fecha_publicación.mes,  
            fecha_publicación__día=fecha_publicación.día,  
            babosa=babosa) })
```

El único bit complejo aquí es analizar la fecha. Primero usa la función strftime en el módulo de tiempo estándar de Python. Esta función toma una cadena que representa una fecha u hora, así como una cadena de formato como la que se pasó a strftime(), y analiza el resultado en una tupla de tiempo. Entonces, todo lo que necesita hacer es concatenar el año, el mes y el día y proporcionar la misma cadena de formato utilizada en el método get_absolute_url(). Luego puede pasar los tres primeros elementos de ese resultado a datetime.date para obtener un objeto de fecha.

Advertencia: comprender los argumentos de las funciones de Python

Las funciones y los métodos en Python pueden pasar y recibir argumentos de dos formas: argumentos posicionales, donde el significado está determinado por el orden en que se pasan los argumentos, y argumentos de palabras clave, cuyos nombres se incluyen directamente con los valores.

Esto se corresponde perfectamente con la lista integrada de Python y los tipos de diccionario, por lo que se proporcionan dos accesos directos para facilitar el paso de argumentos. Pasar una lista como argumento y prefijarla con un solo asterisco (*) hará que cada elemento de la lista, en orden, se use como un argumento posicional separado. Pasar un diccionario y prefijarlo con dos asteriscos (**) hará que las claves del diccionario se utilicen como nombres para argumentos de palabras clave independientes y los valores del diccionario se conviertan en los valores de estos argumentos.

Cuando una función de Python necesita aceptar conjuntos arbitrarios de argumentos opcionales, o aceptar muchos diferentes argumentos ent basados en diferentes situaciones, es común definirlo así:

```
def my_func(*args, **kwargs):
```

La función luego tendrá acceso a una lista llamada args que contiene todos los argumentos posicionales pasados a él y un diccionario llamado kwargs que contiene todos los argumentos de palabras clave que se le pasan. Luego, la función puede mirar esas variables para determinar qué debe hacer.

Así es como Django ORM puede aceptar argumentos de búsqueda basados en los campos de su modelo. Sus métodos no tienen firmas de argumentos fijos; en cambio, los métodos aceptan conjuntos arbitrarios de argumentos de palabras clave definidos como **kwargs y luego analizan esos argumentos para determinar qué campos consultar.

Finalmente, devuelve una respuesta donde el contexto de la plantilla será la entrada. la entrada es se recupera a través de los argumentos de búsqueda, que buscan entradas que coincidan con el año, el mes, el día y el slug de la URL.

Debido a que usó unique_for_date en el campo slug, esta combinación es suficiente para identificar de manera única cualquier entrada en la base de datos. El método de obtención que está utilizando aquí también es nuevo. filter devuelve un QuerySet que representa el conjunto de todos los objetos que coinciden con la consulta, pero get intenta devolver un objeto, y solo uno. (Si ningún objeto coincide con su consulta, o si más de un objeto coincide, generará una excepción).

Adelante, cree la plantilla coltrane/entry_detail.html y complétela de la forma que desee. me gusta. Luego agregue el nuevo patrón de URL al archivo urls.py del proyecto si aún no lo ha hecho, vuelva a cargar la página de índice de entradas en su navegador y haga clic en el enlace a uno de ellos para ver la nueva vista en acción.

Sin embargo, la vista no es perfecta. Si intenta una URL con el formato correcto para una entrada inexistente (digamos, /weblog/1946/sep/12/no-entry-here/), obtendrá un mensaje de error y un rastreo. La excepción es Entry.DoesNotExist, que es la forma en que Django le dice que no hay ninguna entrada que coincida con sus criterios. Sería bueno devolver un HTTP 404 "Página no encontrada" error en este caso. Puede hacerlo manualmente envolviendo la consulta en un bloque de prueba, capturando la excepción DoesNotExist y luego devolviendo una respuesta adecuada. Pero eso sería un trabajo repetitivo. Intentar recuperar algo que puede existir o no, y devolver un 404 si no existe, es algo que debe hacer mucho en el desarrollo web. Así que en lugar de hacerlo

manualmente, puede usar una función de ayuda que Django proporciona para este propósito exacto: get_object_404(). Primero, cambie la declaración de importación en la parte superior de views.py a esto:

```
desde django.shortcuts import get_object_or_404, render_to_response
```

Entonces puedes reescribir la vista así:

```
def entry_detail(solicitud, año, mes, día, slug):
    importar fecha y hora, hora
    sello_de_fecha = hora.strptime(año+mes+día, "%Y%b%d")
    fecha_publicación = fechahora.fecha(*sello_fecha[3])
    entrada = get_object_or_404(Entrada, fecha_publicación__año=fecha_publicación.año,
                                fecha_publicación__mes=fecha_publicación.mes,
                                fecha_publicación__día=fecha_publicación.día,
                                babosa = babosa)
    volver render_to_response('coltrane/entry_detail.html',
                             { 'entrada': entrada })
```

El atajo `get_object_or_404()` usará la misma búsqueda `get()` que acaba de probar, pero captura la excepción `DoesNotExist` y vuelve a generar la excepción `django.http.Http404`. El código de procesamiento HTTP de Django reconoce esta excepción y la convertirá en una respuesta HTTP 404.

Usando las vistas genéricas de Django

Hasta ahora, solo ha escrito dos vistas: un índice de entradas y una vista de detalles para ellas, pero ya parece que esto podría volverse tedioso y aburrido. Vas a necesitar vistas para las últimas entradas; para hojearlos por día, mes y año; y para navegar por categorías y etiquetas. Y

Lo que es peor, mucho será terriblemente repetitivo: hacer una consulta basada en una fecha y devolver una o más entradas como resultado. ¿No sería bueno si pudieras evitar hacer todo ese trabajo a mano?

Resulta que puedes hacerlo usando las vistas genéricas integradas de Django. Hay varios patrones de vistas extremadamente comunes que necesitan las aplicaciones web, independientemente del tipo de contenido que presenten. Entonces, Django incluye varios conjuntos de vistas, que están diseñados para funcionar con cualquier modelo y que se encargan de estas tareas comunes. En términos generales, estas tareas se dividen en cuatro grupos:

- Realización de redireccionamientos simples y simplemente representación de una plantilla basada en una URL
- Visualización de listas de objetos y objetos individuales
- Creación de archivos basados en fechas
- Creación, recuperación, actualización y eliminación (a veces llamados CRUD) de objetos

El weblog se basará en gran medida en archivos basados en fechas, así que le mostraré cómo funciona. Vaya al archivo `urls.py` y elimine el patrón que dirige a su vista `entry_detail`. Reemplázalo con esto:

```
(r'^weblog/(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?<slug>[\w-]+)/$,
'django.views.generic.date_based.object_detail', entry_info_dict),
```

Esto hace uso de una variable llamada `entry_info_dict`, que no ha definido. Así que encima de la lista de patrones de URL (pero debajo de las declaraciones de importación), defínalos así:

```
entrada_info_dict = {
    'conjunto de consultas': Entrada.objects.todos(),
    'campo_fecha': 'fecha_publicación',
}
```

Ahora, realice un cambio en la plantilla `entry_detail.html`. En cualquier lugar hay una referencia a la entrada variable (que estaba proporcionando su vista), cámbiela a objeto. También puede eliminar la vista `entry_detail` que escribió anteriormente porque ya no la necesita. A continuación, regrese y haga clic en la URL de una entrada en su navegador. Se recuperará correctamente de la base de datos y se mostrará como se especifica en su plantilla. Las URL para entradas inexistentes devolverán un 404, solo como lo hizo su vista `entry_detail` una vez que comenzó a usar `get_object_or_404()`.

¿Cómo hizo eso Django? La respuesta es bastante simple. La vista genérica quiere recibir un par de argumentos que le indiquen lo que debe hacer y, a partir de ahí, puede confiar en el hecho de que la API de la base de datos y el sistema de plantillas de Django funcionan de la misma manera en todas las situaciones.

El argumento del conjunto de consultas es la clave aquí porque (como recordará del Capítulo 3) muchos de los métodos de consulta de bases de datos de Django en realidad devuelven un tipo especial de objeto llamado `QuerySet`, que puede filtrarse y modificarse aún más antes de realizar su consulta real. En este caso, pasa la vista genérica `Entry.objects.all()`, que es un `QuerySet` que representa todas las entradas en la base de datos. También le da el argumento `date_field`, que le dice a la vista genérica qué campo en el modelo representa la fecha en la que desea filtrar. El resto de los argumentos requeridos están todos en la URL: la vista genérica recibe el año, el mes, el día y el slug de la misma manera que los recibió la vista `entry_detail`, y realiza la misma consulta de base de datos que estaba haciendo.

Pero debido a que puede reutilizar la vista genérica con diferentes conjuntos de argumentos, puede usarla para crear archivos basados en fechas para *cualquier* modelo, lo que significa que no tiene que escribir todo el código repetitivo una y otra vez. (En particular, puede reutilizar la vista genérica con un valor diferente para el argumento del conjunto de consultas y, posiblemente, `date_field` y/o `slug_field`, que se usan si el slug del modelo el campo no se llama `slug`). Todo lo que necesita hacer es configurar el patrón de URL correcto y entregarle el conjunto necesario de argumentos en un diccionario.

Todas las vistas genéricas basadas en fechas se encuentran en el módulo `django.views.generic.date_based`. Hay siete de ellos, pero necesitará usar solo cinco para la funcionalidad de su weblog:

- `object_detail`: Proporciona una vista de un objeto individual (como ya has visto).
- `archive_day`: proporciona una vista de todos los objetos en un día determinado.
- `archive_month`: proporciona una vista de todos los objetos en un mes determinado.
- `archive_year`: proporciona una lista de todos los meses que tienen objetos en ellos en un año determinado y, opcionalmente, una lista completa de todos los objetos en ese año. (Esto es opcional porque podría ser una lista extremadamente larga).
- `archive_index`: proporciona una lista de los últimos objetos.

Entonces, reescribamos el archivo `urls.py` para usar vistas genéricas para las entradas. Terminará luciendo como el siguiente código (pero por simplicidad, todavía estoy usando el proyecto `cms` que ya se creó):

```
*  
desde django.conf.urls.defaults importar desde  
django.contrib import admin  
admin.detección automática()
```

```
desde coltrane.models entrada de importación
```

```
entry_info_dict =  
    { 'conjunto de consultas':  
        Entry.objects.all(), 'date_field':  
        'pub_date', }  
  
urlpatterns = patrones(",  
    (r'^admin/', include(admin.site.urls)), (r'^search/  
    $', 'cms.search.views.search'), (r'^weblog/$',  
    'django.views.generic.date_based.archive_index', entry_info_dict),  
    (r'^weblog/(?P<año>\d{4})/$', 'django.views.generic.date_based.archive_year',  
    entry_info_dict), (r'^weblog/(?P<año>\d{4})/(?P<mes>\w{3})/$',  
    'django.views.generic.date_based.archive_month', entry_info_dict ),  
    (r'^weblog/(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$',  
    'django.views.generic.date_based.archive_day', entry_info_dict), (r'^weblog/  
    (?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/ÿ (?P<slug>[\w]+)/$',  
    'django.views.generic.date_based.object_detail', entry_info_dict), (r'',  
    include('django .contrib.flatpages.urls')),  
)  
)
```

Deberá crear plantillas para cada vista. Todas las vistas genéricas aceptan una opción argumento para especificar el nombre de una plantilla personalizada a usar (el argumento, apropiadamente, se llama `template_name`), pero por defecto usarán lo siguiente:

- `archive_index` utilizará `coltrane/entry_archive.html`.
- `archive_year` utilizará `coltrane/entry_archive_year.html`.
- `archive_month` utilizará `coltrane/entry_archive_month.html`.
- `archive_day` utilizará `coltrane/entry_archive_day.html`.
- `object_detail` utilizará `coltrane/entry_detail.html`.

Advertencia: cómo se determinan los nombres de las plantillas

Los nombres de plantilla predeterminados utilizados por las vistas genéricas de Django se basan en dos piezas de información: el modelo con el que trabaja la vista genérica y la aplicación en la que vive ese modelo. En este caso, el modelo es la clase `Entry` y la aplicación es `coltrano`. Por razones de consistencia, Django pone ambos en minúsculas al generar el nombre de la plantilla predeterminada.

La vista `object_detail`, como ya ha visto, hace que la entrada esté disponible en una variable denominada `objeto`. En las vistas de archivo diarias y mensuales, obtendrá una lista de entradas como la variable `object_list`. En ambos casos, puede personalizar estas vistas a través de un argumento opcional denominado `template_object_name`. El archivo anual, como se explicó anteriormente, de forma predeterminada le dará simplemente una lista de los meses en los que se han publicado las entradas. Esta será la variable `date_list` en la plantilla. La vista `archive_index` proporcionará a su plantilla una variable llamada `Latest`, que contendrá las últimas entradas (hasta un máximo de 15). Puedes usar el `for` tag en las plantillas apropiadas (tal como lo hizo anteriormente en su índice de entrada manual) para recorrer estas listas.

Los archivos diarios, mensuales y anuales también le dan a la plantilla una representación variable adicional indicando la fecha o rango de fechas con las que están trabajando: día, mes y año, respectivamente. como lo has hecho visto en las plantillas para las vistas de entrada que escribió a mano, puede usar el filtro de plantilla de fecha para formatear las fechas que se muestran en sus plantillas como desee.

Advertencia: Completar las plantillas de entrada

Si está interesado en ver un conjunto completo de plantillas de ejemplo (simples), consulte el código de muestra de este libro (descargable desde el sitio web de Apress). Tenga en cuenta que hacen uso de algunas características que aún no se han introducido, pero debería poder comprender la mayor parte de lo que sucede en ellos.

Desacoplar las URL

En este punto, entre los modelos que ha definido, la interfaz administrativa de Django y las vistas genéricas basadas en fechas, tiene una aplicación de weblog bastante buena. Pero ya hay un gran problema: en realidad no es reutilizable porque sus URL están "acopladas" a la configuración particular. has juntado:

- El conjunto de patrones de URL para las entradas se encuentra en el archivo `urls.py` del proyecto, lo que significa que deberá copiarlos en cualquier otro proyecto que necesite un weblog.
- Los patrones de URL y los métodos `get_absolute_url()` de los modelos de entrada y categoría (aunque aún no ha configurado vistas para categorías) están todos codificados y asumen un diseño de URL particular para el sitio. Es un diseño bastante sensato, pero algunos usuarios pueden querer una configuración diferente (por ejemplo, `/blog/` como raíz del weblog en lugar de `/weblog/`).

Arreglemos eso. En primer lugar, ya ha visto que Django ofrece la función `include()` para conectar un conjunto de URL en un punto específico de un proyecto (como lo ha hecho con la aplicación administrativa). Así que vamos a crear un conjunto reutilizable de URL que viva dentro de la aplicación de blog. Vaya a su directorio y cree un archivo llamado `urls.py`, luego copie las declaraciones de importación y los patrones de URL apropiados en él:

desde django.conf.urls.defaults importar

*

desde coltrane.models importación

```

entry_info_dict =
    { 'conjunto de consultas':
        Entry.objects.all(), 'date_field':
        'pub_date', }

urlpatterns = patrones("", (r'^$', 
    'django.views.generic.date_based.archive_index', entry_info_dict), (r'^(?P<año>\d{4})/$',
    'django .views.generic.date_based.archive_year', entry_info_dict), (r'^(?P<año>\d{4})/(?
    P<mes>\w{3})/$', 'django.views. generic.date_based.archive_month', entry_info_dict), (r'^(?
    P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', 'django.views.generic.date_based.archive_day',
    entry_info_dict), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?P<slug>[-\w]+)$',
    'django.views.generic.date_based.object_detail', entry_info_dict),
)

)

```

En el archivo urls.py del proyecto, puede eliminar la importación del modelo de entrada y la variable entry_info_dict, así como los patrones de URL para las entradas (las que comienzan con ^weblog/). Puede reemplazarlos todos con un patrón de URL:

```
(r'^weblog/', include('coltrane.urls')),
```

Tenga en cuenta que el módulo URLConf dentro de la aplicación weblog no incluye el prefijo weblog/ en ninguno de sus patrones de URL. Se basa en el proyecto para decidir dónde colocar este conjunto de URL.

También puede reducir la escritura repetitiva aquí: todas las vistas utilizadas en la URLConf del weblog comienzan con django.views.generic.date_based, que no es divertido escribir una y otra vez. Mientras tanto, hay una cadena vacía llamativa como lo primero en la lista. Esa cadena vacía no es una URL. Es un parámetro especial que le permite especificar un prefijo de vista, en caso de que todas las funciones de vista tengan rutas de módulo idénticas. Aprovechemos eso:

```

urlpatterns = patrones('django.views.generic.date_based', (r'^$', 
    'archive_index', entry_info_dict), (r'^(?P<año>\d{4})/$',
    'archive_year', entrada_info_dict), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/$',
    'archive_month', entrada_info_dict), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$',
    'archive_day', entry_info_dict), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?P<slug>[-\w]+)$',
    'objeto_detalle', entrada_info_dict),
)

)

```

Ahora Django antepondrá automáticamente django.views.generic.date_based a todos estos nombres de funciones de vista antes de intentar cargarlos, lo cual es mucho mejor.

Ahora necesita lidiar con el problema de los métodos `get_absolute_url()`. en la entrada modelo, `get_absolute_url()` devuelve una URL con `/weblog/` codificado en él, y eso no es bueno. Alguien podría conectar estas URL en una parte diferente del diseño de URL de su sitio. La solución es un par de funciones en Django: una le permite dar nombres a sus patrones de URL y la otra le permite especificar que una función como `get_absolute_url()` debería devolver un valor al buscar patrones de URL con nombres particulares.

Primero, debe realizar un cambio más en la `URLConf` del weblog:

```
urlpatterns = patrones('django.views.generic.date_based',
    (r'^$', 'archive_index', entry_info_dict, 'coltrane_entry_archive_index'),
    (r'^(?P<año>\d{4}/$', 'archive_year', entry_info_dict, 'coltrane_entry_archive_year'),
    (r'^(?P<año>\d{4})/(?P<mes>\w{3})/$', 'archive_month', entry_info_dict, 'coltrane_entry_archive_month'),
    (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', 'archive_day', entry_info_dict, 'coltrane_entry_archive_day'),
    (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?P<slug>[-\w]+)/$', 'objeto_detalle', entrada_info_dict, 'coltrane_entry_detail'),
)
```

Ha agregado un nombre a cada uno de estos patrones de URL. Los nombres se componen del nombre de su aplicación (para evitar colisiones de nombres con patrones de URL en otras aplicaciones) y una descripción de para qué es la vista.

Ahora puede reescribir el método `get_absolute_url()` en el modelo `Entry`:

```
def get_absolute_url(auto):
    return ('coltrane_entry_detail', (), {'año': self.pub_date.strftime("%Y"),
                                         'mes': self.pub_date.strftime("%B").inferior(),
                                         'día': self.pub_date.strftime("%d"),
                                         'babosa': self.babosa})
get_absolute_url = modelos.permalink(get_absolute_url)
```

El método `get_absolute_url()` ahora devuelve una tupla, cuyos elementos son los siguientes:

- El nombre del patrón de URL que desea utilizar
- Una tupla de cualquier argumento posicional que se incluirá en la URL (en este caso, no hay ninguno)
- Un diccionario de los argumentos de palabras clave que se incluirán en la URL

La última línea es un nuevo concepto: un *decorador*. Los decoradores son funciones especiales que no hacen nada por su cuenta, pero se puede utilizar para cambiar el comportamiento de otras funciones. El enlace permanente el decorador que está usando aquí (que vive en `django.db.models`) en realidad reescribirá el `get_absolute_url()` para realizar una búsqueda inversa de URL. Escaneará la `URLConf` del proyecto para buscar el patrón de URL con el nombre especificado, luego usará la expresión regular de ese patrón para crear la cadena de URL correcta y completar los valores adecuados para cualquier argumento que deba incrustarse. en la URL.

Según la URLConf que configuró para este proyecto, el decorador de enlaces permanentes encontrará el prefijo / weblog/ y seguirá el comando include() hasta coltrane.urls, donde encontrará el patrón denominado coltrane_entry_detail y completará la expresión regular con los valores correctos. Para una entrada publicada el 10 de octubre de 2007 con el slug test-entry, este proceso generará la URL /weblog/2007/oct/10/test-entry/. Si cambiara la URLConf raíz para incluir las URL de weblog en blogs/, generaría /blogs/2007/oct/10/test-entry/.

Advertencia: sintaxis de Python Decorator

También es posible usar una sintaxis ligeramente diferente para los decoradores en Python. Puede colocarlos directamente sobre la definición de la función o el método y anteponerles un símbolo de arroba (@). En este caso, eso habría significado colocar @models permalink directamente encima de esta línea:

```
def get_absolute_url(auto):
```

Esta sintaxis se introdujo en Python 2.4, por lo que si usa 2.4 o una versión posterior, funcionará. Sin embargo, generalmente lo evito en mis aplicaciones Django, porque Django también funciona con Python 2.3, donde la única sintaxis disponible es llamar al decorador debajo de la definición de función o método. En general, es una buena idea escribir su código para que sea compatible con la mayor cantidad posible de versiones de Python.

Y ahora ha desvinculado por completo las URL de entrada del proyecto y de cualquier suposiciones sobre diseños de URL de sitios particulares. Estas URL se pueden conectar a cualquier proyecto en cualquier punto de su jerarquía de URL, y entre include() y el decorador permalink(), las URL generadas siempre serán correctas.

Mirando hacia el futuro

Una vez más, ha logrado mucho sin escribir mucho código real. El mayor obstáculo en la aplicación de blogs hasta ahora ha sido simplemente manejar el diseño de un primer "real"

La aplicación Django y todas las opciones variadas que ofrece Django para reducir el código tedioso y repetitivo. Y es lo suficientemente flexible como para ser reutilizado en cualquier proyecto en el que necesite un Blog.

En este punto, tiene una gran cantidad de los conceptos más importantes de Django bajo su cinturón: el modelo básico/URL/vista/arquitectura de plantilla, la sintaxis de cada componente y los principios generales de desacoplamiento y reutilización de código (a veces llamado DRY, abreviatura de "Don't Repítase", una directriz de desarrollo de software que dice que, siempre que sea posible, debe tener una, y solo una, versión autorizada de un dato o funcionalidad). Es posible que desee hacer una pausa aquí y revisar lo que ha escrito hasta ahora porque comenzará a acelerar el ritmo y a escribir código mucho más rápido. Una vez que se sienta cómodo con los conceptos y características presentados hasta este punto, continúe con el próximo capítulo. Allí terminará los modelos de weblog escribiendo la clase Link y luego completará el resto de las vistas básicas. Después,

profundizará un poco más en el sistema de plantillas de Django y en algunas características más avanzadas.

Capítulo 5



Expandiendo el Weblog

Hasta ahora ha escrito dos modelos para su aplicación de weblog, Categoría y Entrada, y ha configurado crear vistas que mostrarán las entradas en el weblog. Todavía tiene trabajo por hacer para configurar todas las diferentes vistas que desea para las entradas; sin embargo, antes de hacerlo, regresemos y terminemos los modelos de datos del weblog agregando la clase de modelo final.

Escribir el modelo de enlace

Así como los campos en el modelo de entrada lógicamente se dividieron en grupos según cómo se usarían, el modelo que usará para representar enlaces, una clase llamada Enlace, necesitará campos para varios propósitos diferentes:

- **Campos principales que representan el enlace:** un título, una descripción y, por supuesto, la URL a la que enlazar.
- **Metadatos:** Esto incluye la fecha en que se publicó el enlace y el nombre del usuario que lo publicó, así como si permitir comentarios para el enlace.
- **Categorización:** logrará esto con etiquetas.
- **Integración con un servicio externo de publicación de enlaces:** en este caso, utilizará Delicious (<http://delicious.com>).

Comencemos con los campos centrales básicos para el modelo (al igual que con los modelos Categoría y Entrada, este código va en `coltrane/models.py`). Al igual que antes, lo construirá de forma incremental (así que no ejecute `syncdb` todavía):

Enlace de clase (`modelos.Modelo`):

```
título = modelos.CharField(max_length=250)
descripción = modelos.TextField(en blanco=True)
description_html = modelos.TextField(en blanco=True)
url = modelos.URLField(único=True)
```

Hay un nuevo tipo de campo aquí: `URLField`. Como sugiere el nombre, está destinado a almacenar una URL.

En la base de datos, será simplemente una columna VARCHAR como la mayoría de los otros tipos de campos basados en texto, pero en los formularios generados automáticamente (como los que se muestran en la interfaz de administración), se realizará una validación adicional para este campo:

- El valor ingresado se comparará con la sintaxis de una URL HTTP, por ejemplo, debe comenzar con `http://` o `https://`.
- No podrá ingresar una URL inexistente o "rota". De forma predeterminada, Django emitirá una solicitud HTTP a la URL durante la validación y se negará a aceptar la URL si devuelve un estado de error HTTP (como "404 No encontrado" o "500 Error interno del servidor"). Puede deshabilitar esta verificación usando el argumento de palabra clave `verificar_existe=False` al configurar `URLField`.

Además, tenga en cuenta el argumento de la palabra clave `unique=True`. Como se mencionó en el Capítulo 4, esto generará una restricción ÚNICA en el nivel de la base de datos y Django también la aplicará. Este argumento de palabra clave evitará que los usuarios publiquen el mismo enlace repetidamente.

Finalmente, la descripción del enlace es opcional; es posible que no siempre desee ingresar una. y eso usa dos campos, tal como lo hizo el extracto y el cuerpo en el modelo `Entry`. En un momento, agregará un método `save()` personalizado para aplicar la conversión de texto a HTML.

Ya vio en el modelo `Entry` cómo agregar una clave externa a un usuario para representar el persona que publicó una entrada, y puede hacer lo mismo con el modelo `Enlace`:

```
posted_by = models.ForeignKey (Usuario)
```

Del mismo modo, puede agregar una fecha de publicación y un slug:

```
pub_date = modelos.DateTimeField(predeterminado=datetime.datetime.now)  
slug = modelos.SlugField(unique_for_date='pub_date')
```

Puede agregar etiquetas tal como lo hizo con el modelo de Entrada:

```
etiquetas = TagField()
```

y dos campos booleanos: uno para determinar si se deben permitir comentarios y otro para determinar si publicar el enlace a un servicio externo. En ambos casos, los establecerá de forma predeterminada en `True`:

```
enable_comments = modelos.BooleanField(predeterminado=Verdadero)  
post_elsewhere = modelos.BooleanField('Publicar en Delicious', predeterminado=Verdadero)
```

Uso Delicious como mi servicio de agregación de enlaces, así que lo puse en la etiqueta del campo; pero más adelante, si decide que desea utilizar un servicio diferente, no dude en cambiarlo.

Cuando escriba el método `save()` personalizado para este modelo, verá cómo enviar el enlace al servicio externo.

Finalmente, agreguemos un par de campos más para obtener un poco de metadatos adicionales. es bastante común para tomar nota de dónde vio un enlace útil, y podría usar la descripción para eso (es decir, podría ingresar "Enlace encontrado a través de Slashdot"), pero a menudo es más útil para modelar eso directamente. Por lo tanto, agregará dos campos más: uno para almacenar el nombre de la persona o el sitio que señaló el enlace y otro para almacenar la URL donde vio el enlace. Hará que ambos sean opcionales para que no tengan que completarse cuando no sean aplicables:

```
via_name = models.CharField('Via', max_length=250, blank=True,  
                           help_text='El nombre de la persona cuyo sitio usted  
vio el enlace en. Opcional.')  
via_url = models.URLField('A través de URL', en blanco=Verdadero,  
                           help_text='La URL del sitio donde vio el  
Enlace. Opcional.')
```

También puede agregar un orden predeterminado por el campo pub_date:

metaclasa:

```
pedido = ['-pub_date']
```

y un método __unicode__() para que cada enlace tenga una representación de cadena útil. Al igual que con las entradas, usará el campo de título para esto:

```
def __unicode__(uno mismo):
    volver self.title
```

Y finalmente, agregará un método save() personalizado, que debe hacer dos cosas:

- Si se completó algo para el campo de descripción, save() debería ejecutar Markdown sobre él y almacene el resultado en el campo description_html.
- Si el campo post_elsewhere es Verdadero y esta es la primera vez que se guarda el vínculo, el El método save() también debería publicarlo en Delicious.

La primera parte es fácil, y puede manejarla de la misma manera que manejó la extracto opcional sobre las entradas:

```
def guardar(auto):
    si auto.descripcion:
        self.description_html = markdown(self.description)
    super(Enlace, auto).guardar()
```

La segunda parte es un poco más complicada. Necesitará alguna forma de comunicarse con el público API de publicación de enlaces que proporciona Delicious. Afortunadamente, puedes hacer esto usando una fuente abierta módulo de Python llamado pydelicious; descárguelo de <http://code.google.com/p/pydelicious/>.

Advertencia: Instalación de módulos Python de terceros

Python proporciona un mecanismo para empaquetar e instalar módulos para que pueda distribuirlos y reutilizarlos fácilmente. La mayoría de los módulos Python de terceros y las aplicaciones Django que encontrará funcionarán de esta manera, por lo que podrá descargar un paquete, abrirlo y, dentro del directorio resultante, escribir python setup.py install para instalarlo.

El módulo pydelicious en realidad implementa bastantes métodos útiles de la API de Delicious, pero el único que necesita aquí es el de publicar un enlace. Esto se implementa en pydelicious como una función llamada add(), que toma cinco argumentos:

- El nombre de usuario de la cuenta para publicar el enlace
- La contraseña de la cuenta para publicar el enlace
- La URL del enlace
- El título del enlace
- Las etiquetas para el enlace, como una cadena con etiquetas separadas por espacios

Sería tentador codificar simplemente la información de su propia cuenta para las partes de nombre de usuario y contraseña, pero eso causaría problemas en el futuro: no podría compartir la aplicación de blog con otros (porque obtendrían su nombre de usuario y contraseña en el código), y no podrá reutilizar la aplicación con varios blogs que publican en diferentes cuentas.

Una buena solución a ese problema es solicitar que se coloque un nombre de usuario y una contraseña en el archivo de configuración de Django. De esta forma, cada sitio que utilice la aplicación de blog puede especificar un nombre de usuario y una contraseña diferentes. Y no tendrá que preocuparse por la seguridad porque no distribuirá su archivo de configuración de todos modos (tiene otra información confidencial, como las credenciales de su base de datos). Llamará a estas configuraciones `DELICIOUS_USER` y `DELICIOUS_PASSWORD` para indicar claramente lo que significan.

Así que agregue una línea en la parte superior de `models.py` para importar la configuración de Django que está usando:

desde la configuración de importación de `django.conf`

Advertencia: Acceso a la configuración

Puede acceder a su archivo de configuración de Django de la misma manera que accedería a cualquier otro módulo de Python, importándolo desde su ubicación en su computadora (usando `import cms.settings`, por ejemplo). Sin embargo, generalmente es una mejor idea usar desde la configuración de importación de `django.conf`. Esto habilitará una función en Django que proporciona automáticamente valores predeterminados para muchas configuraciones si no las ha completado.

Si se siente raro inventar nuevos ajustes, no se preocupe. Definir y hacer uso de configuraciones adicionales es una práctica perfectamente normal y recomendada para las aplicaciones Django (siempre y cuando cada aplicación documente cualquier configuración adicional que requiera). Además, mantener toda la configuración para un proyecto de Django en un solo lugar, el archivo de configuración, hace que sea más fácil de entender y administrar un proyecto que tener la configuración de "Django" y la configuración de "aplicación" repartidas en varias ubicaciones.

Solo hay una cosa más que necesito cubrir antes de que puedas escribir el `save()` terminado método. Django representará la URL, el título y las etiquetas como cadenas Unicode. Por lo general, la práctica de Django de garantizar que las cadenas almacenadas en los campos del modelo sean Unicode es algo bueno: elimina muchos de los dolores de cabeza de lidiar con codificaciones de caracteres. Pero en este caso, también es un poco problemático porque las cadenas Unicode no se traducen directamente en una serie de bytes binarios, por lo que no son adecuadas para enviarse "por cable" en una llamada API basada en web.

Por lo tanto, deberá convertir los valores de estos campos en cadenas basadas en bytes antes de pasarlo a la API de Delicious. Django proporciona una función auxiliar, `django.utils.encoding.smart_str()`, que hará esto. En muchos casos, probablemente también podría usar el `str()` incorporado de Python funcionar y salirse con la suya. Sin embargo, `smart_str()` de Django puede manejar algunas situaciones que `str()` no puede, y también por defecto codifica el resultado en UTF-8 en lugar de ASCII (que es el predeterminado para la mayoría de las instalaciones de Python).

Así que ahora puedes agregar el código apropiado al método `save()`, y listo:

```

def save(self): if
    self.description:
        self.description_html = markdown(self.description) if not self.id and
    self.post_elsewhere: import pydelicious from django.utils.encoding import
        smart_str pydelicious.add(settings.DELICIOUS_USER,
        settings.DELICIOUS_PASSWORD, smart_str(self.url),
        smart_str(self.title), smart_str(self.tags))

super(Enlace, auto).guardar()

```

Es importante tener en cuenta if not self.id y self.post_elsewhere porque resuelven toda la lógica para determinar si el enlace debe publicarse externamente. La verificación de self.id es la clave porque le dice si el enlace se está guardando por primera vez (no sería útil volver a publicar el enlace una y otra vez cada vez que se guarda). Recuerda que si no especificas una clave principal para un modelo, Django agrega una automáticamente en un campo llamado id. Entonces, si ese campo no tiene un valor, aún no debe haberse guardado en la base de datos.

Como toque final al modelo Link, agregará un método get_absolute_url(). Tal como lo hizo con el modelo Entry, utilizará el decorador de enlaces permanentes para permitirle realizar una búsqueda inversa en el módulo URLConf del proyecto:

```

def get_absolute_url(auto):
    return ('coltrane_link_detail', (), { 'año': self.pub_date.strftime('%Y'), 'mes': self.pub_date.strftime('%b')})
    .más bajo(),
    'día': self.pub_date.strftime('%d'), 'slug': self.slug })
get_absolute_url = models.permalink(get_absolute_url)

```

Aún no ha definido ningún patrón de URL para enlaces, por lo que no hay un patrón denominado coltrane_link_detail. Lo agregarás en un momento.

En este punto, tiene el modelo de enlace completamente escrito y puede ejecutar manage.py syncdb para instalar su tabla de base de datos. Como referencia, aquí está la definición completa del modelo con los campos perfectamente organizados y algún texto de ayuda adicional mezclado, como vio para el modelo de Entrada en el Capítulo 4:

```

Enlace de clase (modelos.Modelo):
# Metadatos. enable_comments
= models.BooleanField(default=True) post_elsewhere =
models.BooleanField('Publicar en del.icio.us', default=True, help_text='Si está
marcado, este enlace ѕ se
publicará tanto en su weblog como en su del
cuenta .icio.us.') posted_by = models.ForeignKey(User) pub_date = models.DateTimeField(default=datetime.datetime.now)
slug = models.SlugField( unique_for_date='pub_date', help_text='Debe ser único para la fecha de
publicación.')

```

```
título = modelos.CharField(max_length=250)

# Los bits de enlace reales.

descripción = modelos.TextField(en blanco=Verdadero)
descripción_html = modelos.TextField(editable=False, en blanco=Verdadero)
via_name = modelos.CharField('Vía', max_length=250, en blanco=Verdadero,
                             help_text='El nombre de la persona cuyo sitio ustedÿ
vio el enlace en. Opcional.')
via_url =
    models.URLField('A través de URL', verify_exists=False, blank=True, help_text='La
URL del sitio que vioÿ
el enlace. Opcional.')
tags
    = TagField()
url =
    models.URLField('URL', unique=True)

metaclase:
pedido = ['-pub_date']

def __unicode__(self):
    return self.title

def save(self): si
    no self.id y self.post_elsewhere: import pydelicious
        pydelicious.add(settings.DELICIOUS_USER,
                        settings.DELICIOUS_PASSWORD,
                        smart_str(self.url), smart_str(self.title),
                        smart_str(self.tags) ) if
                        self.description: self.description_html
                        = markdown(self.description)
    super(Link, self).save()

def get_absolute_url(self): return
    ('coltrane_link_detail', (), {
        'año': self.pub_date.strftime('%Y'),
        'mes': self.pub_date.strftime('%b').lower(), 'day':
        self.pub_date.strftime('%d'), 'slug': self.slug })

get_absolute_url = modelospermalink(get_absolute_url)
```

Además, siga adelante y habilite la interfaz administrativa para el modelo Link. Vea si puede averiguar por sí mismo cómo hacer esto, configurando los slugs de prepopulación automática como lo ha hecho anteriormente. Si se queda perplejo, consulte el código fuente de este capítulo (descargable desde el área Código fuente/Descarga del sitio web de Apress).

Vistas para el modelo de enlace

Viste en el último capítulo que las vistas genéricas incorporadas de Django proporcionan una manera fácil de manejar tipos comunes de vistas. Al pasar los parámetros correctos a una vista genérica, a menudo puede ahorrar bastante tiempo y código cuando todo lo que desea es, por ejemplo, mostrar una lista de objetos modelo o un detalle de un solo objeto.

La situación no es diferente con el modelo Link. Desea tener una vista detallada de cada vínculo individual y un archivo basado en fechas para examinar todos los vínculos de la base de datos. Así que abra el archivo urls.py dentro de la aplicación weblog y cambie esta línea

```
desde coltrane.models entrada de importación
```

```
leer
```

```
desde coltrane.models entrada de importación, enlace
```

Luego, al igual que con el modelo Entry, deberá definir un diccionario con argumentos para las vistas genéricas:

```
link_info_dict =  
    { 'conjunto de consultas':  
        Link.objects.all(), 'date_field':  
        'pub_date', }
```

A continuación, puede agregar un nuevo conjunto de patrones de URL a la lista existente:

```
(r'^enlaces/$',  
     'archive_index', link_info_dict,  
     'coltrane_link_archive_index'),  
(r'^enlaces/(?P<año>\d{4})/$',  
     'archivo_año', link_info_dict,  
     'coltrane_enlace_archivo_año'),  
(r'^enlaces/(?P<año>\d{4})/(?P<mes>\w{3})/$',  
     'archive_month', link_info_dict, 'coltrane_link_archive_month'),  
(r'^links/(?P<year>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$',  
     'archive_day', link_info_dict, 'coltrane_link_archive_day'), (r'^links/(?P<year>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/ÿ (?P<slug>[-\w]+)/$', 'objeto_detalle',  
     enlace_info_dict, 'coltrane_enlace_detalle'),
```

Cuando los usó para el modelo de entrada, los nombres de plantilla para las vistas fueron (en orden):

- coltrane/entry_archive.html
- coltrane/entry_archive_year.html
- coltrane/entry_archive_month.html

- coltrane/entry_archive_day.html
- coltrane/entry_detail.html

Ahora que también está utilizando vistas genéricas para el modelo de enlace, necesitará un conjunto de plantillas ligeramente diferente:

- coltrane/link_archive.html
- coltrane/link_archive_year.html
- coltrane/link_archive_month.html
- coltrane/link_archive_day.html
- coltrane/enlace_detalle.html

Los nombres de variables disponibles en estas plantillas serán los mismos que antes, por lo que debe ser capaz de trabajar con ellos fácilmente. Por ejemplo, en la vista detallada, el objeto Enlace estará disponible en una variable denominada objeto. Si lo desea, continúe y configure las plantillas por ahora, pero en el próximo capítulo analizará más detalladamente el sistema de plantillas de Django y cómo puede reducir en gran medida la cantidad de trabajo repetitivo involucrado en la escritura de plantillas.

Configuración de vistas para categorías

En este punto, tiene configuradas la mayoría de las características del weblog. Los modelos están escritos y, gracias a las vistas genéricas, tiene una manera fácil de ver archivos de entradas y enlaces basados en fechas, así como objetos de entrada y enlace individuales en sus propias páginas de detalles. Pero todavía hay dos grupos de vistas que necesita manejar:

- Vistas para navegar por las entradas por categorías
- Vistas para navegar por entradas y enlaces por etiquetas

Comencemos con las categorías. Necesitará dos vistas para las categorías: una para mostrar una lista de todas las categorías en uso y otra para mostrar la lista de entradas en una categoría específica. tan abierto abra el archivo views.py en la aplicación weblog y agregue lo siguiente en la parte superior, después de las declaraciones de importación existentes:

de coltrane.models categoría de importación

Escribir la vista que muestra una lista de categorías es bastante fácil. Todo lo que tienes que hacer es recuperar la lista de la base de datos y entréguela a la plantilla. En aras de la coherencia con cómo hacen las cosas las vistas genéricas, pasará la lista de categorías a la plantilla en una variable llamada object_list, y usará el nombre de la plantilla coltrane/category_list.html (por razones que pronto quedarán claras):

```
def lista_categoria(solicitud):  
    volver render_to_response('coltrane/category_list.html',  
        { 'lista_objetos': Categoría.objetos.todos() })
```

Mostrar una lista de entradas en una categoría particular es solo un poco más complejo. Debido a que cada categoría tiene un SlugField adecuado para usar en una URL, asumirá que la URL coincide con un argumento llamado slug. Lo usará para buscar la categoría (usando `get_object_or_404()` para devolver un error "404 Not Found" si no hay una categoría que coincida con el slug proporcionado en la URL).

Y una vez que tenga el objeto Categoría, acceder a la lista de entradas es fácil. Django sabe sobre la relación establecida por ManyToManyField en Entry, y garantizará que cada categoría tenga un atributo llamado `entry_set`, que se puede usar para acceder a las entradas que se le han asignado. Este atributo se comporta como el atributo de objetos en la entrada modelo. Tiene todos los mismos métodos (`all()` y `filter()`, por ejemplo) que `Entry.objects`, excepto que solo devuelve las entradas asignadas a esa categoría en particular.

A continuación se muestra la vista, utilizando `coltrane/category_detail.html` como nombre de la plantilla, y nuevamente, usando el nombre `object_list` para la variable que contiene la lista de entradas:

```
def category_detail(solicitud, slug):
    categoría = get_object_or_404(Categoría, slug=slug)
    volver render_to_response('coltrane/category_detail.html',
        { 'lista_objetos': categoría.conjunto_entrada.todo(),
          'categoría': categoría })
```

A continuación, puede agregar un par de patrones más en el archivo `urls.py` de la aplicación de weblog. El único problema aquí es que ya especificó un prefijo de `django.views.generic.date_based` para los patrones de URL allí, y estas dos vistas viven en `coltrane.views`. Puede eliminar el prefijo y agregar manualmente `django.views.generic.date_based` a todas esas vistas nuevamente, pero hay una manera más fácil de resolver este problema. Observe cómo comienza la lista de patrones. como esto:

```
urlpatterns = patrones('django.views.generic.date_based',
```

Esta línea llama a una función llamada `patterns()`, que se importa de `django.conf.urls`. predeterminados (como puede ver si observa la parte superior del archivo). La función analiza cada patrón que se le pasa y luego devuelve una lista de patrones de URL en un formato estandarizado con el que Django puede trabajar. Esa lista termina en una variable llamada `urlpatterns`. Debido a que el resultado final es solo una lista ordinaria de Python, puede continuar trabajando con ella. En este caso, vas a tomar Aproveche el hecho de que puede agregar listas de Python usando el operador de signo más (+). Simplemente llame a `patterns()` por segunda vez y agregue el resultado a la variable `urlpatterns` que ya tiene. Sin embargo, esta vez usará un prefijo diferente: `coltrane.views`.

Así que agregue el siguiente código en la parte inferior de `urls.py` (en realidad está usando `+=` en lugar de solo `+` porque significa un código un poco más corto):

```
urlpatterns += patrones('coltrane.views',
    (r'^categorías/$', 'category_list'),
    (r'^categorías/(?P<slug>[-\w]+)/$', 'categoría_detalle'),
)
```

Ahora tiene vistas y direcciones URL configuradas. Tratará con plantillas para ellos en el próximo capítulo. Por ahora, concentrémonos en algunas formas en que puede mejorar lo que tiene aquí.

Uso de vistas genéricas (otra vez)

Esto es realmente más trabajo del que necesitas hacer. Ya ha visto cómo las vistas genéricas facilitan la configuración de archivos basados en fechas, y también son bastante útiles para manejar listas de objetos no basadas en fechas. El módulo `django.views.generic.list_detail` contiene dos vistas, que producen resultados no basados en fechas:

- `object_list` simplemente toma el argumento del conjunto de consultas que ya ha visto y obtiene una lista de objetos
- `object_detail` (que vale la pena mencionar, aunque no lo usará en esta aplicación) toma el argumento `queryset` y un argumento `object_id` que corresponde a la clave principal de un objeto o una combinación de `slug_field` y `slug` argumentos y devuelve una vista detallada de un solo objeto.

Por lo tanto, en realidad no necesita la vista `category_list`. La vista genérica `object_list` hará lo mismo. Vuelva al archivo `urls.py` y realice un cambio más en la declaración de importación que extrae los modelos de `weblog`, cambiarlo de

desde `coltrane.models` entrada de importación, enlace

a

de `coltrane.models` importar Categoría, Entrada, Enlace

Luego regrese al conjunto adicional de patrones que acaba de agregar para las categorías y cámbielo a esto:

```
urlpatrones += patrones("",
    (r'^categorías/$',,
        'django.views.generic.list_detail.object_list',
        { 'conjunto de consultas': Categoría.objetos.todos() }),
    (r'^categorías/(?P<slug>[-w]+)$',
        'coltrane.views.category_detail'),
)
```

La vista genérica `object_list`, de forma predeterminada, utiliza un nombre de plantilla de `coltrane/category_list.html` (por lo que fue una buena idea elegir eso desde el principio para la `categoría_list` original view) y pasa la lista de categorías en una variable llamada `object_list`. Esto tiene el mismo efecto que su vista escrita manualmente (que ahora puede eliminar).

Quizás se pregunte en este punto si es posible usar una vista genérica para la lista de entradas en una categoría. No parece que haya ninguna forma de decirle a la vista genérica que también filtre las entradas en función de las categorías a las que pertenecen, porque el filtrado exacto que debe realizarse variará según la categoría que esté viendo.

Pero hay *una* manera de usar una vista genérica aquí. El truco es recordar que, en Django, una vista es simplemente una función de Python que acepta un objeto `HttpRequest` (y potencialmente un conjunto de argumentos adicionales) y devuelve un objeto `HttpResponse`. Esto significa que es posible escribir una vista que importe y llame a otra vista, así como que devuelva su respuesta.

Si eso suena confuso, así es como podría escribir una variación de la `categoría_detail` vista que usa la vista genérica `object_list`:

```
de django.views.generic.list_detail importar object_list

def category_detail(solicitud, slug):
    categoría = get_object_or_404(Categoría, slug=slug)
    volver object_list(solicitud, queryset=categoría.entry_set.all(),
        extra_context={ 'categoría': categoría }))
```

Analicemos lo que está sucediendo aquí:

1. Importa la vista genérica object_list desde django.views.generic.list_detail
(las otras cosas que usará, como el modelo de Categoría y get_object_or_404()
acceso directo, ya se han importado dentro del archivo views.py).
2. Defina su función de vista, category_detail, para aceptar la solicitud HTTP y un slug.
3. Utiliza get_object_or_404() para obtener la categoría correspondiente al slug
argumento o devolver un error "404 No encontrado".
4. Usted llama a la vista genérica object_list directamente, pasa la solicitud HTTP y establece su argumento
queryset en el conjunto de entradas en esta categoría específica, y devuelve la respuesta directamente.
5. Pasas un argumento adicional, extra_context. La mayoría de las vistas genéricas de Django aceptan esto
argumento, que le permite especificar variables y valores adicionales para ponerlos a disposición de la
plantilla. En este caso, está agregando el objeto Categoría.

En efecto, está "envolviendo" la vista genérica dentro de otra función de vista que realiza un trabajo preliminar
para filtrar el QuerySet eventual que utilizará.

Dado lo simple que era la vista original de detalle de categoría, esto puede parecer una forma extraña de
hacer las cosas y, para este caso específico, probablemente no valga la pena envolver una vista genérica. Pero
este es un patrón *extremadamente* poderoso para mantener en el fondo de tu mente. Habrá muchas ocasiones en
las que necesitará algo como una vista genérica, pero con un poco de filtrado o procesamiento adicional. El uso de
este tipo de contenedor puede, en casos más complejos, a menudo conducir a una reducción significativa en la
cantidad de código que tiene que escribir a mano.

Vistas para etiquetas

Todavía necesita un conjunto de vistas para manejar las entradas de navegación y los enlaces por sus etiquetas.
Sin embargo, resulta que no tienes que escribirlos. La aplicación de etiquetado que está utilizando proporciona un
modelo llamado Tag para representar las etiquetas, y simplemente puede usar la vista genérica object_list para
mostrar una lista de ellas.

Agregue una declaración de importación más en la parte superior del archivo urls.py:

```
de tagging.models importar etiqueta
```

Y agregue otro conjunto de patrones de URL en la parte inferior:

```
urlpatrones += patrones(
    (r'^etiquetas/$', 'django.views.generic.list_detail.object_list',
        { 'conjunto de consultas': Tag.objects.all() }),
)
```

La aplicación de etiquetado también proporciona una vista, escrita en el mismo estilo general que las vistas genéricas integradas de Django, para mostrar todos los objetos de un modelo en particular que tienen una etiqueta en particular. Esta vista es `tagging.views.tagged_object_list`, y necesita darle tres argumentos:

- `queryset_or_model`: Esta será la clase modelo o QuerySet cuyos objetos desea vista, y lo pasará directamente en el patrón de URL.
- `etiqueta`: puede ser un objeto Etiqueta o el nombre de una etiqueta, y configurará el patrón para que se lea de la URL.
- `template_name`: Este es el nombre de la plantilla que utilizará la vista. Si no se especifica, se establecerá de forma predeterminada en `tagging/tag_list.html`, por lo que utilizará algo descriptivo para facilitar el seguimiento de lo que sucede.

Entonces, todo lo que necesita hacer es agregar dos patrones más: uno para buscar entradas por etiqueta y otro para navegar por enlaces por etiqueta. Comienza con el patrón que ya configuró para la lista de etiquetas:

```
urlpatrones += patrones",
    (r'^etiquetas/$',
     'django.views.generic.list_detail.object_list',
     { 'conjunto de consultas': Tag.objects.all() }),
)
```

y luego agregue los dos nuevos patrones:

```
urlpatrones += patrones",
    (r'^etiquetas/$',
     'django.views.generic.list_detail.object_list',
     { 'conjunto de consultas': Tag.objects.all() }),
    (r'^etiquetas/entradas/(?P<etiqueta>[-\w]+)/$',
     'tagging.views.tagged_object_list',
     { 'queryset_or_model': Entrada,
       'template_name': 'coltrane/entries_by_tag.html' }),
    (r'^etiquetas/enlaces/(?P<etiqueta>[-\w]+)/$',
     'tagging.views.tagged_object_list',
     { 'queryset_or_model': Enlace,
       'template_name': 'coltrane/links_by_tag.html' }),
)
```

La vista `tagged_object_list` es en realidad un envoltorio alrededor de la vista genérica `object_list`, como el que viste anteriormente para la vista `category_detail` pero un poco más complejo. (Este es un caso en el que envolver una vista genérica *reduce* significativamente la cantidad de código). Debido a esto, la vista `tagged_object_list` proporcionará la lista de objetos a la plantilla en una variable llamada `object_list`, haciéndola agradable y consistente con todas sus otras vistas.

Limpieza del módulo URLConf

En este punto, el archivo `urls.py` en la aplicación de blogs comienza a volverse difícil de manejar. Actualmente, se parece a lo siguiente:

```
desde django.conf.urls.defaults importar      *

from coltrane.models import Categoría, entrada, enlace
from tagging.models import Tag

entry_info_dict =
    { 'conjunto de consultas':
        Entry.objects.all(), 'date_field':
        'pub_date', }

link_info_dict =
    { 'conjunto de consultas':
        Link.objects.all(), 'date_field':
        'pub_date', }

urlpatterns = patrones('django.views.generic.date_based', (r'^$',
    'archive_index', entry_info_dict, 'coltrane_entry_archive_index'), (r'^(?P<año>\d{4})/$',
    'archive_year', entry_info_dict, 'coltrane_entry_archive_year'), (r'^(?P<año>\d{4})/(
    ?P<mes>\w{3})/$', 'archive_month', entry_info_dict, 'coltrane_entry_archive_month'),
    (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', 'archive_day', entry_info_dict,
    'coltrane_entry_archive_day'), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(
    P<slug>[-\w]+)$', 'object_detail', entry_info_dict, 'coltrane_entry_detail'), (r'^links/
    $', 'archive_index', link_info_dict, 'coltrane_link_archive_index'), (r'^enlaces/(?
    P<año>\d{4})/$', 'archive_year', link_info_dict, 'coltrane_link_archive_year'),
    (r'^enlaces/(?P<año>\d{4})/(?P<mes>\w{3})/$', 'archive_month', link_info_dict,
    'coltrane_link_archive_month'), (r'^links/(?P<year>\d{4})/(?P<mes>\w{3})/(?
    P<día>\d{2})/$', 'archive_day', link_info_dict, 'coltrane_link_archive_day'), (r'^links/
    (?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', (P<slug>[-\w]+)$',
```

```

        'objeto_detalle',
        enlace_info_dict,
        'coltrane_enlace_detalle'),
    )

urlpatterns += patrones(",
(r'^categorías/$',
'django.views.generic.list_detail.object_list', { 'queryset':
Category.objects.all() }), (r'^categories/(?P<slug>[\w]+)/$', 'coltrane.views.category_detail'),

)

urlpatterns += patrones(",(r'^tags/
$', 'django.views.generic.list_detail.object_list', { 'queryset': Tag.objects.all() }),
(r'^tags/ entradas/(?P<etiqueta>[\w]+)/$', 'tagging.views.tagged_object_list',
{ 'queryset_or_model': Entry, 'template_name': 'coltrane/entries_by_tag.html' }),
(r'^tags/links/(?P<tag>[\w]+)/$', 'tagging.views.tagged_object_list',
{ 'queryset_or_model': Link, 'template_name': 'coltrane/links_by_tag.html' }),

)

```

En total, tiene cuatro modelos, dos diccionarios de argumentos de palabras clave para vistas genéricas y tres conjuntos de patrones de URL que se suman para formar el conjunto final. Esto hace que sea un poco complicado seguir exactamente lo que está pasando, así que reorganicemos un poco.

Dentro del directorio de la aplicación de weblog, cree un directorio llamado urls y dentro de él cree cinco archivos:

- `__init__.py` (para indicar que será un módulo de Python)
- `categorias.py`
- `entradas.py`
- `enlaces.py`
- `etiquetas.py`

Lo que vas a hacer es dividir el desorden actual en cuatro grupos lógicos de patrones de URL, cada uno dentro de su propio archivo. A partir de ahí, podrá usar las directivas `include()` para agregar cualquiera o todos estos patrones de URL a cualquier sitio que esté usando la aplicación de blog. Veamos cómo se desglosa esto en cada archivo.

`categorias.py` debe contener este contenido:

```
desde django.conf.urls.defaults importar *
```

de coltrane.models categoría de importación

```
urlpatterns = patrones(" , (r'^$',  
    'django.views.generic.list_detail.object_list', { 'queryset':  
        Category.objects.all() }), (r'^(?P<slug>[-\w]+)$',  
        'coltrane.views.category_detail'),  
)
```

Tenga en cuenta que la tercera línea comienza con urlpatterns = patterns(" , no urlpatterns += patterns(" . Solo habrá un conjunto de patrones por archivo, por lo que no necesita agregar patrones como lo hizo cuando estaban todo en un archivo. Además, las URL ya no tienen la cadena "categories/" . Debido a que ahora se pretende que una directiva include() acceda al archivocategories.py en otro lugar, puede obtener un poco más de flexibilidad al no requiere que las URL contengan la cadena "categorías/" .

Esto es lo que debe contener el nuevo archivo entrys.py:

```
*  
desde django.conf.urls.defaults importar
```

```
desde coltrane.models entrada de importación
```

```
entry_info_dict =  
    { 'conjunto de consultas':  
        Entry.objects.all(), 'date_field':  
            'pub_date', }
```

```
urlpatterns = patrones('django.views.generic.date_based', (r'^$',  
    'archive_index', entry_info_dict, 'coltrane_entry_archive_index'), (r'^(?P<año>\d{4})/$',  
    'archive_year', entry_info_dict, 'coltrane_entry_archive_year'), (r'^(?P<año>\d{4})/(?  
        P<mes>\w{3})/$', 'archive_month', entry_info_dict, 'coltrane_entry_archive_month'), (r'^(?  
        P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', 'archive_day', entry_info_dict,  
    'coltrane_entry_archive_day'), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?  
        P<slug>[-\w]+)$', 'object_detail', entry_info_dict, 'coltrane_entry_detail'),
```

```
)
```

Y pon esto en links.py:

```
*  
desde django.conf.urls.defaults importar
```

```
de coltrane.models enlace de importación
```

```

link_info_dict =
    { 'conjunto de consultas':
        Link.objects.all(), 'date_field':
        'pub_date', }

urlpatterns = patrones('django.views.generic.date_based',
    (r'^$', 'archive_index', link_info_dict, 'coltrane_link_archive_index'), (r'^(?P<año>\d{4})/$',
    'archive_year', link_info_dict, 'coltrane_link_archive_year'), ( r'^(?P<año>\d{4})/(?
    P<mes>\w{3})/$', 'archive_month', link_info_dict, 'coltrane_link_archive_month'), (r'^(?
    P <año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/$', 'archive_day', link_info_dict,
    'coltrane_link_archive_day'), (r'^(?P<año>\d{4})/(?P<mes>\w{3})/(?P<día>\d{2})/(?
    P<slug>[-\w]+)/$', 'objeto_detalle', enlace_info_dict, 'coltrane_enlace_detalle'),
)

)

```

Al igual que hizo con las URL de categorías, eliminó el bit "enlaces/" de estos patrones.

E inserte este contenido en tags.py:

```

from django.conf.urls.defaults import from
coltrane.models import Entry, Link from
tagging.models import Tag

urlpatterns = patrones('', (r'^$', *

'django.views.generic.list_detail.object_list', { 'queryset':
    Tag.objects.all() }), (r'^entries/(?P <etiqueta>[-\w]+)/$', 'tagging.views.tagged_object_list', { 'queryset_or_model':
        Entry, 'template_name': 'coltrane/entries_by_tag.html' }), (r'^links/( ?P<etiqueta>[-\w]+)/$', 'tagging.views.tagged_object_list', { 'queryset_or_model': Link,
        'template_name': 'coltrane/links_by_tag.html' }),

)

```

Una vez más, al igual que con las categorías y los enlaces, el bit "tags/" ha desaparecido.

Una vez que haya configurado estos archivos, debe eliminar el urls.py original de la carpeta de la aplicación de blog.

Ahora puede volver al módulo URLConf raíz del proyecto, que tenía un patrón como este:

```
(r'^weblog/', include('coltrane.urls')),
```

y luego tire de los bits individuales donde los desee:

```
(r'^weblog/categorías/', include('coltrane.urls.categorías')),
(r'^weblog/enlaces/', include('coltrane.urls.enlaces')),
(r'^weblog/tags/', include('coltrane.urls.tags')),
(r'^weblog/', include('coltrane.urls.entradas')),
```

Aunque ahora tiene varios archivos URLConf dentro de la aplicación de blog y necesita múltiples directivas include() para usarlas todas, ha obtenido dos grandes ventajas:

- Las URL de weblog ahora son mucho más manejables porque están divididas en unidades pequeñas que contienen solo conjuntos de URL que lógicamente pertenecen juntas.
- Debido a que ya no están revueltos en un solo archivo, es fácil usar include() para colocar un grupo específico de patrones en cualquier lugar de la jerarquía de URL de un sitio. Esto significa que ya no está atado a prefijos específicos como "enlaces/" o "etiquetas/" si no los quiere.

Como regla general, cualquier aplicación cuyos patrones de URL caigan lógicamente en grupos relacionados como estos deben dividirse en varios archivos separados precisamente por estas razones. Los beneficios superan con creces la desventaja de tener que lidiar con varios archivos.

Manejo de entradas en vivo

Antes de pasar a la última parte del weblog (plantillas y comentarios, que trataré en el próximo capítulo), agreguemos una característica más que falta.

Recordará que cuando configuró el modelo de entrada, le dio un campo llamado estado, que permite que las entradas se marquen como En vivo, Borrador u Oculto. Por el momento, ninguno de sus puntos de vista tiene eso en cuenta. Si agrega una entrada con un estado que no sea Vivo, notará que aún aparece en todas las vistas de archivo y detalles.

Ya ha visto que puede usar el método filter() para obtener solo los objetos que coinciden con ciertos criterios específicos. Al principio, parece una manera fácil de manejar esto. En cualquier lugar que estés usando esto:

```
Entrada.objects.todos()
```

podrías reemplazarlo con esto:

```
Entry.objects.filter(status=Entry.LIVE_STATUS)
```

Recuerde que definió constantes con nombre para los diferentes valores de estado para facilitar este tipo de consultas. Pero esto va a implicar una gran cantidad de tipeo. tendrás que recuerde escribir ese argumento de consulta adicional en cualquier lugar donde esté consultando entradas. Podría ser mucho mejor si pudiera tener una forma separada de consultar entradas que devuelva solo objetos con el campo de estado establecido en En vivo, tal vez algo como Entry.live.all() en lugar de Entry.objects.todos(). En realidad, esto es bastante fácil de hacer, pero requiere la introducción de una característica más importante del sistema modelo de Django: *los administradores*.

Hasta ahora, he estado pasando por alto cómo Django realmente hace consultas a la base de datos. yo solo he estado discutiendo cosas como Entry.objects.all() o FlatPage.objects.filter() sin hablar realmente sobre ese atributo especial llamado objetos o de dónde proviene.

El atributo de objetos es una instancia de una clase especial (`django.db.models.Manager`), que debe "adjuntarse" a una clase de modelo en particular, y que sabe cómo realizar todo tipo de consultas a la base de datos. Además de los métodos que ya ha visto, all() y filter(), tiene una gran cantidad de otros métodos que pueden devolver objetos específicos individuales, devolver listas de objetos, devolver otras estructuras de datos correspondientes a los datos almacenados por un modelo. , cambie el orden utilizado para devolver los resultados y maneje una variedad de otras tareas útiles. Documentación completa de la base de datos.

La API y todos sus métodos y opciones están disponibles en línea en <http://docs.djangoproject.com/en/dev/topics/db/models/>.

Si no especifica un administrador para su modelo, Django agrega uno y lo llama objetos (esto sucede automáticamente para cualquier clase que subclase `django.db.models.Model`). Sin embargo, eres libre de adjuntar un administrador con el nombre que desee, y si lo hace, Django no se molestará con el administrador automático de objetos predeterminado. Por ejemplo, podría definir un modelo como este:

```
clase MiModelo(modelos.Modelo):
    nombre = modelos.CharField(max_length=50)

    object_fetcher = modelos.Manager()
```

Luego, en lugar de usar `MyModel.objects.all()`, por ejemplo, usaría `MyModel.buscar_objeto.todo()`. Todos los métodos de consulta estándar estarán allí, solo en un atributo con el nombre que has especificado.

Sin embargo, lo más importante acerca de los gerentes es que puede definir fácilmente su propias clases de administrador y darles un comportamiento personalizado escribiendo una subclase de `django.db.models.Manager` y reemplazando los métodos que desea personalizar. En este caso, desea escribir un administrador que, cuando se adjunte al modelo `Entry`, devuelva solo las entradas cuyo estado sea En vivo. Puede hacer esto escribiendo una subclase de Manager y anulando un método, `get_query_set()`, que devuelve el objeto `QuerySet` inicial que usarán `all()`, `filter()` y todos los demás métodos de consulta. Hacer esto es sorprendentemente fácil:

```
clase LiveEntryManager(modelos.Manager):
    def get_query_set(auto):
        devuelve super(LiveEntryManager, self).get_query_set().filter(
            status=self.model.LIVE_STATUS)
```

El único truco aquí es que estás usando `self.model.LIVE_STATUS` como el valor para filtrar en. Cada administrador que se ha adjuntado a un modelo puede acceder a esa clase de modelo a través del atributo `self.model`.

Coloque el código anterior en el archivo `models.py` de la aplicación `weblog`, en algún lugar *por encima* de la definición del modelo `Entry`. Luego agregue las siguientes líneas dentro del modelo `Entry`:

```
vivir = LiveEntryManager()
objetos = modelos.Manager()
```

Esto le da al modelo `Entry` dos gerentes. Uno se llama objetos y es solo el estándar. gerente que normalmente obtiene cada modelo. El otro es una instancia de `LiveEntryManager`, lo que significa que ahora puede escribir

```
Entry.live.all()
```

y hará exactamente lo que usted quiere que haga. Tenga en cuenta que debe definir los objetos manualmente. Cuando un modelo tiene un administrador personalizado, Django no configura automáticamente el administrador de objetos para usted.

Ahora puede simplemente realizar una búsqueda y reemplazo en el código del weblog, cambiando cualquier uso de `Entry.objects` a `Entry.live`. Eso se encargará de cualquier situación en la que estés consultando entradas (solo una hasta ahora, pero si hubiera ido mucho más lejos, fácilmente podrían haber sido más).

Sin embargo, hay otros dos lugares en los que deberá preocuparse por filtrar solo las entradas en vivo, cuando recupere las entradas para una categoría o etiqueta específica. Para las categorías, puede resolver esto con bastante facilidad agregando un método en el modelo `Categoría`:

```
def live_entry_set(auto):
    desde coltrane.models importación
    volver self.entry_set.filter(status=Entry.LIVE_STATUS)
```

Y ahora, en cualquier lugar donde haya usado el atributo `entry_set` de una categoría, simplemente puede reemplazarlo con una llamada a `live_entry_set()`. Entonces, por ejemplo, la vista `category_detail` ahora se verá como este:

```
def category_detail(solicitud, slug):
    categoría = get_object_or_404(Categoría, slug=slug)
    volver render_to_response('coltrane/category_detail.html',
        { 'lista_objetos': categoría.live_entry_set() })
```

Con las etiquetas es un poco más complicado, pero aún puedes hacer que funcione. Recuerda que el `tagged_` La vista `object_list` recibe un argumento llamado `queryset_or_model`. Esto significa que puedes pasar la vista a una clase de modelo, como `Link`, o a `QuerySet`. Entonces, ¿dónde estás usando el `tagged_` vista `object_list` con el modelo `Entry` como argumento, cámbielo para usar `Entry.live.all()` en cambio.

Mirando hacia el futuro

La aplicación `weblog` está casi completa ahora. Solo le quedan un par de funciones por agregar, y para ellas usará aplicaciones incluidas con Django más algunas personalizaciones. Los cubriré en el Capítulo 7, pero en el próximo capítulo analizará con más detalle el sistema de plantillas de Django, escribirá plantillas para el blog e incluso escribirá un par de etiquetas de plantillas personalizadas.

Si desea hacer una pausa por un momento y jugar con la aplicación `weblog` antes de pasar al Capítulo 6, no dude en hacerlo. Incluso sin el sistema de comentarios y las técnicas de plantilla, que cubrirá en el próximo capítulo, esta aplicación de `weblog` ya es una pieza de software bastante sólida que ofrece un subconjunto significativo de la funcionalidad de los populares sistemas de `weblog` listos para usar, como `WordPress` (pero con mucho menos código).

Capítulo 6



Plantillas para el Weblog

Su La aplicación weblog está casi completa. En los últimos dos capítulos, ha implementado entradas, enlaces y casi toda la funcionalidad de asistente que deseaba tener con ellos. Solo quedan dos funciones por implementar: un sistema de comentarios y feeds de sindicación, y Django le brindará bastante ayuda con eso, como verá en el próximo capítulo.

Pero hasta ahora, se ha centrado casi exclusivamente en el "back-end" del sitio: el código de Python que modela sus datos, los recupera de la base de datos, establece la estructura de su URL, etc., a expensas del "front-end", o el HTML real que mostrará a los visitantes de su sitio.

Ha visto cómo las vistas genéricas de Django exponen los objetos de su base de datos para su uso en plantillas (a través de la variable `object_list` en los archivos basados en fechas, por ejemplo). Sin embargo, es un gran paso de eso a un weblog atractivo y usable. Echemos un vistazo más profundo al sistema de plantillas de Django y cómo puede usarlo para hacer que la parte delantera sea tan fácil como la trasera.

Lidiando con Elementos Repetitivos: El Poder de Herencia

Está utilizando las vistas genéricas de Django para mostrar tanto las entradas como los enlaces. Ya sea que esté mirando la vista detallada de una entrada o de un enlace, el código de Python real involucrado es la vista genérica `object_detail` basada en la fecha, que proporciona un objeto con nombre variable a la plantilla y representa el objeto de la base de datos que recuperó. La mayor diferencia es que la vista genérica utilizará una plantilla denominada `coltrane/entry_detail.html` para una Entrada y otra denominada `coltrane/link_detail.html` para un enlace.

Debido a que los contextos son tan similares, las plantillas terminarán pareciéndose mucho; por ejemplo, una plantilla simple de `entry_detail` podría tener el siguiente aspecto:

```
<!DOCTYPE html PÚBLICO "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<cabeza>
<title>Weblog: {{ object.title }}</title>
</cabeza>

<cuerpo>
<h1>{{ objeto.título }}</h1>
{{ objeto.cuerpo_html|seguro }}
</cuerpo>
</html>
```

Y un link_detail simple podría verse así:

```
<!DOCTYPE html PÚBLICO "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<cabeza>
<title>Weblog: {{ object.title }}</title>
</cabeza>
<cuerpo>
<h1>{{ objeto.título }}</h1>
{{ objeto.descripcion_html|seguro }}
<p><a href="{{ object.url }}>Visita el sitio</a></p>
</cuerpo>
</html>
```

Por supuesto, para un sitio terminado, querrás hacer un poco más, pero ya es evidente que hay mucha repetición. Hay todo tipo de repeticiones HTML, que son las mismas en ambas plantillas, e incluso elementos como el elemento <title> y el encabezado <h1> tienen el mismo contenido. Escribir todo eso una y otra vez va a ser terriblemente tedioso, especialmente a medida que el HTML se vuelve más complejo. Y si alguna vez realiza cambios en la estructura HTML de un sitio, tendrá que hacerlos en cada plantilla. Django ha sido excelente hasta ahora para ayudarlo a evitar este tipo de trabajo tedioso y repetitivo en el lado de Python, por lo que, naturalmente, sería bueno si también pudiera hacer lo mismo en el lado de HTML.

Y puede El sistema de plantillas de Django admite un concepto de *herencia de plantillas*, que funciona de manera similar a la forma en que funcionan las subclases en el código Python normal. Esencialmente, el sistema de plantillas de Django le permite escribir una plantilla con marcadores de posición (llamados *bloques*) para las secciones de una página. Estos bloques variarán de una plantilla a otra. Luego, escribirá plantillas para "extender" esa plantilla y completar los marcadores de posición.

Para ver la herencia de plantillas en acción, trabajemos con un ejemplo simple. crear un archivo en el directorio raíz de la plantilla del proyecto y asígnele el nombre base.html. No es obligatorio usar este nombre, pero es una práctica común y ayudará a otros a comprender el propósito del archivo. En ese archivo, pon el siguiente código:

```
<!DOCTYPE html PÚBLICO "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<cabeza>
<title>Weblog: {{ object.title }}</title>
</cabeza>
<cuerpo>
<h1>{{ objeto.título }}</h1>
{% contenido del bloque %}
{% bloque final %}
</cuerpo>
</html>
```

Ahora, edite la plantilla coltrane/entry_detail.html para que no contenga *nada* más que esto:

```
{% extiende "base.html" %}

{% contenido del bloque %}
{{ objeto.cuerpo_html|seguro }}
{% bloque final %}
```

A continuación, edite coltrane/link_detail.html para que no contenga nada más que esto:

```
{% extiende "base.html" %}

{% contenido del bloque %}
{{ objeto.descripcion_html|seguro }}
<p><a href="{{ object.url }}>Visita el sitio</a></p>
{% bloque final %}
```

Finalmente, inicie el servidor de desarrollo y visite un enlace o entrada en el weblog, y luego vea la fuente HTML de la página. Verá toda la plantilla HTML que está en base.html; tenga en cuenta que el área que contiene un bloque de contenido vacío se llenará con los resultados apropiados, según esté viendo una entrada o un enlace.

Este es solo un ejemplo simple. A medida que sus plantillas se vuelven más complejas, la capacidad de eliminar piezas repetitivas como esta se convertirá en un salvavidas. Reducirá tanto el tiempo necesario para armar las plantillas como el tiempo necesario para cambiarlas más adelante (porque un cambio en una sola plantilla "base" aparecerá automáticamente en cualquier plantilla que la amplíe).

Cómo funciona la herencia de plantillas

La herencia de plantillas gira en torno a las dos nuevas etiquetas vistas en el ejemplo anterior: {% block %} y {% extends %}. Esencialmente, la etiqueta {% block %} le permite crear una sección de una plantilla y darle un nombre, y posiblemente incluso algún contenido predeterminado. La etiqueta {% extends %} le permite especificar el nombre de otra plantilla, que debería contener uno o más bloques.

y luego simplemente complete el contenido de cualquier bloque que desee usar. El resto del contenido, incluido el contenido predeterminado de los bloques que no anuló, se completará automáticamente a partir de la plantilla que está extendiendo. Además, dentro de un bloque, tendrá acceso al contenido que *habría* ido allí si no hubiera proporcionado el suyo propio. Este contenido se almacena en una variable especial llamada block.super. Entonces, si tuviera una plantilla base que contuviera esto:

```
{% block title %}Mi blog:{% endblock %}
```

puede escribir una plantilla que la amplíe y completar su propio contenido:

```
{% block title %}Mi página{% endblock %}
```

Con block.super, puede acceder al contenido predeterminado del bloque principal para obtener un valor final de Mi weblog: Mi página:

```
{% block title %}{% block.super %} Mi página{% endblock %}
```

Límites de la herencia de plantillas

A medida que comience a trabajar con la herencia en las plantillas, querrá tener en cuenta algunas advertencias:

- Si usa la etiqueta `{% extends %}`, debe ser lo primero en la plantilla. Django necesita saber por adelantado que vas a extender otra plantilla.
- Cada bloque con nombre, si se usa, puede aparecer solo una vez en una plantilla dada. Al igual que HTML le permite tener un solo elemento con una ID dada dentro de una sola página, el sistema de plantillas de Django le permite tener solo un bloque con un nombre dado dentro de una sola plantilla.
- Una plantilla puede extender directamente solo otra plantilla: múltiples usos de `{% extends %}` en la misma plantilla no son válidos. Sin embargo, la plantilla que se amplía puede, a su vez, ampliar otra plantilla, lo que lleva a una cadena de herencia a través de varias plantillas.

Esta capacidad de "encadenar" plantillas heredadas es clave para un patrón común en el desarrollo de plantillas. opción A menudo, un sitio tendrá varias secciones o áreas que no varían mucho entre sí, por lo que las plantillas terminan formando una estructura de tres capas:

1. Una sola plantilla base que contiene el HTML común de todas las páginas.
2. Plantillas base específicas de la sección que completan la navegación y/o la tematización adecuadas. Estos amplían la plantilla base.
3. Las plantillas "reales" que las vistas cargarán y representarán. Estos amplían las plantillas específicas de la sección apropiada.

De hecho, este patrón es tan común y tan útil que lo vas a utilizar para las plantillas de tu blog. Empecemos.

Definición de la plantilla base para el blog

La creación de una plantilla base útil para un sitio consiste en gran medida en determinar cuál será la apariencia general del sitio, escribir el código HTML adecuado para admitirlo y luego determinar qué áreas deberán variar de una página a otra y convertirlas en bloques.

Para este blog, utilicemos un diseño visual común: un encabezado en la parte superior de la página con espacio para el logotipo del sitio y dos columnas debajo. Una columna contendrá el contenido principal de la página y la otra columna tendrá una barra lateral con navegación, metadatos y otra información útil.

En términos de HTML, esto se traduce en tres elementos div: uno para el área del encabezado, uno para el área de contenido y otra para la barra lateral. La estructura se ve así:

```
<!DOCTYPE html PÚBLICO "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<cabeza>
  <título></título>
</cabeza>
```

```
<cuerpo>
  <div id="encabezado"></div>
  <div id="contenido"></div>
  <div id="barra lateral"></div>
</cuerpo>
</html>
```

Tenga en cuenta que seguí adelante y completé algunos atributos de identificación HTML en estas etiquetas div para que sea fácil configurar el diseño con hojas de estilo en cascada (CSS).

Ahora, una cosa que salta a la vista es el hecho de que el elemento del título está vacío. Esto definitivamente es algo que variará, según la parte del sitio en la que te encuentres y lo que estés viendo, así que sigamos adelante y bloqueemos allí:

```
<title>Mi weblog {& block title %}{& endblock %}</title>
```

Cuando amplié esta plantilla, agregará más cosas aquí. El efecto final será conseguir un título como Mi weblog | Entradas | Febrero de 2008, como verás en un momento.

Ahora completemos el encabezado. Probablemente no cambie mucho, por lo que no necesita un bloque aquí:

```
<div id="encabezado">
  <h1 id="branding">Mi weblog</h1>
</div>
```

Nuevamente, agregué un atributo de identificación para que pueda usar fácilmente CSS para diseñar el encabezado más adelante. Por ejemplo, podría usar una técnica de reemplazo de imágenes para reemplazar el texto del h1 con un logotipo.

Debido a que el contenido principal variará bastante, lo convertirá en un bloque:

```
<div id="contenido">
  {& contenido del bloque %}
  {& bloque final %}
</div>
```

Todo lo que queda es la barra lateral. Lo primero que necesitará es una lista de enlaces a diferentes funciones del weblog para que los visitantes puedan navegar fácilmente por el sitio. Puede hacerlo con bastante facilidad (nuevamente, el uso de atributos de identificación facilita volver más tarde y diseñar la barra lateral):

```
<div id="barra lateral">
  <h2>Navegación</h2>
  <ul id="nav-principal">
    <li id="entradas-de-navegación-principal">
      <a href="/weblog/">Entradas</a></li>
    <li id="principal-nav-enlaces">
      <a href="/weblog/enlaces/">Enlaces</a></li>
    <li id="categorías-de-navegación-principal">
      <a href="/weblog/categories/">Categorías</a></li>
    <li id="etiquetas de navegación principal"><a href="/weblog/tags/">Etiquetas</a></li>
  </ul>
</div>
```

Pero una cosa se destaca: aquí tiene URL codificadas. Coincidirán con lo que ha configurado en su módulo URLConf. Pero después de tomarse la molestia de modularizar y desacoplar las URL en el lado de Python, sería una pena dar la vuelta y codificarlas en sus plantillas.

Una mejor solución es usar la etiqueta de plantilla `{% url %}`, que, como el decorador de enlaces permanentes que utilizó en los métodos `get_absolute_url()` de sus modelos, puede realizar una búsqueda inversa en su URLConf para determinar la URL adecuada. Esta etiqueta ofrece bastantes opciones, pero la que le interesa en este momento es bastante simple: puede ingresar el nombre de un patrón de URL y generará la URL correcta.

Usando la etiqueta `{% url %}`, puede reescribir su lista de navegación de esta manera:

```
<ul id="nav-principal">
    <li id="entradas-de-navegación-principal">
        <a href="{% url coltrane_entry_archive_index %}">Entradas</a>
    </li>
    <li id="principal-nav-enlaces">
        <a href="{% url coltrane_link_archive_index %}">Enlaces</a>
    </li>
    <li id="categorías-de-navegación-principal">
        <a href="{% url coltrane_category_list %}">Categorías</a>
    </li>
    <li id="etiquetas-de-navegación-principales">
        <a href="{% url coltrane_tag_list %}">Etiquetas</a>
    </li>
</ul>
```

Ahora no tendrá que hacer cambios en sus plantillas si decide mezclar algunas URL más adelante.

Mientras se ocupa de la navegación, agreguemos un bloque dentro de la etiqueta del cuerpo:

```
<body class="{% block bodyclass %}{% endblock %}">
```

Una técnica común en el diseño web basado en CSS es usar un atributo de clase en la etiqueta del cuerpo para activar cambios en el estilo de una página. Por ejemplo, tendrá una lista de opciones de navegación en la barra lateral, que representan diferentes partes del blog (entradas, enlaces, etc.) y sería bueno resaltar la parte que un visitante está mirando actualmente. Al cambiar la clase de la etiqueta del cuerpo en diferentes partes del sitio, puede usar fácilmente CSS para resaltar el elemento correcto en la lista de navegación.

Para el resto del contenido de la barra lateral, es posible que desee tener una pequeña explicación de lo que está mirando un visitante, como "Una entrada en mi blog, publicada el 7 de febrero de 2008" o "Una lista de entradas en la categoría 'Django'." También puedes agregar un bloque para eso:

```
<h2>¿Qué es esto?</h2>
{% bloquear lo que es%}
{% bloque final %}
```

Ya terminó con la plantilla base, por ahora. (Le agregarás algunas cosas más adelante).
Esto es lo que parece:

```
<!DOCTYPE html PÚBLICO "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es"> <cabeza>

    <title>Mi weblog {% block title %}{% endblock %}</title> </head> <body
class="{% block bodyclass %}{% endblock %}">

    <div id="encabezado">
        <h1 id="branding">Mi weblog</h1> </div>
    <div id="content">

        {% contenido de bloque
        %} {% bloque final %} </
        div> <div id="barra lateral">

        <h2>Navegación</h2>
        <ul id="main-nav"> <li
            id="main-nav-entries">
                <a href="{% url coltrane_entry_archive_index %}">Entradas</a> </li> <li id="main-
nav-links">
                    <a href="{% url coltrane_link_archive_index %}">Enlaces</a> </li> <li id="main-
nav-categories">
                        <a href="{% url coltrane_category_list %}">Categorías</a> </li> <li id="main-
nav-tags">
                            <a href="{% url coltrane_tag_list %}">Etiquetas</a> </li> </ul>
                <h2>¿Qué es esto?</h2> {% block whatis %}{% endblock %} </
        div> </cuerpo> </html>
```

Configuración de plantillas de sección

Ahora configuremos algunas plantillas que manejarán las diferentes áreas principales del blog. Querrá uno para entradas, enlaces, etiquetas y categorías. Llame a la plantilla para las entradas `base_entradas.html`, y todo lo que necesita hacer es ampliar la plantilla base y completar un par de bloques:

```
{% extiende "base.html" %}  
  
{% título de bloque%}| Entradas{% endblock %}
```

```
{% block bodyclass %}entradas{% endblock %}
```

Si tuviera que usar esta plantilla por sí sola, obtendría el resultado de base.html, pero con dos cambios:

- El contenido de la etiqueta del título sería Mi weblog | Entradas.
- El atributo de clase de la etiqueta del cuerpo tendría un valor de entradas, lo que significa que sería es fácil resaltar el elemento Entradas en la barra lateral de navegación.

El resto de las plantillas de sección son bastante fáciles de completar. Por ejemplo, puede escribir un base_links.html como este:

```
{% extiende "base.html" %}  
  
{% título de bloque%}| Enlaces{% endblock %}  
  
{% block bodyclass %}enlaces{% endblock %}
```

También necesitará una plantilla base_tags.html y una plantilla base_categories.html; puedes completarlos usando el patrón que acabo de describir. Estas plantillas son un poco repetitivas, y probablemente siempre lo serán, pero el uso de la herencia de plantillas significa que ha reducido los bits repetitivos al mínimo: está especificando solo las cosas que cambian, no las cosas que permanecen igual.

Visualización de archivos de entradas

Para mostrar las entradas, necesita cinco plantillas:

- La página principal (inicio) que muestra las últimas entradas
- Un archivo anual
- Un archivo mensual
- Un archivo diario
- Una entrada individual

Estos corresponden directamente a las vistas genéricas que está utilizando.

Índice de entrada

Comencemos con el índice principal de entradas. Recordará que la vista genérica buscará la plantilla coltrane/entry_archive.html y proporcionará una variable llamada Latest que contiene una lista de las últimas entradas. Por lo tanto, puede completar la plantilla coltrane/entry_archive.html de la siguiente manera (recordando extender base_entries.html en lugar de base.html):

```
{% extiende "entradas_base.html" %}

{% título del bloque %}{% bloque.super %} | Últimas entradas{% endblock %}

{% contenido del bloque %}
{% para la entrada en el último %}
<h2>{{ entrada.título }}</h2>
<p>Publicado el {{ entry.pub_date|date:"F j, Y" }}</p>
{% si entrada.extracto_html%}
    {{entrada.extracto_html|seguro}}
{% más %}
    {{entrada.body_html|truncatewords_html:"50"|seguro}}
{% terminara si %}
<p><a href="{{ entry.get_absolute_url }}">Lea la entrada completa. . .</a></p>
{% endfor%}
{% bloque final %}

{% bloquear lo que es%}
<p>Esta es una lista de las últimas {{ latest.count }} entradas publicadas en
mi blog.</p>
{% bloque final %}
```

La mayor parte de esto debería ser bastante familiar. Está utilizando la etiqueta `{% for %}` para recorrer las entradas y mostrar cada una. Y en la barra lateral, solo tiene un breve párrafo que describe lo que se muestra en esta página. El código se basa en el método `count()` de Django QuerySet para averiguar cuántas entradas se pasaron a la plantilla en la última variable.

Sin embargo, hay un par de cosas nuevas aquí que vale la pena señalar:

- El uso del filtro de fecha para dar formato a la fecha_publicación de cada entrada: este filtro acepta un formating string, similar al método `strftime()` que ya ha visto, y genera la fecha correspondiente. En este caso, la fecha se imprimirá en el formato 6 de febrero de 2008.
- El uso del filtro `truncatewords_html`: Este filtro toma un número como argumento y saca ese número de palabras de la variable a la que se aplica, añadiendo puntos suspensivos (...) al final. Esto es útil para generar un breve extracto cuando la entrada no tiene su campo de extracto completado.

Archivo anual

La vista genérica que genera el archivo anual proporcionará dos variables:

- `año`: El año que se muestra.
- `date_list`: una lista de objetos de fecha y hora de Python que representan los meses de ese año que tienen entradas.

Esta vista genérica buscará la plantilla `coltrane/entry_archive_year.html`, que puede completar de la siguiente manera:

```

{%
    extiende "entradas_base.html"
}

{%
    título del bloque
}{{
    bloque.super
}} | {{ año }}{%
    bloque final
}

{%
    contenido del bloque
}
<ul>
    {%
        por mes en date_list
    }
    <li>
        <a href="/weblog/entries/{{ año }}/{{ mes|fecha:'b' }}"/>{{ mes|fecha:'F' }}</a>
    </li>
    {%
        endfor
    }
</ul>
{%
    bloque final
}

{%
    bloquear lo que es
}
<p>Esta es una lista de meses en {{ año }} en los que publiqué entradas en
mi blog.</p>
{%
    bloque final
}

```

Aquí está recorriendo la lista de fechas y, para cada mes, muestra un enlace al archivo de ese mes.

Pero aquí hay un problema: puede construir las URL utilizando el filtro de fecha incorporado de Django, pero una vez más está codificando una URL. Anteriormente, solucionaba eso usando `{% url %}` etiqueta con el nombre de un patrón de URL. Puede volver a hacerlo, pero esta vez deberá proporcionar algunos datos adicionales: el año y el mes necesarios para generar la URL correcta para un archivo mensual. Todo lo que tiene que hacer es pasar la etiqueta `{% url %}` a un segundo argumento que contiene una lista separada por comas de los valores que necesita, e incluso puede usar filtros para asegurarse de que las URL tengan el formato correcto:

```

<li><a href="{% url coltrane_entry_archive_mes año,mes|fecha:'b' %}">
    {{ mes|fecha:'F' }}
</a></li>

```

Con la configuración de URL actual, este HTML generará correctamente URL como `/weblog/2008/jan/`, `/weblog/2008/feb/`, y así sucesivamente.

Archivos Mensuales y Diarios

Las vistas genéricas que generan los archivos mensuales y diarios son extremadamente similares. Ambos proporcionarán una lista de entradas en una variable llamada `object_list`, y la única diferencia real es que uno tendrá una variable llamada `mes` (que representa el mes para un archivo mensual) y el otro tendrá una variable llamada `día` (que representa el día para un archivo diario).

Aquí está la plantilla de archivo mensual, que será `coltrane/entry_archive_month.html`:

```

{%
    extiende "entradas_base.html"
}

{%
    título del bloque
}
{{ bloque.super }} | Entradas en {{ mes|fecha:'F, Y' }}
{%
    bloque final
}

```

```
{% contenido del bloque %}  
{% para entrada en object_list %}  
    <h2>{{ entrada.título }}</h2>  
    <p>Publicado el {{ entry.pub_date|date:"F j, Y" }}</p>  
    {% si entrada.extracto_html%}  
        {{entrada.extracto_html|seguro}}  
    {% más %}  
    {{entrada.body_html|truncatewords_html:"50"|seguro}}  
    {% terminara si %}  
    <p><a href="{{ entry.get_absolute_url }}>Lea la entrada completa. . .</a></p>  
{% endfor%}  
{% bloque final %}  
  
{% bloquear lo que es%}  
<p>Esta es una lista de entradas publicadas en mi blog en  
    {{ mes|fecha:"F, Y" }}.</p>  
{% bloque final %}
```

Excepto por un par de cambios en los nombres de las variables y el uso del filtro de fecha para formatear el mes (se imprimirá en febrero de 2008), esto no es muy diferente de lo que ya ha visto. La plantilla de archivo diario (coltrane/entry_archive_day.html) será casi idéntica excepto por el uso de la variable día y el formato apropiado, así que continúe y rellénelo. (Puede encontrar una lista completa de las opciones de formato de fecha disponibles en la documentación de la plantilla de Django en línea en www.djangoproject.com/documentation/templates/.)

Detalle de entrada

La vista genérica que muestra una única entrada utiliza la plantilla coltrane/entry_detail.html y proporciona una variable, objeto, que será la entrada. La primera parte de esta plantilla es fácil:

```
{% extiende "entradas_base.html" %}  
  
    {% título del bloque %}{% bloque.super %} | {{ objeto.título }}{% endblock %}  
  
    {% contenido del bloque %}  
    <h2>{{ objeto.título }}</h2>  
    {{ objeto.cuerpo_html|seguro }}  
    {% bloque final %}
```

La barra lateral es un poco más complicada. Puede comenzar mostrando la fecha de publicación de la entrada:

```
{% bloquear lo que es%}  
<p>Esta es una entrada publicada en mi blog en  
    {{ objeto.pub_date|fecha:"F j, Y" }}.</p>
```

Ahora, sería bueno mostrar las categorías mostrando texto como "Esta entrada es parte de las categorías 'Django' y 'Python'". Pero hay varias cosas a tener en cuenta aquí:

- Para una entrada con una categoría, quiere decir "parte de la categoría". Pero para una entrada con más de una categoría, debe decir "parte de las categorías". Y para una entrada sin categorías, debe decir: "Esta entrada no forma parte de ninguna categoría".
- Para una entrada con más de dos categorías, necesitará comas entre categoría nombres y la palabra "y" antes de la categoría final. Pero para una entrada con dos categorías, no necesita las comas, y para una entrada con una sola categoría, no necesita las comas ni el "y".

Si no hay ninguna categoría para una entrada, {{ object.categories.count }} será 0, que es False dentro de una etiqueta {% if %}, por lo que puede comenzar con una prueba para eso:

```
{% si objeto.categorías.cuenta%}
    . . . llenará esto momentáneamente. . {% más
%
%}
<p>Esta entrada no forma parte de ninguna categoría.</p>
{%- terminara si %}
```

Ahora necesita manejar la diferencia entre "categoría" y "categorías". Porque esto es un problema común, Django incluye un filtro llamado pluralize que puede solucionarlo. El filtro de pluralización, por defecto, no genera nada si se aplica a una variable que se evalúa como el número 1, pero genera una "s" de lo contrario. También acepta un argumento que le permite especificar otro texto para la salida. En este caso, desea una "y" para el caso singular y "ies" para el plural, por lo que puede escribir esto:

```
{% si objeto.categorías.cuenta%}
<p>Esta entrada es parte de la
categoría{{ objeto.categorías.cuentapluralize:"y,ies" }}
```

Obtendrá "categoría" cuando solo hay una categoría y "categorías" de lo contrario. Finalmente, debe recorrer las categorías. Una opción sería unirse a la lista de gatogerías usando comas. En código Python, escribirías esto:

```
', '.join(objeto.categorías.todos())
```

Y el sistema de plantillas de Django proporciona un filtro de unión, que funciona de la misma manera:

```
{{ objeto.categorías.todos|unirse:", " }}
```

Pero desea que se inserte la palabra "y" antes de la categoría final en la lista, y únase no puedo hacer eso La solución es usar la etiqueta {% para %} y aprovechar algunas variables útiles que pone a disposición. Dentro del ciclo {% for %}, las siguientes variables estarán automáticamente disponibles:

- forloop.counter: La iteración actual del bucle, contando desde 1. La cuarta vez a través del bucle, por ejemplo, será el número 4.
- forloop.counter0: Igual que forloop.counter, pero comienza a contar desde 0 en lugar de desde 1. El cuarta vez a través del bucle, por ejemplo, este será el número 3.
- forloop.revcounter: El número de iteraciones que quedan hasta el final del ciclo, contando hasta 1. Cuando quedan cuatro iteraciones, por ejemplo, este será el número 4.
- forloop.revcounter0: Igual que forloop.revcounter, pero cuenta hacia atrás hasta 0 en lugar de 1.

- `forloop.first`: Un valor booleano—será True la primera vez que pase por el bucle y False el resto del tiempo.
- `forloop.last`: Otro valor booleano—este es True la última vez que se recorre el bucle y False el resto del tiempo.

Usando estas variables, puede elaborar la presentación adecuada. Expresado en inglés, la lógica funciona así:

1. Muestre un enlace a la categoría.
2. Si esta es la última vez que pasa por el bucle, no muestre nada más.
3. Si esta es la penúltima vez que pasa por el bucle, muestre la palabra "y".
4. De lo contrario, muestra una coma.

Y aquí está en código de plantilla:

```
{% para categoría en object.categories.all %}  
    <a href="{{ category.get_absolute_url }}>{{ category.title }}</a> {% if forloop.last %}{%  
    else %} {% ifequal forloop.revcounter0 1 %}and {% else %}, {% endifequal %} {% endif  
%} {% endfor %}
```

Hay dos partes importantes aquí:

- `{% if forloop.last %}{% else %}`: Esto no hace absolutamente nada si estás en el último viaje a través del bucle.
- `{% ifequal forloop.revcounter0 1 %}`: Esto determina si estás en el penúltimo viaje a través del bucle para imprimir la "y" antes de la categoría final.

Aquí está el bloque completo de la barra lateral hasta ahora:

```
{% block whatis %}  
<p>Esta es una entrada publicada en mi blog el  
{{ object.pub_date|date:"F j, Y" }}.</p>  
  
{% if object.categories.count %} <p>Esta  
entrada es parte de la  
categoría{{ object.categories.count|pluralize:"y,ies" }} {% for category  
in object.categories.all %}  
    <a href="{{ category.get_absolute_url }}>{{ category.title }}</a> {% if forloop.last %}{%  
    else %} {% ifequal forloop.revcounter0 1 %}and {% else %}, {% endifequal %} {% endif  
%} {% endfor %} </p> {% else %} <p>Esta entrada no forma parte de ninguna categoría.</p>  
    {% endif %} {% bloque final %}
```

El manejo de etiquetas funcionará de la misma manera. {{ object.tags }} devolverá las etiquetas para la Entrada, y un código de plantilla similar puede manejarlas. Y con eso, tienes una plantilla de detalles de entrada bastante buena:

```
{% extiende "entradas_base.html" %}

{% título del bloque %}{% bloque.super %} | {{ objeto.título }}{% endblock %}

{% contenido del
bloque %} <h2>{{ objeto.título }}
</h2> {{ objeto.cuerpo_html }}
{% bloque final %}

{% block whatis %}
<p>Esta es una entrada publicada en mi blog el
{{ object.pub_date|date:"F j, Y" }}.</p>

{% if object.categories.count %}
<p>Esta entrada es parte de la
categoría{{ object.categories.count|pluralize:"y,ies" }} {% for
category in object.categories.all %}
<a href="{{ category.get_absolute_url }}">{{ category.title }}</a> {% if forloop.last
%}{% else %} {% ifequal forloop.revcounter0 1 %}and {% else %}, {% endifequal
%} {% endif %} {% endfor %} </p> {% else %} <p>Esta entrada no forma parte de
ninguna categoría.</p> {% endif %} {% bloque final %}
```

Definición de plantillas para otros tipos de contenido

Las plantillas para mostrar enlaces en el blog no son muy diferentes de las plantillas que muestran las entradas del blog. Extenderán base_links.html en lugar de base_entries.html, por supuesto, pero los nombres de las variables disponibles en las distintas plantillas serán los mismos. La única diferencia es que las plantillas de enlace tendrán acceso a los objetos de enlace, por lo que deberían mostrar los enlaces en función de los campos que haya definido en el modelo de enlace. Este es un ejemplo de cómo se vería coltrane/link_detail.html:

```
{% extiende "base_links.html" %}

{% título del bloque %}{% bloque.super %} | {{ objeto.título }}{% endblock %}

{% contenido del
bloque %} <h2>{{ objeto.título }}
</h2> {{ objeto.descripcion_html }}
```

```
<p><a href="{{ object.url }}>Visita el sitio</a></p>
{%- bloque final %}

{%- bloquear lo que es%}
<p>Este es un enlace publicado en mi blog el {{ object.pub_date|date:"F j, Y" }}.</p>

{%- si objeto.etiquetas.cuenta%}
    <p>Este enlace está etiquetado con
    {%- para etiqueta en object.categories.all %}
        <a href="{{ etiqueta.get_absolute_url }}>{{ etiqueta.título }}</a>
    {%- si forloop.last %}{%- else %}
        {%- ifequal forloop.revcounter0 1 %}y {%- else %}, {%- endifequal %}
    {%- terminara si %}
    {%- endfor%}
</p>
{%- más %}

<p>Este enlace no tiene ninguna etiqueta.</p>
{%- terminara si %}
{%- bloque final %}
```

Tenga en cuenta que debido a que los enlaces tienen etiquetas en lugar de categorías, esta plantilla simplemente recorre las etiquetas de la misma manera que `coltrane/entry_detail.html` recorre las categorías.

Del mismo modo, las plantillas de categorías y etiquetas son fáciles de configurar en este punto. Solo necesitan extender la plantilla correcta para la parte del sitio que representan y usar los campos correctos de los modelos Categoría y Etiqueta, respectivamente (aunque recuerde que la vista detallada de categorías y etiquetas en realidad devolverá listas de objetos Entrada o Enlace para una categoría o etiqueta en particular). Puede encontrar ejemplos completos en el código de muestra del libro, disponible en el sitio web de Apress.

Ampliación del sistema de plantillas con etiquetas personalizadas

En este momento, lo único que aparecerá en la barra lateral de tu blog será la lista de enlaces de navegación y el breve "¿Qué es esto?" propaganda para cada página. Si bien esto es simple y utilizable, sería bueno emular lo que hacen muchos paquetes de blogs preconstruidos populares: mostrar una lista de entradas recientes y enlaces recientes más abajo en la barra lateral para que los visitantes puedan encontrar contenido nuevo rápidamente.

Pero eso plantea un dilema: parece que necesitaría volver y reescribir cada una de sus vistas para consultar también, por ejemplo, las últimas cinco entradas y los últimos cinco enlaces, y luego ponerlos a disposición de la plantilla. Eso sería terriblemente engoroso y repetitivo, y sería aún peor si alguna vez quisiera cambiar la cantidad de elementos recientes que se muestran o agregar nuevos tipos de contenido a su blog. Una vez más, parece que Django debería proporcionar una manera fácil de manejar esto sin mucho código repetitivo.

Y lo hace De hecho, Django proporciona dos formas sencillas de hacerlo. Uno es un mecanismo para escribir una función, llamada *procesador de contexto*, que puede agregar variables adicionales al contexto de cualquier plantilla. La otra forma es extender el sistema de plantillas de Django para agregar la capacidad de obtener contenido reciente usando una etiqueta de plantilla personalizada. Usando este enfoque, podría simplemente usar la etiqueta adecuada en la plantilla `base.html`, y todas las demás plantillas la tendrían automáticamente, cortesía de la herencia de plantilla.

Para esta situación, sigamos adelante y usemos una etiqueta de plantilla personalizada para tener una idea de cómo puede extender el sistema de plantillas de Django cuando necesite agregarle nuevas características.

Advertencia: Separación de preocupaciones

Lo que está a punto de hacer (escribir una etiqueta de plantilla que recupere elementos de la base de datos para mostrarlos) puede parecer extraño, teniendo en cuenta la limpieza con la que Django separa las funciones principales, como la recuperación de datos y la visualización de HTML, entre sí. Sin embargo, no siempre es malo desdibujar un poco esa distinción.

En este caso, desea recuperar estos elementos únicamente con fines de presentación. También desea que aparezcan en todas partes, por lo que escribir la funcionalidad como una extensión del sistema de plantillas de Django, que maneja la presentación del contenido, y aprovechar la herencia de plantillas es una buena manera de manejarlo. Sin embargo, no todo se hace mejor como una extensión del sistema de plantillas, por lo que debe evaluar decisiones como este caso por caso a medida que desarrolla.

Cómo funciona una plantilla de Django

Antes de que pueda sumergirse en la escritura de sus propias extensiones personalizadas para el sistema de plantillas de Django, debe comprender el mecanismo real detrás de esto. Saber cómo funcionan las cosas "debajo del capó" hace que el proceso de escribir la funcionalidad de la plantilla personalizada sea mucho más simple.

El proceso por el que pasa Django al cargar una plantilla funciona, más o menos, así:

1. **Lea el contenido real de la plantilla:** en la mayoría de los casos, esto significa leer un archivo de plantilla en el disco, pero no siempre es así. Django puede funcionar con cualquier cosa que entregue una cadena que contenga el contenido que desea que se trate como una plantilla.
2. **Analice la plantilla, buscando etiquetas y variables:** cada etiqueta en la plantilla, incluidas todas las etiquetas integradas de Django, corresponderá a una función particular de Python definida en alguna parte (dentro de django/template/defaulttags.py en el caso de las etiquetas integradas). Verá en un momento cómo decirle a Django que una etiqueta en particular se asigne a una función en particular. Por lo general, esta función se denomina *función de compilación* de la etiqueta.
porque se llama mientras Django compila una lista de los eventuales contenidos de la plantilla.
3. **Para cada etiqueta, llame a la función adecuada, pasando dos argumentos:** un argumento es la clase de análisis que está leyendo la plantilla (útil para hacer cosas complicadas con la forma en que se procesa la plantilla), y el otro es una cadena que contiene el contenido de la etiqueta. Entonces, por ejemplo, la etiqueta `{% if foo %}` hace que Django pase una función (llamada `do_if()`, en la biblioteca de etiquetas predeterminada de Django) una instancia de la clase de análisis y un objeto que contiene el contenido de la etiqueta "if foo".
4. **Tome nota del valor de retorno de la función de Python llamada para cada etiqueta:** cada función debe devolver una instancia de una clase especial (`django.template.Node`) o una subclase de la misma, y elegir una subclase de `Node` apropiada. en función de la etiqueta en particular.

El resultado es una instancia de la clase `django.template.Template`, que contiene una lista de instancias de `Node` (o instancias de subclases de `Node`). Esta es la "cosa" real que se renderizará para producir la salida. Se requiere que cada nodo tenga un método llamado `render()`, que acepta una copia del contexto de la plantilla actual (el diccionario de variables disponibles para la plantilla) y devuelve una cadena. La salida de la plantilla proviene de la concatenación de esas cadenas.

Una etiqueta personalizada simple

Extender el sistema de plantillas de Django con una etiqueta de plantilla personalizada puede ser un poco complicado al principio, así que empecemos de manera simple. Escribirá una etiqueta que obtenga las últimas cinco entradas y las coloque en una variable de plantilla denominada `últimas_entradas`.

Para comenzar, deberá crear un lugar para que viva el código de esta etiqueta. En la aplicación `coltrane` directorio, agregue un nuevo directorio llamado `templatetags`. En eso, crea dos archivos vacíos: `__init__.py` (recuerde, esto es necesario para decirle a Python que un directorio contiene código de Python cargable) y `coltrane_tags.py`, que será el archivo donde vive su biblioteca de etiquetas de plantillas personalizadas.

A continuación, dentro de `coltrane_tags.py`, agregue un par de declaraciones de importación en la parte superior:

```
desde la plantilla de importación de django
desde coltrane.models entrada de importación
```

Escribir la función de compilación

La etiqueta personalizada se llamará `get_latest_entries`, de modo que en las plantillas eventualmente podrá hacer `{% get_latest_entries %}`, pero puede nombrar su función de compilación (y su clase `Node`) como desee. Sin embargo, generalmente es una buena idea darle a la función un nombre significativo para la etiqueta con la que va, así que llámela `do_latest_entries()`:

```
def do_latest_entries(analizador, token):
```

Los dos argumentos de esta función son el analizador de plantilla y un token. (No usará el analizador de plantillas ahora, pero en el Capítulo 10 escribirá una etiqueta que lo usa para implementar funciones más avanzadas). `token` es un objeto que representa parte de la plantilla que se está analizando. Tampoco necesitará eso todavía, pero más adelante en este capítulo, cuando amplíe la funcionalidad de esta etiqueta, la usará para calcular los argumentos pasados a la etiqueta desde la plantilla.

Lo único que se requiere que haga esta función es devolver una instancia de `django.template.Node`. Definirá el nodo para esta etiqueta en un momento, pero se llamará `LatestEntriesNode`, así que continúe y rellénelo:

```
def do_latest_entries(analizador, token):
    devolver LatestEntriesNode()
```

Escribir el nodo

A continuación, debe escribir la clase `LatestEntriesNode`. Esta debe ser una subclase de `django.template.Node` y debe tener un método llamado `render()`. Django impone dos requisitos a este método:

- Debe aceptar un contexto de plantilla: el diccionario de variables disponibles para el elemento `plato`—como argumento.
- Debe devolver una cadena, incluso si la cadena no contiene nada. Para una etiqueta que produce su salida directamente, la cadena devuelta es el mecanismo por el cual la salida llega a la salida de la plantilla final.

Entonces puede comenzar a escribir su Nodo de la siguiente manera:

```
clase LatestEntriesNode(template.Node):
    def render(auto, contexto):
```

Esta etiqueta simplemente buscará las cinco entradas más recientes y las agregará al contexto como la variable últimas entradas, por lo que no tiene ningún resultado directo. Todo lo que hace es agregar el nuevo elemento al diccionario de contexto y luego devolver una cadena vacía:

```
clase LatestEntriesNode(template.Node):
    def render(auto, contexto):
        contexto['últimas_entradas'] = Entrada.live.all()[:5]
        devolver
```

Tenga en cuenta que incluso cuando una etiqueta no genera nada directamente, el método render() de su nodo *debe* devolver una cadena.

Registro de la nueva etiqueta

Finalmente, debe decirle a Django que la función de compilación debe usarse cuando se encuentra la etiqueta `{% get_latest_entries %}` en una plantilla. Para hacer esto, crea una nueva biblioteca de etiquetas de plantilla y registra su función con ella, así:

```
registro = plantilla.Biblioteca()
registrarse.tag('obtener_últimas_entradas', hacer_últimas_entradas)
```

La sintaxis para esto es simple. Una vez que crea una nueva biblioteca, simplemente llama a su método tag () y pase el nombre que desea darle a su etiqueta y la función que la manejará.

Así es como se ve ahora el archivo completo coltrane_tags.py:

```
desde la plantilla de importación de django
desde coltrane.models Entrada de importación

def do_latest_entries(analizador, token):
    devolver LatestEntriesNode()

clase LatestEntriesNode(template.Node):
    def render(auto, contexto):
        contexto['últimas_entradas'] = Entrada.live.all()[:5]
        devolver

registro = plantilla.Biblioteca()
registrarse.tag('obtener_últimas_entradas', hacer_últimas_entradas)
```

Uso de la nueva etiqueta

Ahora su nueva etiqueta está lista para usar. Abra la plantilla base.html y vaya a la parte de la barra lateral, que todavía se ve así:

```
<div id="barra lateral">
    <h2>Navegación</h2>
    <ul id="nav-principal">
        <li id="entradas-de-navegación-principal">
            <a href="{% url coltrane_entry_archive_index %}">Entradas</a>
        </li>
```

```
<li id="principal-nav-enlaces">
    <a href="{% url coltrane_link_archive_index %}">Enlaces</a>
</li>

<li id="categorías-de-navegación-principal">
    <a href="{% url coltrane_category_list %}">Categorías</a>
</li>

<li id="etiquetas-de-navegación-principales">
    <a href="{% url coltrane_tag_list %}">Etiquetas</a>
</li>
</ul>

<h2>¿Qué es esto?</h2>

{%
    block "bloquear lo que es"
    block "bloque final"
%}
</div>
```

Ahora agregue la lista de las últimas entradas justo debajo de "¿Qué es esto?" bloquear:

```
{% cargar coltrane_tags%}
<h2>Últimas entradas en el weblog</h2>
<ul>
    {% get_latest_entries%}
    {% para entrada en últimas_entradas%}
    <li>
        <a href="{{ entrada.get_absolute_url }}>{{ entrada.título }}</a>,
        publicado hace {{entry.pub_date|timesince}}.
    </li>
    {% endfor%}
</ul>
```

Esto es lo que está pasando:

- La etiqueta `{% load coltrane_tags %}` le dice a Django que desea cargar una biblioteca de plantilla personalizada llamada `coltrane_tags`. Cuando Django vea esto, buscará en todas sus aplicaciones instaladas un directorio `templatetags` que contenga un archivo llamado `coltrane_tags.py`, y cargará las etiquetas definidas allí.
- Una vez que se haya cargado su biblioteca de etiquetas, se puede llamar a la etiqueta `{% get_latest_entries %}`. Esta etiqueta crea la nueva variable de plantilla `últimas_entradas`, que contiene las cinco entradas más recientes.
- Luego, simplemente recorre las `últimas_entradas` usando la etiqueta `{% para %}`, mostrando un enlace a cada una y mostrando cuándo se publicó. Aquí está usando un nuevo filtro llamado `timesince`. Integrado en Django, este filtro da formato a una fecha y hora según hace cuánto tiempo ocurrió algo. El resultado (con la palabra "hace" añadida después) será algo así como "hace 3 días, 10 horas" y le dará al visitante una idea de la última vez que se actualizó el blog.

Escribir una etiqueta más flexible con argumentos

Ahora, también desea mostrar los últimos enlaces publicados en el blog. Puede hacer esto escribiendo una nueva etiqueta `{% get_latest_links %}` y haciendo que agregue una variable `Latest_links` a la plantilla.

contexto. Sin embargo, ese es el comienzo de un camino largo y tedioso de escribir una nueva etiqueta cada vez que agrega un nuevo tipo de contenido a su sitio, por lo que sería mejor convertir su etiqueta existente `{% get_latest_entries %}` en una etiqueta un poco más genérica `{ % get_latest_content %}`, que puede obtener cualquiera de varios tipos de contenido.

Y mientras está en eso, sería bueno agregar un poco más de flexibilidad a la etiqueta al permitirle tomar argumentos para especificar cuántos elementos recuperar, así como el nombre de la variable para colocarlos. De esa manera, podría tener varias listas de contenido reciente que no pisoteen las variables de los demás. Con lo que vas a terminar es una etiqueta que funciona así:

```
{% get_latest_content coltrane.link 5 como últimos_enlaces%}
```

que, como indica la sintaxis, recuperará los cinco objetos Link publicados más recientemente en la aplicación coltrane y los colocará en una variable de plantilla denominada `last_links`.

Escribir la función de compilación para la nueva etiqueta

Puede comenzar de la misma manera que lo hizo con la primera versión de la etiqueta personalizada. Es decir, defina una función de compilación para su nueva etiqueta:

```
def do_latest_content(analizador, token):
```

Pero ahora necesitarás leer algunos argumentos. El contenido completo, que reside en `token.contents`, será una cadena de la forma `get_latest_content coltrane.link 5 as latest_links`. Por lo tanto, puede usar la función de división de cadenas incorporada de Python, que por defecto se divide en espacios, para convertir la cadena en una lista de argumentos:

```
def do_latest_content(analizador, token):
    bits = token.contenido.split()
```

O puede usar `split_contents`, un método del objeto `token` que sabe cómo dividir los argumentos. Este método funciona de forma muy similar al método `split()` de Python, pero sabe cómo tener en cuenta algunos casos especiales:

```
def do_latest_content(analizador, token):
    bits = token.split_contents()
```

Ahora los bits de la variable deben contener una lista similar a `["get_latest_content", "coltrane.link", "5", "as", "latest_links"]`. Debido a que son cinco argumentos en total, puede verificar la longitud de los bits y generar un error de sintaxis de plantilla si no encuentra la cantidad correcta de argumentos:

```
def do_latest_content(analizador, token):
    bits = token.split_contents()
    si len(bits) != 5:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
etiqueta toma exactamente cuatro argumentos")
```

Esto garantiza que nunca intente hacer un uso incorrecto de la etiqueta. Tenga en cuenta que el error de sintaxis dice "cuatro argumentos", no "cinco argumentos". Aunque `bits` tiene cinco elementos, el primer elemento es el nombre con el que se llamó a la etiqueta, no un argumento. (A veces es útil escribir una sola función de compilación y registrarla varias veces con diferentes nombres, lo que le permite representar una familia de etiquetas similares y diferenciarlas por el nombre de la etiqueta que recibe).

A continuación, desea devolver un Node. Se llamará LatestContentNode y deberá pasar algo de información: el modelo del que recuperar contenido, la cantidad de elementos que recuperar y el nombre de la variable para almacenar los resultados. Cuando escriba LatestContentNode en un momento, configurará su constructor para aceptar esta información:

```
def do_latest_content(analizador, token):
    bits = token.split_contents()
    si len(bits) != 5:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
etiqueta toma exactamente cuatro argumentos")
    devuelve LatestContentNode(bits[1], bits[2], bits[4])
```

Tenga en cuenta que debido a que las listas de Python tienen índices que comienzan en 0, el nombre del modelo, aunque es el segundo elemento en bits es bits[1], el número de elementos es bits[2], y así sucesivamente.

Advertencia: ¿Cuánta verificación de errores es demasiado?

También puede agregar una prueba para asegurarse de que el cuarto elemento en bits sea la palabra "como" y generar un error de sintaxis si no lo ve. Pero en este caso, está bien no hacerlo. Para una etiqueta simple como esta, simplemente verificar la cantidad de argumentos suele estar bien, y verificar el "como" solo agregaría más código que probablemente no será necesario. Sin embargo, para etiquetas más complejas, es una buena idea escribir su función de compilación para asegurarse de que la etiqueta se usó correctamente antes de intentar devolver algo de ella.

Ahora debe determinar el modelo del que recuperar el contenido. En la etiqueta original {`%get_latest_entries %`}, simplemente importó el modelo de entrada y lo refirió directamente. Sin embargo, su nueva etiqueta tendrá un argumento como `coltrane.link` o `coltrane.entrada`, por lo que deberá importar la clase de modelo correcta dinámicamente.

Python proporciona una manera de hacer esto a través de una función integrada especial llamada `__import__()`, que toma cadenas como argumentos. Pero cargar una clase de modelo de forma dinámica es una necesidad lo suficientemente común como para que Django proporcione una función de ayuda para manejarla de manera más concisa. Esta función es `django.db.models.get_model()`, y toma dos argumentos:

- El nombre de la aplicación en la que se define el modelo, como una cadena
- El nombre de la clase modelo, como una cadena

Es costumbre poner ambas cadenas completamente en minúsculas porque Django mantiene un registro de modelos instalados con los nombres normalizados en minúsculas. Si lo desea, puede pasar nombres en mayúsculas y minúsculas a `get_model()`, pero debido a que de todos modos estarán en minúsculas, a menudo es más fácil comenzar con ellos de esa manera.

Para ver cómo funciona `get_model()`, vaya al directorio de su proyecto y ejecute el comando `python administrar.py shell`. Esto iniciará un intérprete de Python. En él escribe lo siguiente:

```
>>> desde django.db.models import get_model
>>> modelo_entrada = get_model('coltrane', 'entrada')
```

La función `get_model()` recuperará el modelo `Entry` de la aplicación `coltrane` y lo asignará a la variable `entry_model`. A partir de ahí, puede realizar consultas de la misma manera que lo haría si lo hubiera importado normalmente. Para verificar esto, escriba lo siguiente en el intérprete:

```
>>> modelo_entrada.en vivo.todo()[:5]
```

Verá que esto devuelve las últimas cinco entradas en vivo.

Avancemos y cambiemos la función de compilación `do_latest_content` para usar la función `get_model()` y recuperar la clase modelo. Una forma obvia de hacer esto sería como sigue:

```
de django.db.models importar get_model

def do_latest_content(analizador, token):
    bits = token.split_contents()
    si len(bits) != 5:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
etiqueta toma exactamente cuatro argumentos")
        argumentos_modelo = bits[1].split('.')
        modelo = get_model(model_args[0], model_args[1])
        devuelve LatestContentNode(modelo, bits[2], bits[4])
```

Sin embargo, este código tiene un par de problemas:

- Si el primer argumento no es un par de nombre de aplicación/nombre de modelo separados por un punto (.), o si tiene muy pocas o demasiadas partes, este código podría recuperar el modelo incorrecto o ningún modelo.
- Si los argumentos que pasa a `get_model()` no corresponden realmente a ninguna clase de modelo, `get_model()` devolverá el valor Ninguno, y eso hará que el `LatestContentNode` se tropiece cuando intenta recuperar el contenido.

Así que necesitas un poco de comprobación de errores. Quiere verificar lo siguiente:

- Cuando se divide en el carácter de punto (.), el primer argumento se convierte en una lista de exactamente dos elementos.
- Estos elementos, cuando se pasan a `get_model()`, de hecho devuelven una clase modelo.

Puedes hacerlo en solo unas pocas líneas de código:

```
argumentos_modelo = bits[1].split('.')
si len(argumentos_del_modelo) != 2:
    raise template.TemplateSyntaxError("Primer argumento para'ÿ
'get_latest_content' debe ser una cadena de 'nombre de la aplicación'. 'nombre del modelo'")
    modelo = get_model(*model_args)
    si el modelo es Ninguno:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
la etiqueta obtuvo un modelo no válido: %s" % bits[1])
```

Si te estás preguntando acerca de esta línea:

```
modelo = get_model(*model_args)
```

recuerde que el asterisco (*) es una sintaxis especial de Python para tomar una lista (el resultado de llamar a `split()`) y convertir un conjunto de argumentos en una función. Aquí está la función de compilación terminada:

```
def do_latest_content(analizador, token):
    bits = token.split_contents()
    si len(bits) != 5:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
etiqueta toma exactamente cuatro argumentos")
        argumentos_modelo = bits[1].split('.')
        si len(argumentos_del_modelo) != 2:
            raise template.TemplateSyntaxError("Primer argumento paraÿ
'get_latest_content' debe ser una cadena de 'nombre de la aplicación'. 'nombre del modelo'")
        modelo = get_model(*model_args)
        si el modelo es Ninguno:
            subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
la etiqueta obtuvo un modelo no válido: %s" % bits[1])
        devuelve LatestContentNode(modelo, bits[2], bits[4])
```

Escribiendo el LatestContentNode

Ya sabes que `LatestContentNode` necesita aceptar tres argumentos en su constructor:

- El modelo para recuperar elementos de
- El número de elementos a recuperar
- El nombre de una variable para almacenar los elementos en

Entonces puede comenzar escribiendo su constructor (recuerde que el constructor de un objeto de Python es siempre llamado `__init__()`) y simplemente almacenando esos argumentos como variables de instancia:

```
clase LatestContentNode(template.Node):
    def __init__(self, modelo, num, varname):
        self.modelo = modelo
        self.num = int(num)
        self.varname = varname
```

Tenga en cuenta que obliga a `num` a ser un `int` aquí. Todos los argumentos de la etiqueta llegaron como cadenas, por lo tanto, antes de que pueda usar `num` para controlar la cantidad de elementos que desea recuperar, debe convertirlo en un número real. Aquí hay una manera simple de escribir el método `render()` para lograr eso:

```
def render(auto, contexto):
    contexto[self.varname] = self.modelo.objects.all()[:self.num]
    devolver
```

Al principio, esto se ve bien, pero tiene un problema oculto. Cuando llamas a la etiqueta de la plantilla de esta manera:

```
{% get_latest_content coltrane.entry 5 como últimas_entradas %}
```

la consulta que realiza será el equivalente a

```
Entrada.objetos.todos()[:5]
```

que no es lo que quieres. Esto devolverá *cualquier* entrada, incluidas las entradas que no están destinadas a mostrarse públicamente. En su lugar, desea que haga el equivalente de lo siguiente:

```
Entry.live.all()[:5]
```

Podría escribir un código de caso especial para ver cuándo está trabajando con el modelo Entry, pero esa no es una buena práctica. Si luego necesita usar esta etiqueta en otros modelos con necesidades similares, deberá seguir agregando nuevas piezas de código de casos especiales.

La solución es pedirle a Django que use el administrador predeterminado del modelo. El primer administrador definido en una clase de modelo recibe un estado especial. Se convierte en el administrador predeterminado para ese modelo, además del nombre con el que se definió. También estará disponible como atributo `_predeterminado_` administrador, por lo que en realidad puede escribir esto como:

```
def render(auto, contexto):
    contexto[self.varname] = self.modelo._default_manager.all()[:self.num]
    devolver
```

Debido a que el administrador en vivo se definió primero en el modelo de entrada, esto hará lo correcto.

Advertencia: uso de administradores predeterminados

Siempre que no sepa de antemano con qué modelo trabajará (como en este caso, y en la mayoría de los casos cuando usa `get_model()`), es una buena idea usar `_default_manager`. Cuando un modelo tiene múltiples administradores, o define un solo administrador personalizado que no tiene nombres de objetos, tratar de consultar a través del atributo de objetos puede ser peligroso. Es posible que ese no sea el administrador por el que deben pasar las consultas (como en el caso de Entry con su administrador en vivo especial) y, de hecho, es posible que los objetos ni siquiera existan. Recuerde que cuando un modelo tiene un administrador personalizado, Django no configura automáticamente el administrador de objetos en él, por lo que intentar acceder a los objetos podría generar una excepción.

Registro y uso de la nueva etiqueta

Ahora simplemente puede registrar su nueva etiqueta y está lista para funcionar. El archivo final `coltrane_tags.py` se ve así:

```
de django.db.models importar get_model
desde la plantilla de importación de django

def do_latest_content(analizador, token):
    bits = token.split_contents()
    si len(bits) != 5:
        subir plantilla.TemplateSyntaxError("get_latest_content'ÿ
etiqueta toma exactamente cuatro argumentos")
    argumentos_modelo = bits[1].split('.)')
```

```
if len(model_args) != 2: raise
    template.TemplateSyntaxError("Primer argumento para 'get_latest_content' debe ser una cadena de 'nombre de la aplicación'. 'nombre del modelo'" modelo = get_model(*model_args) si el modelo es Ninguno: aumentar plantilla.TemplateSyntaxError("get_latest_content" tag obtuvo un modelo no válido: %s" % bits[1]) devuelve LatestContentNode(modelo, bits[2], bits[4])
```

```
clase LatestContentNode(template.Node):
    def __init__(self, modelo, num, varname):
        self.modelo = modelo
        self.num = int(num)
        self.varname = varname

    def render(auto, contexto):
        contexto[self.varname] = self.modelo._default_manager.all()[:self.num] volver

registro = plantilla.Librería()
registro.etiqueta('get_latest_content', do_latest_content)
```

Entonces puedes reescribir la barra lateral en la plantilla base.html, así:

```
<div id="barra lateral">
    <h2>Navegación</h2>
    <ul id="main-nav"> <li
        id="main-nav-entries">
            <a href="{% url coltrane_entry_archive_index %}">Entradas</a> </li> <li
        id="main-nav-links">
            <a href="{% url coltrane_link_archive_index %}">Enlaces</a> </li> <li
        id="main-nav-categories">
            <a href="{% url coltrane_category_list %}">Categorías</a> </li> <li
        id="main-nav-tags">
            <a href="{% url coltrane_tag_list %}">Etiquetas</a> </li>
        </ul> <h2>¿Qué es esto?</h2> {% block whatis %} {% endblock %} {% cargar coltrane_tags %} <h2>Últimas entradas en el weblog</h2> <ul> {% get_latest_content coltrane.entry 5 como últimas_entradas %} {% para entrada en últimas_entradas %}
```

```

<li>
    <a href="{{ entrada.get_absolute_url }}>{{ entrada.título }}</a>,
    publicado hace {{entry.pub_date|timesince}}.
</li>
{% endfor%}
</ul>
<h2>Últimos enlaces en el weblog</h2>
<ul>
    {% get_latest_content coltrane.link 5 como últimos_enlaces%}
    {% para enlace en últimos_enlaces%}
    <li>
        <a href="{{ enlace.get_absolute_url }}>{{ enlace.título }}</a>,
        publicado {{ link.pub_date|timesince }} hace.
    </li>
    {% endfor%}
</ul>
</div>

```

Esto asegurará que cada página tenga la lista de las últimas cinco entradas y enlaces, y ofrece dos grandes ventajas sobre la etiqueta original `{% get_latest_entries %}`:

- Cuando agregue nuevos tipos de contenido al blog (en el próximo capítulo agregará comentarios), no tienes que escribir una nueva etiqueta. Simplemente puede reutilizar `get_latest_content` con diferentes argumentos.
- Si decide cambiar el número de entradas o enlaces a mostrar, o las variables que quieras usar para ellos, solo es cuestión de enviar diferentes argumentos a la etiqueta `{% get_latest_content %}`. No tendrá que volver a escribir la etiqueta para cambiar esto.

Mirando hacia el futuro

En el próximo capítulo, concluirá el weblog agregando comentarios, moderación y fuentes RSS. Por ahora, sin embargo, siéntase libre de jugar con el sistema de plantillas y hacer que el blog se vea exactamente como lo desea. Se incluye una hoja de estilo de muestra que implementa el diseño de dos columnas con el código de muestra de este libro (descargable desde el sitio web de Apress), así que no dude en probarlo. Para que Django sirva esto como un archivo sin formato, agregue el siguiente patrón de URL en el módulo `URLConf` raíz del proyecto (una vez más usando la vista de servicio de archivos estáticos que vio en el Capítulo 3):

```
(r'^media/(?P<ruta>.*$',
'django.views.static.serve',
{'document_root': '/ruta/a/hoja de estilo/directorio'}),
```

Simplemente complete la ruta al directorio donde reside el archivo de hoja de estilo en su computadora, y Django lo servirá. (Aunque tenga en cuenta que para la implementación de producción de Django, es mejor no hacer que Django sirva archivos estáticos como este).

Capítulo 7



Terminando el blog

Ahora que tiene un conjunto sólido de plantillas y, lo que es más importante, una sólida comprensión de Sistema de plantillas de Django, es hora de terminar el weblog con las dos funciones finales: un sistema de comentarios con moderación y feeds de sindicación para entradas y enlaces.

Aunque Django proporciona aplicaciones—`django.contrib.comments` y `django.contrib.sitemaps`: que manejan la funcionalidad básica de estas dos funciones, irá un poco más allá, personalizando y ampliando sus funciones a medida que avanza. Esto implicará un poco de código de Python y un poco de plantillas, pero como verá, no es tanto el código que tendría que escribir para implementar estas características desde cero. Así que vamos a sumergirnos.

Comentarios y `django.contrib.comments`

Ya has visto que `django.contrib` contiene algunas aplicaciones útiles. Tanto la interfaz administrativa como el sistema de autenticación que utiliza provienen de aplicaciones en `contrib`, así como de la aplicación de páginas planas que utilizó en su CMS simple. En general, es una buena idea mirar allí antes de empezar a escribir algo por tu cuenta. Mientras escribo esto, `django.contrib` contiene 17 aplicaciones, y hay planes para expandirlo para incluir más aplicaciones de código abierto de la comunidad de Django. Incluso si algo en `contrib` no hace exactamente lo que necesita, a menudo encontrará algo que puede aumentar o algo que puede simplificar un poco de código complicado.

Comentar no es una excepción a esto. El sistema de comentarios de referencia en el que se basará se incluye como `django.contrib.comments`. Es compatible con las funciones básicas que necesitará para poner en marcha un sistema de comentarios y proporciona una base para crear funciones adicionales.

Implementación de herencia y resumen de modelos Modelos

Incluido en `django.contrib.comments` hay un par de modelos—`BaseCommentAbstractModel` y `Comment`—que representan un patrón útil en el desarrollo de Django: modelos abstractos con subclases concretas.

Hasta ahora, ha estado escribiendo modelos que son subclases de la clase de modelo básica incorporada de Django, pero Django también admite modelos que son subclases de otras clases de modelos. Le permite usar cualquiera de los dos patrones comunes cuando realiza tales subclases:

- **Herencia concreta:** Esto es lo que mucha gente piensa cuando imagina cómo funciona la subclaseificación de un modelo. En este patrón, un modelo que subclaseifica a otro creará una nueva tabla de base de datos que se vincula a la tabla de la clase "principal" original con una clave externa. Las instancias del modelo con subclases se comportarán como si tuvieran los campos definidos en el modelo "principal" y los campos definidos en el propio modelo con subclases (bajo el capó, Django extraerá información de ambas tablas según sea necesario).
- **Herencia abstracta:** cuando define una nueva clase de modelo y completa sus opciones usando la meta declaración de la clase interna, puede agregar el atributo `abstract=True`. Cuando usted haga esto, Django no creará una tabla para ese modelo y no podrá crear o consultar directamente instancias de ese modelo. Sin embargo, cualquier subclase de ese modelo (siempre y cuando no declaren `abstract=True`) creará sus propias tablas y también agregará columnas para los campos del modelo abstracto.

En otras palabras, la herencia concreta crea una tabla para cada modelo, como de costumbre. La herencia abstracta crea solo *una* tabla, la tabla para la subclase, y coloca todos los campos dentro de ella.

En general, la herencia concreta es útil cuando desea ampliar los campos o características de un modelo preexistente. La herencia abstracta, por otro lado, es útil cuando tiene un conjunto de campos o métodos comunes (o ambos) que le gustaría tener en múltiples modelos sin definirlos una y otra vez.

La aplicación de comentarios empaquetados de Django aprovecha la herencia abstracta para proveer un modelo básico, `BaseCommentAbstractModel`, que define un conjunto de campos comunes necesarios para casi cualquier tipo de comentario, y declara que es abstracto. También proporciona un segundo modelo, `Comentario`, que subclaseifica este modelo abstracto y lo desarrolla con un conjunto específico de características.

Instalación de la aplicación de comentarios

Instalar el sistema de comentarios es fácil. Abra el archivo de configuración de su proyecto Django (`settings.py`) y agregue la siguiente línea en la lista `INSTALLED_APPS`:

```
'django.contrib.comentarios',
```

A continuación, ejecute `python manage.py syncdb` y Django instalará sus modelos. Si inicia el servidor de desarrollo y visita la interfaz administrativa, verá una nueva sección de comentarios que enumera el modelo de comentarios. (Debido a que el modelo abstracto que subclaseifica no se puede instanciar o consultar directamente, no hay una interfaz de administración para él).

En el archivo `URLConf` raíz del proyecto (`urls.py`), agregue un nuevo patrón de URL:

```
(r'^comentarios/', include('django.contrib.comments.urls')),
```

Ya ha visto este patrón varias veces y, en general, este es el sello distintivo de un pozo. aplicación Django construida. Instalarlo no debería implicar más trabajo que el siguiente:

1. Agréguelo a INSTALLED_APPS y ejecute syncdb.
2. Agregue un nuevo patrón de URL para enrutar a su URLConf predeterminado.
3. Configure las plantillas necesarias.

Escribir una aplicación para que funcione de esta manera lista para usar es una técnica extremadamente poderosa porque permite que incluso sitios muy complejos se construyan rápidamente a partir de aplicaciones reutilizables, y cada una proporciona una funcionalidad particular. Tener este patrón en mente mientras escribe sus propias aplicaciones lo ayudará a producir aplicaciones útiles y de alta calidad. En el Capítulo 11, verá algunas técnicas para incorporar configurabilidad y flexibilidad más allá de este estilo de configuración básica.

Realización de la configuración básica

Para comenzar con la aplicación de comentarios, deberá mostrar un formulario de comentarios para que lo completen los visitantes. Comencemos con eso.

Abra la plantilla de detalles de entrada, coltrane/entry_detail.html, y vaya al bloque de contenido principal, que tiene este aspecto:

```
{% contenido del bloque %}  
<h2>{{ objeto.título }}</h2>  
{{ objeto.cuerpo_html|seguro }}  
{% bloque final %}
```

Continúe y agregue un encabezado que distinguirá el formulario de comentarios:

```
{% contenido del bloque %}  
<h2>{{ objeto.título }}</h2>  
{{ objeto.cuerpo_html|seguro }}  
  
<h2>Publicar un comentario</h2>  
  
{% bloque final %}
```

Ahora solo necesitas mostrar el formulario. El sistema de comentarios incluye una plantilla personalizada biblioteca de etiquetas que, entre otras cosas, puede hacer eso por usted. La biblioteca de etiquetas se llama comentarios, por lo que deberá cargarla con la etiqueta {% load %}:

```
{% contenido del bloque %}  
<h2>{{ objeto.título }}</h2>  
{{ objeto.cuerpo_html|seguro }}  
  
<h2>Publicar un comentario</h2>  
  
{% cargar comentarios%}  
  
{% bloque final %}
```

Ahora, la etiqueta que desea se llama `{% render_comment_form %}`, y su sintaxis se ve así:

```
{% render_comment_form para objeto %}
```

En otras palabras, esta etiqueta solo quiere una variable que contenga el objeto específico al que se adjuntará el comentario, que estará disponible en la plantilla `entry_detail` como la variable `{{ object }}`. Entonces puedes completar la etiqueta de esta manera:

```
{% contenido del bloque %}
```

```
<h2>{{ objeto.título }}</h2>
```

```
{{ objeto.cuerpo_html|seguro }}
```

```
<h2>Publicar un comentario</h2>
```

```
{% cargar comentarios %}
```

```
{% render_comment_form para objeto %}
```

```
{% bloque final %}
```

Tenga en cuenta que aquí *no* coloca las llaves alrededor del objeto. Las llaves, como en `{{ object }}`, se usan solo cuando desea generar el valor de la variable. No son necesarios en una etiqueta de plantilla y, de hecho, provocarán un error. Las etiquetas de plantilla pueden resolver variables por sí mismas (como verá en el Capítulo 10 cuando escriba algunas etiquetas que hacen eso).

Ahora, ve a visitar una entrada y verás que aparece el formulario de comentarios. Si completa un comentario y haga clic en el botón Vista previa, verá una vista previa de su comentario, que se muestra a través de una plantilla predeterminada incluida con Django. De hecho, `django.contrib.comments` incluye suficientes plantillas predeterminadas básicas para respaldar todo lo que hará por ahora; se incluyen plantillas para obtener una vista previa y publicar comentarios, y también para funciones más avanzadas como la moderación de comentarios.

Pero cuando comience a implementar aplicaciones Django en vivo, querrá personalizar estas plantillas para que coincidan con el diseño de su sitio. Las plantillas predeterminadas se incluyen con los comentarios `application`, y residir en el directorio `contrib/comments/templates/comments` dentro de su copia de Django.

Para tener una idea de cómo funcionará esta personalización, cree un nuevo directorio llamado `comentarios` dentro del directorio de plantillas de su proyecto y copie la plantilla `preview.html` de la aplicación de comentarios de Django en este directorio.

La mayor parte del contenido de esta plantilla se refiere a mostrar el formulario de comentarios y cualquier problema de envío. Por ejemplo, algunos campos del formulario son obligatorios y esta plantilla mostrará un mensaje de error si se dejan en blanco (obtendrá más información sobre el sistema de gestión de formularios de Django en el Capítulo 9). Sin embargo, hay una sección que muestra la vista previa real del comentario (si no hubo errores en el formulario). Se parece a esto:

```
<h1>{% trans "Vista previa de tu comentario" %}</h1>
<blockquote>{{ comentario|saltos de línea}}</blockquote>
<p>
  {% trans "y" %} <input type="submit" value="Enviar" name="enviar" class="enviar-publicar" />
  {% trans "Publique su comentario" %} <input id="enviar" type="button" value="Publicar" /> {% trans "o realice cambios" %}:
</p>
```

Hay una etiqueta desconocida aquí, trans, pero no tendrás que preocuparte por eso todavía. Django incluye lo que se conoce como funciones de "internacionalización", que permiten marcar fragmentos de texto para traducirlos a otros idiomas. Si hay traducciones disponibles y el navegador web de un visitante indica un idioma preferido, se sustituirán automáticamente. La etiqueta trans realiza esta función para las plantillas.

El contenido del comentario real se muestra a través de un filtro llamado saltos de línea, que simplemente traduce los saltos de línea en el texto del comentario en etiquetas de párrafo HTML. Debido a que ya está usando Markdown para procesar el contenido del weblog, sería bueno permitir que los visitantes también lo usen para sus comentarios. Django proporciona un filtro de plantilla incorporado que puede manejar esto.

Para habilitar el filtro, deberá agregar una entrada más a su configuración INSTALLED_APPS: django.contrib.markup, que contiene herramientas para trabajar con sistemas comunes de traducción de texto a HTML (incluido Markdown). No necesitará ejecutar syncdb, porque esta aplicación no proporciona modelos; solo necesita enumerar la aplicación en INSTALLED_APPS para que Django le permita usar los filtros de plantilla que proporciona.

Una vez que haya hecho eso, cambie su copia de la plantilla preview.html para que la parte que muestra el contenido del comentario se vea así:

```
{% carga de marcado %}  
<blockquote>{{ comentario|markdown: "seguro" }}</blockquote>
```

Esto aplicará Markdown al contenido del comentario y también habilitará el "modo seguro" de Markdown, que elimina cualquier etiqueta HTML sin procesar del comentario antes de generar el HTML final para mostrar. Esto es importante porque el escape automático normal de Django no se aplicará con este filtro; el filtro de rebajas está *destinado* a devolver HTML, por lo que deshabilita el escape automático para su salida. El uso del modo seguro significa que cualquier HTML malicioso que un usuario intente enviar se eliminará y no provocará una violación de la seguridad de su sitio.

Recuperación de listas de comentarios para mostrar

Todo lo que necesita hacer ahora es recuperar los comentarios y mostrarlos. Al igual que django.contrib.comments proporciona una etiqueta de plantilla personalizada para mostrar el formulario de comentarios, proporciona una etiqueta que puede manejar la recuperación de comentarios. La sintaxis para esto se ve así:

```
{% get_comment_list para objeto como comment_list %}
```

Entonces puede usar la etiqueta en su plantilla entry_detail.html de esta manera:

```
<h2>Comentarios</h2>  
{% carga de marcado %}  
{% get_comment_list para objeto como comment_list %}  
  
{% para comentario en comment_list %}  
  
<p>El {{ comment.submit_date|date:"F j, Y" }},  
{{ comentario.nombre }} dijo:</p>  
  
{{ comentario.comentario|rebaja:"seguro" }}  
{% endfor%}
```

El modelo de comentarios predeterminado de Django configura automáticamente el nombre del atributo para devolver la aprobación. valor práctico; si el comentario fue publicado por un usuario registrado, el nombre será el nombre de usuario de ese usuario. Si el comentario fue publicado por alguien que no inició sesión, el nombre será el nombre que esa persona proporcionó en el formulario de comentarios.

Entonces, el bloque de contenido completo de su plantilla entry_detail.html ahora se ve así:

```
{% contenido del bloque %}  
<h2>{{ objeto.título }}</h2>  
{{ objeto.cuerpo_html|seguro }}  
  
<h2>Comentarios</h2>  
{% cargar comentarios%}  
{% carga de marcado %}  
{% get_comment_list para objeto como comment_list %}  
  
{% para comentario en comment_list %}  
  
<p>El {{ comment.submit_date|date:"F j, Y" }},  
{{ comentario.nombre }} dijo:</p>  
  
{{ comentario.comentario|rebaja:"seguro" }}  
{% endfor%}  
  
<h2>Publicar un comentario</h2>  
  
{% render_comment_form para objeto %}  
  
{% bloque final %}
```

Si desea agregar una línea en la barra lateral para mostrar la cantidad de comentarios en la entrada, el La etiqueta get_comment_count lo recuperará por usted. Podrías usarlo así:

```
{% cargar comentarios%}  
{% get_comment_count para objeto como comment_count %}  
  
<p>Hasta ahora, esta entrada tiene {{ comment_count }}  
comentario{{ comment_count|pluralize }}.</p>
```

Advertencia: el alcance de la etiqueta {% load %}

Debido a la forma en que funciona la herencia de plantillas de Django, una etiqueta personalizada o una biblioteca de filtros cargada a través de la etiqueta {% load %} estará disponible solo en el bloque en el que se cargó. Si necesita reutilizar la misma biblioteca de etiquetas en un bloque diferente, deberá cargarla nuevamente.

Moderación de comentarios

Fuera de la caja, Django cubre la mayor parte de lo que desea para comentar: una manera fácil de permitir que los visitantes publiquen comentarios y luego extraer una lista de los comentarios que están "adjuntos" a un objeto en particular. Pero dada la proliferación de spam de comentarios en la web en los últimos años, es Todavía va a querer algún tipo de sistema de moderación automática para filtrar los comentarios entrantes. Para eso, necesitarás escribir algo de código.

Ambos modelos de comentarios en django.contrib.comments definen un BooleanField llamado `is_public`, y eso es lo que debería usar un sistema de moderación. Ahora, hay un par de formas muy efectivas de filtrar comentarios no deseados:

- Cada vez que se publica un comentario en una entrada que es más de un cierto número de días edad (por ejemplo, 30), márquelo automáticamente como no público. La gran mayoría del spam de comentarios tiene contenido antiguo, en parte porque la mayoría del contenido es antiguo y en parte porque es menos probable que el administrador del sitio lo note.
- Usar un sistema estadístico de detección de spam. Akismet (<http://akismet.com/>) es el oro estándar para esto, con un historial de más de cinco mil millones de comentarios de spam para analizar. Lo mejor de todo es que Akismet ofrece una API basada en web que estima si un comentario es spam o no.

En mi blog personal, recibo alrededor de seis mil comentarios de spam al mes. la combinación La aplicación de estas dos técnicas de filtrado ha impedido, hasta ahora, que todos menos uno o dos de ellos se muestren públicamente.

Entonces, desea encontrar alguna forma de conectarse al sistema de envío de comentarios y auto aplique automáticamente las dos técnicas de filtrado para establecer `is_public=False` en el nuevo comentario cada vez que parezca que será spam. Hay un par de maneras obvias de hacer esto:

- Así como ha definido un método `save()` personalizado en algunos de nuestros propios modelos, podría ir a los modelos de comentarios en Django y editarlos para incluir un `save()` personalizado método que hace el filtrado de spam.
- Puede editar o reemplazar la vista que maneja el envío de comentarios y poner el filtrado de spam allí.

Pero ambos métodos tienen importantes inconvenientes. O está editando el código que viene con Django (lo que dificultará la actualización en el futuro y podría causar problemas de depuración porque tendrá una base de código de Django no estándar), o está duplicando el código que Django ya ha proporcionado para añadir una pequeña modificación.

¿No sería bueno si pudiera escribir algo de su propio código y luego conectarlo Django de alguna manera para asegurarse de que se ejecuta en el momento adecuado?

Uso de señales y el despachador de Django

Resulta que hay una manera de hacerlo. Django incluye un módulo llamado `django.dispatch` que proporciona dos cosas:

- Una forma de anunciar cualquier pieza de código en Django, o en una de sus propias aplicaciones. el hecho de que algo sucedió
- Una forma para que cualquier otra pieza de código "escuche" un evento específico que sucede y tome alguna acción en respuesta

La forma en que esto funciona es bastante simple: django.dispatch proporciona una clase llamada Signal, que representa la ocurrencia de algún evento. Cada instancia de Signal tiene dos métodos importantes:

- enviar: Llamar a este método significa "este evento ha ocurrido".
- conectar: llamar a este método le permite registrar una función que se llamará cada vez que se "envíe" la señal.

Para ver un ejemplo simple, vaya al directorio del proyecto cms e inicie un intérprete de Python al escribiendo **python manage.py shell**. Luego escribe lo siguiente:

```
>>> from coltrane.models import Entrada
>>> desde django.db.models.signals importar post_save
>>> def print_save_message(remitente, instancia, **kwargs):
...     print "¡Se acaba de guardar una entrada!"
>>> post_save.connect(print_save_message, remitente=Entrada)
```

Ahora, busque una entrada y guárdela:

```
>>> e = Entrada.objetos.todos()[0]
>>> e.guardar()
```

Su intérprete de Python imprimirá de repente "¡Se acaba de guardar una entrada!" Esto es lo que sucedió:

1. Importó el despachador y una instancia de Signal, definida en django.db.models.signals
2. Escribiste una función que imprime el mensaje. Los argumentos que recibe (remitente e instancia) acabarán siendo la clase del modelo Entry (que "enviará" la señal que está escuchando) y el objeto Entry específico que se guardará. No estás haciendo nada con estos argumentos, pero cuando construyas el sistema de moderación de comentarios, verás cómo se pueden usar. La función también acepta **kwargs, lo que indica que puede aceptar cualquier argumento de palabra clave. Esto es necesario porque diferentes señales proporcionan diferentes argumentos.
3. Registraste la función usando el método connect() de la señal, para que se llame cuando el modelo Entry envíe la señal post_save.
4. Cuando se guardó la entrada, el código dentro de Django, integrado en la clase de modelo base de la que heredan todos sus modelos, usó el método send() de la señal post_save para enviarla.
5. El despachador llamó a su función personalizada.

Django define alrededor de una docena de señales que puede usar de inmediato, y es fácil de definir y usar el tuyo también. También puede hacer algunos trucos con el despachador que son más complejos, pero lo que ha visto hasta ahora es todo lo que necesita para crear un moderador de comentarios efectivo.

Creación del moderador automático de comentarios

Para construir su sistema de moderación de comentarios, escribirá una función que sepa cómo mirar un comentario entrante y averiguar si es spam. Luego, usará el despachador para asegurarse de que se llame a la función cada vez que se vaya a guardar un nuevo comentario. Tal como lo usaste

la señal post_save en el ejemplo anterior, hay una señal pre_save que puede usar para ejecutar código antes de que se guarde un objeto.

Lo primero que desea hacer cuando recibe un nuevo comentario es mirar la entrada en la que se está publicando. Si esa entrada tiene más de, digamos, 30 días de antigüedad, simplemente establecerá su campo is_public en False y no molestar con más controles. Aquí es donde entra en juego el argumento de la instancia de su función personalizada. Desde el nuevo objeto de comentario que está a punto de guardarse, puede determinar la entrada en la que se está publicando. Así es como se ve el código:

```
importar fecha y hora
```

```
def moderado_comentario(remitente, instancia, **kwargs):
    si no instancia.id:
        entrada = instancia.content_object
        delta = fechahora.fechahora.ahora() - entrada.pub_fecha
        si delta.days > 30:
            instancia.is_public = Falso
```

Hasta ahora, esta función es bastante sencilla. Solo revisas las cosas si el comentario—que será el objeto en el argumento de la instancia; aún no tiene una identificación, lo que significa que no se ha guardado en la base de datos. Si tiene una identificación, presumiblemente ya ha sido verificada. Verificarlo nuevamente dificultaría que un administrador del sitio aprobara manualmente un comentario, ya que el comentario continuaría con este proceso y se marcaría como no público cada vez que se guardara.

Primero, usa el argumento de instancia para encontrar la entrada en la que se publica el comentario. El modelo de comentarios de Django tiene un atributo llamado content_object, que devuelve el objeto al que pertenece el comentario.

A continuación, resta pub_date de la entrada de la fecha y hora actuales. Fecha y hora de Python La clase está configurada para que esto funcione, y el resultado es una instancia de una clase llamada timedelta, que tiene atributos que representan el número de días, horas, etc. entre los dos objetos de fecha y hora involucrados.

A continuación, verifica el atributo de días en ese objeto timedelta. Si es mayor que 30, establece el campo is_public del nuevo comentario a False.

En este punto, ya podría conectar la función y haría un buen trabajo para prevenir el spam:

```
de django.contrib.comments.models import Comentario
de señales de importación django.db.models

señales.pre_save.connect(comentario_moderado, remitente=Comentario)
```

Agregar soporte de Akismet

Ahora agreguemos la segunda capa de prevención de spam: análisis estadístico de spam por el servicio web Akismet. Lo primero que necesitará es una clave API de Akismet: todo acceso al servicio de Akismet requiere esta clave. Afortunadamente, es gratis para uso personal, no comercial. Simplemente siga las instrucciones en el sitio web de Akismet (<http://akismet.com/personal/>) para obtener una clave. Una vez que lo tenga, abra el archivo de configuración de Django para el proyecto cms y agréguele la siguiente línea:

```
AKISMET_API_KEY = 'su clave API va aquí'
```

Al hacer de esta una configuración personalizada, podrá reutilizar el filtrado de spam de Akismet en otros sitios, incluso si tienen diferentes claves de API.

Akismet es un servicio basado en la web. Envías información sobre un comentario al servicio, mediante una solicitud HTTP y envía una respuesta HTTP que le indica si Akismet cree que el comentario es spam. Podría compilar el código necesario para hacer esto, pero, como encontrará a menudo cuando trabaja con Python, alguien más ya lo hizo y puso el código a disposición de forma gratuita.

En este caso, es un módulo llamado akismet, que está disponible del autor, Michael Foord, en su sitio web: www.voidspace.org.uk/python/akismet_python.html. Continúe, descárguelo y descomprímalo (debe venir en un archivo .zip). Esto le dará un archivo llamado akismet.py que puede colocar en su ruta de importación de Python (idealmente, en la misma ubicación que el directorio coltrane que contiene la aplicación de blog).

El módulo akismet incluye una clase llamada Akismet que maneja la API. Esta clase tiene usará dos métodos: uno llamado verificar_clave(), que garantiza que está usando una clave de API válida, y otro llamado comentario_verificar(), que envía un comentario a Akismet y devuelve Verdadero si Akismet cree que el comentario es spam.

Entonces, lo primero que deberá hacer es importar la clase Akismet:

```
de akismet importar Akismet
```

La API de Akismet requiere tanto la clave de API que se le asignó como la dirección del sitio desde el que envía el comentario. Podría codificar la URL de su sitio aquí, pero eso perjudicaría la reutilización del código. Una mejor opción es usar el marco de sitios empaquetados de Django (vive en django.contrib.sites), que proporciona un modelo que representa un sitio web en particular y sabe qué sitio está actualmente activo.

Recordará que en el Capítulo 2, cuando configuró el CMS simple, editó un Sitio objeto para que "supiera" dónde estaba ejecutando el servidor de desarrollo. siempre que estés ejecutando con esta base de datos y archivo de configuración, puede obtener ese objeto de sitio con lo siguiente:

```
del sitio de importación django.contrib.sites.models  
sitio_actual = Sitio.objetos.get_actual()
```

Esto funciona porque el modelo del sitio tiene un administrador personalizado que define get_current() método. El objeto Sitio que devuelve tiene un campo llamado dominio, que puede usar para completar la información que Akismet desea. Esta información es el argumento de palabra clave blog_url cuando está creando una instancia de la API (junto con la clave de API, que proviene de su archivo de configuración y es la clave del argumento de palabra clave):

```
desde la configuración de importación de django.conf  
del sitio de importación django.contrib.sites.models
```

```
akismet_api = Akismet(clave=configuración.AKISMET_API_KEY,  
blog_url="http://%s/" %Sitio.objetos.get_actual().dominio)
```

Luego puede verificar su clave API con el método verificar_clave(). Si es válido, puede enviar un comentario para su análisis con el método comment_check(). El método comment_check() espera tres argumentos:

- El texto del comentario a comprobar
- Algunos “metadatos” adicionales sobre el comentario, en un diccionario
- Un argumento booleano (Verdadero o Falso) que le dice si debe tratar de resolver metadatos por sí solos

El texto del comentario es bastante fácil de obtener, porque es un campo en el comentario mismo. El diccionario de metadatos debe tener al menos cuatro valores, incluso si algunos de ellos están en blanco (porque no necesariamente sabe cuáles son). Estos valores son el tipo de comentario (que, para usos simples como este, es simplemente el comentario de cadena), el valor del encabezado HTTP Referer, la dirección IP desde la que se envió el comentario (también un campo en el modelo de comentarios) y el agente de usuario HTTP del comentarista. Finalmente, le indicará al módulo akismet que continúe y resuelva cualquier metadato adicional que pueda encontrar. Más información significa mayor precisión, especialmente porque el módulo akismet puede, bajo algunas configuraciones de servidor, encontrar información útil automáticamente. El código se ve así (el método comment_check() de Akismet devuelve True si cree que el comentario es spam):

```
de django.utils.encoding import smart_str

si akismet_api.verify_key():
    akismet_data = { 'tipo_comentario': 'comentario',
                     'referente': '',
                     'ip_usuario': instancia.dirección_ip,
                     'agente de usuario': '' }
    si akismet_api.comment_check(smart_str(instancia.comentario),
                                  akismet_data,
                                  build_data=Verdadero):
        instancia.is_public = Falso
```

Recuerde que Django usa cadenas Unicode en todas partes, por lo que siempre que use una API externa, debe convertir cadenas Unicode en cadenas de bytes usando la función auxiliar django.utils.encoding.smart_str().

Pero aquí hay un problema: no conoce los valores de HTTP Referer y User-Agent encabezados. Aunque no son obligatorios, estos valores pueden ayudar a Akismet a determinar con mayor precisión si un comentario es spam. Afortunadamente, hay una manera de obtener esos valores.

Hasta ahora, solo ha estado usando las señales estándar, pre_save y post_save, enviadas por Django cada vez que se guarda un modelo. Pero django.contrib.comments se diseñó teniendo en cuenta casos de uso como este, por lo que también define un par de sus propias señales personalizadas que brindan más información. La señal que querrá usar aquí es django.contrib.comments.signals.comment_will_be_posted, que transmite no solo la clase modelo (Comentario) y el objeto de comentario real, sino también el objeto HttpRequest de Django en el que se envía el comentario. Esto significa que tendrá acceso a todos los encabezados de solicitud y que podrá completar toda la información que solicite Akismet.

Para usar esta señal, primero deberá importarla:

```
de django.contrib.comments.signals import comment_will_be_posted
```

Luego deberá cambiar la definición de la función mode_comment para acomodar Fecha los argumentos que envía esta señal:

```
def moderado_comentario(remitente, comentario, solicitud, **kwargs):
```

Ahora puede reescribir la sección del código que envía el comentario a Akismet para el comprobar spam:

```
if akismet_api.verify_key():
    akismet_data = { 'tipo_comentario': 'comentario',
                     'referente': request.META['HTTP_REFERER'],
                     'user_ip': comentario.ip_address, 'user-agent':
                     request.META['HTTP_USER_AGENT'] } if
    akismet_api.comment_check(smart_str(instancia.comentario), akismet_data,
                               build_data=True): comentario.is_public
                               = False
```

Tenga en cuenta que debido a que ha reescrito la función para aceptar un argumento llamado comentario, debe cambiar todo lo que se refiera a él como instancia. También tenga en cuenta que los valores de los encabezados HTTP residen en request.META, que es un diccionario. Puede identificar la mayoría de los encabezados HTTP en request.META convirtiendo sus nombres a mayúsculas y prefijándolos con HTTP_. Esto significa, por ejemplo, que el encabezado HTTP Referer se convierte en HTTP_REFERER en request.META.

Una vez que lo pone todo junto, la función completa de moderación de comentarios, con filtrado Akismet basado en la edad y estadístico, se ve así:

```
importar fecha y
hora de akismet importar Akismet
de django.conf importar configuración de
django.contrib.comments.models importar comentario de
django.contrib.comments.signals importar comment_will_be_posted de
django.contrib.sites.models importar sitio de django.utils.encoding importar smart_str
```

```
def moderado_comentario(remitente, comentario, solicitud, **kwargs):
```

si no es comentario.id:

```
    entrada = comentario.content_object
    delta = datetime.datetime.now() - entrada.pub_date si delta.days
    > 30: comentario.is_public = False else: akismet_api =
        Akismet(key=settings.AKISMET_API_KEY, blog_url="http://
        %s /" ➔ %Site.objects.get_current().domain) if
        akismet_api.verify_key(): akismet_data = { 'comment_type':
        'comentario',
        'referente': request.META['HTTP_REFERER'],
        'user_ip': comentario.ip_address, 'user-agent':
        request.META['HTTP_USER_AGENT'] }
```

```
si akismet_api.comment_check(smart_str(comment.comment),
                               akismet_data,
                               build_data=Verdadero):
    comentario.is_public = Falso

comment_will_be_posted.connect(comentario_moderado, remitente=Comentario)
```

El mejor lugar para poner esto es cerca de la parte inferior de coltrane/models.py para que connect() line se leerá y ejecutará cuando se importen los modelos del weblog. Esto también elimina la necesidad de al menos una de las importaciones, la línea de fecha y hora de importación, porque ya se ha importado en ese archivo.

Advertencia: rutas de importación e importaciones múltiples de un solo módulo

Cuando importa un módulo de Python por primera vez, todo el código que contiene se analiza y ejecuta. Es por eso que la línea connect() se ejecutará siempre que los modelos del weblog se importen por primera vez. Pero esto abre un error potencial sutil: Python hace esto una vez para cada ruta de importación única utilizada para llevar a cabo la importación. Entonces, por ejemplo, si estuviera importando los modelos orientados a la búsqueda que escribió para el CMS en el Capítulo 3, el código en cms/search/models.py se evaluaría una vez si hiciera la importación de esta manera:

desde cms.buscar modelos de importación

Y se evaluaría nuevamente si luego hiciera otra importación como esta:

de modelos de importación de búsqueda

La utilidad manage.py de Django cambia la ruta de importación de Python por conveniencia y, al hacerlo, hace que las dos líneas anteriores funcionen. Por lo tanto, no es inusual que un proyecto termine teniendo importaciones en ambas formas, como las que se muestran. Desafortunadamente, esto significa que si tiene un fragmento de código que desea ejecutar solo una vez, como la línea connect(), porque solo desea que esa función se registre una vez, en su lugar, se ejecutará una vez para cada forma diferente en que se importa el módulo. .

Lo mejor es elegir un único estilo de importación y utilizarlo de forma coherente. Como regla general, normalmente me atengo a la forma en que aparece la aplicación en mi configuración INSTALLED_APPS. Por ejemplo, si tengo cms.search en INSTALLED_APPS, siempre hago la importación a partir de modelos de importación de cms.search.

Envío de notificaciones por correo electrónico

Muchos sistemas de blogging y CMS que permiten comentar también incluyen una función que notifica automáticamente a los administradores del sitio cada vez que se publica un nuevo comentario. Esto es útil porque les permite mantenerse al día con discusiones activas y también les permite detectar cualquier problema: un comentarista problemático, argumentos que se salen de control o simplemente el spam ocasional que se escapa del filtro. Ha visto lo fácil que es usar el despachador de Django para agregar funciones adicionales cuando se publica un comentario, así que sigamos adelante y agreguemos notificaciones por correo electrónico como toque final.

Enviar correo electrónico desde Django es bastante fácil de hacer y se divide en unos pocos pasos simples:

1. Complete, como mínimo, la configuración EMAIL_HOST y EMAIL_PORT en el archivo de configuración de Django. Estos se utilizarán para determinar el servidor de correo electrónico (SMTP) al que se conecta Django para enviar correo. Si su servidor de correo requiere un nombre de usuario y contraseña para enviar correo, complete EMAIL_HOST_USER y EMAIL_HOST_PASSWORD también. Si su servidor de correo requiere una conexión TLS segura, establezca EMAIL_USE_TLS en Verdadero.
2. Rellene la configuración DEFAULT_FROM_EMAIL para que sirva como la dirección de origen predeterminada para correos electrónicos automatizados . envío de correo electrónico.
3. Importe una función de envío de correo electrónico desde django.core.mail y llámela. La mayoría de las veces usará django.core.mail.send_mail(), que toma un asunto, un mensaje, una dirección de remitente y una lista de destinatarios, en ese orden.

Advertencia: Verificación de la configuración relacionada con el correo electrónico

Por lo general, su proveedor de alojamiento o su proveedor de servicios de Internet (dependiendo de quién proporcione su servicio de correo electrónico) podrá darle los valores correctos para completar configuraciones como EMAIL_HOST. Para verificarlos dos veces, puede usar django.core.send_mail() manualmente en un intérprete de Python (iniciado con python manage.py shell en el directorio de su proyecto) para enviarse un mensaje de prueba. Si la configuración es correcta, recibirá un correo electrónico. Si algo sale mal, Python le informará el mensaje de error en el intérprete.

Si desea suprimir el informe de errores, puede pasar el argumento de palabra clave fail_silently=True a cualquiera de las funciones de envío de correo de Django. Tenga en cuenta, sin embargo, que esto silenciará por completo los errores durante el envío del correo electrónico, lo que significa que no tendrá forma de saber si un mensaje determinado se envió con éxito.

Ahora, puede usar send_mail() y codificar uno o más destinatarios para las notificaciones de comentarios. Pero una vez más, esto perjudicaría la reutilización de su código. Dos sitios diferentes que usan esta aplicación pueden querer que dos conjuntos diferentes de personas reciban notificaciones de comentarios.

Afortunadamente, hay una solución fácil. En el archivo de configuración de Django hay dos configuraciones: ADMINISTRADORES y GERENTES, que lo ayudan a lidar con situaciones como esta. La configuración ADMINS debe ser una lista de programadores u otras personas técnicas que deben recibir notificaciones sobre problemas con su sitio. Cuando implemente en producción, Django automáticamente enviará un correo electrónico a la depuración información a las personas enumeradas en ADMINS siempre que ocurra un error del servidor. La configuración de ADMINISTRADORES, por otro lado, debe ser una lista de personas que no son necesariamente programadores, pero que están involucradas en la administración del sitio. Cada una de estas configuraciones espera un formato como el siguiente:

```
GERENTES = (('Alice Jones', 'alice@example.com'),  
            ('Bob Smith', 'bob@ejemplo.com'))
```

En otras palabras, es una tupla, o lista de tuplas, donde cada tupla contiene un nombre y una dirección de correo electrónico. Cuando se completan, dos funciones en django.core.mail—mail_admins() y mail_managers()—se puede usar como acceso directo para enviar un correo electrónico a esas personas.

Entonces, para agregar una notificación de comentario, puede hacer algo como lo siguiente:

```
from django.core.mail import mail_managers email_body =  
"%s publicó un nuevo comentario en la entrada '%s'." mail_managers("Nuevo  
comentario publicado", email_body % (comment.name, comment.content_object))
```

Esto enviará un correo electrónico a todos los que figuran en la configuración de GERENTES, notificándoles de la nuevo comentario.

Y así tienes la versión final de tu función moderar_comentario:

```
de akismet importar Akismet de  
django.conf importar configuración de  
django.contrib.comments.models importar comentario de  
django.contrib.comments.signals importar comment_will_be_posted de django.contrib.sites.models  
importar sitio de django.core.mail importar mail_managers de django.utils.encoding importar  
smart_str
```

```
def moderado_comentario(remitente, comentario, solicitud, **kwargs):  
    si no es comentario.id:
```

```
        entrada = comentario.content_object delta  
        = fechahora.fechahora.ahora() - entrada.pub_fecha si delta.días >  
        30: comentario.es_público = Falso si no:
```

```
        akismet_api = Akismet(key=settings.AKISMET_API_KEY, blog_url="http://  
        %s/" ♦ %Site.objects.get_current().domain)  
        if akismet_api.verify_key(): akismet_data = { 'tipo_comentario': 'comentario',  
  
            'referente': request.META['HTTP_REFERER'], 'user_ip':  
            comment.ip_address, 'user-agent':  
            request.META['HTTP_USER_AGENT'] } if  
            akismet_api.comment_check(smart_str(comentario.comentario), akismet_data,  
            build_data=Verdadero): comentario.is_public =  
            Falso  
  
            email_body = "%s publicó un nuevo comentario en la entrada '%s'."  
            mail_managers("Nuevo comentario publicado", email_body % (comment.name,  
            comment.content_object))  
  
            comentario.will_be_posted.connect(comentario_moderado, remitente=Comentario)
```

Una vez que esto esté en su lugar, no necesitará hacer nada más. La etiqueta get_comment_list que está utilizando para recuperar comentarios para mostrarlos en sus plantillas es lo suficientemente inteligente como para tomar la

is_public en cuenta cuando recupera los comentarios, por lo que cualquier comentario con is_public establecido en False se excluirá automáticamente.

Uso de las funciones de moderación de comentarios de Django

En este punto, tiene un sistema de moderación de comentarios que implementa un conjunto particular de reglas de moderación, pero desafortunadamente adolece de un par de problemas importantes:

- Está fuertemente ligado a los modelos usados en su aplicación de weblog. Por ejemplo, se supone la existencia de un campo denominado pub_date en el objeto al que se adjuntará un comentario. Esto significa que si alguna vez agrega nuevos modelos a su proyecto (ya sea en la aplicación de blog o en otra aplicación) y permite comentarios sobre ellos, el sistema de moderación podría fallar.
- Las reglas particulares que está utilizando: moderar todos los comentarios después de 30 días, enviar a Akismet, las copias de los comentarios por correo electrónico a los administradores del sitio están codificadas en la aplicación. Esto significa que sería difícil reutilizar esta aplicación en situaciones donde esas reglas no son apropiadas.

Lo ideal sería algún tipo de sistema genérico que te permita decidir qué comentarios someterse a las reglas de moderación y le permite especificar las reglas de moderación por modelo. Esto le permitiría configurar la moderación para comentarios en entradas de weblog, por ejemplo, pero tal vez desactivarla para otros tipos de contenido. Dicho sistema también le permitiría adaptar las reglas de moderación específicas a cada tipo particular de contenido.

Por lo que ya ha visto en el sistema de moderación que acaba de crear, probablemente podría descubrir cómo crear un sistema genérico de este tipo. Principalmente, sería una cuestión de verificar a qué tipo de contenido se "adjuntará" un comentario entrante y luego aplicar las reglas de moderación específicas para ese tipo de contenido. Pero debido a que esto es algo que se necesita con bastante frecuencia, Django le brinda esa infraestructura, lo que le permite escribir solo el código necesario para implementar sus propias reglas de moderación específicas.

El código para el sistema de moderación incorporado de Django reside en django.contrib.comments.moderation, que proporciona dos bits importantes de código:

- django.contrib.comments.moderation.moderator actúa como una especie de registro central para todas las reglas de moderación de comentarios que está utilizando y realiza un seguimiento de qué conjunto de reglas va con qué tipo de contenido.
- django.contrib.comments.moderation.CommentModerator le permite especificar las reglas para un tipo particular de contenido.

En muchos sentidos, el sistema de moderación de Django funciona de manera similar a cómo funciona su interfaz administrativa. Con el administrador, escribe una subclase de la clase ModelAdmin de Django, describe las opciones que desea y la registra con la interfaz administrativa. Con la moderación de comentarios, escribe una subclase de CommentModerator, describe las opciones que desea y las registra en el sistema de moderación.

Por ejemplo, en lugar de usar el sistema de moderación de comentarios que acaba de crear, podría colocar el siguiente código en la parte inferior de coltrane/models.py, y el sistema de moderación de Django marcaría automáticamente los comentarios como no públicos 30 días después de la publicación de una entrada y los enviaría automáticamente. -envíe un correo electrónico al personal de su sitio cada vez que se publique un comentario en una entrada:

```
de django.contrib.comments.moderation import CommentModerator, moderador
```

```
class EntryModerator(CommentModerator):  
    auto_moderate_field = 'pub_date'  
    moderar_después = 30 email_notification =  
    True
```

```
moderador.registrar(Entrada, Moderador de entrada)
```

Esto funcionará porque django.contrib.comments.moderation.moderator escucha las señales enviadas cada vez que se envía un comentario; luego busca las reglas apropiadas y las aplica.

Actualmente (a partir de Django 1.1), el sistema de moderación incorporado no es compatible con Akismet, por lo que necesitará un poco de código personalizado para que funcione. Así es como se ve:

```
de akismet importar Akismet de  
django.conf importar configuraciones de  
django.contrib.comments.moderation importar CommentModerator, moderador de  
django.utils.encoding importar smart_str
```

```
class EntryModerator(CommentModerator):  
    auto_moderate_field = 'pub_date'  
    moderar_después = 30 email_notification =  
    True  
  
    def moderado(yo, comentario, objeto_contenido, solicitud):  
        ya_moderado = super(ModeradorEntrada, yo)  
        ya_moderado(comentario, objeto_contenido) si ya_moderado: return  
        True  
  
        akismet_api = Akismet(clave=configuración.AKISMET_API_KEY,  
                               blog_url="http://%s/" % yo  
        Site.objects.get_current().domain) if  
        akismet_api.verify_key():  
            akismet_data = { 'comment_type': 'comentario',  
                           'referente': request.META['HTTP_REFERER'],  
                           'user_ip': comment.ip_address, 'user-agent':  
                           request.META['HTTP_USER_AGENT'] } return  
            akismet_api.comment_check(smart_str(comentario.comentario), akismet_data,  
                                      build_data=Verdadero)  
  
        falso retorno
```

```
moderador.registrar(Entrada, Moderador de entrada)
```

El código anterior define un método denominado moderar() en su subcomentario Moderator clase. A ese método se le pasarán tres argumentos: el comentario que se publica, el objeto de contenido al que se adjuntará (en el caso de una entrada de blog) y la solicitud HTTP en la que se publica el comentario. Lo primero que debe hacer aquí es usar super() para llamar al método moderar() de la clase principal (CommentModerator), porque podría determinar que el comentario debe ser moderado sin tener que enviarlo a Akismet. El valor de retorno de moderar() es Verdadero o Falso; si es Verdadero, el comentario está moderado (marcado como no público).

Si el método moderar() de la clase principal devuelve Falso, puede enviar el comentario a Akismet y devolver cualquier valor que devuelva el método comment_check() de Akismet (porque también devuelve Verdadero cuando cree que un comentario es spam). Pero tenga en cuenta la línea final de su método moderar(): simplemente devuelve Falso. Esto es importante porque es posible que no obtenga una respuesta útil de Akismet (si su clave de API no es válida, por ejemplo), pero su moderar() todavía se requiere el método para devolver un valor de True o False. Elegir cuál usar como valor de "último recurso" para ese tipo de situación depende de usted; esta línea de código se ejecutará solo si falla la comprobación de Akismetcheck_key().

Adición de fuentes

La última característica que desea para su weblog es la capacidad de tener fuentes RSS o Atom de sus entradas y enlaces. También desea tener fuentes personalizadas que manejen, por ejemplo, entradas en una categoría específica. Crear esta funcionalidad desde cero, escribiendo funciones de vista que recuperan una lista de entradas y representan una plantilla que crea el XML apropiado en lugar de una página HTML, no sería demasiado difícil. Pero debido a que esta es una necesidad común para los sitios web, Django nuevamente brinda ayuda para automatizar el proceso a través de la aplicación incluida django.contrib.syndication.

En esencia, django.contrib.syndication proporciona dos cosas:

- Un conjunto de clases que representan feeds y que se pueden dividir en subclases para una fácil personalización
- Una vista que sabe cómo trabajar con estas clases para generar y servir el apropiado XML

Para ver cómo funciona, comenzemos configurando un feed Atom para las últimas entradas publicadas en el blog

Creación de la clase LatestEntriesFeed

Vaya al directorio coltrane y cree un nuevo archivo vacío, llamado feeds.py. En la parte superior, agregue las siguientes líneas:

```
de django.utils.feedgenerator importar Atom1Feed
del sitio de importación django.contrib.sites.models
de django.contrib.syndication.feeds importar Feed
desde coltrane.models entrada de importación

sitio_actual = Sitio.objetos.get_actual()
```

Ahora puede comenzar a escribir una clase de fuente para las últimas entradas. Llámelo LatestEntriesFeed. Será una subclase de la clase django.contrib.syndication.feeds.Feed que está importando aquí.

Primero debe completar algunos metadatos requeridos. Este va a ser un feed Atom, por lo que se requieren varios elementos. (Los feeds RSS requieren menos metadatos, pero es una buena idea incluir esta información de todos modos, porque los metadatos adicionales son más útiles para las personas que desean recopilar y procesar información de los feeds). Este es un ejemplo:

```
clase LatestEntriesFeed(Feed):
    autor_nombre = "Bob Smith"
    copyright = "http://%s/acerca de/copyright/" % sitio_actual.dominio
    description = "Últimas entradas publicadas en %s" % sitio_actual.nombre
    feed_type = Atom1Feed
    item_copyright = "http://%s/about/copyright/" % sitio_actual.dominio
    item_author_name = "Bob Smith"
    item_author_link = "http://%s/" % sitio_actual.dominio
    link = "/fuentes/entradas/"
    title = "%s: Últimas entradas" % sitio_actual.nombre
```

Continúe y complete la información adecuada para su propio nombre y los metadatos relevantes. Tenga en cuenta que, si bien la mayoría de los elementos aquí variarán automáticamente según el sitio actual, tengo valores codificados en los campos nombre_autor, nombre_autor_elemento y enlace.

Para la reutilización en una amplia variedad de sitios, puede subclasicar esta clase de fuente para anular solo esos valores. O, si tiene una función que puede determinar el valor correcto para un sitio determinado, puede completarlo. (Por ejemplo, puede usar una búsqueda de URL inversa para obtener el enlace campo.) Para obtener una lista completa de estos campos y lo que puede poner en cada uno, consulte la documentación completa de django.contrib.syndication, que se encuentra en línea en www.djangoproject.com/documentation/syndication_feeds/.

Ahora debe decirle a la fuente cómo encontrar los elementos que se supone que debe contener: las últimas 15 entradas activas, en nuestro caso. Para ello, agregue un método denominado items() a la clase de fuente, que devolverá esas entradas:

```
elementos def (uno mismo):
    devuelve Entry.live.all()[:15]
```

Cada elemento debe tener una fecha en el feed. Lo logras usando un método llamado item_pubdate(), que recibirá un objeto como argumento y devolverá un objeto de fecha o fecha y hora para usar con ese objeto. (La clase Feed formateará automáticamente esto de manera adecuada para el tipo de feed que se utilice). En el caso de una entrada, ese es solo el valor del campo pub_date:

```
def item_pubdate(self, item):
    devolver artículo.pub_date
```

Cada elemento también debe tener un identificador único, llamado GUID (abreviatura de *identificador único global*). Este puede ser el campo de identificación de la base de datos, pero generalmente es mejor usar algo menos transitorio. Si fuera a migrar a un nuevo servidor o a una base de datos diferente, los valores de identificación podrían cambiar durante la transición y el GUID para una entrada en particular cambiaría en el proceso.

Para una situación como esta, la solución ideal es algo llamado *URI de etiqueta*. Una etiqueta URI (identificador de recurso de forma uniforme) proporciona una forma estándar de generar un identificador único para algún recurso de Internet, de una manera que no cambiará mientras ese recurso de Internet continúe existiendo en la misma dirección. Si está interesado en los detalles completos del estándar, las etiquetas URI son

especificado por IETF RFC 4151 (www.faqs.org/rfcs/rfc4151.html), pero la idea básica es que una etiqueta URI para un elemento consta de tres partes:

1. La etiqueta: cadena
2. El dominio del elemento, seguido de una coma, seguido de una fecha relevante para el elemento, seguido de dos puntos
3. Una cadena de identificación que es única para ese dominio y fecha

Para la fecha, utilizará el campo `pub_date` de cada entrada. Para la cadena de identificación única, utilizará el resultado de su método `get_absolute_url()`, porque se requiere que sea único.

El resultado, por ejemplo, es que la entrada en www.example.com/2008/jan/12/example-entry/ terminaría con un GUID de

```
tag:example.com,2008-01-12:/2008/ene/12/ejemplo-entrada/
```

Esto cumple con todos los requisitos para un GUID de fuente. Para implementar esto, simplemente defina un método en su clase de alimentación llamado `item_guid()`. De nuevo, recibe un objeto como argumento:

```
def item_guid(self, item): return "tag:  
%s,%s:%s" % (current_site.domain, item.pub_date.strftime('%Y-%m-%d'), item.get_absolute_url() )
```

Una última cosa que puede agregar a su feed es una lista de categorías para cada artículo. Esto ayudará a los agregadores de feeds a categorizar los artículos que publicas. Puede hacer esto definiendo un método llamado `item_categories`:

```
def item_categories(self, item): return [c.title  
for c in item.categories.all()]
```

Una clase de fuente de ejemplo completa, entonces, se ve así:

```
class LatestEntriesFeed(Feed):  
    author_name = "Bob Smith"  
    copyright = "http://%s/about/copyright/" % current_site.domain description = "Últimas  
entradas publicadas en %s" % current_site.name feed_type = Atom1Feed item_copyright  
= "http://%s/about/copyright/" % sitio_actual.dominio item_author_name = "Bob Smith"  
item_author_link = "http://%s/" % sitio_actual.dominio link = "/feeds/entries/" title = "%s:  
Últimas entradas" % sitio_actual.nombre
```

elementos de

```
definición (auto): devuelve Entry.live.all () [: 15]
```

```
def item_pubdate(self, item): return  
    item.pub_date
```

```
def item_guid(yo, elemento):
    devuelve "etiqueta:%s,%s:%s" % (sitio_actual.dominio,
                                      item.pub_date.strftime('%Y-%m-%d'),
                                      elemento.get_absolute_url())

def item_categories(self, item):
    devuelve [c.título para c en item.categories.all()]
```

Ahora puede configurar una URL para este feed. Vaya al archivo urls.py en el directorio del proyecto cms, y agrega dos cosas. Primero, cerca de la parte superior del archivo (encima de la lista de patrones de URL), agregue la siguiente declaración de importación y definición de diccionario:

```
de coltrane.feeds importar LatestEntriesFeed
```

```
feeds = { 'entradas': LatestEntriesFeed }
```

A continuación, agregue un nuevo patrón a la lista de URL:

```
(r'^alimenta/(?P<url>.*)/$',
'django.contrib.syndication.views.feed',
{ 'feed_dict': fuentes }),
```

Esto enrutaría cualquier URL que comience con /feeds/ a la vista en django.contrib.syndication, que maneja los feeds. El diccionario que configura mapea entre slugs de feeds, como entradas y clases de feeds específicas.

Una última cosa que debe hacer es crear dos plantillas. django.contrib.syndication utiliza el sistema de plantillas de Django para representar el título y el cuerpo principal de cada elemento en el feed para que pueda decidir cómo desea presentar cada tipo de elemento. Así que vaya al directorio donde ha estado guardando las plantillas para este proyecto y dentro de él cree un nuevo directorio llamado feeds. Dentro de eso, cree dos nuevos archivos, llamados entries_title.html y entries_description.html. (Los nombres a usar provienen de la combinación del slug del feed, en este caso, las entradas, y si la plantilla es para el título del elemento o su descripción). Cada una de estas plantillas tendrá acceso a dos variables:

- obj: Este es un elemento específico que se incluye en el feed.
- sitio: Este es el objeto Sitio actual, tal como lo devuelve Site.objects.get_current().

Entonces, para los títulos de los elementos, simplemente puede usar el título de cada entrada. En la plantilla entries_title.html, coloque lo siguiente:

```
{{obj.título}}
```

Para la descripción, usará el mismo truco que usó para las plantillas de archivo de entrada. configurarse en el último capítulo. Muestre el campo extract_html si tiene algún contenido; de lo contrario, muestre las primeras 50 palabras de body_html. Por lo tanto, en las entradas_descripción.html, complete lo siguiente:

```
{% si obj.excerpt_html%}
{{obj.excerpt_html|seguro}}
{% más %}
{{obj.body_html|truncatewords_html:"50"|seguro}}
{% terminara si %}
```

Recuerde que el sistema de plantillas de Django escapa automáticamente de HTML en las variables, por lo que aún debe usar el filtro seguro. Con las plantillas en su lugar, puede iniciar el servidor de desarrollo y visitar la URL /feeds/entries/ para ver el feed de las últimas entradas en el weblog.

Escribir un feed para los enlaces más recientes debería ser fácil en este punto. Intenta escribir el LatestLinksFeed clasifique usted mismo y configúrelo correctamente. (Recuerde que los enlaces no tienen categorías asociadas, por lo que debe omitir el método item_categories() o reescribirlo para que devuelva una lista de etiquetas). Un ejemplo completo se encuentra en el código de muestra asociado con este libro, por lo que consúltelo si se pierde (puede encontrar ejemplos de código para este capítulo en el área Código fuente/Descarga del sitio web de Apress en www.apress.com).

Generación de entradas por categoría: un ejemplo de feed más complejo

Ahora, también le gustaría ofrecer fuentes categorizadas para que los lectores que estén interesados en uno o dos temas específicos puedan suscribirse a las fuentes que enumeran solo las entradas de las categorías que les gustan. Pero esto es un poco más complicado porque plantea dos problemas:

- La lista de elementos en el feed debe, por supuesto, saber cómo determinar qué Categoría está mirando y asegúrese de que solo devuelva entradas de esa categoría.
- Varios de los campos de metadatos (el título de la fuente, el enlace, etc.) deberán cambiar dinámicamente en función de la categoría.

Sin embargo, la clase Feed de Django proporciona una manera de lidiar con esto. Una subclase Feed puede definir un método llamado get_object(), al que se le pasará un argumento que contiene los bits de la URL que vinieron después del slug con el que registró el feed, como una lista. Entonces, por ejemplo, si registró un feed con las categorías de slug y visitó la URL /feeds/categories/django/, a get_object() de su feed se le pasaría un argumento que contiene la lista de un solo elemento ["django"]. Desde allí puede buscar la categoría.

Comencemos agregando dos elementos a las declaraciones de importación en la parte superior de su archivo feeds.py para que que ahora se ve así:

```
de django.core.exceptions import ObjectDoesNotExist
de django.utils.feedgenerator importar Atom1Feed
del sitio de importación django.contrib.sites.models
de django.contrib.syndication.feeds importar Feed
de coltrane.models categoría de importación, entrada
```

Esto le da acceso al modelo Categoría, así como a ObjectDoesNotExist, una clase de excepción que define Django. Puede usar esto si alguien intenta visitar una URL para el feed de una categoría inexistente. (Cuando genera ObjectDoesNotExist, Django devolverá una respuesta HTTP 404 "Archivo no encontrado").

Ahora puede comenzar a escribir su clase de alimentación. Porque mucho de esto es similar al existente LatestEntriesFeed, simplemente lo subclasicará y cambiará las partes que deben cambiarse:

```
clase CategoryFeed(LatestEntriesFeed):
    def get_object(self, bits):
        si len(bits) != 1:
            aumentar el objeto no existe
        devuelve Categoría.objetos.get(slug__exact=bits[0])
```

Esto generará ObjectDoesNotExist o devolverá la categoría para la que necesita mostrar entradas. Ahora puede configurar el título, la descripción y el enlace de la fuente, definiendo métodos con esos nombres que reciben el objeto Categoría como argumento (el sistema de fuente de Django es lo suficientemente inteligente como para reconocer que necesita pasar ese objeto al llamar a los métodos):

```
def titulo(self, obj):
    return "%s: Últimas entradas en la categoría %s" % (sitio_actual.nombre,
                                                          obj.título)

def descripción(self, obj):
    return "%s: Últimas entradas en la categoría %s" % (sitio_actual.nombre,
                                                          obj.título)

enlace def(self, obj):
    devolver obj.get_absolute_url()
```

Atributos "simples" frente a métodos en feeds

En general, para cualquiera de los diversos bits de metadatos del feed (título, descripción y enlace, y metadatos para elementos individuales del feed), puede codificarlos utilizando un atributo simple del nombre correcto o generarlos dinámicamente definiendo un método de ese nombre. Para un feed como CategoryFeed que necesita buscar algún objeto (en este caso, una Categoría) a través de su método get_object(), puede definir un método que espera recibir ese objeto.

Nuevamente, para obtener una lista completa de los diferentes campos que puede usar en un feed, cada uno de los cuales funcionará así: consulte la documentación completa de django.contrib.syndication en www.djangoproject.com/documentación/sindicación_feeds/.

También puede cambiar el método items(). Una vez más, el sistema de alimentación de Django es lo suficientemente inteligente para saber que se debe pasar el objeto Categoría, y se asegurará de que eso suceda:

```
elementos def (self, obj):
    devolver obj.live_entry_set()[:15]
```

Recuerde que definió el método live_entry_set() en el modelo Categoría para que devolvería solo las entradas con el estado "en vivo".

Y eso es eso. Ahora su archivo feeds.py debería verse así:

```
de django.core.exceptions import ObjectDoesNotExist
de django.utils.feedgenerator importar Atom1Feed
del sitio de importación django.contrib.sites.models
de django.contrib.syndication.feeds importar Feed
de coltrane.models categoría de importación, entrada

sitio_actual = Sitio.objetos.get_actual()
```

```
class LatestEntriesFeed(Feed):
    author_name = "Bob Smith"
    copyright = "http://%s/about/copyright/" % current_site.domain description =
    "Últimas entradas publicadas en %s" % current_site.name feed_type =
    Atom1Feed item_copyright = "http://%s/about/copyright/" % sitio_actual.dominio
    item_author_name = "Bob Smith" item_author_link = "http://%s/" % sitio_actual.dominio
    link = "/feeds/entries/" title = " %s: Últimas entradas" % sitio_actual.nombre

elementos de
definición (auto): devuelve Entry.live.all () [: 15]

def item_pubdate(self, item): return
    item.pub_date

def item_guid(self, item): return
    "tag:%s,%s:%s" % (current_site.domain,
                       item.pub_date.strftime("%Y-%m-%d"),
                       item.get_absolute_url() )

def item_categories(self, item): return
    [c.title for c in item.categories.all()]

class CategoryFeed(LatestEntriesFeed): def
    get_object(self, bits): if len(bits) != 1:
        aumentar ObjectDoesNotExist

    devuelve Categoría.objetos.get(slug__exact=bits[0])

def title(self, obj): return
    "%s: Últimas entradas en la categoría '%s'" % (current_site.name, obj.title)

def descripción(self, obj): return
    "%s: Últimas entradas en la categoría '%s'" % (sitio_actual.nombre, obj.título)

def link(self, obj): return
    obj.get_absolute_url()

def elementos(self, obj):
    return obj.live_entry_set():15]
```

Puede registrar este feed cambiando la línea de importación en el archivo urls.py de su proyecto de coltrane.feeds importar LatestEntriesFeed

a

de coltrane.feeds import CategoryFeed, LatestEntriesFeed

y agregando una línea al diccionario de fuentes. cambiarlo de

```
feeds = { 'entradas': LatestEntriesFeed }
```

a

```
feeds = { 'entradas': LatestEntriesFeed,
          'categorías': CategoryFeed }
```

Por último, deberá configurar las plantillas feeds/categories_title.html y feeds/categorías_descripción.html. Debido a que solo muestran entradas, siéntase libre de copiar y pegar el contenido de las dos plantillas que usó para LatestEntriesFeed.

Escribir clases de fuentes que muestren entradas o enlaces por etiqueta seguirá el mismo patrón. Se incluyen ejemplos en el código de muestra que puede descargar para este libro, pero nuevamente, le recomiendo que lo pruebe usted mismo antes de echar un vistazo para ver cómo se hace.

Mirando hacia el futuro

Y con eso, ha implementado todas las funciones que se propuso tener para su weblog. Pero, lo que es más importante, ha cubierto una gran cantidad de territorio dentro de Django: modelos, vistas, enrutamiento de URL, plantillas y extensiones de plantillas personalizadas, comentarios y el sistema de señales y fuentes de sindicación de Django. Ya debería sentirse mucho más cómodo trabajando con Django y escribiendo lo que sería, si estuviera desarrollando desde cero sin la ayuda de Django:

ser algunas características bastante complejas.

Así que date una palmadita en la espalda porque ahora tienes muchos conocimientos útiles sobre Django. Tómese también un tiempo para trabajar con la aplicación de blog que ha desarrollado. Trate de pensar en una característica que le gustaría agregar y luego vea si puede averiguar cómo agregarla.

Cuando esté listo, el próximo capítulo iniciará una nueva aplicación: un código compartido sitio con algunas características sociales útiles, que resaltarán el sistema de procesamiento de formularios de Django para el contenido enviado por el usuario y mostrarán algunos usos avanzados de la API de la base de datos.

Capítulo 8



Un sitio social para compartir códigos

Hasta ahora ha estado usando Django para crear aplicaciones de *administración de contenido*. En estos tipos de aplicaciones, un administrador inicia sesión en una interfaz especial y publica algún contenido, después de lo cual el sistema muestra ese contenido públicamente con poca o ninguna interacción de los visitantes generales del sitio. Si bien este tipo de aplicación cubre una gran cantidad de tareas comunes de desarrollo web, no cubre todo, y no es el límite de lo que puede hacer Django.

Entonces, para su tercera aplicación Django, le mostraré cómo crear una aplicación impulsada por el usuario con mucha más interactividad y algunas funciones de estilo social, específicamente, un repositorio basado en la comunidad de código útil y reutilizable.

Puede encontrar un ejemplo en vivo de este tipo de sitio de código compartido en www.djangosnippets.org/, que está dirigido a los usuarios de Django. En los próximos capítulos, verá cómo crear una aplicación similar que puede implementar en cualquier momento que necesite un lugar para que varios usuarios comparten fragmentos de código entre sí.

Compilación de una lista de verificación de características

Al igual que con la aplicación weblog, lo primero que debe hacer es hacerse una idea aproximada de las características que le gustaría incluir. Utilice esta lista de funciones como punto de partida:

- Fragmentos de código con descripciones completas de lo que hacen
- Categorización por lenguaje de programación y resaltado completo de sintaxis con reconocimiento de lenguaje del código renderizado
- Una función de marcador para que los usuarios puedan volver fácilmente y encontrar sus fragmentos favoritos
- Una función de calificación que permite a los usuarios indicar si un código en particular fue útil para a ellos
- Etiquetado para organizar fragmentos y encontrar piezas de código relacionadas
- Listas de los fragmentos más populares por calificación general y por la cantidad de veces que han sido marcado
- Una lista de los autores más activos (usuarios que han enviado la mayor cantidad de fragmentos)

De acuerdo con la tradición de nombrar aplicaciones con el nombre de músicos de jazz notables, estoy voy a llamar a esta aplicación cab, en honor al cantante y director de orquesta Cab Calloway. El taxi fue conocido por su habilidad para cantar scat, cantar con sílabas cortas de palabras a veces sin sentido, lo que parece apropiado para una aplicación enfocada en muchos fragmentos cortos de código.

Configuración de la aplicación

Una vez más, deberá crear un nuevo módulo de Python para contener el código de la aplicación. Debe residir directamente en la ruta de importación de Python, en el mismo directorio que la aplicación coltrane que creó para el weblog. Ahora que sabe cómo hacer esto manualmente, tomemos un atajo. Vaya al directorio donde desea crear la aplicación y escriba lo siguiente:

```
django-admin.py startapp cab
```

Recuerde que en algunos sistemas, deberá escribir la ruta completa al comando django-admin.py.

Anteriormente, se encontró con startapp solo en el contexto de un proyecto específico, donde creó un nuevo directorio de aplicaciones dentro del directorio del proyecto. Sin embargo, funciona bien para crear módulos de aplicaciones independientes y elimina parte del tedio de comenzar con una nueva aplicación. El uso del comando django-admin.py startapp crea un nuevo directorio llamado cab y lo llena con un archivo `__init__.py` vacío y los archivos básicos `models.py` y `views.py` para una nueva aplicación Django.

Con el tiempo, terminará reemplazando el archivo `views.py` con un módulo de vistas que contiene varios archivos, pero para aplicaciones más simples, esta configuración será todo lo que necesita.

Antes de continuar, debe configurar otra cosa. Para resaltar la sintaxis de los fragmentos de código, usará una biblioteca de Python llamada pygments. Su sitio oficial está en <http://pygments.org/>, que tiene documentación y ejemplos interactivos, pero para descargarlo, visite <http://pypi.python.org/pypi/Pygments>, que es la página del proyecto pygments en Python Package Index (anteriormente conocido y a veces todavía se lo conoce como el queso de pitón

Shop, en honor a un famoso sketch cómico de los Monty Python).

El índice de paquetes de Python es un recurso increíblemente útil para los programadores de Python. Derecha ahora está rastreando más de 6000 bibliotecas y aplicaciones de terceros escritas en Python, todas categorizadas y todas con un historial completo de lanzamientos. Cada vez que se pregunte si Python tiene una biblioteca para algo que necesita hacer, debe intentar una búsqueda allí; es muy probable que alguien ya haya escrito al menos parte del código que necesitará y lo haya incluido en el índice.

Mientras escribo esto, la versión actual de pygments es 1.0, por lo que debería poder descargar un paquete llamado Pygments-1.0.tar.gz. Una vez que haya descargado el paquete, ábralo; en la mayoría de los sistemas operativos, puede simplemente hacer doble clic en el archivo. Esto crea un directorio llamado Pigments-1.0. En una línea de comando, vaya a ese directorio y escriba:

```
instalar python setup.py
```

Esto instala la biblioteca de pigments en su computadora. Una vez hecho esto, debería poder para iniciar un intérprete de Python y escribir **pigmentos de importación** sin ver ningún error.

Construcción de los modelos iniciales

Ahora que tiene configurado el módulo de su aplicación y la biblioteca de pigments instalada, puede comenzar a construir sus modelos. Lógicamente, vas a querer un modelo para representar los recortes de código; llamemos a este modelo Snippet. También querrás un modelo para representar el idioma en el que está escrito un fragmento de código en particular. Llamaremos a ese modelo Lenguaje. esto lo hará mucho más fácil almacenar algunos metadatos adicionales, manejar el resultado de sintaxis y ordenar fragmentos por idioma. Cubriré primero el modelo de lenguaje.

El modelo de lenguaje

Abra el archivo models.py en el directorio cab. El script django-admin.py ya ha completado una declaración de importación que extrae las clases modelo de Django, por lo que puede comenzar a trabajar de inmediato. Comience con el modelo de lenguaje que representa los diferentes lenguajes de programación. Necesitará cinco campos:

- El nombre del idioma
- Un slug único para identificarlo en URLs
- Un código de idioma que los pigmentos pueden usar para cargar el resultado de sintaxis apropiado módulo
- Una extensión de archivo para usar al ofrecer un fragmento en este idioma para descargar
- Un tipo MIME para usar al enviar un archivo de fragmento en este idioma

Según lo que ya sabe sobre el sistema modelo de Django, esto es fácil de configurar:

Idioma de clase (modelos.Modelo):

```
nombre = modelos.CharField(max_length=100)
slug = modelos.SlugField(único=True)
language_code = modelos.CharField(max_length=50)
mime_type = modelos.CharField(max_length=100)
```

Debido a que los valores (todas las cadenas) que van en estos campos no serán muy largos, mantuve el longitudes de campo bastante cortas.

Ahora, el orden más lógico para los idiomas es alfabético por nombre, por lo que puede agregarlo y configurar la representación de cadena de un idioma para que sea su nombre:

metaclase:

```
pedido = ['nombre']
```

```
def __unicode__(uno mismo):
    volver self.name
```

También puede definir un método get_absolute_url(). Aunque todavía no hayas configurado ninguna visualizaciones o URL, continúe y escríbalo usando el decorador de enlaces permanentes, para que haga una URL inversa buscar cuando llegue el momento. Cuando escribe las URL, el nombre del patrón de URL que corresponde a un idioma específico será cab_language_detail, y tomará el slug del idioma como argumento:

```
def get_absolute_url(auto):
    return ('cab_language_detail', (), { 'slug': self.slug })
get_absolute_url = modelospermalink(get_absolute_url)
```

Querrá un método más en el modelo de lenguaje para ayudar a los pigmentos con el resultado de sintaxis. pygments funciona leyendo un fragmento de texto mientras usa un fragmento especializado de código llamado *lexer*, que conoce las reglas del lenguaje de programación particular en el que está escrito el texto. La descarga de pygments incluye lexers para un gran conjunto de lenguajes, cada uno identificado por un nombre de código, y pygments incluye una función que, dada la nombre de código de un idioma, devuelve el lexer para ese idioma.

Advertencia: nomenclatura de patrón de URL

Técnicamente, los únicos requisitos que Django impone al nombre de un patrón de URL es que debe ser una cadena y que debe ser único dentro de un proyecto determinado. Sin embargo, como convención general, me gusta que los nombres de mis URL sigan un patrón predecible basado en el nombre de la aplicación, el nombre del modelo involucrado y la acción que realizará la vista. Por lo tanto, la vista detallada de un idioma en la aplicación cab es `cab_language_detail`, mientras que la vista para agregar un fragmento, por ejemplo, es `cab_snippet_add`.

Si bien no es necesario que haga esto, he descubierto que es de gran ayuda para otras personas que necesitan leer el código y, a veces, incluso a mí cuando miro hacia atrás una parte de mi propio código con el que no he trabajado recientemente.

Agreguemos un método al modelo `Language` que use esa función para devolver el lexer apropiado para un idioma dado. La función que desea es `pygments.lexers.get_lexer_by_name()`, lo que significa que deberá agregar una nueva declaración de importación en la parte superior de su archivo `models.py`:

de pigmentos importados lexers

Entonces puedes escribir el método:

```
def get_lexer(auto):
    devuelve lexers.get_lexer_by_name(self.language_code)
```

Ahora el modelo de lenguaje está listo y su archivo `models.py` se ve así:

```
from django.db import models
from pigmentos import lexers
```

Idioma de clase (`modelos.Modelo`):

```
nombre = modelos.CharField(max_length=100)
slug = modelos.SlugField(único=True)
language_code = models.CharField(max_length=50)
mime_type = modelos.CharField(max_length=100)
```

metacase:

```
pedido = ['nombre']
```

```
def __unicode__(uno mismo):
    volver self.name
```

```
def get_absolute_url(auto):
    return ('cab_language_detail', (), { 'slug': self.slug })
get_absolute_url = modelospermalink(get_absolute_url)
```

```
def get_lexer(auto):
    devuelve lexers.get_lexer_by_name(self.language_code)
```

El modelo de fragmentos

Ahora puede escribir la clase que representa un fragmento de código: Snippet. Necesitará tener varios campos:

- Un título y una descripción. Configurará la descripción para que haya dos campos: uno para almacenar la entrada sin procesar y otro para almacenar una versión HTML. Esto es similar a la forma en que configura el extracto y los campos del cuerpo para el modelo de Entrada en su weblog.
- Una clave externa que apunta al idioma en el que está escrito el fragmento.
- Una clave externa al modelo de usuario de Django para representar al autor del fragmento.
- Una lista de etiquetas, para las cuales utilizará el TagField que vio en la aplicación weblog. • El código real, que, nuevamente, almacenará como dos campos para que pueda mantener un renderizado, versión HTML con sintaxis resaltada separada de la entrada original.
- Algunos metadatos que incluyen la fecha y la hora en que se publicó por primera vez el fragmento, y la fecha y hora en que se actualizó por última vez.

Para comenzar, deberá importar el TagField que utilizó anteriormente:

```
de tagging.fields importar TagField
```

También necesitará el modelo de usuario de Django:

```
de django.contrib.auth.models usuario de importación
```

Luego puede construir los campos básicos:

Fragmento de clase (modelos.Modelo):

```
título = modelos.CharField(max_length=255) idioma =
modelos.ForeignKey(Idioma) autor =
modelos.ForeignKey(Usuario) descripción =
modelos.TextField() descripción_html =
modelos.TextFieldeditable=False ) código = modelos.TextField()
código_resaltado = modelos.TextFieldeditable=False) etiquetas =
TagField() fecha_publicación = modelos.DateTimeFieldeditable=False)
fecha_actualizada = modelos.DateTimeFieldeditable=False)
```

Tenga en cuenta que ha marcado varios de estos campos como no editables. Se completarán automáticamente con el método save() personalizado que escribirá en un momento.

El orden lógico de los fragmentos es el orden descendente del campo pub_date. lo harás También quiero darle al modelo Snippet una representación de cadena (que usará el título del fragmento):

metaclase:

```
pedido = ['-pub_date']
```

```
def __unicode__(self): return
self.title
```

Antes de escribir el método save(), continúe y agregue un método que sepa cómo aplicar el resultado de sintaxis. Para esto, necesitará dos elementos más de pygments: los formateadores módulo, que sabe cómo generar código resaltado en varios formatos; y el resultado() función, que pone todo junto para producir una salida resaltada. Así que cambia la importación. Línea de esto:

de pygments importados lexers

a esto:

de pygments importa formateadores, resaltar, lexers

La función de resultado () de pygments toma tres argumentos: el código para resaltar, el lexer para usar y el formateador para generar la salida. El código proviene del campo de código en el modelo Snippet y el lexer proviene del método get_lexer() que definió en el modelo de lenguaje. Luego, simplemente use el formateador HTML integrado en pygments como formateador de salida:

def resaltar (uno mismo):

```
volver resaltar(self.code,
                self.idioma.get_lexer(),
                formateadores.HtmlFormatter(linenos=True))
```

El argumento linenos=True para el formateador le dice a los pigmentos que generen la salida con números de línea para que sea más fácil leer el código e identificar líneas específicas.

ADVERTENCIA: ¿Por qué no resaltar directamente en save()?

Parece extraño estar escribiendo un método tan corto como este, cuando podría simplemente poner el código de resultado de sintaxis directamente en el método save() del modelo. Sin embargo, a menudo es una buena idea dividir cosas como esta en métodos separados. Hacerlo de esta manera significa que puede resaltar un fragmento sin guardarla y también reduce el acoplamiento a un método específico de resultado de sintaxis. Si alguna vez desea cambiar a un sistema de resultado de sintaxis diferente, por ejemplo, solo tendría que volver a escribir este método en lugar de rastrear potencialmente todos los lugares que usan resultado de sintaxis y cambiarlos todos.

Antes de escribir el método save(), continúe e importe el módulo Markdown de Python y utilícelo para generar la versión HTML de la descripción:

de descuento de descuento de importación

También necesitará el módulo de fecha y hora de Python:

importar fecha y hora

Ahora puede escribir el método save(), que debe realizar las siguientes acciones:

- Convierta la descripción de texto sin formato a HTML y guárdenla en el campo description_html.
- Realice el resultado de sintaxis y almacene el HTML resultante en el campo de código_resaltado.

- Establezca pub_date en la fecha y hora actuales si es la primera vez que se publica el fragmento salvado.

- Establezca la fecha_actualizada en la fecha y hora actuales siempre que se guarde el fragmento.

Aquí está el código:

```
def save(self, force_insert=False, force_update=False):  
    si no self.id:  
        self.pub_date = datetime.datetime.now()  
        self.updated_date = datetime.datetime.now()  
        self.description_html = markdown(self.description)  
        self.highlighted_code = self.highlight() super(Snippet,  
        self).save( forzar_insertar, forzar_actualizar)
```

Finalmente, agregue un método get_absolute_url(). La vista que muestra un Snippet en particular es llamado cab_snippet_detail, y toma el id del Snippet como argumento:

```
def get_absolute_url(self): return  
    ('cab_snippet_detail', (), { 'object_id': self.id })  
get_absolute_url = modelos_permalink(get_absolute_url)
```

El modelo terminado se ve así:

Fragmento de clase (modelos.Modelo):

```
título = modelos.CharField(max_length=255) idioma  
= modelos.ForeignKey(Idioma) autor =  
modelos.ForeignKey(Usuario) descripción =  
modelos.TextField() descripción_html =  
modelos.TextFieldeditable=False ) código = modelos.TextField()  
código_resaltado = modelos.TextFieldeditable=False) etiquetas =  
TagField() fecha_publicación = modelos.DateTimeFieldeditable=False)  
fecha_actualizada = modelos.DateTimeFieldeditable=False)
```

metaclasa:

```
pedido = ['-pub_date']  
  
def __unicode__(self): return  
    self.title  
  
def save(self, force_insert=False, force_update=False):  
    si no self.id:  
        self.pub_date = datetime.datetime.now()  
        self.updated_date = datetime.datetime.now()  
        self.description_html = markdown(self.description)  
        self.highlighted_code = self.highlight() super(Snippet ,  
        self).guardar(forzar_insertar, forzar_actualizar)
```

```

def get_absolute_url(auto):
    return ('cab_snippet_detail', (), { 'object_id': self.id })
get_absolute_url = modelos.permalink(get_absolute_url)

def resaltar (uno mismo):
    volver resaltar(self.code,
                    self.idioma.get_lexer(),
                    formateadores.HtmlFormatter(linenos=True))

```

Esto maneja el núcleo de la aplicación, fragmentos de código organizados por idioma, así que ahora puede hacer una pausa y comenzar a trabajar en algunas vistas iniciales para tener una idea de cómo se verán las cosas.

Continúe y cree un archivo admin.py también, y configure una interfaz administrativa básica para estos modelos para que pueda usarlo para comenzar a interactuar con la aplicación.

Prueba de la aplicación

A medida que construya estas vistas y el resto de la aplicación de código compartido de cabina, supondré que ya tiene un proyecto Django configurado con una base de datos y un directorio de plantillas.

Si lo desea, puede seguir usando el proyecto existente con el que ha trabajado para las dos aplicaciones anteriores. Sin embargo, esta aplicación no está realmente relacionada ni con el CMS simple ni con el weblog, así que si desea comenzar un nuevo proyecto ahora para trabajar con esta aplicación, no dude en hacerlo. En cualquier caso, deberá hacer tres cosas:

- 1. Agregue cab a la lista INSTALLED_APPS del proyecto que usará para probar y trabajar con esta aplicación:** si está comenzando un nuevo proyecto, también querrá agregar django.contrib.admin y etiquetando a la lista.
- 2. Ejecute manage.py syncdb para instalar los modelos que ha escrito hasta ahora:** Más tarde, cuando escriba el resto de los modelos, puede ejecutarlo nuevamente para instalarlos. El comando syncdb sabe cómo averiguar qué modelos ya están instalados y configura solo los nuevos unos.
- 3. Use la interfaz de administración para crear algunos objetos de idioma y completar algunos fragmentos:** para obtener una lista de los idiomas compatibles con pygments y los códigos de idioma para los lexers, lea la documentación de lexer de pygments en línea en <http://pygments.org/docs/lexers/>. En el próximo capítulo, verá cómo configurar vistas públicas que permitan a los usuarios comunes enviar fragmentos sin tener que usar la interfaz de administración.

Creación de vistas iniciales para fragmentos e idiomas

Mientras escribía la aplicación weblog, dependía en gran medida de las vistas genéricas de Django para proporcionar archivos basados en fechas y vistas detalladas de las entradas y enlaces. El uso de la exploración basada en fechas no tiene mucho sentido para esta aplicación, pero sin duda puede beneficiarse del uso de las vistas genéricas no basadas en fechas.

En el directorio cab, cree un nuevo directorio llamado urls y en él cree tres archivos:

- `__init__.py`, para marcar este directorio como un módulo de Python
- `snippets.py`, que tendrá las URL para las vistas orientadas a fragmentos
- `languages.py`, que tendrá las URL para las vistas orientadas al idioma

Como hizo con las URL del weblog, mantendrá cada grupo de URL para esta aplicación en su propio archivo. Esto significa que tendrá varios archivos en cab/urls, pero el beneficio en flexibilidad y reutilización vale la pena.

En urls/snippets.py, complete el siguiente código:

```
desde django.conf.urls.defaults importar desde *  
django.views.generic.list_detail importar object_list, object_detail  
de cab.models import Snippet  
  
snippet_info = { 'conjunto de consultas': Fragmento.objetos.todos() }  
  
urlpatrones = patrones("'  
    URL(r'^$',  
        lista_de_objetos,  
        dict(snippet_info, paginate_by=20),  
        nombre='cab_snippet_list'),  
    url(r'^(?P<id_objeto>\d+)/$',  
        objeto_detalle,  
        fragmento_info,  
        nombre='cab_snippet_detail'),  
)
```

Esto establece dos cosas:

- **Una lista de fragmentos, en el orden en que se publicaron: tenga en cuenta el argumento adicional has pasado por aquí—paginate_by.** Esto le dice a la vista genérica que le gustaría que muestre solo 20 fragmentos a la vez. Verás en un momento cómo trabajar con esta paginación en las plantillas.
- **Una vista detallada para objetos Snippet individuales:** Esta es simplemente la vista genérica object_detail vista.

Debería poder configurar las plantillas para esto con bastante facilidad. La plantilla de lista obtiene un variable llamada `{{ object_list }}`, que es una lista de instancias de Snippet, y la plantilla de detalles obtiene una variable llamada `{{ object }}`, que es un Snippet específico. Las vistas genéricas buscan la plantillas `cab/snippet_list.html` y `cab/snippet_detail.html`.

Lo único complicado es manejar la paginación de fragmentos en la vista de lista. La plantilla obtiene solo 20 fragmentos a la vez, por lo que debe mostrar los enlaces Siguiente y Anterior para permitir que el usuario navegue por ellos.

Para manejar esto, la vista genérica proporciona dos variables adicionales:

`paginador`: Esta es una instancia de `django.core.paginator.Paginator`. Sabe cuántas páginas totales de fragmentos hay y cuántos fragmentos totales están involucrados.

`page_obj`: Esta es una instancia de `django.core.paginator.Page`. Conoce su propio número de página y si hay una página anterior o siguiente.

En la plantilla `snippet_list.html`, podría usar algo como esto:

```
<p>{{ página }};
{% si página.has_previous %}
<a href="?page={{ page.previous_page_number }}">Página anterior</a>
{% terminara si %}
{% si la página.has_next_page%}
<a href="?page={{ page.next_page_number }}">Página siguiente</a>
{% endif %}</p>
```

Puede encontrar un ejemplo completo en el código fuente disponible para este libro (descargable desde el sitio web de Apress).

La vista genérica `object_list` sabe buscar la variable de página en la consulta de la URL cadena, y ajusta los fragmentos que muestra en consecuencia. Mientras tanto, el objeto `Page` sabe cómo imprimirse inteligentemente; en la plantilla, `{{ página }}` muestra algo como "Página 2 de 6".

Para configurar estas vistas, agregue un patrón como este al archivo raíz `urls.py` de su proyecto:

```
(r'^fragmentos/', include('taxi.urls.fragmentos')),
```

CSS para pigmentos Resaltado de sintaxis

Habrá notado en la vista de detalles del Fragmento que la muestra de código en realidad no parece estar resaltada de ninguna manera. Esto se debe a que `pygments`, por defecto, simplemente genera HTML con algunos nombres de clase se completaron para marcar cosas como palabras clave de idioma. Espera que use una hoja de estilo para cambiar la presentación de manera apropiada.

Para comenzar a diseñar el código resaltado, consulte algunos de los ejemplos en la demostración en línea de `pygments` en <http://pygments.org/demo/>. `pygments` viene con varios estilos incorporados, y una vez que haya encontrado uno que le guste, puede hacer que genere el CSS apropiado. Tú luego puedes guardarlo en un archivo y usarlo como su hoja de estilo.

Aquí hay un ejemplo simple de cómo obtener la información CSS adecuada de un `pygments` estilo. Esto supone que ha creado un archivo `pygments.css` en el que escribirá los estilos y que ha decidido que le gusta el estilo "murphy". Abra un intérprete de Python y escriba lo siguiente:

```
>>> de pygments importa formateadores, estilos
>>> estilo = estilos.obtener_estilo_por_nombre('murphy')
>>> formateador = formateadores.HtmlFormatter(estilo=estilo)
>>> archivo de salida = open('pigmentos.css', 'w')
>>> archivo de salida.write(formatter.get_style_defs())
>>> archivo de salida.close()
```

El archivo pygments.css ahora contiene una lista de reglas de estilo CSS para el estilo "murphy". Puedes modificarlos un poco siquieres. También puede hacer que los pigmentos agreguen automáticamente información más específica al selector de CSS que utiliza, si sabe que los bloques resaltados aparecerán solo dentro de ciertos elementos de la página. Consulte la documentación de la clase pygments HtmlFormatter para obtener detalles completos sobre cómo funciona el método get_style_defs().

Vistas para idiomas

Para mostrar

una lista de los idiomas en los que se han enviado los fragmentos, puede utilizar de nuevo la vista genérica object_list. Sin embargo, mostrar una lista de fragmentos para un idioma en particular requerirá un poco de código. Deberá escribir un envoltorio alrededor de una vista genérica, como lo hizo en el Capítulo 5, para mostrar la lista de entradas en una categoría particular.

Continúe y elimine el archivo views.py en el directorio de la aplicación cab y cree una vista directorio. En él, pon estos dos archivos:

- __init__.py
- idiomas.py

languages.py es donde colocará su primera vista escrita a mano para esta aplicación.

En views/languages.py, agregue el siguiente código para configurar el contenedor alrededor de la vista genérica:

```
desde django.shortcuts importar get_object_or_404 desde
django.views.generic.list_detail importar object_list desde cab.models
importar Idioma
```

```
def language_detail(solicitud, slug): language
    = get_object_or_404(Language, slug=slug) return
    object_list(request, queryset=language.snippet_set.all(),
                paginate_by=20, template_name='cab/
language_detail.html', extra_context={ 'idioma
idioma })
```

Esto devuelve una lista paginada de fragmentos para un idioma en particular. Ahora puede ir a urls/languages.py y completar un par de patrones de URL:

```
desde django.conf.urls.defaults importar desde *
django.views.generic.list_detail importar object_list desde cab.models
importar idioma desde cab.views.languages importar language_detail
```

```
language_info = { 'conjunto de consultas': Language.objects.all(),
                 'paginate_by': 20 }
```

```
urlpatrones = patrones(",
    URL(r'^$',
        lista_de_objetos,
        idioma_info,
        nombre = 'cab_language_list'),
    url(r'^(?P<slug>[-\w]+)/$',,
        idioma_detalle,
        nombre='cabina_idioma_detalle'),
)
```

Nuevamente, no debería tener problemas para configurar algunas plantillas básicas para manejar estas vistas. Los nombres de las plantillas son cab/language_list.html y cab/language_detail.html.

Para ver estas vistas en acción, agregue una línea como la siguiente al archivo raíz urls.py de su proyecto:

```
(r'^idiomas/', include('cab.urls.idiomas')),
```

Una vista avanzada: Autores principales

Debido a que cualquier usuario de la aplicación podrá enviar un fragmento de código, querrá tener una forma de mostrar los nombres de los usuarios que han enviado la mayor cantidad de fragmentos. Escribamos una vista llamado top_authors para manejar eso.

Dentro del directorio cab/views, cree un nuevo archivo llamado popular.py. Utilizará este archivo para esta vista top_authors, así como para algunas otras vistas que escribirá más adelante para enumerar los fragmentos que tienen la calificación más alta y los marcadores más frecuentes.

Inicie el archivo popular.py con un par de importaciones:

```
de django.contrib.auth.models importar usuario
de django.views.generic.list_detail importar object_list
```

Puede parecer un poco extraño importar una vista genérica aquí, porque es difícil de ver de cualquier manera. puede usar uno para una consulta como esta. De hecho, incluso si ha estado leyendo Django documentación de la API de la base de datos, puede que no sea obvio cómo hacer esta consulta. Entonces, primero, consideremos cómo funcionará la consulta.

La API de la base de datos de Django le permite especificar más que solo consultas que devuelven instancias de tus modelos; también puede escribir consultas que utilicen el soporte subyacente de su base de datos para características más avanzadas. En este caso, desea poder utilizar lo que se denomina consultas "agregadas", que calcular cosas como el número de filas de la base de datos que cumplen alguna condición, el promedio de una colección de filas, etc.

Django proporciona una serie de filtros agregados incorporados, pero el que desea aquí es django.db.models.Count, que le permite escribir una consulta que tenga en cuenta la número de fragmentos que ha publicado un autor en particular. Primero, necesitarás importarlo:

```
de django.db.models import Count
```

Entonces puedes escribir una consulta como esta:

```
User.objects.annotate(score=Count('snippet')).order_by('score')
```

El método de anotación le dice a Django que agregue un atributo adicional a cada Usuario devuelto por este consulta: el atributo se llamará puntuación y contendrá la cantidad de fragmentos publicados por el usuario. El método `order_by` le dice a Django cómo ordenar los resultados de la consulta, y es puntuación aprobada como argumento. El resultado, entonces, será una lista de usuarios ordenados de la mayoría de los fragmentos publicados a la menor cantidad.

Y debido a que este es un Django QuerySet, puede pasarlo a la vista `object_list`:

```
def top_authors(solicitud):
    top_authors_qs = User.objects.annotate(score=Count('snippet')).order_by('score')
    devuelve object_list (solicitud, conjunto de consultas = top_authors_qs,
                          template_name='taxi/top_authors.html',
                          paginar_por=20)
```

Terminará con una lista paginada de usuarios ordenados por sus recuentos de fragmentos. Entonces tú puedes conectar una URL para ello. Agreguemos un nuevo archivo en el directorio urls, `popular.py`, y usémoslo para todas estas vistas principales. En él colocas lo siguiente:

```
*-*- coding: utf-8 -*-
from django.conf.urls.defaults import *
from cab.views import popular

urlpatrones = patrones(),
    url(r'^autores/$',
        populares.principales_autores,
        nombre='cab_top_autores'),
)
```

Una vez más, puede conectar esto en el archivo raíz `urls.py` de su proyecto:

```
(r'^popular/', include('taxi.urls.popular')),
```

Una vez que haya creado la plantilla `cab/top_authors.html`, verá algunos resultados. Por supuesto, los resultados no serán tan impresionantes en este momento, porque la aplicación solo tiene un usuario: usted. Sin embargo, cuando se implementa en vivo en un sitio con múltiples usuarios, la vista `top_authors` será una buena característica.

Mejorar la vista de los mejores autores

Puede mejorar aún más esta función encapsulando la consulta de los mejores autores en un archivo reutilizable camino. En este momento, es un poco complicado, y no querrás escribirlo una y otra vez si alguna vez necesitas reutilizarlo.

Escribamos un administrador personalizado para el modelo Snippet y hagamos que la consulta de los principales autores sea una método en el gerente. Debido a que terminará escribiendo varios administradores personalizados para esta aplicación, avancemos y creemos un archivo `managers.py` en el directorio `cab`. Entonces, dentro es poner el siguiente código:

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models import Count
```

```
class SnippetManager(models.Manager):
    def top_authors(self):
        return User.objects.annotate(score=Count('snippet')).order_by('score')
```

En cab/models.py, agregue una nueva declaración de importación en la parte superior:

de los gerentes de importación de taxis

En la definición del modelo Snippet, agregue el administrador personalizado:

```
objetos = administradores.SnippetManager()
```

Ahora puede reescribir la vista top_authors de esta manera:

```
from django.views.generic.list_detail import object_list
from cab.models import Snippet
```

```
def top_authors(solicitud):
    volver object_list(solicitud, queryset=Snippet.objects.top_authors(),
                      template_name='cab/top_authors.html',
                      paginate_by=20)
```

Eso es mucho mejor.

Adición de una vista top_languages

Mientras agregas estas funciones, continúa y agrega la capacidad de mostrar los idiomas más populares a través de una vista llamada top_languages. Esto implicará una consulta similar a la vista top_authors, por lo que será fácil de escribir ahora.

Sin embargo, una decisión de diseño importante es dónde colocar el método para realizar esta consulta. Podría ponerlo en el SnippetManager y probablemente incluso volver a trabajar el método top_authors() en un método top_objects(). Este nuevo método podría devolver los autores principales, los idiomas principales o, más tarde, cuando haya creado los modelos para ellos, los fragmentos de código más marcados o mejor calificados según el argumento que recibió. Eso reduciría la cantidad de veces que tendría que escribir métodos para hacer este tipo de consulta. Sin embargo, una desventaja de este enfoque es que, lógicamente, la lista de idiomas principales no "pertenece" al modelo Snippet; pertenece al modelo Language. Debido a que es mejor presentar una API lógica para los usuarios de su aplicación que ser perezoso al escribir código, continúa y asigne a Language un administrador personalizado y coloque esta consulta allí.

En cab/managers.py, agregue lo siguiente:

```
class LanguageManager(models.Manager):
    def top_languages(self):
        return self.annotate(score=Count('snippet')).order_by('score')
```

En cab/models.py, puede agregar el administrador en la definición del modelo de lenguaje:

```
objetos = administradores.LanguageManager()
```

En cab/views/popular.py, puede cambiar la declaración de importación de
de cab.models import Snippet
a
de cab.models import Idioma, Fragmento

Escribe esta vista:

```
def top_languages(solicitud):  
    volver object_list(solicitud,  
        queryset=Idioma.objetos.top_languages(),  
        template_name='taxi/top_languages.html',  
        paginar_por=20)
```

y cambie cab/urls/popular.py a lo siguiente:

```
desde django.conf.urls.defaults importar desde *  
cab.views importar popular
```

```
urlpatrones = patrones(''  
    url(r'^autores/$',  
        populares.principales_autores,  
        nombre='cab_top_autores'),  
    url(r'^idiomas/$',  
        populares.principales_idiomas,  
        nombre='taxi_top_languages'),  
)
```

Ahora puede crear la plantilla cab/top_languages.html y agregar algunos fragmentos en varios idiomas para ver cómo cambian los resultados.

Mirando hacia el futuro

Ahora que tiene el núcleo de esta aplicación de código compartido en su lugar, aprenderá a implementar algunas de las interacciones del usuario en el próximo capítulo. Por un lado, obtendrá una introducción al sistema de procesamiento de formularios de Django, para que pueda ver cómo permitir que los usuarios envíen fragmentos sin pasar por la interfaz de administración.

Si desea un pequeño desafío antes de pasar al manejo de formularios, intente escribir una vista que enumere las etiquetas ordenadas por la cantidad de fragmentos que las usan. Echa un vistazo en el etiquetado aplicación para ver cómo funcionan las etiquetas, y consulte la documentación del marco de tipos de contenido de Django (www.djangoproject.com/documentation/contenttypes/) para tener una idea de la relaciones genéricas que utilizan las etiquetas. Si se queda perplejo, puede encontrar un ejemplo de trabajo en el código fuente asociado con este libro (descárguelo del área Código fuente/Descargar de el sitio web de Apress en www.apress.com).

Capítulo 9



Procesamiento de formularios en el Aplicación de código compartido

Todos de sus aplicaciones Django hasta ahora, con la excepción del sistema de comentarios para el weblog, se han centrado exclusivamente en sistemas en los que miembros confiables del personal de un sitio ingresan contenido a través de la interfaz administrativa de Django, en lugar de funciones interactivas que permiten a los usuarios comunes enviar contenido. para ser mostrado. Sin embargo, para esta nueva aplicación, necesitará una forma de permitir que los usuarios envíen sus fragmentos de código. También querrá asegurarse de que sus envíos estén en un formato que funcione con los modelos de datos que ha configurado.

Afortunadamente, Django hará que esto sea bastante fácil mediante el uso de un sistema simple pero poderoso para mostrar y procesar formularios basados en la web. En este capítulo, obtendrá una visión detallada del sistema de manejo de formularios de Django y lo utilizará para crear los formularios que la gente usará para enviar y editar sus ejemplos de código.

Un breve recorrido por el sistema de formularios de Django

El código de manejo de formularios de Django, que vive en el módulo `django.forms`, proporciona tres componentes clave que, en conjunto, cubren todos los aspectos de la construcción, visualización y procesamiento de un formulario:

- Un conjunto de clases de `campo`, similar a los tipos de campos disponibles para los modelos de datos de Django, que representan un tipo particular de datos y saben cómo validarlos.
- Un conjunto de clases de `widgets`, que saben cómo representar varios tipos de controles de formulario HTML (entradas de texto, casillas de verificación, etc.) y leer los datos correspondientes de un Envío de formulario HTTP
- Una clase de formulario que une estos conceptos y proporciona una interfaz unificada para definir los datos que se recopilarán y reglas de alto nivel para validarlos

Un ejemplo sencillo

Para tener una idea de cómo funciona esto, echemos un vistazo a un requisito simple pero común: registros de usuarios.

Advertencia: ¿Adónde va este código?

Este código específico no pertenece lógicamente a la aplicación cab que está desarrollando, y si alguna vez desarrolla código para manejar registros de usuarios, sería mejor colocar ese código en su propia aplicación separada.

Sin embargo, por ahora, no se preocupe por guardar este código en archivos de Python. Es solo un ejemplo útil que muestra la mayor cantidad posible de partes del sistema de manejo de formularios de Django.

Sin embargo, si alguna vez necesita implementar un sistema de registro de usuarios, no dude en consultar este código y adaptarlo a sus necesidades.

Los registros básicos requerirán un formulario de registro que recopila tres datos:

- Un nombre de usuario
 - Una dirección de correo electrónico para asociar con la nueva cuenta
 - Una contraseña que el usuario usará para iniciar sesión

Además, querrá hacer un poco de trabajo de validación personalizado:

- Querrás asegurarte de que el nombre de usuario no esté ya en uso porque no puedes tener dos usuarios con el mismo nombre de usuario.
 - Siempre es una buena idea mostrar dos campos de contraseña y hacer que el usuario escriba lo mismo contraseña dos veces. Esto detectará errores tipográficos y brindará un poco más de seguridad para asegurarse de que los nuevos usuarios obtengan la contraseña que esperan.

Lógicamente, esto se traduce en un elemento HTML <form> con cuatro campos: uno para el nombre de usuario y la dirección de correo electrónico, y dos para manejar la contraseña repetida. Así es como puede comenzar a crear el formulario:

desde formularios de importación de django

clase SignupForm (formularios. Formulario):

```
nombre de usuario = formularios.CharField(max_length = 30)  
correo electrónico = formularios.EmailField()  
contraseña1 = formularios.CharField(max_length=30)  
contraseña2 = formularios.CharField(max_length=30)
```

A parte del uso de clases de django.forms en lugar de django.db.models, esto comienza de forma similar a la forma en que define las clases modelo en Django: simplemente cree una subclase de la clase base apropiada y agrege los campos apropiados.

Pero el código no es del todo perfecto. HTML proporciona un tipo de entrada de formulario especial para manejar contraseñas, `<input type="password">`, que sería una forma más adecuada de representar los campos de contraseña. Puede implementar este tipo de entrada cambiando ligeramente esos dos campos:

El widget PasswordInput se representará como `<input type="password">`, que es exactamente lo que desea.

Esto también muestra una de las principales fortalezas de la forma en que el sistema de formularios de Django separa la validación de datos, que es manejada por el campo, de la presentación del formulario, que es manejada por los widgets.

Es bastante común encontrarse con situaciones en las que tiene una sola regla de validación subyacente que debe funcionar con varios campos que se convierten en diferentes tipos de entradas HTML. Esta separación lo hace fácil: puede reutilizar un solo tipo de campo y simplemente cambiar el widget.

Mientras estás en eso, hagamos un cambio más:

```
contraseña1 = formularios.CharField(max_length=30,  
                                    widget=formularios.PasswordInput(render_value=False))  
  
contraseña2 = formularios.CharField(max_length=30,  
                                    widget=formularios.PasswordInput(render_value=False))
```

El argumento `render_value` de PasswordInput le dice que incluso si tiene algunos datos, no debería mostrarlos. Un error que comete un usuario al ingresar la contraseña debería borrarse por completo el campo para asegurarse de que el usuario lo escriba correctamente la próxima vez.

Validación del nombre de usuario

Todos los campos que ha especificado hasta ahora tienen algunas reglas de validación implícitas asociadas.

El campo de nombre de usuario y los dos campos de contraseña tienen longitudes máximas especificadas, y EmailField confirmará que su entrada parece una dirección de correo electrónico (aplicando una expresión regular). Pero también debe asegurarse de que el nombre de usuario no esté en uso, por lo que deberá definir alguna validación personalizada para el campo de nombre de usuario.

Puede hacer esto definiendo un método en el formulario llamado `clean_username()`. Durante el proceso de validación, el sistema de formularios de Django busca automáticamente cualquier método cuyo nombre comience con `clean_` y termine en el nombre de un formulario en el campo, luego lo llama después de que se hayan aplicado las reglas de validación integradas del campo.

Así es como se ve el método `clean_username()` (suponiendo que el modelo de usuario de Django ya se haya importado usando `django.contrib.auth.models import User`):

```
def clean_username(auto):  
    probar:  
        User.objects.get(username=self.cleaned_data['username'])  
    excepto Usuario.DoesNotExist:  
        devolver self.cleaned_data['nombre de usuario']  
    aumentar formularios.ValidationError("Este nombre de usuario ya está en uso.ÿ  
    Por favor, elija otro.")
```

Este código incluye mucho en unas pocas líneas. En primer lugar, este método se llama solo si el nombre de usuario El campo ya ha cumplido con su requisito integrado de contener menos de 30 caracteres de texto. En ese caso, el valor enviado para el campo de nombre de usuario está en `self.cleaned_data['username']`. El atributo `clean_data` es un diccionario de cualquier dato enviado que haya pasado por la validación hasta el momento.

Usted consulta por un usuario cuyo nombre de usuario coincide exactamente con el valor enviado al nombre de usuario campo. Si no existe tal usuario, Django generará la excepción `User.DoesNotExist`. Esta excepción le indica que el nombre de usuario no está en uso, por lo que sabe que el valor del campo de nombre de usuario es válido. En este caso, simplemente devuelve ese valor.

Si hay un usuario con el nombre de usuario enviado, genera la excepción `ValidationError`. El código de manejo de formularios de Django detectará esta excepción y la convertirá en un mensaje de error que puede mostrar. (Verá cómo hacer esto en un momento, cuando vea la plantilla que muestra este formulario).

Validación de la contraseña

Validar la contraseña es un poco más complicado porque implica mirar dos campos a la vez y asegurarse de que coincidan. Puede hacer esto definiendo un método para uno de los campos y haciendo que mire el otro:

```
def limpiar_contraseña2(uno mismo):
    if self.cleaned_data['contraseña1'] != self.cleaned_data['contraseña2']:
        generar formularios.ValidationError("Debe escribir la misma contraseña cada vez")
    devolver self.cleaned_data['contraseña2']
```

Pero hay una mejor manera de hacer esto. Django te permite definir un método de validación, simplemente llamado `clean()`, que se aplica al formulario como un todo. Así es como podrías escribirlo:

```
def limpio (auto):
    si 'contraseña1' en self.cleaned_data y 'password2' en self.cleaned_data:
        if self.cleaned_data['contraseña1'] != self.cleaned_data['contraseña2']:
            generar formularios.ValidationError("Debe escribir la misma contraseña cada vez")
    devolver self.cleaned_data
```

Tenga en cuenta que en este caso, verifica manualmente si hay valores en `clean_data` para los dos campos de contraseña. Si se generaron errores durante la validación de campos individuales, `clean_data` estará vacío. Por lo tanto, debe verificar esto antes de referirse a cualquier cosa que espere encontrar en él.

Advertencia: los campos del formulario son obligatorios de forma predeterminada

Todos los tipos de campos integrados en el sistema de formularios de Django son obligatorios de forma predeterminada y, por lo tanto, no se pueden dejar en blanco. Si alguno de los campos de la contraseña se dejara en blanco, Django generaría un `ValidationError` antes de llamar al método `clean()`, por lo que no necesitaría generar un error adicional para solicitar un valor.

Para marcar un campo de formulario como opcional, páselle el argumento de palabra clave `required=False`.

Crear el nuevo usuario

En este punto, puede dejar de escribir el código del formulario y pasar a una vista que procese el formulario. Puede escribir la vista para que cree y guarde el nuevo objeto `Usuario`. Pero si alguna vez necesitara reutilizar este formulario en otras vistas, tendría que escribir ese código una y otra vez. Por lo tanto, es mejor escribir un método en el formulario mismo que sepa qué hacer con los datos válidos. Debido a que el método está guardando un nuevo objeto `Usuario` en la base de datos, llámémoslo `save()`.

Advertencia: save() no es solo para la base de datos

La mayoría de las veces, los formularios se utilizan para crear y actualizar objetos de modelo, en cuyo caso save() es la elección natural. Pero los formularios se pueden usar para otros fines (por ejemplo, un formulario de contacto puede enviar un mensaje de correo electrónico en lugar de guardar un objeto).

La convención general en la comunidad de Django es que cada vez que una clase de formulario tiene un método que "sabe" qué acción tomar con los datos válidos, ese método debe llamarse save(), incluso cuando no guarda ningún dato en su base de datos. La ventaja de dar a este tipo de método un nombre consistente y reconocible supera cualquier confusión inicial que pueda causar.

En el método save(), debe crear un objeto Usuario a partir del nombre de usuario, el correo electrónico y la contraseña enviados a su formulario. Suponiendo que ya haya importado el modelo de usuario, puede hacerlo así:

```
def guardar(auto):
    new_user = User.objects.create_user(username=self.cleaned_data['username'],
                                         email=self.cleaned_data['correo'],
                                         contraseña=self.cleaned_data['contraseña1'])
    devolver nuevo_usuario
```

Advertencia: Usuarios y contraseñas

Un gran problema con una base de datos de usuarios y contraseñas es que cualquiera que pueda acceder a la base de datos puede ver todas las contraseñas. Debido a que muchas personas tienden a reutilizar las mismas contraseñas en varios sitios web, esto puede representar un riesgo de seguridad significativo.

Para ayudarlo a proteger a sus usuarios, Django evita almacenar la contraseña "simple" que el usuario realmente usará para iniciar sesión. En su lugar, Django usa un truco matemático llamado función hash, que transforma la contraseña en una contraseña de aspecto aleatorio (pero no en realidad al azar) cadena de letras y números. Ese resultado se almacena en la base de datos en lugar de la contraseña real. La ventaja es que una función hash solo funciona de una manera: si conoce la contraseña, puede aplicar la función hash y obtener siempre el mismo resultado, pero si solo conoce el resultado, no puede retroceder para obtener la contraseña.

Esto proporciona una forma razonablemente segura de almacenar contraseñas. Cuando intenta iniciar sesión, el sistema de autenticación de Django aplica la función hash a la contraseña que ingresó y compara el resultado con el valor en la base de datos. Esto significa que la contraseña "simple" nunca tiene que almacenarse permanentemente en ningún lugar. Pero debido a que este sistema es un poco complicado para trabajar, el modelo de usuario de Django tiene un administrador personalizado que define el método create_user() que está usando aquí. Este método maneja el trabajo de aplicar la función hash a la contraseña y almacenar el valor correcto.

Y aquí está el formulario terminado:

```
de django.contrib.auth.models importar Usuario de
formularios de importación de django
```

```
class SignupForm(formularios.Form):
    nombre de usuario = formularios.CharField(max_length=30)
    correo electrónico = formularios.EmailField()
    contraseña1 = formularios.CharField(max_length=30,
                                         widget=forms.PasswordInput(render_value=False))
    contraseña2 = formularios.CharField(max_length=30,
                                         widget=forms.PasswordInput(render_value=False))

    def clean_username(self): try:
        User.objects.get(username=self.cleaned_data['username']) excepto
        User.DoesNotExist: return self.cleaned_data['username']
        raiseforms.ValidationError("Este nombre de usuario ya está en uso. ♦ Elija
        otro.")

    def clean(self): if
        'password1' en self.cleaned_data y 'password2' en self.cleaned_data: if
            self.cleaned_data['password1'] != self.cleaned_data['password2']: generar
            formularios.ValidationError("You debe escribir la misma ♦ contraseña cada vez")
        devolver self.cleaned_data

    def save(self):
        new_user = User.objects.create_user(username=self.cleaned_data['username'],
                                            email=self.cleaned_data['email'],
                                            password=self.cleaned_data['password1'])
        devolver nuevo_usuario
```

Cómo funciona la validación de formularios

El método que usará en las vistas para determinar si los datos enviados son válidos o no se llama `is_valid()`, y se define en la clase de formulario base de la que derivan todos los formularios de Django. Dentro de la clase `Form`, `is_valid()` pone en marcha las rutinas de validación del formulario, en un orden específico, llamando a `full_clean()` (otro método definido en la clase `Form` base en `django.forms`; consulte la Figura 9-1).

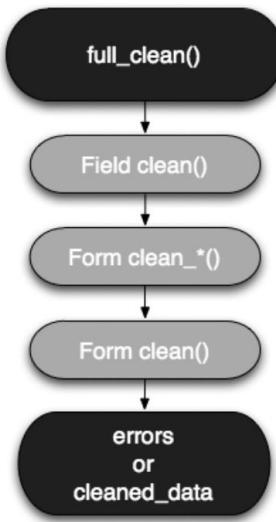


Figura 9-1. El orden en el que se aplican los métodos de validación a un formulario Django

El orden de validación es así:

1. Primero, `full_clean()` recorre los campos del formulario. Cada clase de campo tiene un método llamado `clean()`, que implementa las reglas de validación integradas de ese campo, y cada uno de estos métodos generará un `ValidationError` o devolverá un valor. Si un error de validación se genera, no se realiza ninguna validación adicional para ese campo (porque ya se sabe que los datos no son válidos). Si se devuelve un valor, entra en el diccionario `clean_data` del formulario.
2. Si el método `clean()` incorporado de un campo no generó un error de validación, entonces cualquier Se llama al método de validación personalizado, un método cuyo nombre comienza con `clean_` y termina con el nombre del campo. Nuevamente, estos métodos pueden generar un `ValidationError` o devolver un valor; si devuelven un valor, entra en `clean_data`.
3. Finalmente, se llama al método `clean()` del formulario. También puede generar un `ValidationError`, aunque uno que no esté asociado con ningún campo específico. Si `clean()` no encuentra nuevos errores, debería devolver un diccionario completo de datos para el formulario, normalmente haciendo `return self.cleaned_data`.
4. Si no se generaron errores de validación, el diccionario `clean_data` del formulario se completará con los datos válidos. Sin embargo, si hubo errores de validación, `clean_data` no existirá y un diccionario de errores (`self.errors`) se llenará con errores de validación.
Cada campo sabe cómo recuperar sus propios errores de este diccionario, por lo que puede hacer cosas como `{{ form.username.errors }}` en una plantilla.
5. Finalmente, `is_valid()` devuelve Falso si hubo errores de validación o Verdadero si hubo no lo eran.

Comprender este proceso es clave para aprovechar al máximo el sistema de manejo de formularios de Django. Puede parecer un poco complejo al principio, pero la capacidad de adjuntar reglas de validación a un formulario en varios lugares da como resultado una gran flexibilidad y facilita la escritura de código reutilizable. Por ejemplo, si necesita usar un tipo particular de validación una y otra vez, notará que escribir un método personalizado en cada formulario se vuelve tedioso. Probablemente será mejor que escriba su propia clase de campo, defina un método `clean()` personalizado en ella y luego reutilice ese campo.

Del mismo modo, distinguir los métodos específicos de campo del método `clean()` de "nivel de formulario" abre muchos trucos útiles para validar varios campos juntos. No necesariamente necesitará estos trucos cuando trabaje con un solo campo.

Procesando el Formulario

Ahora, echemos un vistazo a una vista que podría usar para mostrar y procesar este formulario:

```
de django.http importar HttpResponseRedirect
de django.shortcuts import render_to_response

def registro (solicitud):
    if solicitud.método == 'POST':
        formulario = formulario de registro (datos = solicitud.POST)
        si formulario.es_válido():
            nuevo_usuario = formulario.guardar()
            devuelve HttpResponseRedirect("/cuentas/inicio de sesión")
    más:
        formulario = formulario de registro ()
    volver render_to_response('signup.html',
                                {'formulario': formulario })
```

Desglosemos esto paso a paso:

1. Primero verifica el método de la solicitud HTTP entrante. Por lo general, esto será GET o POST. (Existen otros métodos HTTP, pero no se usan con tanta frecuencia, y los navegadores web generalmente solo admiten GET y POST para envíos de formularios).
2. Si, y solo si, el método de solicitud es POST, crea una instancia de `SignupForm` y le pasa `request.POST` como sus datos. En el Capítulo 3, cuando escribió una función de búsqueda simple, vio esa solicitud. GET es un diccionario de datos enviados con una solicitud GET; De manera similar, `request.POST` es el diccionario de datos (en este caso, el envío del formulario) enviado junto con una solicitud POST.
3. Verifica si los datos enviados son válidos llamando a `is_valid()` del formulario método. En el fondo, esto hace coincidir los datos enviados con los campos del formulario y verifica las reglas de validación de cada campo. Si los datos pasan la validación, `is_valid()` devolverá `True`, y el diccionario `clean_data` del formulario se completará con los valores correctos. De lo contrario, `is_valid()` devolverá `False`, y `clean_data` diccionario no existirá.

Escribir un formulario para agregar fragmentos de código

Vaya al directorio cab y cree un archivo llamado Forms.py. En él puedes empezar a escribir tu formulario de la siguiente manera:

```
desde formularios de importación de django
de cab.models import Snippet
```

```
clase AddSnippetForm(formularios.Form):
    def __init__(self, autor, *args, **kwargs):
        super(AddSnippetForm, self).__init__(*args, **kwargs)
        self.autor = autor
```

Además de aceptar un argumento adicional, el autor, que almacena para su uso posterior, está haciendo dos cosas importantes aquí:

- Además del argumento del autor, especifica que el método `__init__()` acepta `*args` y `**kwargs`. Esta es una forma abreviada de Python para especificar que aceptará cualquier combinación de argumentos posicionales y de palabras clave.
- Usas `super()` para llamar al método `__init__()` de la clase padre, pasando los otros argumentos que tu `__init__()` personalizado aceptó. Esto asegura que se llame al `__init__()` de la clase Form base y configure todo lo demás en su formulario correctamente.

El uso de esta técnica (aceptar `*args` y `**kwargs` y pasarlo al método principal) es una forma abreviada útil cuando el método que está anulando acepta muchos argumentos, especialmente si muchos de ellos son opcionales. El método `__init__()` de la clase Form base en realidad acepta hasta siete argumentos, todos ellos opcionales, por lo que este es un truco útil.

Ahora puede agregar los campos que le interesan:

```
título = formularios.CharField(max_length=255)
descripción = formularios.CharField(widget=formularios.Textarea())
código = formularios.CharField(widget=formularios.Textarea())
etiquetas = formularios.CharField(max_length=255)
```

Tenga en cuenta que, una vez más, confía en el hecho de que puede cambiar el widget utilizado por un campo para modificar su presentación. Donde el sistema modelo de Django usa dos campos diferentes: CharField y TextField: para representar diferentes tamaños de campos basados en texto (y tiene que hacerlo, porque funcionan con diferentes tipos de datos en las columnas de la base de datos subyacente), el sistema de formularios solo tiene un CharField. Para convertirlo en un `<textarea>` en el HTML eventual, simplemente cambie su widget a un Textarea, de la misma manera que usó el widget PasswordInput en el formulario de registro de usuario de ejemplo.

Y eso se ocupa de todo excepto del idioma, que de repente se ve un poco complicado. Lo que le gustaría hacer es mostrar una lista desplegable (un elemento HTML `<select>`) de los idiomas disponibles y validar que el usuario eligió uno de ellos. Pero ninguno de los tipos de campo que ha visto hasta ahora puede manejar eso, por lo que deberá recurrir a algo nuevo.

Una forma de manejar esto es con un tipo de campo llamado ChoiceField. Toma una lista de opciones (en el mismo formato que un campo de modelo que acepta opciones; ya lo ha visto en el campo de estado en el modelo de entrada del weblog, por ejemplo) y se asegura de que el valor enviado sea uno de ellos. Pero configurarlo correctamente para que el formulario consulte el conjunto de idiomas cada vez que se usa (en caso de que un administrador haya agregado nuevos idiomas al sistema)

requeriría más piratería en el método `__init__()`. Y representar una relación de modelo como esta es una situación terriblemente común, por lo que esperaría que Django proporcione una manera fácil de manejar esto.

Resulta que Django proporciona una solución fácil: un tipo de campo especial llamado `ModelChoiceField`. Donde un `ChoiceField` normal simplemente tomaría una lista de opciones, un `ModelChoiceField` toma un Django `QuerySet` y genera dinámicamente sus opciones a partir del resultado de la consulta (ejecutada de nuevo cada vez). Para usarlo, deberá cambiar la importación del modelo en la parte superior del archivo para traer también el modelo de idioma:

```
de cab.models import Snippet, Language
```

Y luego puedes simplemente escribir:

```
idioma = formularios.ModelChoiceField(queryset=Idioma.objects.todos())
```

Para este formulario, no necesita ninguna validación especial más allá de lo que le brindan los campos, por lo que puede escribir el método `save()` y listo:

```
def guardar(auto):
    fragmento = Fragmento(título=self.cleaned_data['título'],
                           descripción=self.cleaned_data['descripción'],
                           código=self.cleaned_data['código'],
                           tags=self.cleaned_data['tags'],
                           autor=auto.autor,
                           idioma=self.cleaned_data['idioma'])
    fragmento.guardar()
    fragmento de retorno
```

Debido a que crear un objeto y guardarlo todo en un solo paso es un patrón común en Django, en realidad puede acortarlo un poco. La clase de administrador predeterminada que proporciona Django incluirá un método llamado `create()`, que crea, guarda y devuelve un nuevo objeto. Usando eso, su método `save()` es un par de líneas más corto:

```
def guardar(auto):
    return Snippet.objects.create(title=self.cleaned_data['title'],
                                   descripción=self.cleaned_data['descripción'],
                                   código=self.cleaned_data['código'],
                                   tags=self.cleaned_data['tags'],
                                   autor=auto.autor,
                                   idioma=self.cleaned_data['idioma'])
```

Y ahora su `AddSnippetForm` está completo:

```
desde formularios de importación de django
de cab.models import Snippet, Language
```

```
clase AddSnippetForm(formularios.Form):
    def __init__(self, autor, *args, **kwargs):
        super(AddSnippetForm, self).__init__(*args, **kwargs)
        self.autor = autor
```

```

título = formularios.CharField(max_length=255)
descripción = formularios.CharField(widget=formularios.Textarea())
código = formularios.CharField(widget=formularios.Textarea())
etiquetas = formularios.CharField(max_length=255)
idioma = formularios.ModelChoiceField(queryset=Idioma.objetos.todos())

def guardar(auto):
    return Snippet.objects.create(title=self.cleaned_data['title'],
                                   descripción=self.cleaned_data['descripción'],
                                   código=self.cleaned_data['código'],
                                   tags=self.cleaned_data['tags'],
                                   autor=auto.autor,
                                   idioma=self.cleaned_data['idioma'])

```

Escribir una vista para procesar el formulario

Ahora puede escribir una vista corta llamada `add_snippet` para manejar los envíos. En la cabina/vistas directorio, cree un archivo llamado `snippets.py`, y en él coloque el siguiente código:

```

de django.http importar HttpResponseRedirect
de django.shortcuts import render_to_response
desde cab.forms importar AddSnippetForm

def add_snippet(solicitud):
    if solicitud.método == 'POST':
        formulario = AddSnippetForm(autor=solicitud.usuario, datos=solicitud.POST)
        si formulario.es_válido():
            nuevo_fragmento = formulario.guardar()
            devolver HttpResponseRedirect(nuevo_fragmento.get_absolute_url())
    más:
        formulario = AddSnippetForm(autor=solicitud.usuario)
        devolver render_to_response('cab/add_snippet.html',
                                    { 'formulario': formulario })

```

Este código creará una instancia del formulario, validará los datos, guardará el nuevo Snippet y devolverá un redirigir a la vista detallada de ese fragmento. (Nuevamente, siempre redirija después de una POST exitosa).

Al principio esto se ve muy bien, pero hay un problema al acecho aquí. Te refieres a la solicitud `solicitud.usuario`, que será el usuario conectado actualmente (Django configura esto automáticamente cuando el sistema de autenticación se ha activado correctamente). Pero, ¿qué sucede si la persona que completa este formulario no ha iniciado sesión?

La respuesta es que sus datos realmente no serán válidos. Cuando el usuario actual no ha iniciado sesión, `request.user` es un objeto "ficticio" que representa a un usuario anónimo y no se puede usar como el valor del campo de autor de un fragmento. Entonces, lo que necesita es alguna forma de asegurarse de que solo los usuarios registrados puedan completar este formulario.

Afortunadamente, Django proporciona una manera fácil de manejar esto, a través de un decorador en el sistema de autenticación llamado `login_required`. Simplemente puede importarlo y aplicarlo a su función de vista, y cualquiera que no haya iniciado sesión será redirigido a una página de inicio de sesión:

```
desde django.http importar HttpResponseRedirect desde
django.shortcuts importar render_to_response desde
django.contrib.auth.decorators importar login_required desde cab.forms importar
AddSnippetForm

def add_snippet(solicitud):
    if solicitud.método == 'POST':
        form = AddSnippetForm(author=request.user, data=request.POST) if form.is_valid():
            new_snippet = form.save() return HttpResponseRedirect(new_snippet.get_absolute_url())

    else:
        formulario = AddSnippetForm(autor=solicitud.usuario) return
        render_to_response('cab/add_snippet.html',
                           { 'formulario': formulario })

add_snippet = login_required(add_snippet)
```

Advertencia: Configuración de vistas de inicio/cierre de sesión

El sistema de autenticación de Django, incluido en django.contrib.auth, incluye las vistas y formularios que necesitará para autenticar correctamente a los usuarios e iniciar sesión. Siempre que solo esté probando una aplicación en su propia computadora, puede iniciar sesión a través de la interfaz de administración de Django y luego visite cualquier vista que haya marcado con `login_required`. Pero para una implementación pública en vivo, querrá configurar vistas públicas de inicio/cierre de sesión para usuarios normales.

Para ver cómo usar las vistas de autenticación integradas, consulte la documentación para la autenticación de Django. sistema de ción en línea en <http://docs.djangoproject.com/en/dev/topics/auth/>.

Escribir la plantilla para manejar la vista add_snippet Desde aquí, podría escribir la plantilla `cab/add_snippet.html` de esta manera:

```
<html>
<cabeza>
    <title>Agregar un fragmento</title> </
head> <body> <h1>Agregar un fragmento</
h1> <p>Utilice el siguiente formulario para
    enviar su fragmento; todos los campos
    son obligatorios.</p> <form method="post" action=""> <p>{{ form.title.errors }} <span
        class="error"> {{ form.title.errors|unir:", " }} </span>
```

```
{% endif %}</p>
<p><label for="id_title">Título:</label> {{ form.title }}</
p> <p>{% if form.language.errors %} <span
class="error"> {{ form.language.errors|join:" " }} {%
endif %}</p> <p><label for="id_languages">Idioma:< /
label> {{ form.language }}</p> <p>{% if
form.description.errors %} <span class="error">
{{ form.description.errors|join:" " }} {% endif %}</p>
<p><label for="id_description">Descripción:</label></p>
<p>{{ formulario.descripcion }}</p> <p>{% if form.code.errors
%} <span class="error"> {{ form.code.errors|join:" " }} {% endif
%}</p> <p> <label for="id_code">Código:</label></p>
<p>{{ form.code }}</p> <p>{% if form.tags.errors %} <span
class=" error"> {{ form.tags.errors|join:" " }} {% endif %}</p>
<p><label for="id_tags">Etiquetas:</label> {{ formulario.etiquetas }}<
/p> <p><input type="submit" value="Enviar"></p> </formulari> <
cuerpo> </html>
```

Generación automática del formulario a partir de un modelo

Definición

Aunque el sistema de formularios de Django le permite ser bastante conciso al escribir y usar este formulario, todavía no ha llegado a una solución ideal. Configurar un formulario para agregar o editar instancias de un modelo es algo bastante común, y sería terriblemente molesto seguir escribiendo este tipo de formularios repetitivos una y otra vez (especialmente cuando ya ha especificado la mayor parte o toda la información relevante). una vez en la definición de la clase modelo).

Afortunadamente, existe una manera de reducir drásticamente la cantidad de código que tiene que escribir. Siempre que no necesite demasiado en el comportamiento personalizado de su formulario, Django

proporciona una clase de acceso directo llamada ModelForm que puede generar automáticamente un formulario moderadamente personalizable a partir de una definición de modelo, incluidos todos los campos relevantes y el método save() necesario. En su forma más básica, así es como funciona:

```
de django.forms importar ModelForm  
de cab.models import Snippet
```

```
clase SnippetForm(ModelForm):  
    metaclase:  
        modelo = Fragmento
```

Crear una subclase de ModelForm y proporcionar una clase Meta interna que especifique un modelo esta nueva clase SnippetForm para derivar automáticamente sus campos del modelo especificado. Y ModelForm es lo suficientemente inteligente como para ignorar cualquier campo en el modelo definido con editable=False, por lo que campos como la versión HTML de la descripción no aparecerán en este formulario. Lo único que falta aquí es que aparecerá el campo de autor. Afortunadamente, ModelForm admite algunas personalizaciones, incluida una lista de campos para excluir específicamente del formulario, por lo que simplemente puede cambiar la definición de SnippetForm a lo siguiente:

```
clase SnippetForm(ModelForm):  
    metaclase:  
        modelo = Fragmento  
        excluir = ['autor']
```

Y dejará fuera el campo del autor. Ahora puede simplemente eliminar cab/forms.py y reescribir cab/views/snippets.py así:

```
de django.http importar HttpResponseRedirect  
de django.forms importar ModelForm  
de django.shortcuts import render_to_response  
desde django.contrib.auth.decorators import login_required  
de cab.models import Snippet
```

```
clase SnippetForm(ModelForm):  
    metaclase:  
        modelo = Fragmento  
        excluir = ['autor']
```

```
def add_snippet(solicitud):  
    if solicitud.método == 'POST':  
        formulario = SnippetForm(datos=solicitud.POST)
```

```
if form.is_valid():
    new_snippet = form.save()
    return HttpResponseRedirect(new_snippet.get_absolute_url())
else:
    form = SnippetForm()
    return render_to_response('cab/add_snippet.html',
                             {'formulario':
formulario })
add_snippet = login_required(add_snippet)
```

Sin embargo, esto no es del todo correcto. El Snippet necesita tener un autor completado, pero has dejó ese campo fuera del formulario. Podría regresar y definir un método `__init__()` personalizado nuevamente y pasar `request.user`, pero `ModelForm` tiene un truco más bajo la manga. Puede hacer que `ModelForm` cree el objeto Snippet y lo devuelva sin guardar; haces esto pasando un argumento extra—`commit=False`—a su método `save()`. Cuando haga esto, `save()` aún devolverá un nuevo objeto Snippet, pero no lo *guardará* en la base de datos. Esto le permitirá agregar el usuario usted mismo e insertar manualmente el nuevo Snippet en la base de datos:

```
de django.http importar HttpResponseRedirect de
django.forms importar ModelForm de django.shortcuts
importar render_to_response de django.contrib.auth.decorators
importar login_required de cab.models importar Snippet
```

```
class SnippetForm(ModelForm):
    class Meta: modelo = Snippet
        excluir = ['autor']

def add_snippet(solicitud): if
    request.method == 'POST': form =
        SnippetForm(data=request.POST) if
    form.is_valid(): new_snippet =
        form.save(commit=False) new_snippet.autor
        = request.user new_snippet.save() devuelve
        HttpResponseRedirect(new_snippet.get_absolute_url())

    else:
        form = SnippetForm()
        return render_to_response('cab/add_snippet.html',
                                 {'formulario':
formulario })
add_snippet = login_required(add_snippet)
```

Advertencia: commit=False y relaciones de muchos a muchos

Si el modelo con el que está trabajando tiene un campo ManyToManyField (que estará representado en un formulario por un tipo de campo llamado ModelMultipleChoiceField), deberá dar un paso adicional cuando use el método save() de un ModelForm con commit=False. Las relaciones de muchos a muchos no se pueden configurar hasta que se guarde el objeto principal (porque necesitan saber su ID en la base de datos). Entonces, cada vez que use commit=False en un formulario que tenga una relación de muchos a muchos, el formulario tendrá un método llamado save_m2m(), que almacena los datos para las eventuales relaciones de muchos a muchos. Deberá llamar a ese método manualmente (sin argumentos) después de haber guardado el objeto principal.

Ahora puede abrir cab/urls/snippets.py y agregar una nueva importación:

```
from cab.views.snippets import add_snippet
```

y un nuevo patrón de URL:

```
url(r'^add/$', add_snippet, name='cab_snippet_add'),
```

Simplificación de plantillas que muestran formularios

La plantilla descrita anteriormente seguirá funcionando porque los campos del formulario no han cambiado.

Pero nuevamente, sería bueno si Django proporcionara una manera fácil de mostrar un formulario en una plantilla sin tener que escribir todo el HTML repetitivo y verificar los errores de campo. Ha eliminado el tedio de definir la clase de formulario en sí, entonces, ¿por qué no eliminar el tedio de crear una plantilla?

Para lidiar con esto, cada formulario de Django tiene algunos métodos adjuntos que saben cómo renderice el formulario en diferentes tipos de HTML:

as_ul(): presenta el formulario como un conjunto de elementos de lista HTML (etiquetas), con un elemento por campo

as_p(): presenta el formulario como un conjunto de párrafos (etiquetas HTML <p>), con un elemento por párrafo

as_table(): Representa el formulario como una tabla HTML, con un <tr> por campo

Entonces, por ejemplo, podría reemplazar las plantillas que ha estado haciendo hasta ahora (un conjunto de HTML elementos de párrafo) con solo lo siguiente:

```
{{ formulario.as_p }}
```

Pero hay algunas cosas a tener en cuenta al usar estos métodos:

- Ninguno de ellos genera las etiquetas adjuntas <form> y </form> porque el formulario no "sabe" cómo o dónde planea enviar el formulario. Deberá completar estas etiquetas usted mismo, con los atributos de acción y método apropiados.
- Ninguno de ellos genera ningún botón para enviar el formulario. Una vez más, el formulario no sepa cómo desea que se envíe, por lo que deberá proporcionar una o más etiquetas <input type="submit"> usted mismo.

- El método `as_ul()` no genera las etiquetas circundantes `` y ``, y el `as_table()` no genera las etiquetas `<table>` y `</table>` circundantes. Esto es en caso de que desee agregar más HTML usted mismo (que es una necesidad común para la presentación de formularios), por lo que deberá recordar completar estas etiquetas.
- Finalmente, estos métodos no son fácilmente personalizables. Cuando solo necesitas un pre básico sentation para un formulario (especialmente para la creación rápida de prototipos para que pueda probar una aplicación), son extremadamente útiles, pero si necesita una presentación personalizada, probablemente querrá volver a crear plantillas para el formulario manualmente.

Edición de fragmentos

Ahora tiene un sistema para que los usuarios envíen sus fragmentos de código, pero ¿qué sucede si alguien quiere regresar y editar uno? Es inevitable que alguien envíe accidentalmente algún código que tenga una errata o un error menor, o que encuentre una mejor solución para una tarea en particular. Sería bueno permitir que los usuarios editen sus propios fragmentos en esos casos, así que sigamos adelante y configuremos la edición de fragmentos a través de una vista llamada `edit_snippet`.

Afortunadamente, esto va a ser fácil. `ModelForm` también sabe cómo editar un objeto existente, que se encarga de la mayor parte del trabajo pesado. Todo lo que tienes que hacer, entonces, es manejar dos cosas:

- Averigüe qué objeto Snippet editar.
- Asegúrese de que el usuario que intenta editar el Snippet sea su autor original.

Puede manejar la primera parte con bastante facilidad: puede configurar su vista `edit_snippet` para recibir el id del Snippet en la URL y buscarlo en la base de datos. Luego, puede comparar el campo de autor del fragmento con la identidad del usuario que ha iniciado sesión actualmente para asegurarse de que coincidan.

Entonces, comencemos agregando un par de importaciones más a `cab/views/snippets.py`:

```
desde django.shortcuts import get_object_or_404
de django.http importar HttpResponseRedirect
```

La clase `HttpResponseForbidden` representa una respuesta HTTP con el código de estado 403, lo que indica que el usuario no tiene permiso para hacer lo que estaba tratando de hacer.

Lo usará cuando alguien intente editar un fragmento que no envió originalmente.

Aquí está la vista `edit_snippet`:

```
def edit_snippet(solicitud, snippet_id):
    fragmento = get_object_or_404(Fragmento, pk=id_fragmento)
    if request.user.id != fragmento.author.id:
        devolver HttpResponseRedirect()
    if solicitud.método == 'POST':
        formulario = SnippetForm(instancia=fragmento, datos=solicitud.POST)
        si formulario.es_válido():
            fragmento = formulario.guardar()
            devolver HttpResponseRedirect(fragmento.get_absolute_url())
```

más:

```
formulario = SnippetForm(instancia=fragmento)
devolver render_to_response('cab/edit_snippet.html',
                           { 'formulario': formulario })
edit_snippet = login_required(edit_snippet)
```

Para decirle a una subclase de ModelForm que le gustaría editar un objeto existente, simplemente pase ese objeto como la instancia de argumento de palabra clave; el formulario se encargará del resto. Y tenga en cuenta que debido a que el Fragmento ya tiene un autor, y ese valor no cambiará, no necesita usar commit=False y luego guardar manualmente el Fragmento. El formulario no cambiará ese valor, por lo que simplemente puede dejar que se guarde como está.

Ahora puede agregarle un patrón de URL. Primero cambia la línea de importación en cab/urls/snippets.py para importar también esta vista:

```
desde cab.views.snippets importar add_snippet, edit_snippet
```

y luego agregas el patrón de URL:

```
url(r'^editar/(?P<fragmento_id>\d+)/$', editar_fragmento, nombre='cab_fragmento_editar'),
```

Debido a que el formulario para la vista edit_snippet y la vista add_snippet tendrán los mismos campos, puede simplificar un poco la creación de plantillas usando solo una plantilla y pasando una variable que indica si está agregando o editando (para que los elementos como la página el título puede cambiar en consecuencia). Entonces, cambiemos la línea final de la vista add_snippet para pasar una variable adicional llamada add, establezcamos su valor en True y cambiemos el nombre de la plantilla a cab/snippet_form.html:

```
devolver render_to_response('cab/snippet_form.html',
                           { 'formulario': formulario, 'añadir': verdadero})
```

Luego puede cambiar la misma línea en la vista edit_snippet para usar cab/snippet_form.html y establezca la variable add en False:

```
devolver render_to_response('cab/snippet_form.html',
                           { 'formulario': formulario, 'añadir': Falso })
```

Ahora simplemente puede tener una plantilla, cab/snippet_form.html, que puede tener este aspecto:

```
<html>
  <cabeza>
    <title>{% if add %}Añade un{% else %}Edita tu{% endif %} fragmento</title>
  </cabeza>
  <cuerpo>
    <h1>{%- if add %}Añade un{% else %}Edita tu{% endif %} fragmento</h1>
    <p>Use el siguiente formulario para {% if add %}add{% else %}editar {% endif %}
      tu fragmento; todos los campos son obligatorios</p>
    <método de formulario="publicar" acción="">
      {{ formulario.as_p }}
      <p><tipo de entrada="enviar" valor="Enviar"></p>
    </formulario>
  </cuerpo>
</html>
```

Ahora tiene formularios, vistas y plantillas que permiten a los usuarios agregar y editar su código fragmentos. Aquí está el archivo cab/views/snippets.py terminado, como referencia:

```
desde django.http importar HttpResponseRedirect, HttpResponseRedirect
desde django.forms importar ModelForm desde django.shortcuts importar get_object_or_404,
render_to_response desde django.contrib.auth.decorators importar login_required
desde cab.models importar Snippet
```

```
clase SnippetForm(ModelForm):
```

```
    clase Meta:
```

```
        modelo = Fragmento
```

```
        de exclusión = ['autor']
```

```
def add_snippet(solicitud): if
```

```
    request.method == 'POST': form =
```

```
        SnippetForm(data=request.POST) if
```

```
        form.is_valid(): new_snippet =
```

```
            form.save(commit=False) new_snippet.author
```

```
            = request.user new_snippet.save() devuelve
```

```
            HttpResponseRedirect(new_snippet.get_absolute_url())
```

```
más:
```

```
        formulario = SnippetForm()
```

```
    return render_to_response('cab/snippet_form.html', { 'formulario':
```



```
                                formulario, 'agregar': True })
```

```
add_snippet = login_required(add_snippet)
```

```
def edit_snippet(solicitud, snippet_id): fragmento
```

```
    = get_object_or_404(Snippet, pk=snippet_id) if request.user.id !=
```



```
    snippet.author.id: return HttpResponseRedirect()
```

```
if solicitud.método == 'POST':
```

```
    form = SnippetForm(instancia=snippet, data=request.POST) if
```

```
    form.is_valid(): snippet = form.save() return
```

```
    HttpResponseRedirect(snippet.get_absolute_url())
```

```
else:
```

```
    formulario = SnippetForm(instancia=fragmento)
```

```
    return render_to_response('cab/snippet_form.html', { 'formulario':
```



```
                                formulario, 'agregar': False })
```

```
edit_snippet = login_required(edit_snippet)
```

Mirando hacia el futuro

Antes de continuar, sugeriría tomarse un poco de tiempo para trabajar con el sistema de formularios de Django. Aunque ya debería tener una buena comprensión de los conceptos básicos, probablemente querrá pasar algún tiempo revisando la documentación completa del paquete django.forms (en línea en <http://docs.djangoproject.com/en/dev/topics/forms/>) para tener una idea de todas sus funciones (incluida la variedad de tipos de campos y widgets, así como trucos más avanzados para personalizar la presentación de formularios).

Cuando esté listo para regresar, el próximo capítulo concluirá esta aplicación agregando las funciones de marcadores y calificación, incluidas listas de los fragmentos más populares y las extensiones de plantilla necesarias para determinar si un usuario ya ha marcado o calificado un fragmento. .

Capítulo 10



Terminando el código compartido Solicitud

Con la adición de los formularios para envíos de usuarios, su aplicación de código compartido es casi completo. Solo quedan tres funciones por implementar de la lista original. Luego puede concluir la aplicación con algunas vistas finales. Empecemos.

Fragmentos de marcadores

Actualmente, los usuarios de su aplicación pueden realizar un seguimiento de sus fragmentos favoritos marcándolos en un navegador web o publicando marcadores en un servicio como Delicious. Sin embargo, sería bueno darle a cada usuario la posibilidad de rastrear una lista personalizada de fragmentos directamente en el sitio. Esto reducirá la cantidad de desorden en los marcadores de propósito general de cada usuario y proporcionará una métrica social útil (fragmentos de la mayoría de los marcadores) que puede rastrear y mostrar públicamente.

Para admitir esta función, primero necesita un modelo que represente el marcador de un usuario. Esto es un modelo bastante simple, porque todo lo que necesita hacer es rastrear algunos datos:

- El usuario al que pertenece el marcador
- El fragmento que el usuario marcó como favorito
- La fecha y la hora en que el usuario marcó el fragmento como favorito

Puede administrar esto abriendo cab/models.py y agregando un nuevo modelo de marcador con tres campos para esta información:

Marcador de clase (modelos.Modelo):

```
fragmento = modelos.ForeignKey(fragmento)
usuario = modelos.ForeignKey(Usuario, related_name='cab_bookmarks')
fecha = modelos.DateTimeFieldeditable=False)
```

metaclase:

```
pedido = ['-fecha']
```

```
def __unicode__(uno mismo):
    devuelve "%s marcado por %s" % (self.snippet, self.user)

def guardar(auto):
    si no self.id:
        self.fecha = fechahora.fechahora.ahora()
    super(Marcador, auto).guardar()
```

Solo hay una característica nueva en uso aquí, y ese es el argumento de nombre_relacionado para la clave foránea que apunta al modelo de usuario. El hecho de que haya creado una clave externa para Usuario significa que Django agregará un nuevo atributo a cada objeto Usuario, que podrá usar para acceder a los marcadores de cada usuario. De forma predeterminada, este atributo se llamaría bookmark_set según el nombre de su modelo de marcador. Por ejemplo, podría consultar los marcadores de un usuario de esta manera:

de django.contrib.auth.models usuario de importación

```
u = Usuario.objetos.get(pk=1)
marcadores = u.bookmark_set.all()
```

Sin embargo, esto puede crear un problema: si alguna vez usa cualquier otra aplicación con un sistema de marcado de libros, y si esa aplicación nombra su modelo Marcador, obtendrá un conflicto de nombres porque el atributo bookmark_set de un Usuario no puede referirse simultáneamente a dos modelos diferentes.

La solución a esto es el argumento related_name para ForeignKey, que le permite manualmente especifique el nombre del nuevo atributo en Usuario, que utilizará para acceder a los marcadores. En este caso, utilizará el nombre cab_bookmarks. Entonces, una vez que este modelo esté instalado y tenga algunos marcadores en su base de datos, podrá ejecutar consultas como esta:

de django.contrib.auth.models usuario de importación

```
u = Usuario.objetos.get(pk=1)
marcadores = u.cab_bookmarks.all()
```

En general, es una buena idea usar related_name cada vez que esté creando una relación. de un modelo con un nombre común.

Además, tenga en cuenta que debido a que los usuarios administrarán sus marcadores completamente a través de vistas, no necesita activar la interfaz de administración para el modelo de marcadores.

Continúe y ejecute manage.py syncdb para instalar el modelo Bookmark en su base de datos. Nuevamente, syncdb es lo suficientemente inteligente como para darse cuenta de que solo necesita crear una nueva tabla.

Adición de vistas básicas de marcadores

Ahora puede agregar un par de vistas para permitir que los usuarios marquen fragmentos y eliminen sus marcadores más tarde si lo desean. Cree un archivo en cab/views llamado bookmarks.py y comience con la vista add_bookmark:

```
de django.http importar HttpResponseRedirect
desde django.shortcuts import get_object_or_404, render_to_response
desde django.contrib.auth.decorators import login_required
desde cab.models import Bookmark, Snippet
```

```
def add_bookmark(solicitud, snippet_id):
    fragmento = get_object_or_404(Fragmento, pk=id_fragmento)
    probar:
        Bookmark.objects.get(user__pk=request.user.id,
                             fragmento__pk=fragmento.id)
    excepto Bookmark.DoesNotExist:
        marcador = marcador.objetos.create(usuario=solicitud.usuario,
                                             fragmento=fragmento)
    devolver HttpResponseRedirect(fragmento.get_absolute_url())
add_bookmark = login_required(add_bookmark)
```

La lógica aquí es bastante simple. Comprueba si el usuario ya tiene un marcador para este fragmento y, en caso contrario, en cuyo caso se generará la excepción `Bookmark.DoesNotExist`: crea uno. De cualquier manera, devuelve una redirección al fragmento y, por supuesto, se asegura de que el usuario debe iniciar sesión para hacer esto.

Eliminar un marcador es igualmente fácil:

```
def delete_bookmark(solicitud, snippet_id):
    if solicitud.método == 'POST':
        fragmento = get_object_or_404(Fragmento, pk=id_fragmento)
        Bookmark.objects.filter(user__pk=request.user.id,
                               fragmento__pk=fragmento.id).delete()
        devolver HttpResponseRedirect(fragmento.get_absolute_url())
    más:
        volver render_to_response('cab/confirm_bookmark_delete.html',
                                  { 'fragmento': fragmento })
delete_bookmark = login_required(delete_bookmark)
```

Con la vista `delete_bookmark`, está utilizando dos técnicas importantes:

- En lugar de consultar si el usuario tiene un marcador para este fragmento y luego eliminarlo manualmente (lo que genera la sobrecarga de dos consultas a la base de datos), simplemente use `filter()` para crear un `QuerySet` de cualquier marcador que coincida con este usuario y este fragmento. Luego llama al método `delete()` de ese `QuerySet`. Esto emite solo una consulta: una consulta `DELETE`, cuya cláusula `FROM` la limita a las filas correctas, si existen.
- Está solicitando que la eliminación de marcadores use un HTTP POST. Si el método de solicitud no es POST, muestra una página de confirmación en su lugar.

Vale la pena enfatizar este último punto, porque requerir HTTP POST y una pantalla de confirmación para cualquier cosa que elimine contenido, incluso contenido que parezca trivial como un marcador, es un hábito extremadamente importante en el que se debe entrar. No solo evita la eliminación accidental por parte de un usuario que hace clic en el enlace incorrecto de una página, sino que también agrega una pequeña medida de seguridad contra un tipo común de ataque basado en la web: la falsificación de solicitud entre sitios (CSRF). En un ataque CSRF, un pirata informático atrae a un usuario de su sitio a una página que contiene un enlace oculto o un formulario que apunta a su aplicación. El hacker aprovecha el hecho de que debido a que las solicitudes HTTP provienen del usuario, muchas aplicaciones permiten la modificación o eliminación de contenidos.

Además, generalmente es una buena práctica solicitar POST para cualquier operación que altere o elimine datos en el servidor. La especificación HTTP establece que ciertos métodos, incluido GET, deben considerarse seguros y, en general, no deben tener efectos secundarios.

Advertencia: métodos HTTP seguros e idempotentes

Se puede acceder a la vista que ha escrito para agregar un marcador a través de HTTP GET, lo que parece contradecir la idea de que este tipo de vista debería ser segura.

La especificación HTTP utiliza dos términos diferentes pero relacionados para describir los métodos de solicitud: seguro e idempotente. Una solicitud segura es aquella que no tiene efectos secundarios y simplemente recupera alguna información, mientras que una solicitud idempotente es aquella en la que el efecto de múltiples solicitudes idénticas es el mismo que el efecto de una solicitud. HTTP requiere que las solicitudes GET sean idempotentes, pero no requiere estrictamente que sean seguras.

La vista `add_bookmark` es idempotente, porque varias solicitudes del mismo usuario para marcar el mismo fragmento no crean varios objetos Bookmark. El efecto neto es el mismo que si solo hubiera una solicitud, porque solo se crea un objeto Bookmark.

Sin embargo, la vista `add_bookmark` no es segura en este sentido, porque puede tener un efecto secundario (crear un objeto Bookmark). Esto no viola la especificación HTTP, pero en general, debe tener cuidado al permitir que una solicitud GET tenga efectos secundarios. En este caso, crear un marcador realmente no representa un riesgo. Si alguien fuera engañado para hacer clic en un enlace para marcar un fragmento, por ejemplo, lo peor que podría pasar sería que tuviera que eliminar el marcador. Por lo tanto, generalmente es aceptable permitir que la creación de marcadores se realice a través de una solicitud GET.

Crear una plantilla para la página de confirmación es bastante fácil. Puede mostrar información sobre el fragmento que el usuario está a punto de "desmarcar" y luego puede incluir un formulario simple que envía la confirmación a través de POST:

```
<método de formulario="publicar" acción="">
<p><tipo de entrada="enviar" valor="Eliminar marcador"></p>
</formulario>
```

Advertencia: mayor protección contra CSRF

Requerir un HTTP POST ayuda un poco contra CSRF, porque significa que un atacante no puede simplemente mostrar un enlace a una página en particular y hacer que eso active la eliminación del contenido. Sin embargo, para una protección completa, querrá consultar y habilitar `django.contrib.csrf`, una aplicación incluida con Django que proporciona algunas medidas más sólidas. Inserta y verifica automáticamente una cadena generada aleatoriamente en un POST entrante envío, y devuelve una respuesta HTTP 403 (Prohibido) si el navegador del usuario no publica esa cadena.

Puede encontrar la documentación completa para este sistema en línea en <http://docs.djangoproject.com/en/dev/ref/contrib/csrf/>.

Es bastante fácil configurar direcciones URL para agregar y eliminar marcadores. Puedes crear `cabina/urls/bookmarks.py` y comienza a completarlo:

```
desde django.conf.urls.defaults importar desde
cab.views importar marcadores
```

*

```
urlpatrones = patrones(",
    url(r'^añadir/(?P<fragmento_id>\d+)/$',,
        marcadores.add_bookmark,
        nombre='taxi_bookmark_add'),
    url(r'^eliminar/(?P<fragmento_id>\d+)/$',,
        marcadores.eliminar_marcador,
        nombre = 'taxi_bookmark_delete'),
)
```

Ahora que tiene vistas para administrar marcadores, continúe y escriba una para mostrar una lista de los favoritos del usuario actual. Esto es solo un envoltorio alrededor de object_list vista genérica:

```
de django.views.generic.list_detail importar object_list
```

```
def user_bookmarks(solicitud):
    devuelve object_list(queryset=Bookmark.objects.filter(user__pk=request.user.id),
        template_name='taxi/user_bookmarks.html',
        paginar_por=20)
```

Puede configurar una URL para la vista de modo que la raíz de las URL de marcadores simplemente muestre los marcadores del usuario:

```
url(r'^$', marcadores.user_bookmarks, name='cab_user_bookmarks'),
```

Finalmente, para completar las vistas orientadas a marcadores, agregue una que consulte los fragmentos más marcados. Debido a que esta consulta devuelve objetos Snippet, colóquelo en SnippetManager en cab/managers.py:

```
def most_bookmarked(self):
    return self.annotate(score=Count('bookmark')).order_by('score')
```

Ahora escriba la vista más marcada en cab/views/popular.py:

```
def most_bookmarked(solicitud):
    volver object_list(queryset=Snippet.objects.most_bookmarked(),
        template_name='taxi/most_bookmarked.html',
        paginar_por=20)
```

Luego agregue el patrón de URL en cab/urls/popular.py:

```
url(r'^marcadores/$', popular.más_marcados, nombre='cabina_más_marcados'),
```

Creación de una nueva etiqueta de plantilla:

{% if_bookmarked %}

Para ir con las vistas add_bookmark y delete_bookmark, es posible que desee indicar al mostrar un fragmento si un usuario ya lo ha marcado. De esa forma, puede ocultar los enlaces a las vistas de marcadores que de otro modo podría mostrar o cambiar para mostrar un enlace o un botón para eliminar el marcador.

Puede configurar esto para que forme parte de la vista de detalles del fragmento, pero ese no es necesariamente el único lugar donde puede desear esta funcionalidad. Si está mostrando una lista de fragmentos, por ejemplo, es posible que desee una forma rápida y fácil de determinar dónde mostrar un enlace para marcarlo y dónde no. La solución ideal sería una etiqueta de plantilla, que puede indicar si un usuario ya ha marcado un fragmento específico. Algo que funcione así sería ideal:

```
{% objeto de usuario if_bookmarked %}

<form method="post" action="{% url cab_bookmark_delete object.id %}">
    <p><tipo de entrada="enviar" valor="Eliminar marcador"></p>
</formulario>

{% más %}

<p><a href="{% url cab_bookmark_add object.id %}">Agregar marcador</a></p>
{% endif_marcado %}
```

Advertencia: cableado de las URL

Debido a que está usando la etiqueta `{% url %}` para generar el enlace a la vista `add_bookmark`, debe agregar las URL para la aplicación `cab` al módulo `URLConf` raíz de su proyecto (a través de `llamadas include()`). Si usa la etiqueta `{% url %}` con un nombre de URL que aún no ha configurado en su proyecto, no podrá encontrar la URL correcta y simplemente devolverá una cadena vacía en lugar de una URL.

Pero, ¿cómo puedes escribir esto? Hasta ahora, todas sus etiquetas de plantillas personalizadas han sido bastante simples. Por lo general, solo leen sus argumentos y escupen algo en el contexto.

Escribir esta etiqueta requiere dos nuevas técnicas:

- La capacidad de escribir una etiqueta que se adelanta un poco en la plantilla para encontrar, por ejemplo, el `{% else %}` cláusula y la etiqueta de cierre, y realiza un seguimiento de lo que se muestra
- La capacidad de resolver variables arbitrarias del contexto de la plantilla, como en el caso de un variable como objeto

Afortunadamente, ambos son bastante fáciles de lograr.

Analizando hacia adelante en una plantilla de Django

Recordará del Capítulo 6 cuando escribió sus primeras etiquetas de plantilla personalizadas que la función de compilación para una etiqueta recibe dos argumentos, llamados convencionalmente analizador y token. En ese momento, solo le preocupaba la parte del token porque contenía los argumentos que le interesaban. Sin embargo, ahora se encuentra en una situación en la que el analizador, que es el objeto real que analiza la plantilla, será útil. .

Antes de profundizar demasiado, avancemos y diseñemos la infraestructura para la costumbre. etiqueta. En el directorio `cab`, cree un nuevo directorio llamado `templatetags`, y en ese directorio, cree dos nuevos archivos: `__init__.py` y `snippets.py`. Luego, abre `cab/templatetags/snippets.py` y complete un par de importaciones necesarias:

desde la plantilla de importación de django
desde la importación de `cab.models`

Ahora, puede comenzar a escribir la función de compilación para la etiqueta `{% if_bookmarked %}`:

```
def do_if_bookmarked(analizador, token):
    bits = token.contenido.split()
    si len(bits) != 3:
        aumentar plantilla.TemplateSyntaxError("La etiqueta %s toma dos argumentos" % bits[0])
```

Esta función de compilación examina la sintaxis utilizada para llamar a la etiqueta, que tiene el formato `{% if_bookmarked user snippet %}`, y verifica que tenga la cantidad correcta de argumentos, y si no es así, se recupera inmediatamente con un `TemplateSyntaxError`.

Ahora puede centrar su atención en el argumento del analizador y ver cómo puede ayudarlo.

Quiere seguir leyendo en la plantilla hasta que encuentre un `{% else %}` o un `{% endif_` etiqueta marcada `%}`. Puede hacerlo llamando al método `parse()` del objeto analizador y pasando una lista de cosas que le gustaría que busque. El resultado de este análisis será una instancia de la clase `django.template.NodeList`, que es, como su nombre lo indica, una lista de nodos de plantilla:

```
nodelist_true = parser.parse(('otro', 'endif_marcado'))
```

Está almacenando este resultado en una variable llamada `nodelist_true` porque, en términos de esta etiqueta Comportamiento de estilo `if/else`: corresponde a la salida que desea mostrar si la condición es verdadera (si el usuario ha marcado el fragmento).

La llamada a `parser.parse()` avanza en la plantilla justo *antes* del primer elemento de la lista que le indicó que buscara. Esto significa que ahora desea mirar el siguiente token y averiguar si es un `{% else %}`. Si es así, tendrás que hacer un poco más de análisis:

```
token = analizador.next_token()
if token.contents == 'otro':
    nodelist_false = parser.parse(('endif_marcado',))
    analizador.delete_first_token()
más:
    nodelist_false = plantilla.NodeList()
```

Si lo primero que encuentra el analizador de su lista es de hecho un `{% else %}`, entonces querrá leer de nuevo hasta `{% endif_bookmarked %}` para que el resultado se muestre cuando el usuario *no lo haya hecho*. marcó el fragmento. Esta es otra lista de nodos, que almacena en la variable `nodelist_falso`.

Si, por otro lado, el analizador encuentra un `{% endif_bookmarked %}` sin `{% else %}`, simplemente crea una lista de nodos vacía. Si el usuario no ha marcado el fragmento como favorito, entonces no debería mostrar nada cuando no haya una cláusula `{% else %}`.

Finalmente, devuelve una clase `Node`, pasando los dos argumentos recopilados de la etiqueta y las dos instancias de `NodeList`. Aunque aún no lo ha definido, la clase de `Nodo` que va a utilizar se llamará `IfBookmarkedNode`:

```
devolver IfBookmarkedNode(bits[1], bits[2], nodelist_true, nodelist_false)
```

Resolución de variables dentro de un nodo de plantilla

Ahora puede comenzar a escribir `IfBookmarkedNode`. Obviamente, necesita una plantilla de subclase. `Node`, y necesita aceptar cuatro argumentos en su método `__init__()`. Simplemente almacenará las dos instancias de `NodeList` para su uso posterior cuando represente la plantilla:

```
clase IfBookmarkedNode(template.Node):
    def __init__(self, usuario, fragmento, nodelist_true, nodelist_false):
        self.nodelist_true = nodelist_true
        self.nodelist_false = nodelist_false
```

Pero, ¿qué pasa con las variables de usuario y fragmento? En este momento, son las cuerdas en bruto de la plantilla, y aún no sabe a qué valores se resolverán realmente cuando observe el contexto. Necesita alguna forma de decir que estas son en realidad variables de plantilla que debe resolver más adelante. Afortunadamente, eso es bastante fácil de hacer:

```
self.usuario = plantilla.Variable(usuario)
self.fragmento = plantilla.Variable(fragmento)
```

La clase Variable en django.template se encarga del trabajo duro por usted. Cuando se le da el contexto de la plantilla para trabajar, sabe cómo resolver la variable y le devuelve el valor real al que corresponde.

Ahora puede comenzar a escribir el método render():

```
def render(auto, contexto):
    usuario = self.user.resolve(contexto)
    fragmento = self.fragmento.resolve(contexto)
```

Cada instancia de Variable tiene un método llamado resolve(), que maneja el negocio real de resolver la variable. Si resulta que la variable no corresponde a nada, incluso se encargará de generar una excepción (django.template.VariableDoesNotExist) automáticamente para usted. Por supuesto, ha visto que, por lo general, es una buena idea que las etiquetas de plantillas personalizadas fallen silenciosamente cuando sea posible, así que detecte esa excepción y simplemente haga que la etiqueta no devuelva nada cuando una de las variables no sea válida:

```
def render(auto, contexto):
    probar:
        usuario = self.user.resolve(contexto)
        fragmento = self.fragmento.resolve(contexto)
    excepto plantilla.VariableDoesNotExist:
        devolver
```

Si supera este punto, sabrá que estas variables se resolvieron correctamente y puede usarlas para consultar un marcador existente. Lo único complicado ahora es averiguar qué devolver en cada caso. Tiene dos instancias de NodeList y desea representar una u otra según si el usuario ha marcado el fragmento. Afortunadamente, eso es fácil.

Así como un nodo debe tener un método render() que acepte el contexto y devuelva una cadena, también debe hacerlo NodeList:

```
si Marcador.objetos.filtro(usuario__pk=usuario.id,
                           fragmento__pk=fragmento.id):
    devolver self.nodelist_true.render(contexto)
más:
    devolver self.nodelist_false.render(contexto)
```

Ahora tienes una etiqueta terminada. Despu s de registrarlo, cab/templatetags/snippets.py se ve as :

```
desde la plantilla de importaci n de  
django desde la importaci n de cab.models
```

```
def do_if_bookmarked(parser, token): bits  
    = token.contents.split() if len(bits) !=  
    3: raise template.TemplateSyntaxError("La  
        etiqueta %s toma dos argumentos" % bits[0])  
    nodelist_true = parser.parse(('else', 'endif_bookmarked')) token =  
    parser.next_token() if token.contents == 'else':  
  
        nodelist_false = parser.parse('endif_bookmarked',)  
        parser.delete_first_token() m s:  
  
        nodelist_false = plantilla.NodeList()  
        devolver IfBookmarkedNode(bits[1], bits[2], nodelist_true, nodelist_false)
```

```
clase IfBookmarkedNode(template.Node):  
    def __init__(self, usuario, fragmento, nodelist_true, nodelist_false):  
        self.nodelist_true = nodelist_true  
        self.nodelist_false = nodelist_false self.user  
        = plantilla.Variable(usuario) self.snippet =  
        plantilla.Variable(fragmento)  
  
    def render(self, context):  
        prueba: usuario =  
            self.user.resolve(context) snippet =  
            self.snippet.resolve(context)  
        excepto plantilla.VariableDoesNotExist:  
            devolver  
        if Bookmark.objects.filter(user__pk=user.id,  
                                    snippet__pk=snippet.id):  
            return self.nodelist_true.render(context)  
  
        devolver self.nodelist_false.render(contexto)
```

```
registro = plantilla.Librer a()  
registro.etiqueta('si_marcado', do_si_marcado)
```

Ahora puede simplemente hacer {cargar fragmentos %} en una plantilla y usar la etiqueta {% if_bookmarked %}.

Uso de RequestContext para completar automáticamente Variables de plantilla

Pero solo puede usar la etiqueta `{% if_bookmarked %}` si la plantilla en la que está usando la etiqueta tiene una variable disponible que representa al usuario que ha iniciado sesión actualmente. Esta es una propuesta un poco más complicada porque hasta ahora no ha escrito sus vistas para pasar al usuario actual como una variable a las plantillas que usa. Principalmente eso se debe a que no ha tenido mucha necesidad de hacerlo.

Ha estado haciendo todo con el usuario que inició sesión en el nivel de vista al acceder a la solicitud. usuario, por lo que realmente no se ha encontrado con un caso, hasta ahora, en el que realmente necesite tener una variable para el usuario disponible en las plantillas.

Simplemente podría regresar a este punto y hacer el cambio necesario en todas sus vistas escritas a mano, pero eso inmediatamente trae dos desventajas:

- **Es tedioso y repetitivo:** generalmente, Django lo alienta a evitar cualquier cosa que se puede describir de esa manera.
- **No ayuda para vistas que no escribiste tú mismo:** en muchos casos, simplemente estás envolviendo una vista genérica, y aparte de pasar manualmente el argumento `extra_context` cada vez que usa una vista genérica, no parece haber ninguna forma de resolver esto. Más, este enfoque podría no ser útil si necesita usar vistas de la aplicación de otra persona. Si esa persona no ha escrito vistas para aceptar un argumento similar a `extra_context`, no podrá hacer nada.

Afortunadamente, hay una solución más fácil. Como recordará de las primeras vistas escritas a mano en el Capítulo 3, el diccionario de variables y valores pasados a una plantilla es una instancia de `django.template.Context`. Debido a que esta es una clase ordinaria de Python, puede subclasicarla para agregue un comportamiento personalizable. Django incluye una subclase de Contexto muy útil: `Django.template.RequestContext`: que puede completar automáticamente algunas variables adicionales cada vez que se usa *sin* necesidad de que esas variables se declaren y definan explícitamente en cada vista.

`RequestContext` recibe su nombre del hecho de que hace uso de funciones llamadas *procesadores de contexto* (que mencioné brevemente en el Capítulo 6). Cada procesador de contexto es una función que recibe un objeto Django `HttpRequest` como argumento y devuelve un diccionario de variables basado en ese `HttpRequest`. `RequestContext` luego agrega automáticamente esas variables al contexto, además de cualquier variable que se haya pasado explícitamente al contexto durante el proceso de ejecución de una función de vista.

En uso normal, `RequestContext` lee su lista de funciones de procesador de contexto de la configuración `TEMPLATE_CONTEXT_PROCESSORS`. El conjunto predeterminado incluye un procesador de contexto que lee `request.user` para obtener el usuario actual y lo agrega al contexto como la variable `{{ usuario }}`. Esto resulta ser exactamente lo que quieras aquí. Siempre que una vista use `RequestContext`, su plantilla puede confiar en el hecho de que la variable `{{ usuario }}` estará disponible y corresponderá al usuario actualmente activo.

Usar `RequestContext` es trivialmente fácil; simplemente lo importas:
de `django.template import RequestContext`

Puede usarlo en cualquier lugar donde necesite un contexto para una plantilla. La única diferencia entre un Contexto normal y un `RequestContext` es que este último debe recibir el objeto `HttpRequest` como argumento. Por ejemplo, en una vista, podría escribir esto:

```
contexto = RequestContext(solicitud, { 'foo': 'bar' })
```

También funciona con el atajo render_to_response(), aunque el uso es ligeramente diferente. Por ejemplo, donde normalmente escribirías esto:

```
volver render_to_response('ejemplo.html',
                           { 'foo': 'barra' })
```

En su lugar, escribirías esto:

```
volver render_to_response('ejemplo.html',
                           { 'foo': 'barra' },
                           context_instance=RequestContext(solicitud))
```

Y para los casos en los que está envolviendo una vista genérica, ni siquiera tiene que hacer nada: Las vistas genéricas de Django usan por defecto RequestContext. Hasta ahora, ha escrito solo tres vistas en esta aplicación que no usan vistas genéricas: las vistas delete_bookmark, add_snippet y edit_snippet, para ser precisos, por lo que no es demasiado difícil volver atrás y agregarles el uso de RequestContext. . Debido a que el resto son vistas genéricas o envuelven vistas genéricas, ya están usando RequestContext.

Advertencia: usar RequestContext de forma repetitiva

Aunque RequestContext obviamente hace que sea mucho más fácil manejar situaciones en las que desea tener ciertas variables disponibles globalmente para sus plantillas, indicar manualmente que lo desea cada vez todavía se siente un poco repetitivo. Y si las vistas genéricas usan RequestContext automáticamente, ¿por qué un atajo como render_to_response() no debería usarlo también? De hecho, ¿por qué no es solo la clase de contexto predeterminada?

Una buena razón es el hecho de que RequestContext requiere acceso al objeto HttpRequest y no hay forma de que obtenga ese acceso automáticamente. A menos que se le pase HttpRequest explícitamente, RequestContext no podrá hacer nada. Otra buena razón es que, en muchos casos, querrá representar una plantilla independientemente de cualquier solicitud HTTP que se esté procesando. No es inusual que el sistema de plantillas de Django se utilice para generar mensajes de correo electrónico, archivos que se escriben en el disco y otros elementos que tienen poco que ver directamente con el ciclo de solicitud/respondida HTTP.

Sin embargo, si te encuentras anhelando un atajo, puedes escribir uno fácilmente:

```
de django.shortcuts import render_to_response
de django.template import RequestContext
```

```
def render_response(solicitud, *argumentos, **kwargs):
    kwargs['context_instance'] = RequestContext(solicitud)
    volver render_to_response(*args, **kwargs)
```

Personalmente, tiendo a evitar hacer esto y, como cuestión de estilo, prefiero simplemente escribir el uso de RequestContext cada vez. Me parece que hacerlo me sirve como un recordatorio de que estoy configurando una vista para tener las variables adicionales que completará RequestContext. Además, el código adicional para configurarlo hace que sea fácil de detectar cuando vuelvo más tarde y leo una función de vista. El manejo manual de RequestContext también evita el problema de escribir código que depende en gran medida de una función de acceso directo que podría no distribuirse junto con una aplicación en particular, lo que a su vez mejora la reutilización de su código.

Adición del sistema de clasificación de usuarios

Lo único que queda por implementar de la lista de características es un sistema de calificación que permite a los usuarios marcar fragmentos particulares que encontraron útiles (o no útiles, según sea el caso). Una vez más, comience con un modelo de datos. Al igual que con el sistema de marcadores, es bastante simple. Necesita recopilar cuatro piezas de información:

- El fragmento que se califica
- El usuario que hace la calificación
- El valor de la calificación, en este caso, un +1 o un -1, para una simple votación "hacia arriba o hacia abajo" sistema
- La fecha de la calificación

Puede crear fácilmente este modelo de calificación en cab/models.py:

Clasificación de clase (modelos.Modelo):

```
CALIFICACIÓN_AUMENTADA = 1
CALIFICACIÓN_ABAJO = -1
RATING_CHOICES = ((RATING_UP, 'útil'),
                  (RATING_DOWN, 'no útil'))
fragmento = modelos.ForeignKey(fragmento)
usuario = modelos.ForeignKey(Usuario, related_name='cab_rating')
calificación = modelos.IntegerField(choices=RATING_CHOICES)
fecha = modelos.DateTimeField()

def __unicode__(uno mismo):
    return "%s rating %s (%s)" % (self.user, self.snippet,
                                   self.get_rating_display())

def guardar(auto):
    si no self.id:
        self.fecha = fechahora.fechahora.ahora()
    super(Valoración, auto).guardar()
```

Al igual que con el modelo de marcadores, está configurando related_name explícitamente en la relación con el modelo de usuario para evitar posibles conflictos de nombres con otras aplicaciones que puedan definir sistemas de clasificación. Mientras tanto, el valor de calificación usa un campo entero, con constantes con nombres apropiados, para manejar los valores reales de calificación "hacia arriba" y "hacia abajo", de la misma manera que el campo de estado en el modelo de Entrada del weblog. Sin embargo, hay un elemento nuevo: en el método __unicode__(), está llamando a un método llamado get_rating_display(). Cada vez que un modelo tiene un campo con opciones como esta, Django agrega automáticamente un método, cuyo nombre se deriva del nombre del campo, que devolverá el valor legible por humanos para el valor actualmente seleccionado.

Mientras está en el archivo cab/models.py, también puede agregar un método al modelo Snippet que calcula la puntuación total de un fragmento sumando todas las calificaciones adjuntas. Este método volverá a usar el soporte agregado de Django, pero con un tipo diferente de filtro agregado: django.db.models.Sum. Este filtro, como su nombre lo indica, suma un conjunto de valores en la base de datos y devuelve la suma.

También utilizará un método diferente para aplicar el agregado. Anteriormente, usó la función de anotación porque necesitaba agregar información adicional a los resultados devueltos por la consulta. Pero ahora solo desea devolver directamente el valor agregado y nada más, por lo que usará un método diferente llamado agregado. Si tiene un objeto Snippet en una variable llamada snippet y desea la suma de todas las calificaciones adjuntas, puede escribir la consulta de esta manera:

```
de django.db.models importar Suma  
calificación_total = fragmento.conjunto_calificación.agregado(Suma('calificación'))
```

Luego puede agregar esta funcionalidad como un método get_score en el modelo Snippet (recuerde bajar para colocar la declaración de importación para el agregado Sum en la parte superior del archivo models.py):

```
def get_score(self):  
    devuelve self.rating_set.aggregate(Sum('rating'))
```

Finalmente, en cab/managers.py, puede agregar un método más en SnippetManager para calcular los fragmentos mejor calificados (nuevamente, recuerde agregar la declaración de importación para Sum agregar):

```
def top_rated(self):  
    return self.annotate(score=Sum('rating')).order_by('score')
```

Esto se ocupa de todas las consultas personalizadas que necesitará, así que continúe y ejecute manage.py syncdb para instalar el modelo de calificación.

Fragmentos de calificación

Permitir que los usuarios califiquen fragmentos es bastante fácil. Todo lo que necesita es una vista que obtenga una identificación de fragmento y una calificación "hacia arriba" o "hacia abajo", luego agregue un nuevo objeto de calificación. La lógica de la vista es simple. Cree un archivo de vista más, cab/views/ratings.py, y coloque este código en él:

```
de django.http importar HttpResponseRedirect  
desde django.shortcuts import get_object_or_404  
desde django.contrib.auth.decorators import login_required  
de cab.models import Clasificación, Fragmento
```

tasa de definición (solicitud, snippet_id):

```
fragmento = get_object_or_404(Fragmento, pk=id_fragmento)  
si 'calificación' no está en request.GET o request.GET['rating'] no está en ('1', '-1'):  
    devolver HttpResponseRedirect(fragmento.get_absolute_url())  
probar:  
    calificación = Clasificación.objetos.get(user__pk=request.user.id,  
                                              fragmento__pk=fragmento.id)  
excepto Rating.DoesNotExist:  
    calificación = Clasificación (usuario = solicitud.usuario,  
                               fragmento=fragmento)  
    rating.rating = int(solicitud.GET['rating'])  
    calificación.guardar()  
    devolver HttpResponseRedirect(fragmento.get_absolute_url())  
tasa = login_required(tasa)
```

Solo dos cosas moderadamente complicadas están sucediendo aquí:

- Esperará que se acceda a esta vista con una cadena de consulta como ?rating=1 o ?rating=-1, por lo que verifica que esta cadena está presente y que tiene un valor aceptable. De lo contrario, simplemente redirija de nuevo al fragmento.
- Para evitar que un usuario llene la boleta intentando calificar el mismo fragmento una y otra vez, asegúrese de que la vista simplemente cambie el valor de una calificación existente si se encuentra una.

Configurar la URL para esta vista debería ser bastante fácil. Simplemente puede agregar un archivo cab/urls/ratings.py y configurar el patrón de URL necesario:

```
desde django.conf.urls.defaults importar desde *  
cab.views.ratings tasa de importación
```

```
urlpatrones = patrones(",  
    url(r'^(?P<fragmento_id>\d+)$', tarifa, nombre='taxi_fragmento_tarifa'),  
)
```

Adición de una etiqueta de plantilla { % if_rated %} Continúe y agregue

una etiqueta de plantilla { % if_rated %} que se asemeje a la etiqueta { % if_bookmarked %} que desarrolló anteriormente en este capítulo. La función de compilación debería parecerle familiar (una vez más, esto va a cab/templatetags/snippets.py):

```
def do_if_rated(analizador, token):  
    bits = token.contents.split() if len(bits) != 3: raise template.TemplateSyntaxError("La  
    etiqueta %s toma dos argumentos" % bits[0]) nodelist_true = parser.parse(('else', 'endif_rated'))  
    token = parser.next_token() if token.contents == 'else':  
  
        nodelist_false = parser.parse('endif_rated'))  
        parser.delete_first_token() más:  
  
        nodelist_false = plantilla.NodeList()  
        devolver IfRatedNode(bits[1], bits[2], nodelist_true, nodelist_false)
```

Una vez más, utiliza la capacidad de analizar adelante en la plantilla para calcular la estructura de las posibilidades if/else para la etiqueta y almacenar un par de instancias de NodeList para pasar como argumentos a la clase Node, que puede llamar IfRatedNode. Primero, debe cambiar la declaración de importación en la parte superior del archivo de

desde la importación de cab.models

a

desde cab.models import Marcador, Calificación

Entonces puedes escribir la clase IfRatedNode:

```

clase IfRatedNode(plantilla.Nodo):
    def __init__(self, usuario, fragmento, nodelist_true, nodelist_false):
        self.nodelist_true = nodelist_true
        self.nodelist_false = nodelist_false
        self.user =
            plantilla.Variable(usuario)
        self.snippet =
            plantilla.Variable(fragmento)

    def render(self, context): prueba:
        usuario =
            self.user.resolve(context)
        snippet =
            self.snippet.resolve(context)
        excepto plantilla.Variable.DoesNotExist:
            devolver
        if Rating.objects.filter(user__pk=user.id,
            snippet__pk=snippet.id): return
            self.nodelist_true.render(context) else: return
        self.nodelist_false.render(context)

```

En la parte inferior del archivo, puede registrar la etiqueta:

```
registrarse.tag('si_clasificado', hacer_si_clasificado)
```

Recuperación de la calificación de un

usuario Ahora que tiene la etiqueta { % if_rated % }, puede agregar una segunda etiqueta complementaria para recuperar la calificación del usuario para un fragmento en particular. Esta nueva etiqueta { % get_rating % } te permite configurar una plantilla como esta:

```

{% load snippets %}
if_rated user snippet %
    get_rating user snippet as rating %} <p>Calificó
    este fragmento como <strong>{{ rating.get_rating_display }}</strong>.</p>
{% endif_rated%}

```

Cuando un usuario ha calificado un fragmento, este código debería terminar mostrando algo como "Calificó este fragmento como útil".

La función de compilación de esta nueva etiqueta, do_get_rating, es sencilla:

```

def do_get_rating(parser, token):
    bits =
        token.contents.split() if len(bits) != 5:
        raise template.TemplateSyntaxError("La
            etiqueta %s toma cuatro argumentos" % bits[0]) if bits[3] != 'como': raise
        template.TemplateSyntaxError("El tercer argumento para %s debe ser 'como'" % bits[0]) return
        GetRatingNode(bits[1], bits[2], bits[4])

```

La clase Node, a la que llamará GetRatingNode, también es fácil de escribir. solo necesitas resuelva las variables de usuario y fragmento, recupere la calificación y colóquela en el contexto:

```
clase GetRatingNode(plantilla.Nodo):
    def __init__(self, usuario, fragmento, varname):
        self.usuario = plantilla.Variable(usuario)
        self.fragmento = plantilla.Variable(fragmento)
        self.varname = varname

    def render(auto, contexto):
        probar:
            usuario = self.usuario.resolve(contexto)
            fragmento = self.fragmento.resolve(contexto)
        excepto plantilla.Variable.DoesNotExist:
            devolver
            calificación = Calificación.objetos.get(user__pk=user.id,
                                                    fragmento__pk=fragmento.id)
        contexto[self.varname] = calificación
        devolver
```

A continuación, registra la etiqueta:

```
registrarse.tag('get_rating', do_get_rating)
```

Luego puede usar la etiqueta de esta manera (en la vista detallada de un fragmento, por ejemplo):

```
{% cargar fragmentos %}
{% objeto de usuario if_rated %}
    {% fragmento de usuario get_rating como calificación %}
        <p>Califícate este fragmento {{ rating.get_rating_display }}.</p>
    {% más %}
        <p>Califica este fragmento:
            <a href="{% url cab_snippet_rate object.id %}?rating=1">útil</a> o
            <a href="{% url cab_snippet_rate object.id %}?rating=-1">no útil</a>.</p>
    {% endif_rated%}
```

Mirando hacia el futuro

En este punto, ha implementado todo en su lista de funciones original para la aplicación de código compartido. Los usuarios pueden enviar y editar fragmentos, etiquetarlos y ordenarlos por idioma. También tiene funciones de marcadores y calificaciones, así como algunas vistas agregadas para mostrar cosas como los fragmentos mejor calificados y más marcados y los idiomas más utilizados. En el camino, aprendió a trabajar con el sistema de formularios de Django y aprendió algunos trucos avanzados para trabajar con el mapeador relacional de objetos y el motor de plantillas.

Por supuesto, aún podría agregar muchas más funciones en este punto:

- Dando seguimiento a sus experiencias con la aplicación weblog, podría agregar fácilmente comentarios (con moderación) y feeds.
- Puede tomar prestadas las etiquetas de plantilla de recuperación de contenido que escribió para el weblog y utilícelos para recuperar los fragmentos más recientes o adáptelos para realizar algunas de las consultas personalizadas que ha escrito para esta aplicación.
- Podría crear un montón de nuevas vistas y consultas; incluso con el simple conjunto de modelos que tiene aquí, hay mucho espacio para formas interesantes de explorar esta aplicación, y lo que ha configurado hasta ahora solo rasca la superficie.
- Podría explorar formas de integrar esta aplicación con algunas de las otras que ha escrito y utilizado (tal vez un sitio de código compartido con un blog que señala los fragmentos favoritos del personal del sitio).

A estas alturas, ha llegado a un punto en el que puede comenzar a desarrollar estas funciones en su Adquiera y adapte esta aplicación para que funcione exactamente de la manera que usted desea. Considere algunas de estas ideas y piense en cómo las implementaría, luego siéntese y escriba el código. Luego comience a pensar en algunas cosas que le gustaría que *no estén* en la lista anterior, y pruébelas también. Porque si has llegado hasta aquí, estás listo para hacer uso de tu conocimiento y poner a Django a trabajar para ti.

En reconocimiento de eso, no voy a dictar más listas de funciones o implementaciones. para ti. En cambio, en los próximos dos capítulos, cambiaré un poco de tema y hablaré sobre algunas de las mejores prácticas generales para desarrollar sus aplicaciones Django y aprovecharlas al máximo.

CAPÍTULO 11



Desarrollo práctico Técnicas

Hasta que ahora, se ha centrado en lo que pueden hacer las bibliotecas y los componentes de Django y cómo puede aprovecharlos en sus aplicaciones. Pero el código de Django y el código que escribes usando Django constituyen solo la mitad de la historia cuando se trata de un desarrollo web práctico y eficiente. La otra mitad consiste en técnicas más generales y mejores prácticas específicas. Algunos de estos se aplican ampliamente a cualquier tipo de desarrollo de software, mientras que otros se aplican más específicamente a Python, Django y la Web. En cualquier caso, comprenden un sólido conjunto de pautas que pueden mejorar drásticamente su productividad y su capacidad para producir continuamente código funcional y útil.

En este capítulo, repasaré algunas técnicas para organizar, mantener e implementar su código. Mientras lee, tenga en cuenta que la mayoría de estos temas son lo suficientemente extensos como para llenar libros enteros. En su mayor parte, proporcionaré una breve descripción general de una práctica determinada y algunos ejemplos que muestran cómo es útil, pero debe continuar con su propia investigación para encontrar las herramientas y técnicas específicas que mejor se adapten a sus necesidades.

Uso de sistemas de control de versiones para rastrear su código

Hay un problema común que afecta a todos los desarrolladores de software tarde o temprano; me ha afectado más veces de las que puedo contar, y si pasas mucho tiempo escribiendo código, eventualmente también te morderá. Este némesis universal es el ataque del insecto repentino.

Imagina que tienes una aplicación en la que has estado trabajando durante un tiempo. Has implementado mencionó todas las características en su lista de verificación inicial, y todas funcionan. Luego, decide aventurarse un poco y agregar un par de bits adicionales para que su aplicación realmente brille. Pero aproximadamente a la mitad, se detiene, guarda su trabajo, inicia el servidor de desarrollo de Django para probar lo que tiene hasta ahora y... desastre. En lugar de ver su nueva y brillante aplicación en acción, recibe mensajes de error feos. Has introducido un error. ¿Pero donde está? Y, lo que es más importante, ¿cómo puede recuperar una versión anterior de su código que aún funcionaba?

Normalmente no tendría suerte: cada vez que realiza cambios en un archivo lleno de código y lo guarda, está destruyendo todo lo que tenía antes. Si sus cambios introdujeron un error en el código y no se dio cuenta a tiempo, se enfrenta a un problema importante.

Este es precisamente el tipo de problema que un sistema de control de versiones (VCS) puede resolver por usted.

En pocas palabras, un VCS es una pieza de software diseñada para hacer dos cosas:

- Realice un seguimiento de lo que cambió y cuándo, a medida que realiza cambios en sus archivos
- Proporcionar la capacidad de recuperar instantáneamente y sin dolor cualquier versión anterior de cualquier archivo has estado trabajando con

El uso de un VCS generalmente no impone muchos cambios en su flujo de trabajo de desarrollo.

La mayoría de las veces, simplemente agrega un paso adicional para llevar a cabo cada vez que guarda uno o más archivos, un paso adicional en el que "confirma" sus cambios.

Un ejemplo sencillo

Para la mayor parte de mi trabajo diario, uso un VCS llamado Mercurial. Me gusta porque es extremadamente simple y extremadamente poderoso. Además, está escrito en Python, por lo que puedo personalizarlo fácilmente si es necesario. Tiendo a trabajar casi exclusivamente en la línea de comandos, y Mercurial proporciona un comando llamado hg que me permite acceder a sus funciones. (El nombre del comando es un juego de palabras: "Hg" es el símbolo químico del mercurio).

Supongamos que quiero escribir un script de Python nuevo y simple. Creo un directorio para él, y en el directorio escribo:

inicio hg

Esto le dice a Mercurial que quiero que este directorio sea un *depósito de Mercurial*, una ubicación donde realizará un seguimiento de mis archivos y cualquier cambio realizado en ellos. A continuación, podría crear un archivo llamado hello.py y colocar la siguiente línea de código en él:

imprimir "¡Hola!"

Hasta ahora, Mercurial no conoce este archivo, pero puedo escribir un comando para decirle que este El archivo debe convertirse en parte de mi repositorio:

hg agregar hola.py

Y luego puedo comprometerme: crear un registro permanente de este archivo y su contenido, junto con un mensaje explicativo, escribiendo un comando más:

hg commit -m "Aregar archivo hello.py"

Ahora Mercurial conoce este archivo y hará un seguimiento de los cambios en el archivo a partir de este momento. También comenzará a mantener un registro de todos los cambios que he realizado, que puedo ver escribiendo **hg log**. La salida se ve así:

conjunto de cambios: 0: 55f0a856fa92

etiqueta: consejo
usuario: james bennett
fecha: mié 18 de marzo 01:16:24 2009 -0500
resumen: Agregar archivo hola.py

Ahora, supongamos que me gustaría cambiar hello.py para imprimir "¡Hola, ahí!" en lugar de solo "¡Hola!" yo puede cambiar el código para leer:

imprimir "¡Hola, allí!"

y luego guarde el archivo hello.py. Si ahora le pido a Mercurial que me muestre el estado actual del repositorio (escribiendo **hg st**), se mostrará:

```
M hola.py
```

Esto significa que el archivo hello.py ha sido modificado desde la versión más reciente que Mercurial lo sabe. Puedo usar hg commit para decirle a Mercurial que grabe la nueva versión:

```
hg commit -m "Cambiar mensaje impreso por hello.py"
```

Y ahora si pido un registro de cambios en mi repositorio, veré una nueva entrada:

```
conjunto de cambios: 1:50ca08429c16
```

```
etiqueta: consejo
usuario:      james bennett
fecha:        mié 18 mar 01:20:05 2009 -0500
resumen:     Cambiar mensaje impreso por hello.py
```

```
conjunto de cambios: 0: 55f0a856fa92
```

```
usuario:      james bennett
fecha:        mié 18 de marzo 01:16:24 2009 -0500
resumen:     Agregar archivo hola.py
```

Más importante aún, ahora puedo hacer dos cosas muy útiles. Primero, puedo ver un resumen de las diferencias entre las dos versiones de hello.py (el comando para hacerlo se llama hg diff, pero no lo mostraré aquí porque su salida puede ser algo complicada de leer). En segundo lugar, puedo volver instantáneamente a la versión anterior del archivo. Todo lo que tengo que hacer es escribir:

```
hg revertir -r 0 hola.py
```

El -r 0 significa "volver al número de revisión 0". A cada cambio en mi depósito de Mercurial se le asigna un número (a partir de cero), y el registro muestra cuál es ese número. Cada cambio también tiene un identificador mucho más largo (para la revisión 0, es 55f0a856fa92) que puede identificar de forma única el cambio incluso si se fusiona con otro repositorio con un número de cambio secuencial diferente. Pero por ahora eso no es demasiado importante para preocuparse.

Después de ejecutar el comando de reversión, Mercurial vuelve a poner hello.py como era originalmente. finalmente, imprimiendo solo "¡Hola!" Si estoy satisfecho con esa versión del archivo, puedo emitir otro compromiso para mantenerlo así:

```
hg commit -m "Restaurar el mensaje hello.py original"
```

Esto producirá una nueva entrada en el registro (número de revisión asignado 2). Ahora puedo seguir trabajando, con mi archivo de vuelta tal como lo quiero. Tenga en cuenta que hasta que emita una confirmación, Mercurial no registra ningún cambio de forma permanente, incluso si realizó ese cambio volviendo a una versión anterior de un archivo.

Un buen VCS puede hacer mucho más por usted, pero este ejemplo debería darle una idea general de por qué es tan útil. A cambio de un poco más de trabajo (recordar confirmar cada vez que cambia su código), un VCS le brinda una manera rápida y fácil de rastrear lo que ha hecho, ver el historial de cualquier archivo que haya estado trabajando y restaurar una versión anterior si accidentalmente estropea algo.

Herramientas de control de versiones y opciones de alojamiento

Hay bastantes VCS disponibles, y muchos de ellos son gratuitos para que cualquiera los descargue y use. Estos tres parecen ser los más populares:

- Mercurial (<http://www.selenic.com/mercurial/wiki/>)
- Git (<http://git-scm.com/>)
- Subversión (<http://subversion.tigris.org/>)

Los tres son sólidos y estables, y admiten las funciones básicas que necesita de un VCS.

Mercurial y Git son generalmente un poco más fáciles de poner en marcha para administrar su propio local.

trabajo de desarrollo, sin embargo; Subversion requiere un poco más de configuración y no facilita la creación rápida de nuevos repositorios controlados por versiones.

Además, los tres tienen buenas opciones de hospedaje gratuitas o de bajo costo, lo cual es importante si desea compartir su código con otros desarrolladores o ponerlo a disposición del público en general. Cualquier buen VCS tiene algún tipo de opción para permitir el acceso a un repositorio a través de HTTP u otros protocolos de red. De esta manera, tanto usted como otros desarrolladores pueden descargar una copia del código y, dependiendo de cómo se haya configurado el acceso al repositorio, cargar los cambios nuevamente al repositorio para que otros los vean.

Configurar un repositorio para que sea accesible a través de Internet y configurar los controles de acceso para garantizar que solo las personas de su confianza puedan publicar los cambios puede ser algo complicado, por lo que tener un servicio especializado de alojamiento de repositorios es extremadamente útil. Aquí hay algunos servicios populares de hospedaje de código:

- Para Mercurial, Bitbucket (<http://bitbucket.org/>) ofrece una variedad de planes de hospedaje, incluido uno que es gratuito. También proporciona un seguimiento básico de errores y la capacidad de configurar un wiki para su proyecto.
- Para Git, GitHub (<http://github.com/>) ofrece de manera similar planes gratuitos y pagos, y también proporciona un wiki de proyecto y un rastreador de errores.
- Para Subversion, Google Project Hosting (<http://code.google.com/hosting/>) es gratuito para proyectos de código abierto y proporciona seguimiento de errores y un wiki además de un repositorio. En el momento de escribir este artículo, Google también ha anunciado, pero aún no se ha hecho público disponible: soporte para alojar repositorios de Mercurial.

Elección y uso de un VCS

Si toma solo un consejo de este capítulo, que sea este: elija un VCS, aprenda a trabajar con él y utilícelo en todos sus proyectos. La cantidad de tiempo y problemas que ahorrará como resultado supera con creces el tiempo que dedicará a aprender a trabajar con él.

Incluso si esto suena un poco aterrador, no se preocupe; tiene muchas buenas opciones para aprender a trabajar con un VCS y para hacer que el proceso sea lo más fácil posible. Para cada una de las tres herramientas de VCS que he mencionado aquí, puede aprovechar tanto los libros completos como los complementos útiles que brindan interfaces más sencillas:

- Mercurial es el tema de *Mercurial: The Definitive Guide* de Bryan O'Sullivan, que está disponible de forma gratuita en línea en <http://hgbook.red-bean.com/hgbook.html>. Y TortoiseHg, disponible de forma gratuita en <http://bitbucket.org/tortoisehg/stable/wiki/Home>, proporciona una interfaz gráfica fácil de usar.
- Git es el tema de un libro editado por la comunidad, disponible en línea en <http://book.git-scm.com/>, así como varios libros impresos. Hay múltiples aplicaciones disponibles para proporcionar interfaces gráficas; consulte <http://git-scm.com/tools> para obtener una lista.
- El libro estándar sobre Subversion es *Version Control with Subversion* de Ben Collins Sussman, Brian W. Fitzpatrick y C. Michael Pilato, y está disponible de forma gratuita en línea en <http://svnbook.red-bean.com/>. TortoiseSVN, una interfaz gráfica gratuita para Subversion, está disponible en <http://tortoisessvn.tigris.org/>.

También hay muchos otros VCS disponibles además de estos tres; siéntete libre de investigarlos antes de decidir cuál te gustaría usar.

Uso de entornos de Python aislados para administrar Software

Cuando escribió su primera aplicación Django, el sistema de búsqueda para su CMS simple, usó el comando `manage.py startapp` de Django para crear los archivos directamente dentro del directorio de su proyecto. Como ya mencioné, hacer esto perjudica la reutilización de una aplicación, por lo que generalmente es mejor escribir aplicaciones como módulos que viven directamente en su ruta de importación de Python. De acuerdo con esta práctica, el proceso estándar de instalación de paquetes de Python coloca el código de cualquier módulo de Python de terceros o aplicaciones de Django en un directorio que se encuentra en su ruta de acceso de Python.

Pero colocar todo su código en directorios en su ruta de Python lo expone a un nuevo conjunto de problemas:

- Si tiene solo uno o dos directorios donde coloca los módulos de Python, volverse cada vez más concurrido.
- Eventualmente podría encontrarse con dos aplicaciones o bibliotecas diferentes que, sin embargo, tienen el mismo nombre de módulo de Python (este es un riesgo particular con los módulos que usan nombres genéricos como etiquetado), aunque solo se le permite un módulo de un determinado nombre.
- Tarde o temprano, tendrá dos aplicaciones diferentes que dependen de dos diferentes versiones de algún módulo en particular. Por ejemplo, es posible que tenga una aplicación anterior que requiera una versión anterior de la biblioteca y una aplicación más nueva que requiera una versión más reciente. Este es esencialmente el mismo problema que tener dos bibliotecas con el mismo nombre, pero ocurre con mucha más frecuencia.

Para abordar estos problemas de administración de código, puede configurar muchos directorios diferentes, cada uno con un conjunto particular de bibliotecas y aplicaciones, y luego cambiar su ruta de Python para que apunte a cualquiera con la que esté trabajando actualmente. Pero esto es tedioso, repetitivo y propenso a errores, justo el tipo de cosas que he estado tratando de ayudarlo a evitar.

Una mejor solución es usar herramientas automatizadas, y hay una herramienta muy buena que resuelve todos los problemas antes mencionados y más. Se llama virtualenv y puede descargarlo gratis en <http://pypi.python.org/pypi/virtualenv>.

Lo que hace virtualenv es crear un nuevo entorno de Python aislado o "virtual". Logra esto creando un nuevo directorio que contiene estos elementos:

- Una copia del intérprete de Python
- Un directorio para scripts de Python ejecutables
- Un directorio de paquetes de sitio para módulos de Python
- Una herramienta llamada easy_install que puede usar para descargar e instalar nuevos módulos de Python
- Scripts que puede utilizar para "activar" el entorno virtual

Este intérprete de Python se configurará para usar el directorio de paquetes del sitio creado por virtualenv en lugar del directorio de paquetes del sitio de todo el sistema que normalmente usa Python. Cuando el entorno virtual está activo, cualquier paquete de Python que instale (ya sea a través de easy_install, alguna otra herramienta o una instalación manual de setup.py) se instalará en el directorio de paquetes del sitio del entorno virtual.

Esto significa que, en lugar de lidiar con complicadas gimnasias de ruta de importación, simplemente puede crear un nuevo entorno virtual para cada proyecto en el que trabaje, y cualquier biblioteca que instale será "visible" solo para ese entorno. Esto resuelve todos los problemas de la lista anterior:

- Debido a que cada proyecto tiene solo las bibliotecas y aplicaciones que realmente necesita, no termine con directorios abarrotados en su ruta de importación.
- Debido a que los diferentes entornos virtuales no interfieren entre sí, puede tener dos proyectos que usan dos módulos diferentes con el mismo nombre: cada proyecto verá solo el módulo que está instalado en su entorno.
- Si tiene dos proyectos que necesitan diferentes versiones de una biblioteca, simplemente puede instalar la versión adecuada en el entorno virtual de cada proyecto. Una vez más, los dos entornos no interferirán entre sí y cada proyecto verá solo la versión de la biblioteca que necesita.

El uso de virtualenv tiene la ligera desventaja de crear copias adicionales del intérprete de Python y las bibliotecas estándar (y posiblemente varias copias de las bibliotecas que instala en diferentes entornos), lo que consume más espacio en su disco duro. Pero en casi un año de usar virtualenv (y crear una gran cantidad de entornos virtuales con él), no he notado ninguna pérdida significativa en el espacio disponible de mi computadora portátil. virtualenv hace un buen trabajo al mantener los entornos virtuales lo más livianos posible, y sus beneficios compensan con creces la cantidad insignificante de espacio en disco que utilizan los entornos.

Una vez que haya descargado virtualenv y lo haya instalado (elija el paquete fuente y use el script setup.py que proporciona para instalarlo), tendrá un script llamado virtualenv.py. Para crear un nuevo entorno virtual de Python, abra una línea de comando y escriba:

```
python /ruta/a/virtualenv.py django_environment
```

(Reemplace `/path/to/virtualenv.py` con la ubicación real del script `virtualenv.py` en su computadora).

Esto creará un nuevo directorio llamado `django_environment`, que contiene el nuevo entorno de Python. En su interior se encuentra un directorio llamado `bin`, cuyo contenido depende del sistema operativo de tu computadora:

- En Windows, habrá dos secuencias de comandos por lotes de Windows que usará para activar y desactivar el entorno virtual: `activar.bat` y `desactivar.bat`, respectivamente. Escribir `\ruta\al\virtualenv\bin\activate.bat` (con la ruta correcta para el directorio del entorno virtual) activa el entorno y hacer lo mismo con `deactivate.bat` lo desactiva.
- En los sistemas Mac OS X, Linux y UNIX, habrá un solo script llamado `activar`, que está escrito en el lenguaje de scripting bash estándar de UNIX. Para ejecutarlo, simplemente escriba fuente `activar` desde una línea de comando en el directorio `bin` del entorno. Para desactivar el entorno, simplemente cierre la ventana de la terminal o escriba el comando `desactivar`.

Una vez que haya activado su entorno virtual, al escribir **python** (en la misma sesión de línea de comandos) se ejecutará el intérprete de Python del entorno virtual. Luego, el intérprete buscará módulos de Python en el directorio de paquetes del sitio del entorno virtual.

Desde ahí puedes, por ejemplo, instalar Django escribiendo:

fácil_instalar Django

Esto descargará el último paquete de lanzamiento de Django y lo instalará en el directorio de paquetes del sitio del entorno virtual. También puede colocar cualquier otro módulo de Python que desee en el directorio de paquetes del sitio del entorno virtual, y solo ese entorno virtual podrá verlos.

Crear un nuevo entorno virtual cada vez que inicia un proyecto es un buen hábito, ya que simplificará en gran medida el proceso de instalación y administración del código de Python y evitará que los diferentes proyectos se tropiecen entre sí.

Si está utilizando Mac OS X, Linux o algún otro sistema UNIX, también le recomiendo que consulte `virtualenvwrapper` (<http://www.doughellmann.com/projects/virtualenvwrapper/>), que proporciona un conjunto de utilidades para que sea aún más fácil de administrar, activar y desactivar, y trabajar con entornos virtuales de Python creados por `virtualenv`. (Desafortunadamente, `virtualenvwrapper` no está disponible para Windows).

Y una vez que haya terminado el trabajo de desarrollo inicial en una aplicación y esté listo para implementarlo, `virtualenv` puede continuar ayudándolo a mantener las cosas en orden. En el servidor donde alojo mi sitio web personal y varios otros proyectos, tengo `virtualenv` instalado y creo un nuevo entorno virtual para cada sitio que implemento. Esto me permite evitar fácilmente que diferentes conjuntos de aplicaciones interfieran entre sí, lo que ayuda cuando tengo proyectos con requisitos conflictivos. Cuando estoy listo para implementar código nuevo, `virtualenv` me permite asegurarme de actualizar solo el código usado por los sitios específicos que estoy cambiando.

Además, el soporte para usar un entorno Python virtual creado por `virtualenv` está integrado en `mod_wsgi`, un módulo que permite que las aplicaciones web de Python se ejecuten en un servidor web Apache. Consulte <http://code.google.com/p/modwsgi/wiki/VirtualEnvironments> para obtener más información .

Uso de herramientas de compilación

Si bien el uso de virtualenv resuelve muchos problemas en el desarrollo normal de Python del día a día e incluso en la implementación, hay una tarea aún más difícil con la que no puede ayudar: rastrear todas las dependencias de su proyecto y crear *compilaciones reproducibles*. En pocas palabras, esta tarea implica crear rápidamente una copia completamente funcional del código de su aplicación y todas las bibliotecas u otras aplicaciones de las que depende, desde cero.

Hay muchas situaciones en las que necesitará poder hacer esto:

- Si está configurando su aplicación en un servidor web, deberá asegurarse de que todos los el código está presente y todo está configurado correctamente.
- Si usa personalmente varias computadoras (por ejemplo, una PC de escritorio en su hogar u oficina, y una computadora portátil en el camino), deberá asegurarse de que todos tengan copias idénticas de su aplicación y todo lo que necesita.
- Si está trabajando como parte de un equipo para desarrollar un proyecto más grande, todos deben tener el mismo código y todas las bibliotecas necesarias para admitirlo.

Si bien puede escribir manualmente un largo conjunto de instrucciones para todos y cada uno de los proyectos en los que trabaja, y luego revisarlo cada vez para asegurarse de configurar todo correctamente, usar un proceso automatizado es mucho más fácil. En general, el software que lo ayuda a configurar dicho proceso se llama una *herramienta de compilación*. Puede usar una herramienta de compilación para redactar una especificación de todas las cosas que necesita su aplicación, y luego puede ejecutarla para obtener todo y configurarlo exactamente como lo necesita.

Para Python, dos piezas populares de software pueden servir como herramientas de construcción:

- zc.buildout, disponible en <http://pypi.python.org/pypi/zc.buildout/>
- pip, disponible en <http://pypi.python.org/pypi/pip/> (pip en realidad se anuncia solo como un instalador de paquetes, pero brinda suficiente funcionalidad para funcionar como una herramienta de compilación/implementación en muchas situaciones comunes)

De los dos, zc.buildout ofrece más funciones, pero como resultado es un poco más complicado ponerlo en marcha. Por otro lado, pip no puede hacer tanto, pero podrá sumergirse en él más fácilmente. Recomendaría estudiar ambos para determinar cuál se adapta mejor a sus necesidades, pero para darle una idea de algunos ejemplos simples, le mostraré un poco de lo que puede hacer pip.

Especificamente, le mostraré cómo usar pip para crear, congelar y replicar un entorno.

Instalar pip es bastante fácil, especialmente si ya instaló la herramienta easy_install (partes de las cuales son un requisito previo). Como ya ha visto, virtualenv configura easy_install para usted automáticamente, por lo que en un entorno virtual simplemente puede escribir:

pip fácil de instalar

Y easy_install hará el resto.

En esencia, pip es una herramienta de instalación de paquetes (y, de hecho, pretende servir como una herramienta de reemplazo). Esto significa que puede usarlo para instalar rápidamente módulos de Python y rastrear automáticamente sus dependencias al mismo tiempo. Por ejemplo, puedes usar pip para instalar Django:

pip instalar Django

Pero donde pip realmente brilla es en una característica llamada *archivo de requisitos*. Esto es simplemente un simple archivo de texto que contiene una lista de paquetes, que se puede especificar mediante:

- El nombre del paquete, si el paquete aparece en el índice de paquetes de Python
- La URL de un paquete de Python, si el paquete no está en el Índice de paquetes de Python o si prefieres usar una versión alternativa de otro lugar
- La ubicación del repositorio controlado por versión de un proyecto en la Web (Mercurial, Git, y Subversion son compatibles)

Una vez que tenga un archivo de requisitos, puede pasar el nombre de ese archivo (en lugar de un paquete name) para pip install, e instalará todos los paquetes enumerados en el archivo, junto con las dependencias que esos paquetes especifiquen.

Más importante aún, pip proporciona una forma de crear un archivo de requisitos a partir de su conjunto actual de Módulos de Python instalados a través del comando pip freeze. Entonces, por ejemplo, si creó un entorno virtual con virtualenv e instaló su aplicación y todo lo que necesita, puede escribir esto en el entorno virtual:

```
pip congelar my_django_environment.txt
```

Esto creará el archivo my_django_environment.txt y completará una lista de todo lo instalado en su entorno virtual; en otras palabras, su aplicación y todo lo que necesita. Luego, puede, por ejemplo, cargar una copia de ese archivo en su servidor web y escribir esto (si tiene pip instalado allí):

```
pip instalar my_django_environment.txt
```

Esto instalará todas las aplicaciones y bibliotecas enumeradas, replicando el código que está en su entorno de desarrollo en su servidor.

También puede usar pip junto con virtualenv. Dentro de un entorno virtual activo, pip se instalará en el directorio de paquetes del sitio de ese entorno, pero también puede hacer que pip cree un nuevo entorno virtual para usted e instale todo en él, así:

```
pip install -E new_django_environment/ my_django_environment.txt
```

Si virtualenv está instalado, este comando creará el nuevo entorno virtual new_django_environment e instale todo, desde el archivo de requisitos.

Así que ahora puede configurar un flujo de trabajo simple pero extremadamente poderoso:

1. En su computadora, cree un nuevo virtualenv, instale pip en él y otros módulos de Python como los necesites.
2. Cuando esté listo para implementar su aplicación, use pip freeze para crear requisitos archivo de su entorno virtual.
3. En su servidor, use pip install para instalar todos los módulos necesarios (ya sea en una entorno que ya ha creado, o en uno que pip cree para usted).

También puede usar un proceso similar para reproducir fácilmente su entorno de desarrollo en una computadora diferente, o para permitir que nuevos compañeros de trabajo obtengan rápidamente sus propias copias y las ejecuten.

Tenga en cuenta que pip requiere que cualquier software que desee instalar esté disponible como Python estándar paquete, y zc.buildout (si decide usarlo en su lugar) funciona mejor con paquetes de Python. En el próximo capítulo, explicaré cómo puede crear un paquete desde una aplicación Django para que estas y muchas otras herramientas puedan trabajar sin esfuerzo con sus propias aplicaciones.

Uso de una herramienta de implementación

La última gran pieza de cualquier proceso de desarrollo práctico es una manera fácil de mover su código desde su propia computadora donde lo está desarrollando al servidor web donde se ejecutará (o varios servidores web, según sea el caso). Si bien las herramientas como pip y zc.buildout pueden ayudarlo a configurar inicialmente una copia funcional de su código, incluso el proceso relativamente simple de ejecutar las herramientas en uno o más servidores cada vez que actualiza algo puede volverse tedioso rápidamente. Además, algunas cosas, como los archivos de configuración y otra información de configuración, no encajarán en el tipo de empaquetado y flujo de trabajo de creación que proporcionan estas herramientas.

Por supuesto, podría simplemente usar FTP o un protocolo similar para cargar archivos a su servidor web cada vez que necesite hacer cambios, pero una vez más, eso se vuelve tedioso y repetitivo. Lo mejor es tener alguna forma de especificar lo que se debe hacer para una actualización y luego hacer que suceda automáticamente. Puede lograr esto con una herramienta de implementación.

Como con la mayor parte del material que cubro aquí, este es un tema extremadamente amplio sobre el cual usted querrá hacer al menos un poco de su propia investigación. Pero para darle una idea de lo que una buena herramienta de implementación puede hacer por usted, mostraré algunos ejemplos de una herramienta que uso llamada Fabric.

Fabric está, por supuesto, escrito en Python. Puede encontrarlo en línea en <http://www.nongnu.org/fab/index.html>, y tanto easy_install como pip pueden instalarlo automáticamente (mediante el comando pip install Fabric). Fabric proporciona un comando llamado fab, y la implementación gira en torno a escribir un tipo de script de Python llamado *fabfile*.

Como he dicho, normalmente uso Mercurial como mi VCS, y la mayoría de mis proyectos en realidad residen en repositorios de Mercurial a los que se puede acceder a través de la Web. (Por ejemplo, todo el código de este libro se mantiene en un repositorio de Mercurial alojado en Bitbucket). Esto significa que cuando realicé cambios en mi copia del código, puedo "empujarlos" al repositorio en línea y otras personas pueden "tirar" del repositorio para obtener el código actualizado.

Así que aquí hay un ejemplo de un archivo fab (que debe llamarse fabfile.py) que empujaría el código de mi copia de una aplicación a un repositorio en línea, iniciaría sesión en un par de servidores web, descargaría el código actualizado y volvería a cargar el solicitud:

```
config.fab_hosts = ['servidor1.ejemplo.com', 'servidor2.ejemplo.com']

def desplegar():
    local("cd miaplicacion/")
    local("empuje hg")
    ejecutar("cd /home/code/myapp/")
    ejecutar("hg tirar -u")
    sudo("reinicio de httpd")
```

Las diversas funciones proporcionadas por Fabric ejecutarán estos comandos:

- local() ejecuta un comando en su computadora. En este caso, navega al directorio que contiene la aplicación y ejecuta hg push para enviar los cambios al repositorio en línea.
- run() ejecuta un comando en el servidor web. Aquí, va al directorio que contiene la aplicación y ejecuta hg pull -u para actualizar el código con los últimos cambios en el repositorio en línea.
- sudo() ejecuta un comando en el servidor web, pero requiere privilegios administrativos (le pedirá que escriba una contraseña). Debido a que el código se acaba de actualizar, este script usa sudo() para reiniciar el servidor web y recargar la aplicación.

La variable config.fab_hosts simplemente contiene una lista de nombres de servidores en los que Fabric debe ejecutar comandos. Fabric utiliza Secure Shell (SSH) como una conexión cifrada para conectarse al servidor remoto y, si es necesario, le pedirá contraseñas cuando necesite iniciar sesión.

Ejecutar el script anterior es simple; simplemente escribirías:

implementación fabulosa

desde dentro del directorio que contiene el script fabfile.py, y Fabric ejecutaría el deployment() y ejecutar todos los comandos que especifica.

Fabric admite una serie de otras funciones útiles: carga de archivos y directorios, lo que permite una configuración detallada de los servidores en los que iniciar sesión, etc. Y debido a que el script fabfile.py está escrito únicamente en Python, es extremadamente extensible. Incluso este simple ejemplo debería darle una idea del poder de una buena herramienta de implementación y del tiempo que puede ahorrar al usar dicha herramienta.

Simplificando su proceso de desarrollo de Django

Hasta ahora, este capítulo se ha centrado en herramientas y técnicas que se aplican ampliamente a muchos tipos de proyectos, pero hay algunas cosas específicas de Django que puede hacer para simplificar el proceso de desarrollo e implementación de aplicaciones Django. Algunos de estos se combinan muy bien con otros consejos de este capítulo, pero la mayoría de ellos lo ayudarán con cualquier aplicación de Django, independientemente de cualquier otra herramienta que esté usando.

Vivir sin proyectos

Cuando comenzó su primer proyecto de Django al comienzo de este libro, usó django admin.py startproject, que creó un módulo de Python simple que contenía el archivo manage.py secuencia de comandos auxiliar, el archivo settings.py para su configuración y el archivo urls.py para la configuración de URL raíz. Este comando es útil, por supuesto, pero en realidad no es necesario.

Para que funcione, Django solo tiene que apuntar a un archivo de configuración válido y, de forma predeterminada, busca una variable de entorno llamada DJANGO_SETTINGS_MODULE para indicarle dónde encontrar la configuración. El valor de esta variable debe ser la ruta de importación de Python del archivo de configuración, como

como configuración de cms. (Tenga en cuenta que las variables de entorno son bastante diferentes de las variables de Python; por lo general, se aplican ampliamente a la forma en que trabaja con su computadora, como en el caso de la variable de entorno PYTHONPATH que configuró en el Capítulo 4 para decirle a Python dónde buscar su código).

Siempre que configure esta variable de entorno correctamente (por lo general, debe hacer esto de todos modos cuando implementa Django en un servidor web), no hay necesidad de manage.py, que simplemente configura DJANGO_SETTINGS_MODULE para usted y luego ejecuta cualquier comando que haya solicitado usando el mismo código que django-admin.py. (Esta es la razón por la que, en el Capítulo 8, pudo crear los archivos básicos para la aplicación de código compartido cab usando django-admin.py startapp).

El archivo urls.py tampoco es necesario; Django realmente determina la configuración de la URL raíz mirando la configuración ROOT_URLCONF, y django-admin.py startproject crea un archivo de configuración que apunta esta configuración al archivo urls.py del proyecto (por ejemplo, ROOT_URLCONF se configuró en cms. direcciones URL para su proyecto CMS de los Capítulos 2 y 3).

Finalmente, ya ha visto que desarrollar aplicaciones directamente como módulos de Python independientes, en lugar de colocarlos dentro del directorio de un proyecto, generalmente es mejor para la capacidad de reutilización. Verá más sobre ese tema en el próximo capítulo.

Por lo tanto, en realidad no se necesita un "proyecto" de Django. En lugar de crear proyectos cada vez que comience a trabajar en un nuevo sitio impulsado por Django, puede crear solo un directorio llamado config, y dentro de él directorios llamados settings y urls (y, como siempre, archivos __init__.py en cada uno para decirle a Python que estos directorios son módulos de Python). Luego, puede colocar los archivos de configuración y los archivos de configuración de URL para sus sitios en los directorios adecuados y consultarlos de manera coherente. (usando nombres como config.settings.site1, config.settings.site2, config.urls.site1, etc.).

Este tipo de configuración ofrece algunas ventajas. Si administra una gran cantidad de sitios impulsados por Django, es mucho más fácil tener toda la configuración en un solo lugar que buscar constantemente en muchos directorios de proyectos diferentes para encontrar lo que está buscando. Y si utiliza un esquema de nomenclatura coherente, le resultará mucho más fácil desarrollar herramientas automatizadas que conozcan todos los sitios que está administrando. Por ejemplo, un script que simplemente busque todos los archivos en un directorio de "configuración" será mucho más sencillo de escribir y mantener que uno que tenga que escanear varios directorios de proyectos.

Por supuesto, ya no tendrá la secuencia de comandos de ayuda de manage.py, pero django-admin.py puede hacer todo lo que puede hacer manage.py, y acepta un argumento --settings que le dice qué configuración usar. Entonces, si tiene un directorio de configuración/ajustes como se describió anteriormente y desea ejecutar syncdb para un sitio en particular, puede escribir:

```
django-admin.py syncdb --settings=config.settings.site1
```

Y django-admin.py se encargará de ello por usted. También puede simplemente configurar DJANGO_ variable de entorno SETTINGS_MODULE (aunque, como se explica en el Capítulo 4, el proceso para hacerlo varía según su sistema operativo).

Decidir si trabajar sin proyectos es adecuado para usted dependerá, por supuesto, de lo que esté haciendo. Si no planea tener muchos sitios impulsados por Django, podría ser más sencillo simplemente usar startproject para crear un directorio de proyecto para cada uno y usar manage.py como de costumbre. Pero si va a trabajar con una gran cantidad de sitios (más de cuatro o cinco) de manera regular, es posible que desee explorar una configuración "sin proyecto" para ver si puede facilitarle la vida. .

Uso de rutas relativas en la configuración

Cuando estaba configurando su primer proyecto Django, lo configuró para usar una base de datos SQLite, que está contenido en un solo archivo, y especificó un directorio donde se almacenarían sus plantillas. Las configuraciones relevantes, DATABASE_NAME y TEMPLATE_DIRS, simplemente se completaron con las ubicaciones apropiadas en su computadora, y funcionó bien.

Pero tan pronto como comience a pensar en implementar su aplicación en un servidor web, o tener varias personas trabajando en el mismo proyecto, tener este tipo de nombres de directorios y archivos codificados comienza a causar problemas. ¿Qué pasa si diferentes desarrolladores guardan las cosas en diferentes lugares en sus computadoras? ¿Qué pasa si algunos desarrolladores usan Windows y otros usan Mac OS o Linux (que usan diferentes formas de especificar ubicaciones en el sistema de archivos)? ¿Qué pasa con la

¿Servidor web? Probablemente no tendrá el mismo diseño de directorio que su propia computadora.

La solución a esto es evitar colocar este tipo de ubicaciones codificadas en su configuración, y eso es extremadamente fácil de hacer. Un archivo de configuración de Django es solo código de Python, y aunque en su mayoría se compone de variables con valores asignados a ellas, tiene todo el poder de Python disponible dentro de ese archivo.

Un caso simple sería un directorio de proyecto que contenga el directorio de plantillas además de los archivos settings.py, manage.py y urls.py. Entonces podría configurar TEMPLATE_DIRS así:

```
importar sistema operativo

PLANTILLA_DIRS = (
    os.path.join(os.path.abspath(os.path.dirname(__file__)), 'plantillas'),
)
```

El módulo os.path de Python le permite combinar y trabajar fácilmente con rutas de archivos y directorios, y sabe cómo generar cadenas que serán rutas válidas para el sistema operativo que está utilizando. Y cada archivo de Python tiene acceso a una variable especial llamada __file__, que contiene la ruta completa a ese archivo. Al juntar este módulo y la variable, el fragmento de código anterior establece TEMPLATE_DIRS para incluir un directorio llamado templates dentro del mismo directorio que el archivo de configuración, *sin importar dónde se encuentre el archivo de configuración*.

Puede hacer lo mismo con la configuración DATABASE_NAME, por ejemplo, para especificar una base de datos SQLite que resida en el mismo directorio que la configuración. Podría haber configurado la base de datos para su proyecto CMS simple de esta manera:

```
DATABASE_NAME = os.path.join(os.path.abspath(os.path.dirname(__file__)), 'cms.db')
```

Por supuesto, este proceso se vuelve repetitivo si lo hace para varias configuraciones. Para una mejor solución, calcule la ubicación del archivo de configuración una vez, guárdelo en una variable y consúltelo según sea necesario:

```
AJUSTES_DIR = os.path.abspath(os.path.dirname(__file__))
```

Entonces podría especificar la base de datos de esta manera:

```
DATABASE_NAME = os.path.join(AJUSTES_DIR, 'cms.db')
```

El uso de este tipo de *rutas relativas* (llamadas así porque son relativas a la ubicación del archivo de configuración) resuelve todos los problemas asociados con las rutas codificadas de archivos y directorios. Las rutas relativas hacen que sea trivialmente fácil para varios desarrolladores trabajar en el mismo proyecto, y le facilitan el uso del mismo archivo de configuración para el desarrollo en su computadora y para la implementación en su servidor web.

Manejo de configuraciones que cambian para diferentes entornos

Por supuesto, hay algunas configuraciones que simplemente no pueden permanecer iguales tanto para el desarrollo local en su computadora como para la implementación real en su servidor web. Por ejemplo, puede hacer su trabajo de desarrollo usando un archivo de base de datos SQLite, pero probablemente no quiera usar SQLite para su sitio real; lo más probable es que utilice una base de datos MySQL, PostgreSQL u Oracle servidor en su lugar. Aún así, sería bueno minimizar la cantidad de cambios que tiene que hacer.

Una vez más, el hecho de que un archivo de configuración de Django consista en código Python es importante. Django tiene una configuración llamada DEBUG, que está establecida en True de forma predeterminada (esto habilita cosas como las páginas de error que vio en capítulos anteriores, que mostraban mucha información útil de depuración cuando algo salía mal). Pero siempre querrá establecer DEBUG en False en un sitio en vivo. Entonces podría cambiar la configuración de la base de datos según el valor de la configuración DEBUG, así:

si DEPURAR:

```
MOTOR_BASE_DATOS = 'sqlite3'  
# ... otras configuraciones para usar con SQLite irían aquí
```

más:

```
MOTOR_BASE_DE_DATOS = 'mysql'  
# ... la configuración para usar con MySQL iría aquí
```

Luego, todo lo que tendría que hacer es cambiar el valor de la configuración DEBUG al cambiar entre desarrollo local e implementación en vivo. Y esto también funciona bien con otras configuraciones; por ejemplo, Django incluye un sistema de almacenamiento en caché que puede mejorar drásticamente el rendimiento de su sitio al permitirle almacenar datos en cualquiera de varios tipos de cachés. Cualquier cosa, desde los resultados de una consulta de base de datos compleja hasta una página web completamente renderizada, se puede almacenar en caché, lo que significa que puede ahorrarle mucho procesamiento a su servidor si tiene páginas que no cambian mucho o no necesitan ser personalizadas en por usuario. Una de las configuraciones involucradas en el almacenamiento en caché se llama CACHE_BACKEND, que puede configurar como ficticia (lo que significa que no se realiza el almacenamiento en caché) o en el nombre y la ubicación de un caché que debe usar Django. (Para obtener detalles completos, consulte la documentación de almacenamiento en caché de Django en <http://docs.djangoproject.com/en/dev/topics/cache/>.)

Debido a que probablemente no desee realizar ningún almacenamiento en caché durante el desarrollo, podría volver a utilizar DEBUG como señal para cambiar la configuración:

si DEPURAR:

```
CACHE_BACKEND = 'ficticio'
```

más:

```
# ... la configuración para el caché del sitio real iría aquí
```

Pero aunque esta opción es útil para cambiar los valores de algunas configuraciones, aún causa un par de problemas:

- El archivo de configuración se volverá más complicado, gracias a todos los diferentes opciones que deben configurarse de una forma u otra según el valor de DEBUG.
- Aún tiene que recordar cambiar el valor de la configuración DEBUG, y—porque otras configuraciones cambiarán cuando lo haga; es posible que no pueda realizar ninguna prueba local de su sitio sin DEBUG configurado en Verdadero.

Una solución más flexible es simplemente escribir su archivo de configuración como de costumbre e incluir todos los valores correctos para la implementación en vivo en su servidor web real, y luego agregar algunas líneas como esta en la parte inferior:

```
probar:  
    desde la importación local_settings *  
excepto ImportError:  
    pasar
```

Este código intentará encontrar un archivo llamado local_settings.py en el mismo directorio que el archivo de configuración. Si tiene éxito, importará todo lo definido en el archivo. Si no existe tal archivo, no pasará nada.

La ventaja de esta solución es que simplemente puede crear un archivo local_settings.py y en él coloque cualquier cosa que le gustaría anular para el desarrollo local. Cuando realmente implemente su sitio, ese archivo no existirá y nada se anulará, pero en su propia computadora, los valores definidos en su local_settings.py tendrán prioridad sobre la configuración normal.

(Debido a que esto sucede en la parte inferior de settings.py, cualquier valor que importe anulará la configuración definida más arriba en el archivo).

Esta solución también permite que cada desarrollador de su equipo mantenga su propia configuración local.py, que facilita mucho algunos tipos de personalizaciones. Por ejemplo, un desarrollador podría tener un servidor de base de datos ejecutándose en su computadora, por lo que podría usarlo para el desarrollo en lugar de un archivo de base de datos SQLite.

Incluso podría llevar esto más lejos y escribir código en el archivo de configuración para buscar configuraciones anuladas en múltiples ubicaciones diferentes, dependiendo de algún otro parámetro. Una vez más, el hecho de que el archivo de configuración sea solo código de Python hace que esto sea extremadamente fácil.

Pruebas unitarias de sus aplicaciones

Anteriormente, cuando estaba explicando algunas de las utilidades detrás del uso de un VCS, mencioné un caso común: realiza algunos cambios en su código y, de repente, deja de funcionar. Sería bueno tener alguna manera fácil de averiguar de inmediato si un nuevo cambio o una nueva pieza de código ha roto algo en su aplicación, para que pueda solucionarlo rápidamente. Eso es precisamente lo que las pruebas unitarias pueden hacer por usted.

Se podrían escribir (y se han escrito) bibliotecas enteras de libros sobre pruebas unitarias, pero en pocas palabras consiste en escribir pequeños fragmentos de código que prueban diferentes partes de su aplicación. Una prueba unitaria recibe su nombre del hecho de que cada una prueba alguna "unidad" de su código, alguna función o método en particular, por ejemplo, llamándolo y verificando que el resultado sea el esperado.

Django incluye un marco robusto de pruebas unitarias y lo alienta a escribir pruebas a medida que escribir código; es por eso que el comando startapp crea un archivo tests.py para usted. La documentación completa para el marco de prueba de Django está disponible en línea en <http://docs.djangoproject.com/en/dev/topics/testing/>, pero mostraré algunos ejemplos para darle una idea de cómo funciona y cómo es útil.

El marco de prueba de Django admite dos formas diferentes de escribir pruebas, ambas basadas en testing bibliotecas incluidas con Python. Uno de estos, basado en el módulo doctest de Python, le permite escribir pruebas que se parecen a lo que escribiría en un intérprete de Python. Es decir, escribes unas cuantas líneas de código con el mensaje >>> del intérprete delante de ellas seguidas de una línea que muestra cuál debería ser el resultado, y la prueba pasa si coincide con el resultado real de ejecutar ese código.

La otra forma de escribir pruebas se basa en el módulo unittest de Python; configurarlo requiere un poco más de trabajo, pero hace que las pruebas más complejas sean más fáciles de escribir.

Aquí hay un ejemplo simple de una prueba basada en unittest, que podría ir en el archivo tests.py para su aplicación de blog de coltrane:

```
de django.test importar TestCase
```

```
clase EntryTests(TestCase):
    def test_entry_archive_view(self):
        respuesta = self.client.get('/weblog/')
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(respuesta, 'coltrane/entry_archive.html')
```

Este configura un *caso de prueba*, una colección de pruebas unitarias que se ejecutarán juntas, con una prueba en él. Esta prueba utiliza una característica del marco de prueba de Django: cada objeto TestCase de Django tiene un atributo llamado cliente, que puede emitir solicitudes HTTP simuladas a su aplicación e inspeccionar las respuestas que devuelve. Específicamente, esta prueba (de la vista de archivo de entrada del weblog) emite una solicitud HTTP GET a la URL /weblog/, que (la prueba supone) es un índice de las entradas del weblog.

Luego hace dos *afirmaciones*: que la respuesta HTTP devuelta de esa solicitud tenía un código de estado de 200 (que HTTP define como "OK", lo que significa que no se produjeron errores), y que la vista utilizó la plantilla coltrane/entry_archive.html.

Si tenía este código en el archivo tests.py de su aplicación de blog y un proyecto con el aplicación weblog instalada (y las URL apropiadas configuradas), puede ejecutar:

```
python manage.py prueba coltrane
```

y se ejecutaría la prueba anterior. Si alguna de las afirmaciones en la prueba falla (por ejemplo, porque la vista devolvió un error del servidor o usó la plantilla incorrecta), el marco de prueba de Django le dirá cuál falló y por qué. Si ambas aserciones pasaron, el marco de prueba simplemente mostraría un mensaje diciendo que la prueba pasó.

Este ejemplo también demuestra la regla cardinal de las pruebas unitarias: cada prueba debe cubrir una y solo una parte lógica del código de su aplicación. El método único test_entry_archive_

view en este TestCase hace justamente eso: prueba el comportamiento de una vista. Un conjunto de pruebas adecuado para la aplicación de weblog también tendría métodos adicionales para probar cada una de las otras vistas, y tendría clases TestCase adicionales para probar las vistas en busca de enlaces, categorías y etiquetas, así como las funciones de moderación de comentarios.

Querrá leer la documentación de prueba de Django para tener una idea de todas las funciones. Además de lo que proporcionan las bibliotecas de prueba integradas de Python, Django agrega una serie de cosas, que incluyen:

- **El cliente HTTP de prueba:** Esto le permite enviar solicitudes a su aplicación e inspeccionar las respuestas.
- **Una serie de aserciones adicionales:** estas incluyen assertTemplateUsed, que vio en el código anterior. Esta afirmación verifica que una vista de Django usa una plantilla particular.
También puede hacer afirmaciones sobre el contenido del contexto de la plantilla.
- **Un sistema de carga de accesorios:** el marco de prueba de Django no usa su base de datos real porque eso podría sobrescribir o eliminar los datos en los que confía. En su lugar, crea una base de datos temporal que existe solo durante la ejecución de la prueba. Con la ayuda de Django, puede crear archivos llamados accesorios que contienen datos para cargar durante la ejecución de la prueba para que se pueda probar el código que realiza consultas a la base de datos.
- **Un sistema de correo electrónico simulado:** Esto le permite verificar el comportamiento de las aplicaciones que envían Email.

En general, es una muy buena idea escribir pruebas para una aplicación Django mientras escribe el código. sí mismo. De esa manera, cada vez que realice un cambio, puede ejecutar las pruebas (con "prueba de gestión.py" o "prueba de django-admin.py") para verificar que nada se haya roto y que las funciones nuevas que haya agregado funcionen correctamente. .

Las pruebas unitarias también pueden vincularse convenientemente con su VCS. Muchas herramientas de VCS ahora admiten una función llamada *bisección*, que lo ayuda a identificar el cambio exacto que introdujo un error. Necesita tres cosas para realizar la bisección:

- Un número de revisión para una versión del código que funcionó
- Un número de revisión (posterior) para una versión del código que no funciona
- Un comando que devuelve un código de salida estándar que indica éxito o fracaso (Django's el sistema de prueba hace esto, a través de la prueba de manage.py)

El VCS luego comenzará en un punto a mitad de camino entre las dos revisiones, ejecutará el comando especificado y verá si falla. Si el comando falla, el VCS retrocede hasta un punto medio entre la revisión "buena" y la que acaba de probar, y repite el proceso. (Si el comando tiene éxito, avanza a la mitad del camino entre la revisión que acaba de probar y la revisión "mala" conocida). Moviéndose de esta manera, el VCS finalmente identificará la revisión en la que el código dejó de funcionar.

Aunque requiere un poco más de trabajo y una sólida disciplina, escribir y mantener un conjunto completo de pruebas unitarias para cada aplicación que escriba tiene tantos beneficios que siempre vale la pena. Un buen conjunto de pruebas lo ayuda a identificar y corregir errores rápidamente y le brinda la confianza de que cuando implemente sus aplicaciones, funcionarán exactamente como usted desea.

Mirando hacia el futuro

En el próximo (y último) capítulo, cubriré un conjunto de principios para crear aplicaciones Django que puede usar y reutilizar en múltiples proyectos. Aunque ya se ha encontrado con algunas de las ideas básicas, la capacidad de escribir código una vez y reutilizarlo varias veces es una de las características más sólidas que ofrece Django, y merece una cobertura más detallada.

Sin embargo, antes de continuar, lo animo a que dedique algún tiempo a investigar algunos de los temas generales que se tratan en este capítulo y a pensar cómo puede adaptarlos a su flujo de trabajo de desarrollo. Incluso si nunca termina implementando una sola aplicación Django, la mayoría de las herramientas y técnicas que he mencionado pueden ayudarlo a convertirse en un programador de propósito general mejor y más eficiente.

Capítulo 12



Escribir Django reutilizable Aplicaciones

Así que Hasta ahora, este libro se ha ocupado principalmente de cubrir varios aspectos de Django en el contexto de la construcción de un conjunto de aplicaciones específicas. A través del proceso de escribir el código para esas aplicaciones, ha visto los principales componentes de Django en acción y ha aprendido cómo pueden reducir drásticamente la cantidad de trabajo necesario para crear aplicaciones web útiles. Pero eso es solo una pequeña parte de lo que Django puede hacer para ayudarlo a reducir el tiempo y el esfuerzo de desarrollo. Al fomentar ciertas prácticas recomendadas y facilitar su seguimiento mientras escribe, Django también lo ayuda a mejorar la calidad, la flexibilidad y la reutilización de su código. Y a la larga, esa es una ganancia mucho mayor.

Una y otra vez, has visto cómo los componentes incluidos en Django, o las aplicaciones incluido junto con él, puede ayudarlo a iniciar el proceso de desarrollo de una nueva aplicación al manejar tareas comunes para usted. Cuando estás desarrollando con Django, no necesitas preocuparte por escribir mucho código para manejar las consultas de tu base de datos. Es fácil enrutar direcciones URL específicas a partes de su aplicación o generar HTML a través de plantillas. Y cuando usa las aplicaciones incluidas con Django, puede obtener una gran cantidad de funciones de forma "gratuita". Por ejemplo, ha visto cómo Django proporciona funciones como cuentas de usuario y autenticación, generación de fuentes RSS, comentarios enviados por los usuarios e incluso una interfaz administrativa dinámica para el contenido del sitio.

A partir de ahí, el siguiente paso natural es considerar formas de escribir nuevas aplicaciones que pueda reutilizar una y otra vez, tal como reutiliza los propios componentes de Django y las aplicaciones incluidas en `django.contrib`. Las aplicaciones en `django.contrib` brindan buenos ejemplos para observar porque, aparte del hecho de que están incluidas en la descarga de Django, no tienen nada de especial o mágico. Todos ellos, incluso la interfaz administrativa, son simplemente aplicaciones que se han escrito teniendo en cuenta la flexibilidad y la reutilización, por lo que no son diferentes de cualquier otra aplicación Django bien diseñada.

A medida que adquiere experiencia con Django y comienza a crear una biblioteca de aplicaciones que ha escrito por usted mismo, descubrirá que desarrollar sus propias aplicaciones reutilizables es sorprendentemente fácil. Además, al hacerlo pone un poderoso recurso a su alcance: en lugar de volver a implementar una función en particular cada vez que la necesite, simplemente puede escribirla una vez y reutilizarla una y otra vez. Esto le da una ventaja impresionante en cada nuevo proyecto.

En este capítulo, analizaré en profundidad algunas pautas prácticas para desarrollar estos tipo de aplicaciones reutilizables, y le mostraré algunas técnicas específicas que pueden facilitar el proceso.

Una cosa a la vez

Un adagio popular en el desarrollo de software establece que un programa en particular debe “hacer una cosa y hacerlo bien”. Esto se remonta a los primeros días del sistema operativo UNIX, que consistía, en parte, en una colección de programas pequeños y simples que los usuarios podían encadenar para crear efectos poderosos. Debido a esto, UNIX a menudo se contrasta con los sistemas operativos que tienden a usar aplicaciones grandes y complejas repletas de muchas características.

Si bien las aplicaciones complejas tienen su lugar, la filosofía de construir un sistema a partir de una colección de partes independientes más pequeñas abre mucha flexibilidad. En lugar de realizar cambios en una pieza de software grande y complicada cuando necesita nuevas funciones y realizar un seguimiento de cómo interactúan todas sus funciones entre sí, puede crear diferentes arreglos de aplicaciones más simples y escribir código nuevo solo cuando aún no lo haga. tienes las piezas necesarias para construir lo que necesitas.

Aunque UNIX originalmente aplicó esta idea a tareas como el procesamiento de texto, este enfoque es igual de poderoso cuando se aplica al desarrollo web. Al mantener una biblioteca de aplicaciones pequeñas y autónomas, cada una de las cuales maneja alguna característica particular, usted obtiene la capacidad de reutilizarlas una y otra vez, en diferentes combinaciones y configuraciones, como bloques de construcción para nuevos sitios.

mantenerse enfocado

Uno de los mayores peligros en el desarrollo de software es el proceso de desplazamiento de *características* o *alcance*. Supongamos que tiene una idea para una característica interesante que está al menos algo relacionada con lo que está trabajando, así que continúa y la agrega. Pero una vez que esa característica está en su lugar, comienza a pensar en formas de desarrollarla y mejorarla con aún más características y capacidades, y comienza a escribir más y más código para admitir estas características. Eventualmente, terminas con un gran lío enredado que se ha desviado significativamente de su propósito original.

Sin embargo, cuando estás escribiendo código para un sistema modular como Django, a menudo es un poco más fácil para detectar las señales de advertencia de deslizamiento de características y volver a encarrilarse. Un sitio complejo con muchas características pero solo una pequeña cantidad de aplicaciones enumeradas en INSTALLED_APPS a menudo indica que una o más de las aplicaciones que está usando están tratando de hacer demasiado.

De manera similar, la estructura relativamente simple de una aplicación Django (modelos, vistas, URL y tal vez algunos formularios personalizados o etiquetas de plantilla) rápidamente comenzará a sentirse abarrotada si está tratando de incluir demasiadas funciones. A veces, realmente necesitará mantener una gran cantidad de clases de modelos o grupos lógicos de vistas y patrones de URL en una sola aplicación, pero a menudo la cantidad de trabajo de contabilidad que deberá realizar para mantener tanto código organizado indicará que su aplicación no está tan bien enfocada como podría ser.

Como regla general, la forma más fácil de mantenerse al día es responder una pregunta simple: “¿Qué hace esta aplicación?” En lugar de enumerar todas las características, intente resumir el propósito de la aplicación. Por ejemplo, con la aplicación de weblog, la respuesta a esta pregunta sería: “Déle al personal del sitio una interfaz sencilla para publicar entradas y enlaces en un weblog, y mantenga estas entradas organizadas a través de etiquetas y categorías temáticas”. Para django.contrib.auth, la respuesta sería: “Proporcionar un mecanismo para almacenar información de la cuenta del usuario y autenticar a los usuarios para que puedan interactuar con el sitio”.

Si encuentra que su respuesta a esta pregunta es demasiado larga, más de una oración o dos, en muchos casos, podría ser hora de dar un paso atrás y evaluar si su aplicación está tratando de hacer demasiadas cosas a la vez.

Una vez que tenga esta mentalidad, descubrirá que aborda las ideas de nuevas características con escepticismo. En lugar de pensar en las funciones únicamente en términos de lo genial que sería tenerlas en su sitio, también comenzará a pensar en términos de cómo se relacionan con el propósito de su aplicación. Esto hace que sea mucho más fácil eliminar cosas que no pertenecen y rechazarlas o archivarlas para implementarlas en otro lugar.

Ventajas de las aplicaciones estrictamente enfocadas

Una vez que esté desarrollando aplicaciones con este tipo de enfoque estricto, descubrirá que es mucho más fácil reutilizarlas. Por ejemplo, una aplicación bien enfocada suele ser mucho más sencilla de configurar e instalar, porque normalmente no tiene que preocuparse por configurar una gran cantidad de plantillas o realizar un seguimiento (y posiblemente capacitar al personal de su sitio para usar) muchas de nuevos modelos de datos.

También descubrirá que es mucho más fácil adaptar una aplicación estrictamente enfocada cuando se encuentra con situaciones en las que necesita agregar una nueva función o crear más flexibilidad, porque generalmente tiene menos código para revisar y editar y generalmente está bien organizado. . Muchas aplicaciones de Django extremadamente útiles consisten en solo tres o cuatro archivos cortos de código.

Finalmente, notará que de repente le resulta mucho más fácil lidiar con los problemas reales y específicos que su aplicación está tratando de resolver. Cuando ya no mantiene una gran cantidad de funciones no relacionadas en una sola aplicación, puede examinar el dominio de su problema particular con mucho más detalle y encontrar muchas soluciones más completas y flexibles.

Un buen ejemplo del mundo real de esto sería expandir el sistema simple de registro de usuarios que presenté en el Capítulo 9 para enseñarle sobre el sistema de procesamiento de formularios de Django. Sería tentador ir simplemente desde el formulario de registro básico del sistema y ver y comenzar a agregar funciones que tienen menos relevancia para el proceso de registro de usuarios. Por ejemplo, puede permitir que el usuario complete un perfil de usuario específico del sitio o configurar preferencias para controlar cómo se le presenta el sitio.

Sin embargo, ese es el comienzo de la fluencia de características. Si bien los perfiles de usuario y los sistemas de preferencias son características importantes y útiles, no tienen mucho que ver con el proceso de registro de usuarios, y solo hacer que ese proceso sea correcto puede ser bastante complicado por sí solo.

Por otro lado, una característica más relevante para el proceso de registro podría ser un paso de activación explícito, en el que le envía al nuevo usuario un correo electrónico indicándole que confirme la cuenta. Además, si necesita que los usuarios se registren en varios sitios, probablemente deba especificar diferentes formas de recopilar la información inicial de la cuenta. Por ejemplo, algunos sitios pueden necesitar que los nuevos usuarios lean y acepten los términos del servicio u otras políticas, mientras que otros pueden tener restricciones sobre quién puede registrarse. Finalmente, muchos sitios también quieren alguna forma de prevenir registros automáticos por parte de spambots. Muchos spambots pueden navegar automáticamente a través de un sistema de activación basado en correo electrónico, por lo que es posible que desee agregar detalles adicionales al proceso de registro, como generar opcionalmente una imagen con texto y requerir que el nuevo usuario la lea y escriba el texto en un campo del formulario.

Este es un escenario común en el desarrollo de aplicaciones: incluso algo que parece simple a primera vista puede tener mucha complejidad al acecho justo debajo de la superficie. Mantener sus aplicaciones bien enfocadas lo ayudará a mantener su atención en lidiar con esa complejidad, para que no termine con solo una solución parcial al problema que originalmente se propuso resolver.

Desarrollo de múltiples aplicaciones

La idea de que cualquier aplicación dada debe hacer una cosa y hacerlo bien es solo la mitad del proceso de construir sistemas complejos a partir de partes pequeñas e independientes. La otra mitad es la noción de que debes comenzar con una idea inicial y terminar desarrollando varias aplicaciones que implementen diferentes partes de ella.

Hasta cierto punto, esta es una consecuencia natural del desarrollo de aplicaciones estrictamente enfocadas. Si no se deja arrastrar por las funciones dentro de una aplicación dada, naturalmente terminará con una lista de funciones que le gustaría tener pero que lógicamente no pertenecen a esa aplicación. Entonces, el próximo paso obvio es desarrollar una aplicación separada con un enfoque apropiado para las funciones que desea implementar.

Adquirir el hábito de "desenrollar" nuevas aplicaciones siempre que tenga un nuevo conjunto de características para implementar puede ser complicado al principio, no solo porque es fácil ser víctima de la aparición de características, sino también porque es extremadamente tentador ver el desarrollo web en una forma que equipara una aplicación con un sitio web.

Ahora, a veces esto no es una mala idea. Por ejemplo, muchas herramientas populares de ging de weblog disponibles en el mercado adoptan este enfoque y brindan no solo funciones básicas como entradas y enlaces, sino también sus propias interfaces administrativas, sus propios sistemas de usuario y autenticación, sus propios sistemas de plantillas, etc. El desarrollo de una sola aplicación que proporcione todas las funciones de su sitio web puede ser una forma extremadamente útil de trabajar en ciertos casos, por ejemplo, cuando una aplicación en particular está dirigida a usuarios no técnicos o solo moderadamente técnicos que simplemente desean descargar e instalar una sola aplicación, paquete y tener su sitio funcionando inmediatamente.

Sin embargo, cuando escribe aplicaciones destinadas a ser utilizadas y reutilizadas por otros desarrolladores, o simplemente por usted mientras trabaja en diferentes proyectos, este puede ser un método desastroso para desarrollar una aplicación. Perdería rápidamente la capacidad de mezclar y combinar funciones específicas a medida que construye nuevos sitios y, por lo general, tendría que compensar agregando sistemas que le permitieran desarrollar complementos u otras adiciones a una sola aplicación grande. Esto solo aumenta la complejidad del código y la cantidad de trabajo que tiene que hacer cada vez que necesita agregar o reutilizar una característica.

La alternativa, ver un sitio web como una colección de aplicaciones estrechamente enfocadas, cada una de las cuales proporciona una función o un conjunto de funciones en particular, da como resultado mucha más flexibilidad y, a menudo, fomenta un mejor código dentro de cada aplicación, como ya ha visto. Django está diseñado para adaptarse a este estilo de desarrollo:

- **En lugar de manejar todo a través de una sola aplicación monolítica, Django tiene que especificar una lista de aplicaciones para usar (la configuración INSTALLED_APPS):**
También puede designar qué aplicaciones son responsables de qué funcionalidad configurando la configuración de la URL raíz.
- **En lugar de forzar que todo el código de un sitio en particular exista dentro de un solo Django usa la ruta de importación estándar de Python para buscar las aplicaciones que enumera en INSTALLED_APPS:** esto evita vincular su código a cualquier estructura de directorio específica y le permite reutilizar una sola copia de una aplicación en múltiples proyectos en lugar de tener que hacerlo sin cesar. cópielo en nuevos directorios de proyectos y mantenga todas esas copias actualizadas mientras trabaja en el código.

- A través de abstracciones como el modelo Site en django.contrib.sites, Django lo alienta a pensar en términos de reutilización de aplicaciones en múltiples sitios, incluso cuando esos sitios comparten una base de datos y posiblemente incluso una sola instancia de la interfaz administrativa: django.contrib.admin puede proporcionar fácilmente administración para múltiples sitios a través de una configuración llamada ADMIN_FOR, que enumera los módulos de configuración de todos los sitios para administrar.

El efecto neto de esto es que, aunque puede hacerlo si está realmente decidido, tratar de incorporar todas sus funciones en una aplicación grande a menudo le dará la sensación de que está nadando contra la corriente. Tan pronto como comience a dividir las cosas de manera lógica según la función, encontrará que el desarrollo es mucho más fácil.

Dibujar las líneas entre las aplicaciones

Por supuesto, esto plantea la cuestión de cómo saber cuándo debe dividir una función o un conjunto de funciones y comenzar a desarrollar una o más aplicaciones nuevas e independientes. Hasta cierto punto, aprender a reconocer la necesidad de desarrollar nuevas aplicaciones es algo que viene con la experiencia, pero puede seguir algunas buenas pautas generales para ayudar con la toma de decisiones. proceso.

La señal más obvia de que necesita comenzar a desarrollar una aplicación nueva e independiente es cuando descubre que hay una función particular, o algunas funciones relacionadas, que desea tener pero que lógicamente no pertenecen a la aplicación en la que está trabajando. . Por ejemplo, probablemente desee tener algún tipo de sistema de registro de usuarios de acceso público para acompañar la aplicación de código compartido de taxi que desarrolló en los últimos capítulos, pero ese sistema obviamente no pertenece a esa aplicación, por lo que debe desarrollarlo por separado.

Este proceso de toma de decisiones se vuelve algo más complicado cuando se consideran conjuntos de características que están al menos algo relacionadas. La discusión en la sección anterior sobre agregar perfiles de usuario y preferencias al sistema de registro es un buen ejemplo de esto, porque todas las características involucradas se relacionan de alguna manera con el manejo de cuentas de usuario. Puede presentar un caso para manejarlos juntos, porque casi siempre se usarán juntos. La mayoría de las veces, un sitio que tiene usuarios que se registran a través de un sistema de registro público también tendrá algún tipo de características o preferencias de perfil que pueden aprovechar.

En estos casos, suele ser útil pensar en términos de *ortogonalidad*. En general, en el software desarrollo, dos entidades son ortogonales si un cambio en una no afecta a la otra. Las preferencias de los usuarios, entonces, son ortogonales a los registros de usuarios, porque podría, por ejemplo, cambiar la forma en que funciona el proceso de registro (digamos, agregando un paso de activación explícito o construyendo medidas para derrotar a los robots de spam) sin cambiar la forma en que los usuarios configuran sus preferencias. Cuando las características son claramente ortogonales entre sí como esta, casi siempre pertenecen a aplicaciones separadas.

Finalmente, la reutilización puede ser un buen criterio para determinar si alguna característica en particular merece ser dividida en su propia aplicación. Si puede imaginar un caso en el que le gustaría usar esa función, y solo esa función, en otro sitio, es muy probable que deba estar en una aplicación separada para que la reutilización sea más fácil.

División de la aplicación de código compartido

Para ver un ejemplo instructivo de la aplicación de estas pautas, considere la aplicación de código compartido de cabina que desarrolló en los últimos capítulos. Lo desarrolló como una sola aplicación, pero es posible que haya notado que contiene varias características que podrían dividirse fácilmente en aplicaciones separadas (aunque todas serían necesarias si tuviera que implementar un sitio real de código compartido públicamente).

Por ejemplo, el sistema de calificación que desarrolló fue útil y necesario para las características sociales que deseaba tener, pero según las tres pautas enumeradas anteriormente (características no relacionadas, ortogonalidad y reutilización), sería un fuerte candidato para convertirse en propio. solicitud:

- **Funciones no relacionadas:** Proporcionar un mecanismo para que los usuarios califiquen fragmentos de código no está tan estrechamente relacionado con el objetivo principal de la aplicación, que es proporcionar los medios para que los usuarios envíen y editen los fragmentos en primer lugar.
- **Ortogonalidad:** El sistema de calificación es en gran parte ortogonal al resto de la aplicación. Por ejemplo, puede cambiarlo de una simple calificación "hacia arriba" o "hacia abajo" a una puntuación numérica o a un sistema en el que los usuarios otorgan calificaciones como "tres estrellas de cuatro", sin afectar la forma en que las personas envían, editan y fragmentos de marcador.
- **Reutilizar:** es fácil imaginar otros sitios o proyectos en los que le gustaría tener un sistema para que los usuarios califiquen el contenido, pero en los que no necesariamente le gustaría tener las funciones de fragmentos de código junto con él.

Lo mismo ocurre con el sistema de marcadores, casi precisamente por las mismas razones: no es relacionado con el "propósito" central de la aplicación (que, nuevamente, es la funcionalidad del fragmento de código). Es ortogonal a las otras características. Y proporcionar a los usuarios la capacidad de marcar sus piezas favoritas del contenido del sitio es algo que sería útil en muchos tipos diferentes de sitios.

Construyendo para la flexibilidad

La división lógica de la funcionalidad en múltiples aplicaciones es solo una parte del proceso de hacer que esa funcionalidad sea reutilizable. Como ya ha visto, es fácil imaginar un caso en el que incluso una característica aparentemente "simple" puede variar bastante de un proyecto a otro. Un buen ejemplo de esto sería un formulario de contacto. Muchos tipos diferentes de sitios necesitan algún tipo de función que permita a los visitantes completar un formulario y enviar información al personal del sitio, pero los casos de uso pueden variar enormemente. Por ejemplo, algunos sitios pueden querer un formulario que permita a los visitantes enviar un mensaje a los propietarios del sitio para proporcionar comentarios o informar problemas. Otros sitios, a menudo sitios de negocios, probablemente querrán recopilar más información e incluso querrán diferentes tipos de formularios para diferentes situaciones. Por ejemplo, un formulario podría manejar consultas de ventas, mientras que otro podría manejar solicitudes de servicio al cliente. Es posible que otros sitios deseen complementar las reglas de validación del formulario con comprobaciones de spam (quizás mediante el uso de Akismet o alguna otra forma de análisis automatizado).

Al principio, parece que no habría forma de desarrollar una sola aplicación que pueda manejar todos estos casos (y esto es solo una pequeña muestra de los casos de uso de un formulario de contacto). Es posible que sospeche que tendrá que esforzarse y escribir una versión diferente de la aplicación cada vez que la use. Sin embargo, con un poco de planificación y un poco de código, una aplicación Django puede volverse lo suficientemente flexible para manejar todas estas variaciones en el tema subyacente y más.

Manejo flexible de formularios

Si va a escribir una aplicación de formulario de contacto, puede comenzar definiendo un formulario de contacto simple como este:

```
desde django importar formularios
desde django.core.mail importar mail_managers clase
ContactForm(forms.Form): nombre =
    formularios.CharField(max_length=255) correo
    electrónico = formularios.EmailField() mensaje =
        formularios.CharField(widget=forms.Textarea()) def save(self):
            mensaje = "%s (%s) escribió:\n\n%s" % (self.cleaned_data['name'],
                self.cleaned_data['email'], self.cleaned_data['mensaje'])

            mail_managers(asunto="Comentarios del sitio", mensaje=mensaje)
```

Una vista simple llamada contact_form podría procesar este formulario:

```
from django.http import HttpResponseRedirect from
django.shortcuts import render_to_response from django.template
import RequestContext def contact_form(request): if
request.method == 'POST': form = ContactForm(data=request.POST)
    if form.is_valid(): form.save() return HttpResponseRedirect("/"
contacto/enviado/")
```

más:

```
    formulario = ContactForm()
    return render_to_response('contact_form.html', { 'formulario':
        formulario },
        context_instance=RequestContext(solicitud))
```

Para los casos más simples, esto estaría bien. Pero, ¿cómo podría manejar una situación en la que necesita usar un formulario diferente, uno con campos adicionales, por ejemplo, o reglas de validación adicionales?

La solución más sencilla es recordar que una vista de Django es simplemente una función y que puede definirlo para que tome cualquier argumento adicional que desee manejar. Puede agregar un nuevo argumento a la vista que especifica la clase de formulario a usar, y puede hacer referencia a ese argumento cada vez que necesite crear una instancia de un formulario desde dentro de la vista:

```
def contact_form(solicitud, form_class): if
    request.method == 'POST': form =
        form_class(data=request.POST) if form.is_valid():
            form.save() return HttpResponseRedirect("/"
contacto/enviado/")
```

más:

```
formulario = formulario_clase()
volver render_to_response('formulario_contacto.html',
{ 'formulario': formulario },
context_instance=RequestContext(solicitud))
```

Puede mejorar esto ligeramente proporcionando un valor predeterminado para el nuevo argumento:

```
def contact_form(solicitud, form_class=ContactForm):
```

Así es como funcionan muchos de los parámetros opcionales de las vistas genéricas de Django: la función de vista acepta una gran cantidad de argumentos y proporciona valores predeterminados razonables. Luego, si necesita cambiar ligeramente el comportamiento, simplemente pase el argumento apropiado.

Si está desarrollando un sitio comercial que desea manejar consultas de ventas a través de un formulario, puede definir una clase de formulario para manejar eso, quizás llamada SalesInquiryForm, y luego configurar un patrón de URL como este:

```
url(r'^consultas/ventas/$',
Formulario de contacto,
{ 'form_class': formulario de consulta de ventas},
name='formulario_consulta_ventas'),
```

El argumento `form_class` que pasa aquí anula el valor predeterminado en la vista `contact_form` y, siempre que recuerde definir un método `save()` en su clase `SalesInquiryForm`, simplemente funciona. Si necesita varios formularios de diferentes tipos, puede reutilizar `contact_form`

view varias veces, pasando un argumento `form_class` diferente cada vez, de la misma manera que reutilizó previamente las vistas genéricas pasando diferentes conjuntos de argumentos.

Manejo flexible de plantillas

Por supuesto, el simple hecho de cambiar la clase del formulario podría no ser de mucha ayuda, porque la vista siempre usará la misma plantilla (`formulario_contacto.html`) para representarla. Pero, una vez más, puede realizar un pequeño cambio en la vista y agregar cierta flexibilidad al manejo de la plantilla. En este caso, puede emular directamente las vistas genéricas de Django, que aceptan un argumento llamado `template_name` para anular la plantilla predeterminada que usarían:

```
def contact_form(solicitud, form_class=ContactForm,
template_name='formulario_de_contacto.html'):
if solicitud.método == 'POST':
    formulario = form_class(data=request.POST)
    si formulario.es_válido():
        formulario.guardar()
        devolver HttpResponseRedirect('/contacto/enviado/')
más:
    formulario = formulario_clase()
    volver render_to_response(template_name,
{ 'formulario': formulario },
context_instance=RequestContext(solicitud))
```

Luego puede cambiar el patrón de URL para especificar una plantilla diferente:

```
url(r'^consultas/ventas/$',  
    Formulario de contacto,  
    { 'form_class': formulario de consulta de ventas,  
      'template_name': 'sales_inquiry.html' },  
    name='formulario_consulta_ventas'),
```

Poder cambiar tanto el formulario que usa la vista como la plantilla que usa para mostrar ese formulario le brinda una gran flexibilidad para reutilizar esta vista. Ahora puede configurar fácilmente múltiples formularios y personalizar las plantillas para cada uno con cualquier presentación específica o instrucciones que desee agregar.

Procesamiento posterior al formulario flexible

Aquí falta una cosa más: independientemente de los argumentos que pase a la vista, siempre se redirigirá a la URL /contacto/enviado/ después del envío exitoso. Arreglemos eso agregando un último argumento llamado `Success_url`:

```
def contact_form(solicitud, form_class=ContactForm,  
                 template_name='formulario_de_contacto.html',  
                 Success_url='/contacto/enviado/'):  
    if solicitud.método == 'POST':  
        formulario = form_class(data=request.POST)  
        si formulario.es_válido():  
            formulario.guardar()  
            devolver HttpResponseRedirect(url Éxito)  
    más:  
        formulario = formulario_clase()  
    volver render_to_response(template_name,  
                               { 'formulario': formulario },  
                               context_instance=RequestContext(solicitud))
```

Ahora tiene control total sobre todo el proceso de visualización, validación y procesamiento del formulario:

```
url(r'^consultas/ventas/$',  
    Formulario de contacto,  
    { 'form_class': formulario de consulta de ventas,  
      'template_name': 'sales_inquiry.html',  
      'success_url': 'consultas/ventas/enviadas/' },  
    name='formulario_consulta_ventas'),
```

Ahora puede manejar todos los casos enumerados anteriormente (diferentes combinaciones de formularios, campos adicionales y validación adicional) con nada más complicado que pasar los argumentos correctos a la vista `contact_form`, exactamente de la misma manera que ha estado pasando argumentos a Vistas genéricas de Django. Podría agregar aún más flexibilidad a esta vista emulando algunos otros argumentos comunes aceptados por las vistas genéricas. Por ejemplo, el `extra_context`

Sería útil admitir el argumento para que las variables de plantilla personalizadas adicionales puedan estar disponibles.

Por supuesto, es importante no exagerar y agregar tantos argumentos que la vista se vuelve demasiado compleja para usar o escribir, y admite una gran cantidad de argumentos opcionales.

puede ser complicado. El equilibrio correcto entre flexibilidad y complejidad variará de una situación a otra, pero debe tratar de respaldar al menos algunos argumentos. Si bien no tiene que usar los siguientes nombres para ellos, elegir un conjunto estándar de nombres de argumentos y ceñirse a ellos mejorará en gran medida la legibilidad de su código. Además, cuando esté escribiendo una vista, es una buena idea dar a sus argumentos los mismos nombres que los argumentos similares aceptados por las vistas genéricas de Django. En mis propias aplicaciones, generalmente trato de admitir al menos el siguiente argumento:

- `form_class`, cuando estoy escribiendo una vista que maneja un formulario
- `success_url`, cuando estoy escribiendo una vista que redirige después de un procesamiento exitoso (de un formulario, por ejemplo)
- `template_name`, como en las vistas genéricas
- `extra_context`, también como en vistas genéricas

Además, siempre me aseguro de usar `RequestContext` para la representación de plantillas. Esto habilita tanto el conjunto estándar de procesadores de contexto, que agregan cosas al contexto como la identidad del usuario que ha iniciado sesión actualmente, así como cualquier procesador de contexto personalizado que se haya agregado a la configuración del sitio.

Manejo flexible de URL

En los ejemplos anteriores, el valor predeterminado para el argumento `Success_url` era una URL codificada. Sin embargo, en las aplicaciones que ha desarrollado en este libro, ha trabajado duro para mantenerse alejado de hacer eso. Por ejemplo, en los modelos, cuando definió `get_absolute_url()`, siempre usó el decorador `permalink()` para asegurarse de que usa una búsqueda de URL inversa basada en la configuración de URL actual. Y en sus plantillas, vio cómo usar el `{% url %}`

etiqueta para realizar una búsqueda de URL inversa similar y para asegurarse de que siempre genera las URL correctas para los enlaces.

Sin embargo, no ha encontrado este problema en una vista y ninguna de las soluciones que ha visto hasta ahora funcionará en este contexto. Pero hay otra función que hará lo que quieras: `django.core.urlresolvers.reverse()`. Este es en realidad el mecanismo subyacente tanto para el decorador `permalink()` como para la etiqueta `{% url %}`. Al usar el reverso, puede hacer referencia fácilmente a cualquier patrón de URL y hacer que busque automáticamente y genere la URL correcta. Entonces, si configura un patrón de URL con un nombre de `contact_form_sent`, por ejemplo, podría volver a escribir `contact_form` ver la lista de argumentos de esta manera (después de importar `reverse()`, por supuesto):

```
def contact_form(solicitud, form_class=ContactForm,  
                 template_name='formulario_de_contacto.html',  
                 exit_url=reverse('formulario_de_contacto_enviado')):
```

Y la URL adecuada se completaría mediante una búsqueda inversa en su módulo `URLConf` en vivo.

Siempre que necesite hacer referencia a una URL o devolverla, siempre debe usar la utilidad de búsqueda inversa que sea adecuada para lo que está escribiendo:

- `django.db.models permalink()`: Use este decorador cuando esté escribiendo el `get_` de un modelo método `absolute_url()` u otros métodos en un modelo que devuelven una URL.
- `{% url %}`: use esta etiqueta cuando esté escribiendo una plantilla.
- `django.core.urlresolvers.reverse()`: use esta función en cualquier otro código de Python.

Para facilitar el uso de las búsquedas inversas, cualquier módulo URLConf incluido en su aplicación debe dar nombres sensatos a todos sus patrones de URL (preferiblemente con el prefijo del nombre de la aplicación para evitar conflictos de nombres, como ha estado haciendo anteriormente con nombres de patrones de URL como `cab_snippet_detail`).

Aprovechando las API de Django

También vale la pena señalar que muchas de las propias API de Django funcionan de la misma manera, o de manera extremadamente similar, con muchos tipos diferentes de modelos. Por ejemplo, un QuerySet de Django tiene los mismos métodos, `all()`, `filter()`, `get()`, etc., independientemente del modelo contra el que termine consultando. Esto significa que a menudo puede escribir código que acepte un QuerySet como argumento y simplemente le aplique métodos estándar.

Advertencia : evaluación de QuerySet

Tenga en cuenta que cada objeto QuerySet individual evalúa y realiza su consulta solo una vez. Despues de eso, simplemente almacena una copia de sus resultados. En muchos casos, esto no será un problema, porque su código llama a métodos como `filter()`, que modifican el QuerySet original y fuerzan una nueva consulta cuando solicita resultados. Sin embargo, si no está modificando el QuerySet, querrá llamar a su método `all()` y trabajar con el nuevo objeto QuerySet que devuelve. Esto evitara cualquier problema potencial de un QuerySet ya evaluado con resultados rancios.

De manera similar, puede usar el ayudante de ModelForm que vio en el Capítulo 9 como una forma de generar rápida y fácilmente un formulario para agregar o editar cualquier tipo de objeto. Debido a que ModelForm funciona de la misma manera para cualquier modelo (aunque las personalizaciones, como la función de exclusión, generalmente se completan según el modelo), puede usarlo con cualquiera de los múltiples modelos, incluso si no sabe de antemano qué modelo estarás trabajando.

Mantenerse genérico

Además de escribir vistas que toman argumentosopcionales para personalizar su comportamiento, también puede crear flexibilidad en su código sin vista al no vincularlo a modelos específicos o ideas específicas de cómo debería funcionar. Para ver lo que quiero decir, piense en la aplicación de blog: cuando agregó la función de moderación de comentarios, hizo algunas suposiciones que limitaron su flexibilidad. La solución en ese caso fue usar el sistema de moderación incorporado de Django, que fue diseñado para ser verdaderamente genérico.

Y aunque el sistema de moderación de Django es un poco más complejo que la función de moderación de comentarios que escribió originalmente para el weblog, vale la pena con una flexibilidad increíble. Puede configurar un conjunto diferente de reglas de moderación para cada modelo en el que permita comentarios, y cuando necesite admitir reglas de moderación personalizadas que no están cubiertas por el código en la clase `CommentModerator`, puede subclasicarlo, escribir el código apropiado para sus reglas de moderación personalizadas y luego use esa subclase para manejar su moderación de comentarios.

Este es un tipo de situación que se repite con frecuencia en el desarrollo de aplicaciones de Django: una función que comienza ligada a una aplicación en particular, o incluso a un modelo en particular, resulta útil en otros contextos y se vuelve a escribir para que sea genérica. De hecho, así es precisamente como Django

Se desarrolló un sistema de moderación de comentarios. Comenzó como una pieza de código estrechamente ligada a una aplicación de registro web de terceros en particular y luego evolucionó hasta convertirse en un sistema de moderación genérico que podía funcionar con cualquier modelo en cualquier aplicación. En ese momento, se dividió en una aplicación separada (todavía de terceros), diseñada para mejorar y ampliar el sistema de comentarios de Django. Esa aplicación resultó ser bastante popular, por lo que en Django 1.1 las funciones de moderación se incorporaron directamente en django.contrib.comments, que es el lugar más lógico para que estén.

Distribución de aplicaciones Django

Una vez que haya escrito una aplicación para que pueda reutilizarla fácilmente, el paso final es hacerla fácilmente distribuible. Incluso si nunca tiene la intención de lanzar públicamente una aplicación que haya escrito, seguir este paso puede ser útil. Terminará con una buena versión empaquetada de su aplicación que puede copiar fácilmente de una computadora a otra, y un mecanismo simple para instalarla, lo que garantiza que la aplicación terminará en una ubicación que se encuentra en la ruta de importación de Python. .

El primer paso para crear una aplicación Django de fácil distribución es asegurarse de que está desarrollando su aplicación como un módulo que puede vivir directamente en la ruta de importación de Python, en lugar de uno que deba colocarse dentro de un directorio de proyecto. Desarrollar de esta manera hace que sea mucho más fácil mover una copia de una aplicación de una computadora a otra, o tener múltiples proyectos usando la misma aplicación. Recordará que las últimas dos aplicaciones que creó en este libro han seguido este patrón y, en general, siempre debe desarrollar aplicaciones independientes de esta manera.

Advertencia: código que está estrechamente relacionado con un proyecto

A veces tendrá un código que está estrechamente relacionado con un proyecto en particular. Por ejemplo, es algo común escribir una vista que maneja la página de inicio de un sitio y hacer que esa vista maneje requisitos que son tan específicos del sitio que no tendría sentido reutilizar esa vista en otros proyectos.

Si lo desea, puede colocar un código como este en una aplicación que esté directamente dentro del directorio del proyecto, pero tenga en cuenta que para casos comunes como este, no es necesaria una aplicación. Django no requiere que las funciones de vista estén dentro de un módulo de aplicación (las vistas genéricas propias de Django no lo están, por ejemplo). Así que simplemente puede poner vistas específicas del proyecto directamente dentro del proyecto. Solo necesita crear una aplicación si también está definiendo modelos o etiquetas de plantilla personalizadas.

Herramientas de empaquetado de Python

Debido a que una aplicación Django es solo una colección de código de Python, simplemente debe usar las herramientas estándar de empaquetado de Python para distribuirlo. La biblioteca estándar de Python incluye el módulo distutils, que proporciona la funcionalidad básica que necesitará: crear paquetes, instalarlos y registrarlos en el índice de paquetes de Python (si desea distribuir su aplicación al público).

La forma principal en que usará distutils es escribiendo un script, llamado convencionalmente `setup.py`, que contiene información sobre su paquete. Luego usará ese script para generar el paquete. En el caso más simple, este es un proceso de tres pasos:

1. En un directorio temporal (no uno en su ruta de importación de Python), cree un archivo `setup.py` vacío y una copia del directorio de su aplicación, que contenga su código.
2. Complete el script `setup.py` con la información adecuada.
3. Ejecute `python setup.py sdist` para generar el paquete; esto crea un directorio llamado `dist` que contiene el paquete.

Advertencia: una configuración para el envasado continuo

Una molestia menor con este proceso es que, como desarrollador de un paquete, debe tener una copia del código de la aplicación en el mismo directorio que el archivo `setup.py`; de lo contrario, no podrá generar el paquete. (Si simplemente está instalando un paquete que otra persona ha producido, no necesita hacer esto).

Si bien es bastante fácil hacer una copia temporal del código de su aplicación para que pueda crear el paquete, esto puede ser tedioso para hacer una y otra vez. En cambio, a menudo mantengo una estructura de directorios permanente que tiene un directorio para cada paquete que mantengo. Dentro de cada directorio se encuentra el script `setup.py`, cualquier otro archivo relacionado con el paquete y el código de la aplicación real. Luego coloco un enlace (un enlace simbólico en los sistemas UNIX o un acceso directo en Windows) al código de la aplicación en un directorio en mi ruta de importación de Python.

Descubrí que esta es una forma mucho más fácil de trabajar con una aplicación que evoluciona con el tiempo (y, por lo tanto, debe empaquetarse varias veces para diferentes versiones). Siéntete libre de usar una técnica o experimento similar para encontrar una configuración que se adapte a ti.

El otro método común para distribuir paquetes de Python usa un sistema llamado `setuptools`. `setuptools` tiene algunas similitudes con `distutils`: ambos usan un script llamado `setup.py`, y la forma en que usa ese script para crear e instalar paquetes es la misma. Pero herramientas de configuración agrega una gran cantidad de funciones además de las `distutils` estándar, incluidas formas de especificar dependencias entre paquetes y formas de descargar e instalar automáticamente paquetes y todas sus dependencias. Puede obtener más información sobre las herramientas de configuración en línea en <http://peak.telecommunity.com/DevCenter/setuptools>. Sin embargo, usemos `distutils` para el ejemplo aquí, porque es parte de la biblioteca estándar de Python y, por lo tanto, no requiere que instale ninguna herramienta adicional para generar paquetes.

Escribir un script `setup.py` con `distutils`

Para ver cómo funciona la biblioteca `distutils` estándar de Python, analicemos cómo empaquetar una aplicación simple. Vaya a un directorio que *no esté* en su ruta de importación de Python y en él coloque lo siguiente:

- Un archivo vacío llamado `setup.py`
- Un archivo vacío llamado `hola.py`

En hello.py, agregue el siguiente código:

```
print "¡Hola! ¡Soy una aplicación de Python empaquetada!"
```

Obviamente, esta no es la aplicación de Python más útil jamás escrita, pero ahora que tiene un poco de código, puede ver cómo escribir el script de empaquetado en setup.py:

```
from distutils.core import setup
setup(name="hola",
      version="0.1",
      description="Una aplicación de Python empaquetada simple",
      author="Tu nombre aquí",
      author_email="Su dirección de correo electrónico aquí",
      url="La URL de su sitio web aquí",
      py_modules=["hola"],
      download_url="URL para descargar este paquete aquí")
```

Ahora puede ejecutar python setup.py sdist, que crea un directorio dist que contiene un archivo llamado hola-0.1.tar.gz. Este es un paquete de Python y puede instalarlo en cualquier computadora que tenga Python disponible. El proceso de instalación es simple: abra el paquete (el archivo es un archivo comprimido estándar que la mayoría de los sistemas operativos pueden desempaquetar) y creará un directorio llamado hello-0.1 que contiene un script setup.py. Al ejecutar python setup.py install en ese directorio, se instala el paquete en la ruta de importación de Python.

Por supuesto, este es un ejemplo muy básico, pero muestra la mayor parte de lo que necesitará saber para crear paquetes de Python. Los diversos argumentos de la función de configuración en su archivo setup.py brindan información sobre el paquete y distutils hace el resto. Esto solo se complica si su aplicación consta de varios módulos o submódulos, o si también incluye archivos que no son de Python (como archivos de documentación) que deben incluirse en el paquete.

Para manejar múltiples módulos o submódulos, simplemente enumérelos en el argumento py_modules. Por ejemplo, si tiene una aplicación llamada foo, que contiene un submódulo llamado foo.templatetags, usaría este argumento para decirle a distutils que los incluya:

```
py_modules=["foo", "foo.templatetags"],
```

El script de configuración espera que el módulo foo esté junto a él en el mismo directorio, por lo que mira dentro de foo para encontrar foo.templatetags para su inclusión.

Archivos estándar para incluir en un paquete

Cuando creó el paquete de ejemplo anterior, el script setup.py probablemente se quejó de que no se encontraron algunos archivos estándar. Aunque técnicamente no son necesarios, normalmente se incluyen varios archivos con un paquete de Python, y distutils le avisa cuando no están. Como mínimo, debe incluir dos archivos en cualquier paquete que planee distribuir:

- **Un archivo llamado LICENSE o LICENSE.txt:** debe contener información de derechos de autor. Para muchos paquetes de Python, esto es simplemente una copia de una licencia estándar de código abierto con el nombre del autor debidamente llenado.
- **Un archivo llamado README o README.txt:** Esto debería proporcionar algunos datos básicos legibles por humanos. información sobre el paquete, su contenido y punteros a documentación o información adicional.

También puede encontrar estos otros archivos comunes en muchos paquetes:

- **AUTORES o AUTHORS.txt:** para el software desarrollado por un equipo de colaboradores, suele ser una lista de todos los que han contribuido con el código. Para proyectos grandes, esto puede crecer hasta un tamaño impresionante. El archivo AUTORES de Django, por ejemplo, enumera a todos los que han contribuido con el código del proyecto y ejecuta varios cientos de líneas.
- **INSTALL o INSTALL.txt:** suele contener instrucciones de instalación. A pesar de Todos los paquetes de Python ofrecen el mecanismo de instalación estándar setup.py, algunos paquetes también pueden ofrecer métodos de instalación alternativos o incluir instrucciones detalladas para casos especiales.
- **CHANGELOG o CHangelog.txt:** suele incluir un breve resumen de la aplicación histórica, anotando los cambios entre cada versión lanzada.

Incluir este tipo de archivos en un paquete de Python es bastante fácil. Mientras que el script setup.py especifica los módulos de Python que se empaquetarán, puede enumerar archivos adicionales como estos en un archivo llamado MANIFEST.in (en el mismo directorio que setup.py). El formato de este archivo es extremadamente simple y, a menudo, se parece a esto:

```
incluir LICENCIA.txt  
incluir README.txt  
incluir REGISTRO DE CAMBIOS.txt
```

Cada declaración de inclusión va en una línea separada y nombra un archivo que se incluirá en el paquete. Para uso avanzado, como empaquetar un directorio de archivos de documentación, puede usar una declaración de inclusión recursiva. Por ejemplo, si los archivos de documentación residen en un directorio llamado docs, puede usar esta declaración para incluirlos en el paquete:

```
recursivo-incluir documentos *
```

Documentación de una aplicación

Finalmente, una de las partes más importantes de una aplicación Django distribuible y reutilizable es una buena documentación. No he hablado mucho sobre la documentación porque me he centrado principalmente en el código, pero la documentación es esencial cada vez que escribe código que alguien más podría terminar usando (o que podría necesitar usar nuevamente después de no buscarlo). Un rato.

Una cosa que puede y debe hacer con frecuencia es incluir algunos archivos de documentación en el paquete de su aplicación. En general, puede asumir que otros desarrolladores sabrán cómo funcionan Python y Django, por lo que no necesita documentar cosas como usar setup.py install o agregar la aplicación a la lista INSTALLED_APPS de un proyecto Django. Sin embargo, debe explicar qué hace su aplicación y cómo funciona, y debe proporcionar al menos un resumen de cada uno de los siguientes elementos:

- Cualquier modelo proporcionado por su aplicación, sus usos previstos y cualquier administración personalizada usuarios o métodos personalizados útiles que ha configurado para ellos
- Una lista de vistas en su aplicación, junto con los nombres de plantilla que esperan y cualquier variables que ponen a disposición en el contexto de la plantilla
- Una lista de las etiquetas o filtros de plantillas personalizadas que haya proporcionado y lo que hacen

- Una lista de los formularios personalizados que haya proporcionado y para qué sirven
- Una lista de los módulos Python de terceros o las aplicaciones Django en las que se basa su aplicación e información sobre cómo obtenerlos

Además de estos esquemas o, más a menudo, como un precursor de ellos, también debe incluir documentación directamente en su código. Python facilita proporcionar documentación junto con el código que está escribiendo al proporcionar *cadenas* de documentación a sus módulos, clases y funciones de Python. Una docstring es simplemente una cadena literal de texto, incluida como lo primero en la definición de un módulo, clase o función. Para ver un ejemplo de cómo funciona esto, inicie un intérprete de Python y escriba:

```
>>> def suma(n1, n2):
...
...     """Suma dos números y devuelve el resultado.
...
...     """
...
...     devolver n1 + n2
...
...
...     """
```

Esto define una función simple y le da una cadena de documentación. Utiliza comillas triples (el ““ en el principio y final de la cadena de documentación) porque Python permite que las cadenas entre comillas triples se ejecuten en varias líneas.

Las cadenas de documentos terminan siendo útiles de tres maneras principales:

- **Cualquiera que esté leyendo su código también puede ver las cadenas de documentación y obtener información adicional.** **información de ellos:** Esto es posible porque están incluidos directamente en el código.
- **La herramienta de ayuda automatizada de Python sabe cómo leer una cadena de documentos y mostrarle información útil.** **información:** En el ejemplo anterior, podría escribir ayuda (agregar) en el intérprete, y Python le mostraría la firma del argumento de la función e imprimiría su cadena de documentación.
- **Otras herramientas pueden leer cadenas de documentación y ensamblarlas automáticamente en documentación en una variedad de formatos:** varias herramientas estándar o semiestándar pueden leer una aplicación completa, por ejemplo, e imprimir documentación organizada de las cadenas de documentación en formato HTML o PDF.

Documentación mostrada dentro de Django

Este último punto es particularmente importante, porque Django puede filtrar su código en busca de cadenas de documentos y usarlas para mostrar documentación útil a los usuarios. La interfaz administrativa generalmente contiene un enlace etiquetado como "Documentación" (en la esquina superior derecha de la página principal), que lleva al usuario a una página que enumera toda la documentación que Django puede producir (si las herramientas de documentación de Python necesarias están disponibles; ver la siguiente sección para más detalles). Esto incluye:

- **Una lista de todos los modelos instalados, organizados por las aplicaciones a las que pertenecen:** Para cada modelo, Django muestra una tabla que enumera los campos definidos en el modelo y cualquier método personalizado, así como la cadena de documentación de la clase del modelo.
- **Una lista de todos los patrones de URL y las vistas a las que se asignan:** para cada vista, Django muestra la cadena de documentación.
- **Listas de todas las etiquetas y filtros de plantilla disponibles, ambos del propio conjunto incorporado de Django y desde cualquier biblioteca de etiquetas personalizadas incluidas en sus aplicaciones instaladas:** para cada etiqueta o filtro, Django muestra la cadena de documentación.

Finalmente, darle a su código buenos docstrings le da una ventaja en la producción independiente documentación para su solicitud. De todos modos, es una buena práctica escribir cadenas de documentación útiles, porque muchas herramientas en Python las utilizan. Una vez que los tenga, puede copiarlos en archivos para usarlos como documentación de referencia independiente para distribuir con sus aplicaciones.

Qué documentar

En general, debe ser liberal al escribir cadenas de documentos para clases y funciones en su código. Es mejor tener documentación cuando no la necesitas que necesitar documentación cuando no la tienes. En general, la única vez que *no debe* preocuparse por darle a algo una cadena de documentación es cuando está escribiendo algo que es estándar y conocido. Por ejemplo, no necesita proporcionar una cadena de documentación para el método `get_absolute_url()` de un modelo, porque es un método estándar para definir modelos, y puede confiar en que las personas que lean su código sabrán por qué está ahí y qué está haciendo. . Sin embargo, si está proporcionando un método `save()` personalizado, a menudo *debe* documentarlo, ya que una explicación de cualquier comportamiento especial que proporcione será útil para las personas que lean su código.

Por lo general, una buena cadena de documentación proporciona una breve descripción general de lo que está haciendo el código asociado. La cadena de documentación de una clase debe explicar qué representa la clase, por ejemplo, y cómo se pretende que se use. La cadena de documentación para una función o método debe explicar lo que hace y mencionar cualquier restricción en los argumentos o el valor de retorno.

Además, al escribir cadenas de documentos, debe tener en cuenta los siguientes elementos, que son específicos de Django:

- **Las clases de modelo deben incluir información sobre los administradores personalizados adjuntos al modelo:** sin embargo, no es necesario que incluyan una lista de campos en sus cadenas de documentación, ya que se genera automáticamente.
- **Las cadenas de documentos para las funciones de visualización siempre deben mencionar el nombre de la plantilla que se utilizará:** además, deben proporcionar una lista de variables que están disponibles para la plantilla.
- **Las cadenas de documentación para las etiquetas de plantillas personalizadas deben explicar la sintaxis y los argumentos que esperan las etiquetas:** idealmente, también deben brindar al menos un ejemplo de cómo funciona la etiqueta.

Dentro de la interfaz de administración, Django puede formatear automáticamente gran parte de esta documentación si tiene instalado el módulo Python docutils (puede obtenerlo en <http://docutils.sourceforge.net/> si aún no está instalado en su computadora). El paquete docutils incluye una sintaxis liviana llamada reStructuredText (comúnmente abreviado como reST), y Django sabe cómo transformar esto en HTML. Si lo desea, puede usar esta sintaxis en sus cadenas de documentación para obtener una documentación con un buen formato.

Django también hace uso de un par de extensiones personalizadas a la sintaxis reST para permitirle referirse fácilmente a elementos específicos de Django, como clases de modelo o funciones de visualización. Para ver cómo funciona esto, considere una vista simple que podría entrar en su aplicación de blog de coltrane:

```
def últimas_entradas(solicitud):
    devolver render_to_response('coltrane/entry_archive.html',
                                { 'último': Entrada.objetos.todos()[:15] })
```

Ahora, nunca necesitará escribir esta vista, porque Django proporciona una vista genérica que tiene el mismo propósito, pero puede usarlo para mostrar algunos trucos de documentación. Aquí está la misma vista con una cadena de documentación útil:

```
def últimas_entradas(solicitud):

    Vista de las últimas 15 entradas publicadas. Esto es similar a
    la :vista:'django.views.generic.date_based.archive_index'
    vista genérica.

    **Modelo:**
    "coltrane/entry_archive.html"
    **Contexto:**
    "más reciente"
    """ Una lista de objetos :model'coltrane.Entry'.

    devolver render_to_response('coltrane/entry_archive.html',
                                { 'último': Entry.live.all()[:15] })
```

Mucho de lo que sucede aquí es bastante simple: los saltos de línea se convierten en saltos de párrafo en la documentación con formato HTML; los asteriscos dobles se convierten en texto en negrita para los encabezados; y la lista de variables de contexto se convierte en una lista de definición HTML, con el nombre de variable más reciente (rodeado de acentos graves) en una fuente monoespaciada.

Advertencia: Aprendizaje reStructuredText

Para la mayoría de los usos, no necesitará saber mucho más sobre la sintaxis reST de lo que se explica en el ejemplo. Sin embargo, si desea obtener más información al respecto, hay disponible en línea un manual completo y una extensa documentación (como es de esperar de una herramienta diseñada para facilitar la documentación) en <http://docutils.sourceforge.net/docs/user/rst/quickstart.html>. El paquete docutils también incluye herramientas para leer archivos escritos con sintaxis reST y generar resultados con un buen formato en HTML y otros formatos. Es una herramienta extremadamente útil para familiarizarse y se adapta a grandes proyectos de documentación. Por ejemplo, originalmente escribí y edité el texto de este libro en sintaxis reST antes de traducirlo a otros formatos para su publicación.

Sin embargo, aquí están ocurriendo dos cosas especializadas: la mención de una vista genérica y la mención del modelo Entry. Estos hacen uso de las extensiones específicas de Django y se transforman en un enlace a la documentación de la vista genérica y un enlace a la documentación del modelo Entry, respectivamente.

Además de los accesos directos :view: y :model: que se muestran en el ejemplo anterior, hay otros tres disponibles:

- :tag:: Debe ir seguido del nombre de una etiqueta de plantilla. Se vincula a la etiqueta documentación.
- :filtro:: Debe ir seguido del nombre de un filtro de plantilla. Se vincula con el filtro documentación.
- :template:: Debe ir seguido de un nombre de plantilla. Se vincula a una página que muestra ubicaciones en la configuración TEMPLATE_DIRS de su proyecto donde se puede encontrar esa plantilla, o no muestra nada si no se puede encontrar la plantilla.

Mirando hacia el futuro

Se puede decir mucho más sobre el desarrollo de aplicaciones Django para obtener el máximo uso posible y reutilizarlas, pero lo que he cubierto aquí es un buen comienzo.

Aprender cuándo aplicar estos principios generales a aplicaciones específicas y, lo que es igualmente importante, cuándo *no* aplicarlos (no existen reglas universales de desarrollo de software), se logra mejor a través de la experiencia de escribir y usar aplicaciones Django. Considere hacer una lista de ideas de aplicaciones que le interesen y pruebe algunas de ellas, incluso si nunca termina usándolas en una situación seria. Siéntase libre de regresar y jugar con las aplicaciones que ha creado en este libro. Hay mucho espacio para expandirlos y agregar nuevas funciones, o incluso para derivar aplicaciones completamente nuevas a partir de ellos. Además, tenga en cuenta que existe todo un ecosistema de aplicaciones Django ya escritas y disponibles en línea, lo que proporciona una gran base de código que puede estudiar.

Recuerde siempre que Django tiene una comunidad grande y amigable de desarrolladores y usuarios. que responden preguntas en listas de correo y en salas de chat. Entonces, cada vez que te quedes perplejo (y todos nos quedamos perplejos de vez en cuando), puedes recurrir a ellos en busca de ayuda.

Sobre todo, recuerde lo que mencioné en el Capítulo 1, cuando vio Django por primera vez: el trabajo de Django es hacer que el desarrollo web sea *divertido* nuevamente, liberándolo de todo el tedio y el trabajo repetitivo que tradicionalmente ha sido parte del proceso. . Así que encuentra una o dos ideas que te gusten, deja que Django se encargue del trabajo pesado por ti y diviértete escribiendo tu código.

Índice

SÍMBOLOS

(signo de almohadilla) para comentarios de Python, 13 % extiende % etiqueta, 99 ?P construcción, 66 __import__() función, 117 __init__.py archivo, 7, 27 __unicode__() método, 34, 62, 198 _default_manager, 120 + = (más igual) en el modelo de enlace, 85 % for % loop, 108–109 % free_comment_form % etiqueta, 126 % if_bookmarked % template tag, 191–192 % load % tag, 128 % url % template tag, 102, 232 500 Internal Error del servidor, 78

A

herencia abstracta, 124 script de activación, 211 función add(), argumentos a favor, 79 vista add_snippet, 178–179, 184 aplicación de administración agregando una nueva página plana a, 15–16 plantillas y, 24–25 sistema de documentación de administración, 22 formulario de administración para agregando una categoría, 51 para palabras clave de búsqueda, 36 interfaz de administración, agregando categorías a, 51 admin/patrón de URL, 13 admin/change_form.html plantilla, 25 admin/flatpages/change_form.html plantilla, 25 admin/flatpages/flatpage/change_form plantilla .html, 25 página de inicio de la interfaz administrativa, 13 plantillas/directorio usado por, 19 archivos admin.py, 156 configuración ADMINS, 136 método agregado, 199 consultas agregadas, 160 clave API Akismet, 131 importación de clase Akismet, 132 métodos en, 132 método de anotación, 161 API, Django, 233

creación de

aplicaciones para la flexibilidad, 228 desarrollo de múltiples, 226–228 frente a proyectos en Django, 44–45 razones para separar, 228 reconocimiento de la necesidad de generar nuevas, 227 técnicas para desarrollar técnicas reutilizables, 224 estrechamente enfocadas, 225 pruebas unitarias, 219 –221 archive_day, 71 archive_index, 71 archive_month, 71 archive_year, 71 argumentos, proporcionando valor predeterminado para nuevo, 230 método as_li(), 183 método as_p(), 182 método as_table(), 182–183 método as_ul(), 182 asociativo matrices, 27 Fuentes Atom, 140–141, 147 AUTORES/AUTHORS.txt archivo, 237

B

plantilla base_entries.html, 103 plantilla base_links.html, 104 plantilla base_tags.html, 104 BaseCommentAbstractModel, 123 plantilla base.html, 98 función bisectriz (VCS), 221 Bitbucket, 208 campos en blanco frente a campos nulos, 54 block.super variable, 99 bloque de adición de blog en la etiqueta del cuerpo, 102 argumento de palabra clave blog_url, 132 relleno de encabezado para, 101 bloque de barra lateral, 109 modelo de marcador

consultar los marcadores del usuario, 188 ejecutar consultas en, 188 atributo bookmark_set, 188 marcar vistas básicas de marcadores, 188 eliminar marcadores, 189 fragmentos, 187–188

BooleanField, 56

llaves, uso de etiquetas de plantilla, 126

sitios de folletos, 9 herramientas
de compilación, código de bytes
212–214, almacenamiento de Python de, 8

C

aplicación cab, 149–150 cab/
models.py, 187 cab/
snippet_form.html template, 184 cab/templatetags/
snippets.py, 192 cab/urls/popular.py, 163 cab/urls/
snippets.py agregando un nuevo patrón de URL a, 182
cambio de entrada de línea de importación,
184 archivo cab/views, 188–189 cab/views/
popular.py, 163 archivo cab/views/snippets.py
agregar importaciones para editar fragmentos, 183
terminado, 185 categorías modelo de categoría, 95
categoría objeto, 85 clase CategoryAdmin, 50
consideraciones para mostrar, 108 bucle, 108
configuración de vistas para, 84–85 categorización
para el modelo de enlace, 77 fuentes categorizadas que
agregan elementos para importar declaraciones, 144
problemas asociados con, 144 escritura de clase de
fuente para, 144 plantilla change_form.html, 25
archivos CHANGELOG/CHANGELOG.txt, 237
CharField, 57 opción de opciones, 57 método clean(),
171 método de validación clean(), 168 método
clean_username(), 167 diccionario de datos limpiados,
171 directorio de plantilla de elección de proyecto CMS
para, 19–20 personalización simple, 23 ensamblaje,
12–18 cms subdirectorio, creación, 6 código acoplado
a proyectos, 234 formulario para agregar fragmentos,
174–176 sistemas de control de versiones para rastrear,
205–209 directorio coltrane, 65–66 archivo coltrane_tags.py,
114 plantilla coltrane/entry_archive_year.html, 105

comas, finales, 20
modelo de comentarios, 123
moderador de comentarios, 130
método comment_check(), 133 en módulo
akismet, 132 argumentos esperados

por, 132 comentarios, 123 función de
moderación de comentarios, 134 sistema
de moderación de comentarios, características
de, 138–140

Clase CommentModerator, 233 comentarios

permitir y prohibir, 56 aplicación
(Django), 124–125 biblioteca de etiquetas
de comentarios, 125 sistema de envío
de comentarios, 129 notificación por correo
electrónico de, 135 moderación, 129

Python, 13
recuperar y mostrar, 127
commit=False en formularios, 182
cambio de función de compilación para
recuperar clase de modelo, 118 comprobación de
errores, 119 problemas con cambios, 118 escritura
para % if_bookmarked % etiqueta, 193 herencia
concreta, 124 método de conexión, 130 argumento
de vista de contact_form, reescritura, 232 efecto de
pasar argumentos correctos a, 231–232 creación de
aplicaciones de formulario de contacto para flexibilidad,
228–229 procesamiento flexible posterior al
formulario, 231–232 vista simple para procesamiento,
229

clase de contexto, 27
función de procesador de contexto, 111, 196
programas de conversión, HTML, 60 método
count() (QuerySet), 40 método create(), 176
método create_user(), 169 falsificación de
solicitud entre sitios (CSRF), 189–190 creación
simple de etiquetas personalizadas, 113 registro de
nuevas, 114 función de compilación de escritura para,
113

bases de datos

configuración de la BASE DE DATOS,
9–10, 217 usando diferentes, 10 filtros
de fecha para la aplicación de blogs, 66 archivos
basados en fechas, 70 restricciones basadas en
fechas admitidas por Django, 55 clase de fecha y hora, 53

coltrane/entry_archive.html plantilla, 104 coltrane/
entry_detail.html plantilla
agregando encabezado de formulario de
comentarios a, 125 creando, 68 editando, 99
plantilla coltrane/link_detail.html, 99, 111

- módulo `datetime`, 154
- `DateTimeField`, 53 extensión
- de archivo `.db`, 10
- configuración DEBUG (Django), 218–219
- decoradores, 74 palabra clave def, 28
- archivo `default.html`, 20 del `ocio.us`, 77
- Delicious, 77–78 herramientas de implementación, Proceso de desarrollo 214–215, Django

- rutas relativas en escenarios, 217–218
- escenarios que cambian con los ambientes, 218–219
- aplicaciones de prueba unitaria, 219–221
 - sin proyectos, 215–216 método `distinto()`, 38 distribución de aplicaciones Django, 234
- módulo `distutils`
 - para distribuir aplicaciones Django, 234 escribiendo el script `setup.py` con, 235–236 etiquetas `div`
- (elementos), 100–101 Django accediendo al archivo de configuración, 80 API, 233 construyendo el primer sitio, 9 script de administración integrado, 6 aplicaciones de contribución en, 12 sintaxis de búsqueda de base de datos, 29 objeto de sitio predeterminado creado por, 15 desarrollo de, 2 `DJANGO_SETTINGS_MODULE` variable de entorno, 215–216 `django-admin.py`, 9, 150, 216 `django.contrib`, 123 aplicación `django.contrib.admin`, 12 `django.contrib.auth`, 11, 56 `django.contrib.comments`, 123, 126, 127 `django.contrib.comments.moderation`.

- CommentModerator, 138
- aplicación `django.contrib.flatpages`, 12
- `django.contrib.sites`, 11, 15 aplicación
- `django.contrib.syndication`, 123
- `django.contrib.syndication.feeds.Feed` class, 141
 - `django.core.mail`, 137
- `django.core.urlresolvers.reverse()` función, 232
- `django.db.models.Count`, 160
- `django.db.models.get_model()` función,

- 117
- `django.db.models.permalink()`, 232
- `django.db.models.Sum` filtro, 198 función
- `django.shortcuts.render_to_response`, 31
 - aplicación de etiquetado django, 59

- módulo `django.template`, 18 clase
- `django.template.Context`, 196
- `django.template.loader.select_template`, 25
- `django.template.Node`, 114
- `django.template.Template` class, 112
- `django.utils.encoding.smart_str()` función, 80, 133
- `django.views.generic.date_based` module, 70

- módulo `django.views.generic.list_detail`, 86
- documentación mostrada dentro, 238–239 ejemplo de página plana, 20 manejo de consultas de base de datos, 94 instalación, 4–5 introducción a, 5 previsión, 8 lanzamientos empaquetados frente a código de desarrollo, 4 proceso para cargar plantillas, 112 página de error del servidor, 17 configuración de la base de datos, 8, 11 uso de extensiones personalizadas para REST

- sintaxis, 240
- Distribución de aplicaciones
 - Django, 234 qué documentar, 239 escritura, 44

- Proyecto Django
 - cambiando dirección y puerto, 6
 - configurando cms, 9–12 creando, 5
 - explorando, 8

- Plantillas de Django
 - para categorías, 109
 - análisis previo, 192
 - funcionamiento de, 112
 - docstrings, Python, 238–240
 - documentación para aplicaciones
 - distribuibles, 237 enlaces, 22

- E edición
 - de la vista `edit_snippet`, 197
 - fragmentos, 183–186 envío de correo electrónico desde
 - Django, 136 verificación de la configuración para, 136 entradas que agregan la lista de lo último, 115 categorización y etiquetado, 58
 - archivoentries.py, desglose, 91 detalle de entrada, 107 índice de entrada, 104, 105
 - Clase de modelo de entrada, 61–62, 130
 - plantillas de entrada, 72 plantilla de entrada_detalle, 97

vista entry_detail, 67
 entry_detail.html template, 127–128 entry_info_dict
 variable, 69 Entry.HIDDEN_STATUS, 58
 Entry.LIVE_STATUS, 58 Entry.objects.all(), 70
 consultas con el campo de estado establecido en
 Live, 93 tipos de, 57 escribir sin HTML, 60
 comprobación de errores, 117 campo de extracto, 54

F

Fabric software, 214, 215 arrastre
 de características, 224 entradas
 destacadas, 55–56 ejemplo de
 clase de fuente, 142 GUID de
 fuente, 142 agregación de fuentes
 a la aplicación de blog, 140
 categorizado, 144–147 directorio, creación de
 archivos en, 143 archivo feeds.py, 145 , 147
 clases de campos en el módulo
 django.newforms, 165 núcleo necesario para
 el modelo Link, 77–78 en Django, 29, 34 método
 filter() en QuerySet, 40 uso de entradas, 93 :filter atajo,
 241 accesorios (archivos), 221 Clase FlatPage, 27–28
 Objeto FlatPage, 21 plantilla flatpages/default.html, 30
 variable flatpage.title, 21 Foord, Michael, 132 Campo
 ForeignKey, 55 etiquetas for/endfor, 30 Clase de
 formulario añadiendo campos a, 175 en django.newforms
 módulo, 165 formularios para agregar fragmentos de
 código, 174 agregar el método personalizado __init__() a,
 174 campos, requisito para, 168 etiquetas de formulario y /
 formulario, 182 código de manejo de formularios en el
 módulo django.newforms, 165 formularios.py, 175 generar
 desde definición de modelo, 179–182 procesamiento en
 aplicación de código compartido, 165 representación en
 diferentes tipos de HTML, 182 plantillas de simplificación
 th en pantalla, 182–183 validación, 170–172 método
 full_clean(), 171 funciones frente a valores devueltos, 55

G

relaciones genéricas, 59, 60
 vistas genéricas, 86–87, 230
 método get_absolute_url()
 agregar a la interfaz de administración,
 62 agregar al modelo, 52 definir, 151 en
 el modelo de entrada, 66 reescribir en
 el modelo de entrada, 74

etiqueta get_comment_list, 137
 método get_content_object(), 131 método
 get_model(), 117–118 método get_object(),
 144 función get_object_or_404(), 68 función
 get_template, 29–30

GitHub, 208

Alojamiento de proyectos de Google, 208
 GUID (identificadores únicos globales), 141–142

H

tablas hash, 27
 texto de ayuda que

 se agrega a los campos de la interfaz de
 administración, 62 argumento, que se agrega al
 campo en el modelo, 51 opción oculta, 58 código
 resaltado, estilo, 158
 Botón de historial en página plana, 16
 opciones de alojamiento (VCS), 208
 HTML, agregar campos para almacenar, 60
 Encabezados HTTP (protocolo de transferencia
 de hipertexto), 134
 clase HttpResponse, 27
 Clase HttpResponseRedirect, 183

I

métodos HTTP idempotentes, 190 etiquetas
 if, 33, 66 IfRatedRoute, 200 declaración de
 importación, cambio para el modelo de
 calificación, 200 directivas include(), 90 función
 include(), 72 página de índice, elementos
 enumerados en, 15 resumen de herencia, 124
 concreto, 124 template, 98–99 input type=, 166, 182
 INSTALL o INSTALL.txt file, 237 INSTALLED_APPS
 configuración agregando aplicaciones a, 12–13
 agregando aplicación coltrane a, 48 cambiando,
 11 argumento de instancia, 131 IntegerField, 57
 is_public field, 131 item_pubdate () método, 141

método items()

- agregando a la clase de feed, 141
- cambiando para feeds categorizados, 145

j

jscripts/directorio, 23

k

argumentos de palabra clave

- en Python, 68 restricción
- única generada por, 78 palabras clave que mejoran la función de búsqueda de CMS con, 33 campo de palabra clave en el modelo de datos de Django, 34 método keyword_results[0].get_absolute_url(), 40

l

Modelo de lenguaje, 151, 176

LatestContentNode, escritura, 119–120

LatestEntriesFeed, configuración, 140 lexers in

pygments download, 151 bibliotecas, Django, 2–3

LICENSE/LICENSE.txt file, 236 linebreaks filter, 127

Link class, 77–81 Vincule el modelo agregando el

método save() personalizado a, 79 agregue una clave

externa a, 78 agregue más patrones a, 88 campos

básicos para, 77 diccionario de definición

- para vistas genéricas, 83 definición completa del modelo, 81 instale la tabla de la base de datos para, 81 plantilla link_detail, 98 servicio de agregación de enlaces, 78 archivo links.py, 91–92 plantilla utilizada para la vista genérica, 97 usando el método _unicode_() con, 79 escritura, 77–83 entradas en vivo, 93–95 módulo cargador, 27 decorador requerido por inicio de sesión, 177–178

vistas de inicio/cierre de sesión, 178

Manager, subclase de escritura de, 94

configuración de MANAGERS, 136 archivo

managers.py, 161 relaciones de muchos a

muchos commit=False y, 182 cómo funcionan, 59

ManyToManyField, 58–59 filtro de reducción,

127 archivos multimedia, 24 software Mercurial,

208, 214 *Mercurial: The Definitive Guide*, 209 Meta

clase, 50 metadatos, 78 módulo mod_wsgi, 211

clase ModelForm

agregando patrón de URL para, 184

personalización admitida por, 180 indicando

editar objeto existente, 184 usando, 180 modelos

diseño para la aplicación weblog, 47–52 clases de modelos, 62–63 definiciones de modelos, 179–182

ModelChoiceField, 175 ayudante de ModelForm, 233

recuperación de contenido de, 117–119 archivo models.py

agregando categoría a, 50 creación de un modelo de datos Django en, 33–34 parcial para aplicación weblog,

63 función mode_comment, 134 reglas de moderación, configuración, 233 sistema de moderación para filtrar comentarios entrantes, 129 plantilla de archivo mensual,

107 archivos mensuales/diarios, 106 aplicaciones múltiples, desarrollo, 226–228 MyModel.object_fetcher.

método all(), 94

norte

estilo de nomenclatura en Python, 28

paquete de formularios nuevos, 186

NodeList, 194 campos nulos frente a

campos en blanco, 54

o

vista genérica object_detail, 86, 97 vista

genérica object_detail, 72 vista genérica

object_list, 86, 158 mapeador relacional de

objetos (ORM), 22 atributo de objetos

(django.db.models.clase Manager), 94 ORDEN POR título ASC,

52 orden de URL patrones, 17 método order_by, 161

ortogonalidad, 227–228 módulo os.path (Python), 217

METRO

números mágicos, 58

función mail_managers(), 136–137 archivo

manage.py, 7 script de manage.py, 6, 216 comando

startapp de manage.py, 45 tablas de la base de

datos syncdb de manage.py creadas por, 13

instalación de tabla de modelo de categoría con,

- 48 en ejecución, 12 en ejecución para instalar

- el modelo en la base de datos, 188 administradores en el

- sistema modelo Django, 94

Construcción P ?

P, 66 herramientas de empaquetado, Python, 234–235 campo de página (modelo de datos Django), 34 Error de página no encontrada, 17 variables de página/paginador (modelo Snippet), 158 argumento del analizador, 193 contraseñas Widget PasswordInput, 167 validación, 168 función de patrones (), 85 decorador de enlaces permanentes, 74, 81 errores de permiso, 6 software pip, 212, 214 marcadores de posición, plantillas de escritura con, 98 atributos simples frente a métodos de fuentes, 145 señal post_save, 130 señal pre_save, 131 argumento prepopulated_fields, 50 plantilla preview.html, 126 claves principales, 35 proyectos frente a aplicaciones en Django, 44–45 creación de Django, 5 campo pub_date agregando orden predeterminado para, 79 proporcionando valor predeterminado para, 55 mostrando para blog, 107 argumento py_modules, 236 .pyc extensión, 8 módulo pydelicious, 79 función de resaltado de pygments, 154 método pygments.lexers.get_lexer_by_name(), 152 biblioteca de Python, 150 resaltado de sintaxis, 158–159 advertencia de Python sobre el aprendizaje, 3 sintaxis del decorador, 75 tutorial sobre la importancia de la lectura, 9 interpretación interactiva eter, 3 introducción a, 3–4 entornos aislados para software man

gestión, 209, 211 Módulo

Markdown, importación, 154 aplicaciones de nombres, 45 estilo de nombres, 28 python setup.py install, 236 python setup.py sdist, 236 sintaxis de expresiones regulares, 66–67 detener el servidor, 7 comprender los argumentos de la función, 68 módulos de Python que proporcionan cadenas de documentación, 238 instalación de terceros, 79

Paquete Python

índice, 150 herramientas de empaquetado, 234–235 archivos estándar para incluir, 236–237

Cambio de ruta

de Python, 46 poner código en el directorio, 46

Q q variable, 32

consultas agregadas, 160 ejecución (Django), 40

Clase

QuerySet (Django), 40 objeto, métodos en, 233 argumento queryset, 70 argumento queryset_or_model, 88

R

Clasificación de modelo/objeto, 199 archivo README/README.txt (Python), 236 declaraciones de inclusión recursiva, 237 expresiones regulares, 14 Reinhardt, Django, 2 argumento de nombre_relacionado, 188 rutas relativas en la configuración, 217–218 método render(), 113 , 194–195 función render_to_response, 31 compilaciones reproducibles, 212 RequestContext vs. Context, 196 importar, 196 llenar variables de plantilla con, 197 para representación de plantilla, 232 usar repetitivamente, 197 escritura de atajos para, 197 request.GET.get('q', '') método, 32 método resolve(), 194 aprendizaje reStructuredText (reST), 240 sintaxis, 240 valores devueltos frente a funciones, 55 editores de texto enriquecido (RTE), 23–26

S

métodos HTTP seguros, 190

Ejemplo de formulario de consulta de ventas, 230 método save() agregando código para el modelo de enlace a, 80–81 creando un objeto de usuario con, 169 en la clase de modelo, 61 motivo para no resaltarlo, 154 guardando un nuevo objeto de usuario con, 169 escribiendo para el modelo Snippet, 154 –155 desplazamiento del alcance, 224 búsqueda/directorio, 27

- buscar palabras clave, 36, 45
 - agregar sistemas de búsqueda
 - al proyecto CMS, 26 escribir vista de búsqueda para, 27 vista de búsqueda agregar HttpResponseRedirect
 - a, 39–40 agregar keyword_results a, 38–39 agregar soporte de palabras clave en, 38 mejorar en proyecto CMS, 31–33 reescribiendo para mostrar un formulario de búsqueda vacío, 32 trabajando sin agregar a INSTALLED_APPS, 35 consideraciones de seguridad en aplicaciones web, 33 método SELECT COUNT (QuerySet), 40 self.id, comprobando, 81
 - self.post_elsewhere+, 81 método de envío, 130 error del servidor página (Django), 17 cambios de configuración con entornos, 218–219 archivo, acceso (Django), 80 rutas relativas en, 217, 218 archivo settings.py, 8 script setup.py para empaquetado continuo, 235 para generar paquete de distribución, 235 escribir con distutils, 235–236 sistema de herramientas de configuración (paquetes de Python), 235 barra lateral
 - agregar explicaciones en, 102
 - agregar líneas a, 128 para blogs, 101–102 reescribir en plantilla base.html, 121 clase de señales, 130 y despachador de Django, 129–130 formulario de registro, 170 plantilla signup.html, 173 objeto de sitio, 132 sitio -directorios de paquetes, 46 campos de slug que cambian la definición de, 50 tipos de campo, 47 slugs que se agregan al modelo de enlace, 78 y normalización, 50 modelos Snippet que agregan un administrador personalizado a la definición de,
- fragmentos
- que generan automáticamente formularios para agregar, 182
 - marcadores favoritos, 187–188 edición, 183–186 e idiomas, 156 ordenamiento lógico para, 153 clasificación, 199
- Administrador de fragmentos, 199
- División de aplicaciones de fragmentos, 228 pruebas, 156 sitios de código
- social compartido
- construyendo en Django, 149
 - construyendo modelos iniciales, 150
 - lista de comprobación de características, 149 configurando una aplicación, 150 regla general de desarrollo de software para no perder el rumbo, 224 importancia de mantenerse enfocado, 224–225 avance del alcance, 224 escribiendo aplicaciones reutilizables en Django, 223
- spam, filtrado de comentarios para, 129
- método split_contents, 116 ataques de inyección SQL, 33 SQLite, 10 seguimiento de pila, 18 aplicaciones independientes y acopladas, 45 comando startapp, 27–31
- comando startproject (django-admin.py), 7 análisis estadístico de spam, 131–135 campos de estado, 58, 93 palabras vacías en campos slug, 50 función str() frente a smart_str() (Django), formato de 80 cadenas (Python), 52 tipos de, 34 función strftime (Python), 67 estilo guía (Python), 62 Subversion, 208 argumento Success_url, 231 llamada al método super(), 175 usando, 61 comando syncdb, 156

- 162
- creación de campos básicos, 153
 - variables adicionales para fragmentos, 157
 - campos en, 153 relleno del campo de autor, 174 terminado, 155–156 formulario terminado para, 176 configuración de plantillas para, 157 plantilla snippet_list, 158

T

- aplicación de etiquetado, 87
- tagging.views.tagged_object_list vista, 88 etiquetas que se agregan al modelo de enlace, 78 aplicación a modelos, 59 plantilla de detalles de entrada para, 110 sistema de plantilla que se extiende con personalización, 111
- proporcionado por el sistema de plantillas Django, 19

- registro y uso de nuevo, 120–122 método `tag()`, 114 modelo de etiqueta, 87 :atajo de etiqueta, 241 etiqueta URI, 141–142 TagField, importación al modelo Snippet, 153 `tagging.views.tagged_object_list` vista, 88 archivo `tags.py`, 92 usando nuevas, 114–115 vistas para, 87–88 escribiendo función de compilación para, 116–117 escribiendo más flexible con argumentos, 115–116 plantillas llamando a métodos de objetos, 30 encadenando heredado, 100 eligiendo entre múltiples, 25 creando para generar HTML, 29–30 base de definición para blog, 100 para mostrar entradas, 104 mostrar formularios con, 182–183 filtros, aplicar, 60 manejo flexible de, 230 cómo se determinan los nombres, 71 herencia, 97–100 para vistas genéricas del modelo de enlace, 84 cargadores, 19 para otros tipos de contenido, 110 acceso directo, 241 `tagging.views.tagged_object_list` vista, 88 `TEMPLATE_CONTEXT_PROCESSORS` conjunto, 196 `TEMPLATE_DIRS`, 19 `template.Node`, 193–195 templates/ directorio, 19–20 `TemplateSyntaxError`, 193 directorio `templatetags`, 192 sistema de plantillas en Django, 18–22 variables, 196 aplicaciones de prueba (prueba unitaria), 27, 219, 221 `TextFields`, 47, 175 convertidor de texto a HTML Markdown como, 61 método `save()` para aplicar, 78 módulos Python de terceros, instalación, 79 Configuración de TIME_ZONE, 11 instancia de clase `timedelta`, 131 TinyMCE, 23 elementos de título, adición de bloques para, 101 plantillas de título, renderizado para elementos de fuente, 143–144 herramientas, VCS, 208 autores principales, 160–161 vista `top_languages`, 162–163 vista `top_user`, reescritura, 162 código de seguimiento con VCS, 205–209 comas finales, 20 etiqueta `trans`, 127 filtro `truncatewords_html`, 66 tuplas que representan secuencias de elementos con, 20 utilizado por Python para el número de versión, 5 **tu** descomentando código, 13 restricción_única_por_fecha admitida por Django, 54 usada en el campo `slug`, 68 restricción_única_por_año, 55 restricciones de unicidad, 54–55 aplicaciones de prueba unitaria, 27, 219–221 patrones de URL agregando nuevos al modelo de enlace, 83–84 cambiando a especificar diferentes plantillas, 230–231 orden de, 17 reemplazar, 73 configurar el modelo de calificación, 200 limpiar el módulo `URLConf`, 93 realizar cambios en el blog, 74 proporcionado por la aplicación Django, 44 incorporar bits individuales, 93 en la aplicación del blog, 88–90 configuración de URL (localizadores uniformes de recursos), 14 desacoplamiento, 72–75 directorio, 157 manejo flexible de, 232–233 denominación de patrones, 152 configuración para agregar y eliminar marcadores, 190 configuración para `LatestEntriesFeed`, 143 archivo `URLConf (urls .py)`, 124 `URLField` (modelo de enlace), 77–79 `urls/snippets.py`, 157 cableado, 192 archivo `urls.py` agregando una nueva línea, 23–24 copiando instrucciones de importación y patrones de URL, 72–73 reparando, 17 reescribiendo para usar vistas genéricas para las entradas, 70 configurando en `cms directo ry`, 66 usuarios crear nuevo, 168 no especificar el actual como predeterminado, 56 y contraseñas, 169 sistema de calificación, 198–202 registros, 165–167 modelo de usuario, importar al modelo Snippet, 153 Excepción `User.DoesNotExist`, 167 campo de nombre de usuario, validación, 167–168 nombre de usuario /contraseña en `del.icio.us`, 80 variable, 196

V

validación

- personalizada para formularios de registro, 166 formularios, visualización/procesamiento, 172–174 orden de, 171–172
- Excepción ValidationError, 168
- variables proporcionadas por el sistema
 - de plantilla de Django, 19
 - Clase de variable, 194
- opciones de nombre_verbose, 50
- método de verificar_clave(), 132
- elección/uso de sistemas de control
 - de versiones (VCS), 208–209
 - ejemplo, 206–207 descripción general, 205–206 opciones de alojamiento y herramientas, 208
 - control de versiones con Subversion, 209
 - vista función, 28
- Botón Ver en el sitio, 16 vistas
- agregando nuevos argumentos
 - a (Django), 229–230 agregando top_languages al modelo Snippet, 163
 - crear un archivo para calificar fragmentos, 199–200 desventajas de cambiar la escritura a mano, 196
 - genérico (Django), 70, 233
 - manejo específico del proyecto, 234
 - para la clase HttpResponseRedirect, 183–184 mejora de los principales autores, 161–162 para idiomas, 159–160 para enumerar los marcadores de usuario actuales, 191 inicio/cierre de sesión, 178 consulta de la mayoría de fragmentos marcados, 191
 - configuración para categorías, 84–85
 - configuración para el modelo Snippet, 158
 - especificación de prefijos para, 73
 - comenzando con un índice simple, 65
 - para etiquetas, 87–88 usando coltrane/category_detail, 85 usando genérico (Django), 69, 86–87 vistas archivo .py, 27, 65 escritura en formulario de proceso, 177–179 herramienta virtualenv, 210–211

W

- aplicaciones web, seguridad y, 33 desarrollo web, 226–228 marco web definición y uso de, 1–2 usando Django como, 1–8 páginas
 - web, Django vs. escrito a mano, 19
 - servidores web, iniciando para ver la interfaz administrativa, 13 sitios web, para descargar el software Fabric, 214 GitHub, 208

pip, 212

- herramienta virtualenv, 210 zc.buildout, 212
- sitios web, para obtener más información, servicio web Akismet, 129, 131 código fuente de Apress/área de descarga, 163 documentación del marco de tipos de contenido, 163
- Documentación del sistema de autenticación de Django, 178 Documentación de la API de la base de datos de Django, 41, 94 Instrucciones de instalación de Django, 10 Documentación de configuración de Django, 10 Aplicación django.contrib.syndication, 141 Documentación de django.newforms, 186 descargar Django, 4 para descargar Python, 3 Alojamiento de proyectos de Google, 208 Estándar IETF RFC 4151, 141 Mercurial, 208 pydelicious, 79 pygments Biblioteca de Python, 150 Documentación de Python, 52 Guía de estilo de Python en línea, 28 herramientas de configuración información del sistema, 235 aplicación de fragmentos, 149 TinyMCE RTE, 23 campos básicos de aplicación de weblog en, 53–54 modelo de construcción para entradas, 52 creación independiente, 45–47 creación de plantillas para cada vista, 71–72 creación de directorio de direcciones URL en, 90 con tecnología de Django, 43 en expansión, 77–95 lista de verificación de funciones, 43–44 finalización, 61–62, 123 tareas de vista genérica, 69 instalación de django.contrib.comments, 124–125 nuevos tipos de campo en, 47 plantillas de sección para, 103–104 plantillas para, 97 índice de visualización de todas las entradas creadas en, 66 escribir Modelo de enlace para, 62–65 escribir las primeras vistas para, 65–69 clases de widgets (módulo django.newforms), 165

Windows, zonas horarias en, 11

Y

- archivo anual, 105–106

Z

- software zc.buildout, 212
- formato zoneinfo, 11

You Need the Companion eBook

Your purchase of this book entitles you to buy the companion PDF-version eBook for only \$10. Take the weightless companion with you anywhere.

We believe this Apress title will prove so indispensable that you'll want to carry it with you everywhere, which is why we are offering the companion eBook (in PDF format) for \$10 to customers who purchase this book now. Convenient and fully searchable, the PDF version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find and copy code—or perform examples by quickly toggling between instructions and the application. Even simultaneously tackling a donut, diet soda, and complex code becomes simplified with hands-free eBooks!

Once you purchase your book, getting the \$10 companion eBook is simple:

- ① Visit www.apress.com/promo/tendollars/.
- ② Complete a basic registration form to receive a randomly generated question about this title.
- ③ Answer the question correctly in 60 seconds, and you will receive a promotional code to redeem for the \$10.00 eBook.

Apress
THE EXPERT'S VOICE™



2855 TELEGRAPH AVENUE | SUITE 600 | BERKELEY, CA 94705

All Apress eBooks subject to copyright protection. No part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. The purchaser may print the work in full or in part for their own noncommercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.

Oferta válida hasta el 12/09.